# EFFICIENT MAPPING OF ADAS (ADVANCED DRIVER ASSISTANCE SYSTEM) ALGORITHMS ONTO MULTICORE ARCHITECTURES

by

Kerem Par

B.S., Computer Engineering, Boğaziçi University, 1991

M.S., Computer Engineering, Boğaziçi University, 1993

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2013

EFFICIENT MAPPING OF ADAS (ADVANCED DRIVER ASSISTANCE SYSTEM) ALGORITHMS ONTO MULTICORE ARCHITECTURES

APPROVED BY:

| | |
|---|---|
| Prof. Oğuz Tosun<br>(Thesis Supervisor) | ..... |
| Prof. Can Özturan | ..... |
| Prof. Haluk Topçuoğlu | ..... |
| Assoc. Prof. Alper Şen | ..... |
| Assist. Prof. Zeki Bozkuş | ..... |

DATE OF APPROVAL: 21.01.2013

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Prof. Oğuz Tosun for his invaluable guidance and help throughout this study. I greatly appreciate him for his contribution. It was an invaluable experience for me to work with him.

I am grateful to Prof. Haluk Topçuoğlu and Prof. Can Özturan for serving on my thesis committee and for their helpful comments. I would like to thank to Prof. Alper Şen and Prof. Zeki Bozkuş for their participation in my thesis jury.

I would like to present my thanks to Prof. Füsun Özgüner, Prof. Ümit Özgüner, Prof. Tankut Acarman, and Prof. Okyay Kaynak for their contributions and help.

I acknowledge the Boğaziçi University Scientific Research Fund (BAP) for supporting this research under the contract number 5522.

I also would like to thank my colleagues Ufuk Peker, Ali İhsan Danışman for their continuous encouragement and support. Special thanks to Mustafa Erdem for his invaluable contribution and help.

Finally, I would like to thank to my family, my parents, my wife Hürrem and my son Berk for their love, encouragement, patience and support which made this thesis possible.

# ABSTRACT

## EFFICIENT MAPPING OF ADAS (ADVANCED DRIVER ASSISTANCE SYSTEM) ALGORITHMS ONTO MULTICORE ARCHITECTURES

This thesis aims to address the real-time performance requirements of ADAS (Advanced Driver Assistance System) and autonomous vehicle applications on emerging multicore CPU and manycore GPU architectures. A parallel particle filter based vehicle localization and map matching algorithm which fuses GPS, odometer and digital maps, and a parallel template-matching based traffic sign recognition algorithm which employes a Kinect sensor and digital map fusion are proposed. Implementations were performed on multicore CPUs using OpenMP programming model and on manycore GPUs using CUDA programming model. Real data were collected via a vehicle equipped with sensors for various road and weather conditions and performance tests were conducted on a parallel system having two six-core CPUs and two 512-cores GPUs. The execution times and speedup of parallel processing is examined. The effect of number of particles on the success rate of the localization algorithm is also observed. Test results show that up to 75 times speedups for particle filter based localization and map matching algorithm and up to 35 times speedups for the traffic sign recognition algorithm can be achieved on GPUs compared to implementations on sequential systems, and evidently the algorithms can be used with real-time performance in the vehicle environment. It is concluded that the emerging general purpose multicore/manycore processors can constitute a unified vehicle computing platform where ADAS applications can be implemented in parallel and run with real-time performances by replacing specialized hardware and/or software platforms used for each application.

# ÖZET

## ADAS (İLERİ SÜRÜŞ DESTEK SİSTEMİ) ALGORİTMALARININ ÇOK ÇEKİRDEKLİ MİMARİLER ÜZERİNE VERİMLİ UYARLANMASI

Bu tezde ileri sürücü destek sistemleri (ADAS) ve otonom araç algoritmalarının yeni gelişen çok çekirdekli işlemci (CPU) ve grafik işleme ünitesi (GPU) mimarileri üzerine verimli olarak uyarlanması ve gerçek zamanlı performans gereksinimlerinin karşılanması hedeflenmektedir. GPS, odometre ve sayısal haritaların tümleştirildiği bir paralel parçacık filtresi tabanlı konumlandırma ve harita eşleme algoritması ile Kinect kamera ve sayısal harita tümleştirmesinin kullanıldığı bir paralel şablon eşleme tabanlı trafik işareti tanıma algoritması önerilmektedir. Algoritmalar, OpenMP programlama modeli kullanılarak çok çekirdekli işlemciler üzerinde ve CUDA programlama modeli kullanılarak grafik işleme üniteleri üzerinde gerçeklenmiştir. Sensörler ile donatılmış bir test aracı ile değişik yol ve hava şartlarında gerçek veri toplanmış ve algoritmaların performans testleri her biri altı çekirdekli iki işlemcisi ve her biri 512 çekirdekli iki grafik işleme ünitesi bulunan bir sistem üzerinde gerçekleştirilmiştir. Çalışma zamanları ve paralel işleme hızlanmaları incelenmiştir. Parçacık sayılarının konumlandırma algoritmasının başarım oranı üzerindeki etkisi gözlemlenmiştir. Test sonuçları, grafik işleme üniteleri üzerinde sıralı sistemlerdeki gerçeklenmelerine oranla, parçacık filtresi tabanlı konumlandırma ve harita eşleme algoritması için 75 kata varan, trafik işareti tanıma algoritması için ise 35 kata varan hızlanmalar elde edilebildiğini ve algoritmaların araç ortamında gerçek zamanlı olarak kullanılabileceğini göstermektedir. Genel amaçlı çok çekirdekli işlemci ve grafik işleme ünitesi mimarilerinin, her bir uygulama için kullanılan özel donanım ve yazılım platformlarının yerine, ileri sürücü destek sistemi algoritmalarının paralel olarak gerçeklenebileceği ve gerçek zamanlı olarak çalıştırılabileceği birleşik bir araç işlemci platformu oluşturabileceği sonucuna varılmaktadır.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF SYMBOLS

| | |
|---|---|
| $h$ | Height of template in pixels |
| $L$ | Link or road segment in map database |
| $Lat$ | Latitude |
| $Lon$ | Longitude |
| $m$ | Number of different sizes for each template to be used for matching |
| $N_{eff}$ | Effective sample size |
| $n$ | Number of templates in the template database |
| $r$ | Number of ROIs detected in the frame |
| $s$ | Number of different starting positions for matching in each ROI |
| $u_t$ | Measured inputs at a given time $t$ |
| $v_t$ | Unmeasured forces or faults at a given time $t$ |
| $w$ | Width of template in pixels |
| $w_t^i$ | Weight of particle $i$ at a given time $t$ |
| $x_t$ | System state at a given time $t$ |
| $\hat{x}_t$ | System state estimation at a given time $t$ |
| $x_t^i$ | State of particle $i$ at a given time $t$ |
| $y_t$ | Measurement at a given time $t$ |
| $\Theta$ | Orientation |

# LIST OF ACRONYMS/ABBREVIATIONS

| | |
|---|---|
| 3D | Three Dimensional |
| ABS | Anti-lock Braking Systems |
| ACC | Adaptive Cruise Control |
| ADAS | Advanced Driver Assistance Systems |
| ANN | Artificial Neural Network |
| ARB | OpenMP Architecture Review Board |
| CAN | Controller Area Network |
| CMOS | Complementary Metal Oxide Semiconductor |
| CPU | Central Processing Unit |
| CUDA | Compute Unified Device Architecture |
| DARPA | Defense Advanced Research Projects Agency |
| DASWS | Driver Assistance and Safety Warning System |
| DGPS | Differential Global Positioning System |
| DR | Dead Reckoning |
| DRAM | Dynamic Random Access Memory |
| DSP | Digital Signal Processor |
| ECC | Error Correcting Code |
| EGNOS | European Geostationary Navigation Overlay Service |
| EKF | Extended Kalman Filter |
| ESS | Effective Sample Size |
| FCW | Forward Collision Warning |
| FPGA | Field Programmable Gate Array |
| FRC | Functional Road Class |
| GCDC | Grand Cooperative Driving Challenge |
| GDDR | Graphics Double Data Rate |
| GFLOP | Giga-Floating Point Operations |
| GNSS | Global Navigation Satellite System |
| GPGPU | General Purpose GPU Processing |

| | |
|---|---|
| GPS | Global Positioning System |
| GPU | Graphics Processing Unit |
| HCI | Human-Computer Interaction |
| HT | Hough Transform |
| IMU | Inertial Measurement Unit |
| INS | Inertial Navigation Sensor |
| IR | Infrared |
| ISA | Instruction Set Architecture |
| ITS | Intelligent Transportation Systems |
| ITS | Intelligent Vehicle |
| KF | Kalman Filter |
| LDW | Lane Departure Warning |
| LADAR | Laser Detection and Ranging |
| LIDAR | Light Detection and Ranging |
| MEMS | Microelectromechanical Systems |
| MLP | Multi Layer Perceptron |
| MRPT | Mobile Robot Programming Toolkit |
| OCR | Optical Character Recognition |
| ODR | Optical Digit Recognition |
| OpenCL | Open Computer Language |
| OpenCV | Open Computer Vision |
| OpenMP | Open MultiProcessing |
| PCI | Peripheral Component Interconnect |
| PF | Particle Filter |
| POI | Point of Interest |
| ROI | Region of Interest |
| SAD | Sum of Differences |
| SBAS | Satellite Based Augmentation System |
| SFU | Special Function Unit |
| SIFT | Scale Invariant Feature Matching |
| SIMD | Single Instruction Multiple Data |

| | |
|---|---|
| SIMT | Single Instruction Multiple Thread |
| SIR | Sampling Importance Resampling |
| SM | Streaming Multiprocessor |
| SMC | Sequeantial Monte Carlo |
| SMP | Shared Memory Parallel Computer |
| SMT | Simulataneous Multithreading |
| SP | Streaming Processor |
| SURF | Speeded Up Robust Features |
| SVM | Support Vector Machine |
| TSR | Traffic Sign Recognition |
| UKF | Unscented Kalman Filter |
| VANET | Vehicular AdHoc Network |
| WAAS | Wide Area Augmentation System |

# 1. INTRODUCTION

Today's road vehicles already offer the driver a significant amount of different assistance systems to increase comfort and enhance road safety. Yet the era of advanced driver assistance systems (ADAS) and highly automated driving just seems to be beginning, as available computer power and sensor technology is being further enhanced and the ensuing possibilities are explored for an ever increasing number of new safety and assistance systems and functionalities.

Advanced driver assistance systems (ADAS) profoundly increase safety, improve vehicle operating efficiency, reduce fuel consumption, and enhance driving comfort within the connected automobile. Some of the ADAS applications are parking aids, lane departure warning, lane keeping assistance, adaptive cruise control, vision enhancement (night vision), traffic sign recognition, collision warning systems, collision avoidance, pedestrian detection, emergency brakes, driver impairment monitoring, and automated vehicle control.

With the rise of multicore and many-core processors, the way of computing has been evolving into a new era. The high computational power, energy efficiency and programmability of these emerging general purpose multicore processors make them a good candidate for a unified vehicle computing platform to host advanced driving assistance systems and autonomous vehicle applications by replacing specialized hardware and/or software platforms for each application. On the other hand, meeting the real-time performance requirements of those applications on such a platform is a challenge. Parallelization and using parallel programming techniques is one of the key methods to speed up applications on multicore and manycore architectures.

This thesis addresses the challenge of meeting the real-time performance requirements of ADAS and autonomous vehicle applications by efficiently mapping them on multicore and/or many-core architectures. Specifically, we present parallel implementation and performance analysis of a complete system for sign recognition with map

fusion including localization and map matching, both on a multicore processor using Open Multi-Processing (OpenMP) and on a graphics processing unit (GPU) using Compute Unified Device Architecture (CUDA).

We have surveyed the current generation of advanced driver assistance systems. We have analyzed the algorithms used in the domain, especially sensory data acquisition (GPS, odometer, video camera, LIDAR, radar, etc.), sensor fusion (Bayesian filters e.g. Particle filter), and inference (situation analysis) algorithms. We have examined multiple ways in which such systems gather and analyze data. State estimation and sensor fusion algorithms play an important role in ADAS applications. Vehicle localization through fusion of GPS and dead reckoning sensors like odometer is one of the fundamental tasks in intelligent vehicles. It is also one of the representative tasks for vehicle internal state sensing. First, we have focused on computationally intensive state estimation algorithms, namely particle filters, which is known to be a successful state estimation tool for nonlinear systems, but its computational complexity has often been a prohibitory factor for it to be employed in real-time applications. We proposed a parallel particle filter based vehicle localization and map matching algorithm which fuses GPS, odometer and digital maps. We performed implementations of the algorithm both on multicore CPUs using OpenMP and manycore GPUs using CUDA programming model. We conducted performance tests with real data collected via a vehicle, equipped with a GPS and an odometer, on a parallel system having two six-core CPUs and two 512-cores GPUs. We examined the execution times and parallel speedups. We also examined the effect of number of particles on the success rate of the algorithm to determine the optimum number of particles to be used on such a platform.

In the second part of the research, we focused on a traffic sign recognition application which is also one of the fundamental camera based ADAS applications and constitute a good representative of vehicle external environment sensing. We proposed a parallel traffic sign recognition algorithm which utilizes Kinect sensor for image acquisition and sign detection. We used a template matching based algorithm which is a simple but also a computationally complex algorithm for the recognition phase. We also employed digital map and vehicle location fusion which improves the success

rate of the recognition. We performed implementations of the algorithm both on multicore CPUs using OpenMP and manycore GPUs using CUDA programming model. We conducted performance tests with real data collected via a vehicle, equipped with a Kinect sensor together with a GPS and an odometer, on a parallel system having two six-core CPUs and two 512-cores GPUs. Test data consist of various weather and lighting conditions. We examined the execution times and parallel speedups. We also performed and tested a multi GPU implementation using CUDA.

The proposed system is unique, with many features, since it is not limited to speed signs, uses topological features of digital maps, and shows good performance in various lighting conditions. The system utilizes a Kinect sensor, which simplifies sign detection radically and lowers overall system cost.

The target architecture is a combination of a multicore CPU and a many-core graphics processing unit (GPU), which is very likely to take place in a production vehicle environment as a unified computing platform in the near future. Both modules run on the same platform. The system overview can be seen in Figure 1.1.

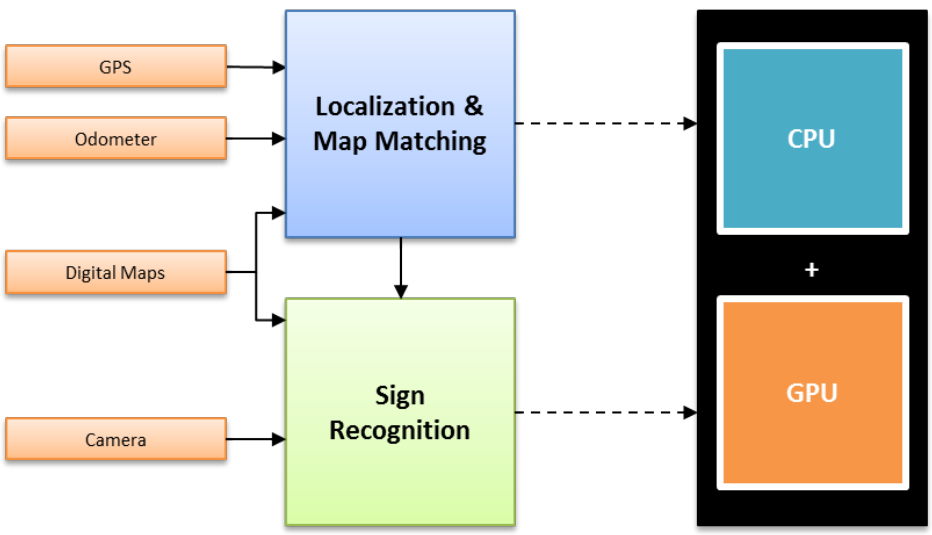



Figure 1.1. System Overview.

Both particle filter-based localization and map matching and template matching based sign recognition are computationally intensive applications where high success

rates and real-time performance cannot both be achieved simultaneously using sequential implementations. The proposed system achieves both targets by employing parallelization on a hybrid multicore/many-core architecture.

We consider a multi-hypothesis localization and map matching algorithm where map topology information is used in terms of route-ability as the likelihood calculation in the particle filter to increase map matching performance, at the same time further increasing the computational cost of the algorithm.

We first characterized the execution profile of the particle filter algorithm for different number of particles using a sequential implementation. Critical function blocks in terms of execution time were identified. We also investigated the effect of the number of particles employed by the algorithm on the error rate of localization and map matching. We observed that increasing the number of particles up to a certain level drastically decreases the error rate. This also directly contributes to the execution time of the algorithm. We then mapped the algorithm to the multicore CPU and the GPU platforms to accelerate bottlenecks and to see if the required speedups are realizable.

We conducted performance tests on a six-core CPU and a 512-core GPU platform by using real GPS and odometer data captured in vehicle environment comprising various speed and road conditions. Test results show that nearly 150 times speedups for the critical parts and and real-time performance is possible as the parallel structure handles the high computational cost of using map topology information and employing high number of particles in the particle filter.

For our template-matching based sign recognition algorithm, which can be applied to a wide range of traffic signs, we employed a similar approach; we first observed detection rates using the Kinect sensor and recognition rates with the map fusion, and we also characterized the execution profile using a sequential implementation of the algorithm. We then tested the parallel implementations on our test system having two six-core CPUs and two 512-core GPUs with real video and positioning data captured in the vehicle environment under various road and lighting conditions. We achieved

recognition performances over 80% for all types of road and lighting conditions including night time. We have presented experimental results for different number of threads for multicore CPU implementation. We have tested CPU+GPU implementations on single GPU and two GPU environment. We reached speedups up to 35.2 times which corresponds to 13 frames per second on video sequences with VGA (640x480) resolution. This shows that our template matching based recognition approach with map fusion using Kinect sensor which is also simple, effective but compute intensive technique can be used with real time performance in a vehicle environment.

We propose a generic template-based approach which can be applied to a wide range of traffic signs and the parallel implementation on a multicore CPU and GPU platform. Our approach uses a new sensor (Kinect) which provides both color and infrared images of the traffic scene, which enhances the detection stage, and we also use digital map information to augment template matching in the classification stage in order to increase the robustness of the recognition and to contribute to real-time performance.

### 1.1. Contributions

The thesis achieves the following contributions:

- Proposed parallel algorithms for vehicle localization and map matching and traffic sign recognition which are among the fundamental tasks in ADAS and representatives of internal state sensing and external environment sensing, respectively.
- Proposed parallel implementations of a particle filter based localization and map matching algorithm which fuses GPS, odometer and digital map information on multicore CPUs using OpenMP programming model, and on manycore GPU architectures using CUDA programming model.
- Proposed parallel implementations of a traffic sign recognition algorithm which utilizes Kinect sensor and digital map and localization fusion on multicore CPUs using OpenMP programming model, and on manycore GPU architectures using CUDA programming model.

- Tests were performed on real data captured in the vehicle environment comprising various road and lighting conditions.
- Presented test results on commercial testbeds, namely on six-core Intel® Xeon® 5660 CPUs and 512-core NVIDIA® GeForce GTX580 GPUs.
- For localization and map matching, achieved up to 75 times speedups for the overall parallel algorithm on GPU platform over sequential implementation.
- For sign recognition, achieved up to 35 times speedups on GPU platform over sequential implementation. The system performs very well even in poor lighting and night conditions.
- Derived guidelines to map ADAS algorithms onto multicore/manycore architectures.

## 1.2. Outline of the thesis

The outline of the remaining of the thesis is as follows: Chapter 2 makes an introduction to ADAS and autonomous vehicles domain. Key components of ADAS, sensing, sensor fusion and state estimation algorithms, and some sample applications are summarized. Chapter 3 gives background information about multicore and manycore computing, specifically GPU computing, CUDA programming model, OpenMP programming model and parallel architectures that were used in this work. Chapter 4 makes an introduction to particle filters, summarizes related work about parallelization efforts, then describes the particle filter based localization and map matching algorithm and details of its parallel implementations with CUDA and OpenMP. Chapter 5 makes an introduction to traffic sign recognition, summarizes related work about parallelization efforts, gives information about Kinect sensor, describes proposed traffic sign recognition algorithm and details of its parallel implementations with CUDA and OpenMP. Chapter 6 presents experimental results of both localization and map matching and sign recognition implementations on multicore CPU and GPU architectures after introducing the test platform. Chapter 7 includes the conclusions.

# 2. ADAS AND AUTONOMOUS VEHICLES

Every year in Europe alone, more than 40,000 casualties and 1.4 million injuries are caused by vehicle-related accidents [1]. Although advances in passive safety have made passenger cars ever safer, the safety potential of further improvements in passive safety features is limited. However, active safety systems offer possibilities for improving traffic safety by assisting the driver in his driving task. In addition, advanced driver assistance systems (ADASs) have the potential to significantly reduce the number of road accidents. An ADAS is a vehicle control system that uses environment sensors (e.g. radar, laser, vision) to improve driving comfort and traffic safety by assisting the driver in recognising and reacting to potentially dangerous traffic situations.

The following list includes current generation of advanced driver assistance systems:

- Lane Departure Warning
- Lane Keeping Assistance
- Adaptive Cruise Control
- Parking Aids
- Vision Enhancement (Night Vision)
- Traffic Sign Recognition
- Collision Avoidance
- Collision Warning Systems
- Pedestrian Detection
- Blind Spot Detection
- Emergency Brakes
- Driver Impairment Monitoring

Since an ADAS can even autonomously intervene, an ADAS-equipped vehicle is popularly referred to as an “intelligent vehicle”. As explained in more detail in several surveys [2–5], the following types of intelligent vehicle systems can be distinguished:

- *Driver information* systems increase the driver's situation awareness, e.g. advanced route navigation systems.
- *Driver warning* systems actively warn the driver of a potential danger, e.g. lane departure warning, blind spot warning, and forward collision warning (FCW) systems. This warning then allows the driver to take appropriate corrective actions in order to mitigate or completely avoid the event.
- *Intervening* systems provide active support to the driver, e.g. an adaptive cruise control (ACC) system. ACC is a comfort system that maintains a set cruise control velocity, unless an environment sensor detects a slower vehicle ahead. The ACC then controls the vehicle to follow the slower vehicle at a safe distance.
- *Integrated passive and active safety systems.* In addition to passive safety systems that are activated during the crash, a pre-crash system can mitigate the crash severity by deploying active and passive safety measures before a collision occurs. Pre-crash safety measures, such as brake assist and seat belt pre-tensioners, have recently been introduced on the market.
- *Fully automated* systems are the next step beyond driver assistance, and operate without a human driver in the control loop. Automated highway systems, using fully automated passenger cars, are expected to significantly benefit traffic safety and throughput.

According to several surveys ADASs can prevent up to 40% of traffic accidents, depending on the type of ADAS and the type of accident scenario.

ADAS applications extensively make use of sensors for understanding the vehicle's internal state, the environment or outside world and also the driver's status. Some examples of these sensors are camera, GPS, inertial measurement unit, odometer, radar, LIDAR, digital maps and oem vehicle sensors through vehicle's CAN Bus. The ADAS applications usually require fusion of information collected by sensors for a better understanding the situation. This process is called "sensor fusion". So, the first building block or module can be named as "sensing and sensor fusion", sometimes called "perception". After sensing is completed, the information is processed at higher

levels in the next stage, and converted to some actions such as alerting the driver for potential dangers through a human computer interaction or control some mechanical components of the vehicle through actuators, for example, helping the driver for parking or staying within highway lane markings [6, 7]. The basic building blocks of an ADAS application are illustrated in Figure 2.1.

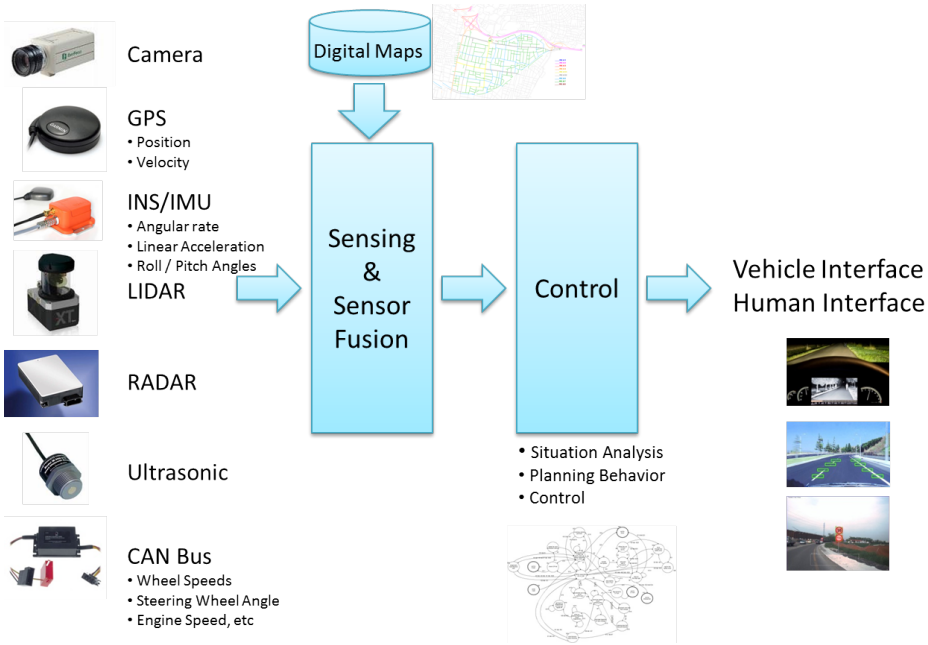



Figure 2.1. ADAS Architecture.

The ultimate goal for ADAS applications is fully autonomous driving. Autonomous vehicle applications share the similar algorithms with ADAS applications. The most extensive use of sensors and information fusion are seen in autonomous vehicle applications. DARPA has organized the Grand Challenges and the Urban Challenge from 2004 to 2007, which remarkably promoted the technologies of intelligent vehicles around the world [8, 9]. Some examples of the vehicles developed for these challenges are DARPA Grand Challenge winner "Stanley" in 2005 [10], DARPA Urban Challenge winner "Boss" in 2007 [11]. Another example from Europe is the Grand Cooperative Driving Challenge winner "AnnieWAY" in 2011 [12]. The vehicles can be seen in in Figures 2.2a, 2.2b and 2.2c, respectively. Some other examples of such vehicles can be examined in [13–16]. These can be considered as the most advanced intelligent vehicle projects.

Recently, autonomous cars have started to be driven not only on test tracks but also on the public roads and became legal in U.S. An example is Google's driverless car project which has already logged for more than 190.000 miles (about 300.000 kilometers) driving in city traffic, busy highways, and mountanious roads with only occasional human intervention [17]. Nevada and California recently became the first U.S. states to make self-driving cars legal starting with this car. The vehicle can be seen in Figure 2.2d. The car uses a laser range finder mounted on the roof of the car. The device generates a detailed 3D map of the environment. The car then combines the laser measurements with high-resolution maps of the world, producing different types of data models that allow it to drive itself while avoiding obstacles and respecting traffic laws. The vehicle also carries other sensors, which include: four radars, mounted on the front and rear bumpers, that allow the car to "see" far enough to be able to deal with fast traffic on freeways; a camera, positioned near the rear-view mirror, that detects traffic lights; and a GPS, inertial measurement unit, and wheel encoder, that determine the vehicle's location and keep track of its movements.

A complete survey of components, historical development, research and experiments on autonomous vehicles can be found in [18–20]. The following sections make a brief presentation of sensing, estimation, sensor fusion and examples of autonomy and ADAS tasks mainly based on the classification and descriptions in [18].

## 2.1. Sensing

Sensors are applied in all levels of vehicle control and autonomy, ranging from engine control, ABS braking and stability enhancement systems, passive driver assistance systems such as navigation, infotainment, backup hazard warning, and lane change assistance, active safety systems such as lane maintenance and crash avoidance, and, of course, full vehicle automation.

In broad terms, sensors can be grouped according to the function they provide. Internal vehicle state sensors provide information about the current operation and state of the vehicle, including lower-level functions such as engine operations and higher-


(a) Stanley

(b) Boss

(c) AnnieWAY

(d) Google's Autonomous Vehicle

Figure 2.2. Autonomous Vehicles.

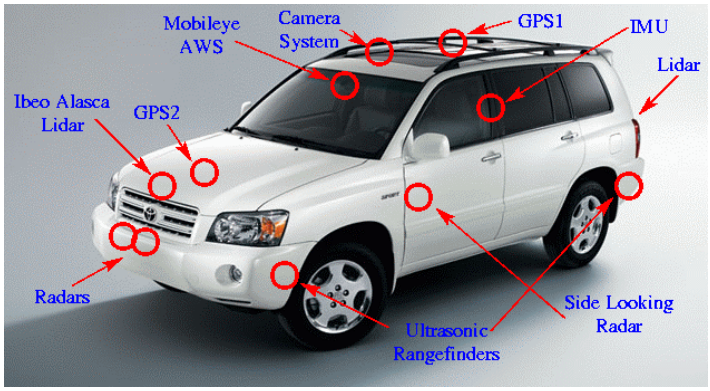
level states such as vehicle motion and position. External environment sensors provide information about the world outside the vehicle, potentially including road and lane information, the location and motion of other vehicles, and stationary physical objects in the world. Finally, driver state and intention sensors provide information about the state or intentions of the driver. These sensors can include seat occupancy and passenger weight (pressure or infrared sensors), audio sensors, internal cameras, eye trackers, breath alcohol sensors, and haptic transducers.

In this section individual sensors and technologies that are generally applied for vehicle control and automation are shortly reviewed. Since it is often both advantageous and necessary to combine the information from multiple sensors to provide a full and error-free understanding of the current state of the vehicle and the world, a description of estimation and sensor fusion approaches are also included.

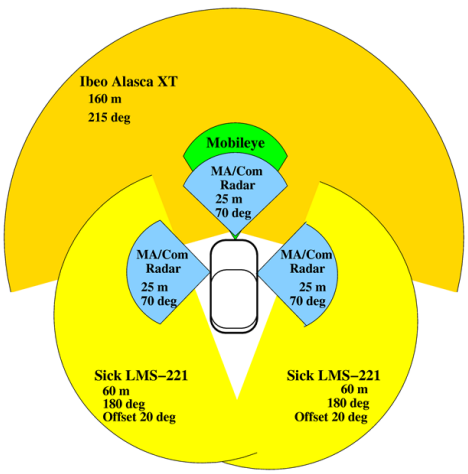
Conceptually, any device or technology that provides information can be considered, and treated, as a sensor. In vehicle automation applications, common examples of this include map databases and wireless vehicle-to-vehicle and vehicle-to-infrastructure communications and other cooperative infrastructure technologies including visual signs, tags, or markers and radar reflective surfaces.

Figure 2.3 shows an example of sensing and sensor fusion in autonomous vehicles employed in Ohio State University ACT (Autonomous City Transport) vehicle developed for DARPA Urban Challenge in 2007 (OSU ACT) [21].

The sensors generally produce enourmous amounts of data to be processed in real-time. For example, the type of LIDAR used in Google's driverless car produces one million point information per second. Sensors are not only significant for autonomous vehicle applications but they are also an important part of today's standard cars. As an example, the current model of Ford Focus has more than 145 actuators, 4,716 signals, and 74 sensors — radar, sonar, cameras, accelerometers, temperature and even rain sensors — that produce more than 25 gigabytes of data per hour. That data is analyzed by more than 70 on-board computers [22].




(a) OSU ACT




(b) Sensors

Figure 2.3. Sensing in Autonomous Vehicles [21].

### 2.1.1. Vehicle Internal State Sensing

Autonomous vehicles use all the standard sensors available in a car for self-sensing.

2.1.1.1. OEM Vehicle Sensors. Modern vehicles have sophisticated electronic control systems that require a number of sensors and measurements. The outputs of these sensors may appear on the vehicle's internal communication bus (CANBUS), or they be electronically tapped for use. Commonly available measurements include:

- Wheel speed, usually measured by a Hall effect sensor, which produce a digital signal whose frequency is proportional to speed;
- Vehicle dynamic state, possibly including yaw rate and lateral and longitudinal acceleration;
- Driver inputs, for example, steering wheel position, throttle and brake pedal positions, turn signals, headlights, windshield wipers, and so forth;
- Transmission gear and differential state;
- Brake pressure, either at the master cylinder or for each wheel;
- Engine and exhaust variables, for example, coolant temperature, RPM, and spark plug firing timing.

2.1.1.2. Global Positioning Systems (GPS). A Global Positioning System (GPS) is a key component of a present day intelligent vehicle. GPS can be used to find absolute position and velocity. This is very useful for an autonomous vehicle that has access to a precise map, as it can understand where it is with respect to its destination and with respect to the road network or, for an off-road vehicle, topographic features and obstacles. This information is needed, for example, to compute optimal routes or driving directions. On- or off-road, when combined with a vehicle-to-vehicle wireless communication system, it can also provide relative position and relative speed information.

The information available from a GPS receiver is:

- Absolute position in a geodetic coordinate system, for example, latitude-longitude-altitude;
- Velocity and course over ground information (horizontal speed and orientation relative to true north);
- Precise time and synchronized pulse per second;
- Raw information that can be used for precise postprocessing applications.

Generally speaking, in areas with an unobstructed view of sky, a standard inexpensive or embedded GPS receiver can achieve position accuracies on the order of 5–15 meters. This is sufficient for providing navigation and routing instructions for a human driver, but is insufficient for resolving which lane a vehicle is currently occupying.

The GPS constellation consists of approximately 24 satellites arranged in 6 orbital planes at an altitude of 12,500 miles (20,200 km). GPS is the most established and commonly used Global Navigation Satellite System (GNSS) for vehicle navigation and localization. It is operated by the United States Department of Defense. There are three other GNSSs over the world: Russian GLONASS, European Galileo, and Compass from China. Galileo is under construction and will be directly compaticle with GPS, whereas the GLONASS requires a somewhat different receiver structure. The future GNSS receivers are expected to be able to use more than one system.

A GPS receiver determines its position by knowing the distance from itself to at least four satellites of known position. For this purpose each satellite broadcasts its orbit and a navigation signal. The time it takes this signal to reach the receiver is used to calculate this distance, known as a “pseudorange”. A pesudorange mesurement is affected by noise from a number of sources, for example receiver clock bias, atmospheric transmission errors, error in the broadcasted satellite orbit (ephemeris), and multipath (signal reflections off of nearby objects). These noise sources will induce errors in the calculated receiver positions.

The most persistent errors are multipath and reduced satellite visibility. Others, such as atmospheric and ephemeris errors that can contribute tens of meters errors to

the estimated position, can be compensated by differential means including Differential GPS (DGPS) or satellite-based augmentation systems (SBASs) that, through geostationary satellites, regionally provide correction information free of charge for the GPS and the GLONASS. In North America, there is the Wide Area Augmentation System (WAAS) provided by the U.S. Federal Aviation Administration, in Europe, there is the European Geostationary Navigation Overlay Service (EGNOS).

An appropriately capable GPS receiver, using one of these basic differential correction data services broadcast over a large area of the planet, can achieve position accuracies on the order of 1–2 meters. This is sufficient for some safety applications, and can sometimes resolve lane identity, but is insufficient for autonomous vehicle operations.

Methods known as "dead reckoning" (DR) and "map matching" are commonly used to compensate for satellite visibility and multipath. Dead reckoning uses measurements of the vehicle's motion from on-board sensors, such as accelerometers and gyroscopes, to extrapolate from the last-known vehicle position, heading and/or speed. The major challenge when using dead reckoning is the problem of accumulated error; small sensor measurement errors accumulate over time into large position errors. This especially problemetic when GPS position estimates are unavilable for a long period of time, as there is no capability to correct the accumulated error.

More information about fundamentals of GPS can be found in [18, 23–27].

2.1.1.3. Inertial Measurement Unit (IMU). Positioning technologies based on standalone GPS receivers are vulnarable and have to be supported by additional information sources to obtain the desired accuracy, integrity, and availability and continuity of service. It is very common to augment a GPS receiver with an inertial measument unit (IMU) to maintain continuity of the localization estimates when GPS cannot provide a position, such as when the vehicle is in a tunnel or surrounded by tall buildings. Often, the accelerometers and rate gyroscopes are integrated into a single 6 degree of freedom

sensor known as an inertial measurement unit (IMU). An IMU provides 3-D positon, velocity, and attitude information which can be measured internally without reference to an external point. In combination with decreasing cost, power consumption and the size of the MEMS inertial sensors, incorporating MEMS (microelectromechanical systems) IMUs to positioning systems become attractive.

2.1.1.4. Magnetic Compass (Magnetometer). One approach to measuring the absolute (yaw) angle of a vehicle is to use an electronic compass that measures the magnetic field of the Earth. Normally this would be accomplished using two or three magnetic sensors arranged along orthogonal axes. These sensors are quite sensitive to electromagnetic interference, which can arise from computers, radio transceivers, power electronics, and the internal combustion engine.

### 2.1.2. External Environment Sensing

A number of different sensors have been developed for sensing the external environment of an autonomous vehicle. Many have been developed initially for safety warning or safety augmentation systems that are now being deployed on some high-end vehicles. These include radar sensors, scanning laser range finders, known as light detection and ranging (LIDAR) or sometimes laser detection and ranging (LADAR), single camera or stereo camera image processing systems, and ultrasonic rangefinders.

2.1.2.1. Radar. Radar is a popular active sensing technology for road vehicles used in sensing both near and far obstacles. A radar system tends to be designed based on the desired safety or control function. For applications like imminent crash detection and mitigation, lane change, and backup safety systems, a shorter range but wider field of view is desired, perhaps on the order of 30 meters of range with a 65–70° wide field of view. For applications like advanced cruise control and crash avoidance, a longer range but narrower field of view is required, perhaps on the order of a 120-meter range and a 10–15° field of view. Radars are popular choices because they are robust mechanically and operate effectively under a wide range of environmental conditions.

They are generally unaffected by ambient lighting or the presence of rain, snow, fog, or dust. They generally provide range and azimuth measurements as well as range rates.

2.1.2.2. LIDAR. A scanning laser range finder system, or LIDAR, is a popular system for obstacle detection. A pulsed beam of light, usually from an infrared laser diode, is reflected from a rotating mirror. Any nonabsorbing object or surface will reflect part of that light back to the LIDAR, which can then measure the time of flight to produce range distance measurements at multiple azimuth angles. LIDAR sensors can produce very precise measurements. Higher-end sensors can produce multiple distance measurements per laser pulse, which can be helpful when trying to see through dust, rain, mostly transparent surface (such as glass windows), and porous objects such as wire fences. Some recent sensors can scan in multiple planes in order to provide a semi-three-dimensional view of the external world. While they tend to be heavily used in research applications, they have not seen wide use in automotive OEM safety applications.

2.1.2.3. Image Processing Sensors. Vision and image processing sensors have been heavily used in automated vehicle research, and have been deployed in some safety applications. In general terms, single camera systems are often used for lane marker or lane edge detection and for basic, low-accuracy object detection and localization, and are currently applied in lane departure warning systems and a few forward collision warning systems. Researchers have also developed road sign reading applications. Multiple camera systems, for example, stereo vision systems, can provide a depth map for objects in the world and can be used for obstacle detection. Although, to date, they have been applied mostly in research applications, a few OEM automotive products are beginning to appear.

Their primary advantage of vision based sensors is their use of low-cost, off-the-shelf components and that their implementation is almost entirely in software. Their primary disadvantage is that they are almost always implemented as passive sensors, and thus must cope with the full range of ambient and uncontrolled conditions, includ-

ing lighting, shadowing and reflection, and atmospheric weather, dust, and smoke.

## 2.2. Estimation

Even under the best of circumstances sensors provide noisy output measurements. In autonomous vehicles, measurements from sensors of varying reliability are used to ascertain the location and velocity of the car both with respect to the roadway or locations on a map and with respect to obstacles or other vehicles. In order to use sensor measurements for control purposes the noise components may need to be eliminated, measurements may need to be transformed to match variables, and measurements from multiple sensors may need to be fused. Fusion of data may be required because a single sensor or sensor reading is insufficient to provide the needed information, for example the shape of an obstacle or open pathway, or it may be needed to provide redundancy and reduce uncertainty.

The most widely used state estimation tools in ADAS applications are Bayesian filters [28]. Bayesian filtering is the general term used to discuss the method of using a predict/update cycle to estimate the state of a dynamical system from noisy sensor measurements. Two types of Bayesian filters are Kalman filters and particle filters.

The basic operation of the filters involves predict/update cycles as shown in Figure 2.4. The time update projects the current state estimate ahead in time. The measurement update adjusts the projected estimate by an actual measurement at that time. The following sections briefly describe two types of Bayesian filters.

### 2.2.1. Kalman Filter

The Kalman filter addresses the general problem of trying to estimate the state $x$ of a system given a known model [29–31]. It is an optimal linear filter for systems with Gaussian noise. We consider the linear, discrete time, time-varying system modeled as

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{2.1}$$

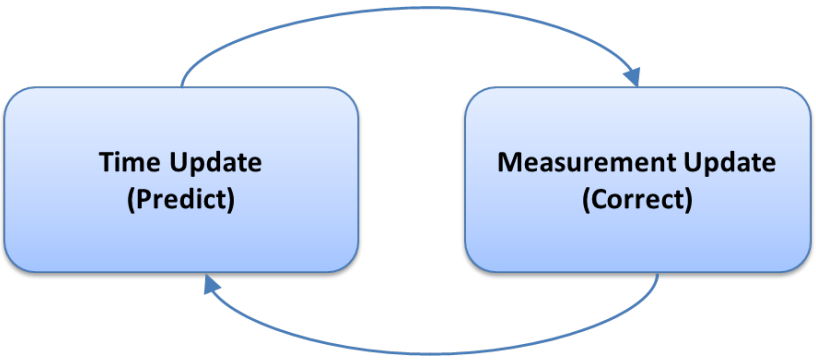



Figure 2.4. Schematic illustration of Predict/Update cycle of a Bayesian filter.

with a measurement $z$ that is

$$z_k = Hx_k + v_k \tag{2.2}$$

The random variables $w_k$ and $v_k$ represent the process and measurement noise, respectively. They are assumed to be independent of each other, white, and with normal probability distributions

$$P(w)\ N(0, Q) \tag{2.3}$$

$$P(v)\ N(0, R) \tag{2.4}$$

where $Q$ is the *process noise covariance* and $R$ is the *measurement noise covariance* matrices.

The matrix $A$ is the state transition matrix which relates the state at the previous time step $k-1$ to the state at the current step $k$. The matrix $B$ is the control input matrix which relates the optional control input $u$ to the the state $x$. The matrix $H$ in the measurement equation is the output matrix which relates the state to the measurement $z_k$.

When the system whose state is to be estimated is linear and the process and measurement noises are Gaussian, the basic Kalman filter can be used. Figure 2.5 shows the basic Kalman filter operation. The Kalman filter estimates the state of a system based only on the last estimate and the most recent set of measurements available, and therefore is a recursive filter. The first task during the measurement update is to compute the Kalman gain $K_k$. The next step is to actually measure the process to obtain $z_k$ and to generate a posteriori state estimate by incorporating the measurement. The final step is to obtain a posteriori error covariance estimate. After each time and measurement update pair, the process is repeated with the previous posteriori estimates used to predict the new priori estimates.

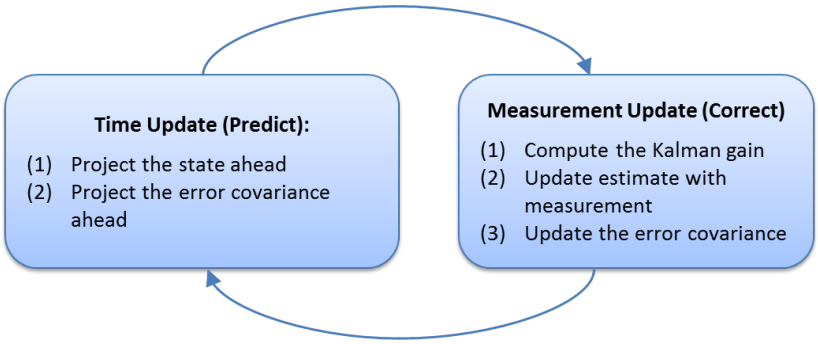



Figure 2.5. Basic Kalman Filter Operation.

### 2.2.2. Extended Kalman Filter (EKF)

If the process we wish to estimate is non-linear in nature and the noises are not Gaussian, the basic Kalman filter does not work as they are only for linear processes. A Kalman filter that linearizes about the current mean and covariance is referred to as an *extended Kalman Filter* or EKF. In this case, the process has again a state vector $x$, but the process is governed by the non-linear stochastic difference equation

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.5}$$

with a measurement $z$ that is

$$z_k = h(x_k, v_k) \tag{2.6}$$

where random variables $w_k$ and $v_k$ again represent the process and measurement noise. In this case, the *non-linear* function $f$ relates the state at the previous time step $k-1$ to the state at the current time step $k$. The *non-linear* function $h$ in the measurement equation relates the state $x_k$ to the measurement $z_k$. The basic operation of the EKF is the same as the linear discrete Kalman filter. However, it is important to note that a fundamental flaw of EKF is that the distributions of the various random variables are no longer normal after undergoing nonlinear transformations. The EKF is simply an *ad hoc* state estimator that only approximates the optimality of Bayes' rule by linearization.

### 2.2.3. Particle Filter

When the system models are non-linear and the noises are non-Gaussian, particle filtering gives better results at the price of additional computational effort. Particle Filters, also known as Sequential Monte Carlo (SMC) methods, are iterative methods that track a number of possible state estimates, so-called particles, across time and gauge their probability by comparing them to measurements [32–37].

The basic idea of particle filters is that any pdf (probablilty distribution function) can be represented as a number of samples (particles). Each particle has one set of values for the state variables. This method can represent any arbitrary distribution, making it good for non-Gaussian, multi-modal pdfs. We can represent a pdf by drawing lots of samples from it, so that the density of the samples in one area of the state space represents the probability of that region. The key idea is that we can find an approximate representation of a complex model (any arbitrary probaility distribution) rather than an exact representation of a simplified model (Gaussian).

The *sampling importance resampling* (SIR) algorithm is one of the most widely

used sequential Monte Carlo methods, which allows system state estimates to be computed on-line while the state changes as it is the case for tracking algorithms. More detailed information about particle filters are given in Section 4.1.

## 2.3. Sensor Fusion

A vehicle is usually fitted with multiple sensors and technologies, and the outputs of these individual sensors must be combined to produce a final overall view of the world. The primary justifications for this approach are:

- Different sensors and sensor technologies have different perceptive abilities for the environment.
- Different or multiple sensors have different fields of view.
- A single sensor does not cover the application's required field of view.

However, when multiple sensors or sensor technologies are deployed, there is always the chance of conflict between the sensor measurements. For example, the same object in the environment may be reported at two different positions (i.e., 30 meters versus 28 meters), or one sensor may detect an object and another may not. These conflicts may arise because of:

- Different perceptive characteristics of each sensor: for example, range, accuracy, field of view, sensitivity, or response to environmental factors;
- Actual changes over time in the environment;
- Faulty sensor element, electronics, noise, or false data input;
- Thresholds or other differences between processing algorithms.

There are various techniques to deal with these conflicts, including tracking and filtering (e.g., extended Kalman filter, particle filter), confidence and hypothesis testing approaches, voting schemes, and evidence-based decision theory [38, 39].

## 2.4. Digital Maps

Digital map databases is another key component of ADAS and autonomous vehicle applications. The information they provide can be considered, and treated, as an additonal sensor. They help the vehicle to understand the environment locally for short-term vehicle control and also for long-term planning. This generally involves a process called map-matching. Map-matching is the process of aligning one or a sequence of observed vehicle positions with the road network on a digital map [40, 41]. It is a fundamental pre-processing step for many applications.

On the other hand, a fully autonomous vehicle will require planning behaviors over a longer timescale. Ideally, a fully autonomous vehicle would allow navigation from any origin to any destination without direct human input. The long term planning requires geographic information for an area and a planning algorithm that can use these maps to generate a plan for the vehicle path and behavior, as well as mechanisms for replanning when actual current conditions do not match the contents of the maps. The use of map databases is reasonable due to the availability of GPS-based positioning.

The available map data falls into two categories: raster data and vector data. Raster-based data divides an area into a collection of cells or grids, often of uniform size, and each grid cell is assigned one or more values representing the information being mapped. Raster data consumes a large amount of memory but requires simpler data processing techniques as the information is represented at a more direct level. Examples of raster data include digital elevation maps and land cover or usage maps.

Vector data expresses information in terms of curves, connected line segments, or even discrete points. Curves or line segments can also enclose and represent areas. Vector data is a more complex and rich representation and thus generally requires less storage but more complicated processing algorithms. Examples include digital road maps where streets are represented by a road centerline.

The road network is often contained in a vector database. Roads are represented

by piecewise-linear segments modeling the location of the road centerline between two endpoints defined by intersections or dead-ends. Each entry in the database represents one such road "segment", and each road segment is uniquely identified by its index in the database.

Raster data can be processed in much the same way as grid cell sensor data for obstacle avoidance and planning. Possible algorithms include potential field methods, Voronoi decompositions with grid maps treated as graphs, and discrete gradient descent optimization approaches.

Vector data can be processed using tools from graph theory. A graph consists of nodes, also known as vertices, and edges. Edges may be directed and weighted. A road map can easily be translated into a graph by, for example, labeling intersections as nodes and streets as the edges connecting nodes. Various properties of a given graph, including connectedness, adjacency, reachability, and optimal flow and routing, can be computed from this representation.

## 2.5. Examples of Autonomy and ADAS Tasks

### 2.5.1. Vehicle Localization

A key element of intelligent vehicle technology is vehicle localization. All aspects of the system, from sensor processing and fusion to navigation and behavioral decision making to low-level lateral and longitudinal control, require accurate vehicle position, velocity, and vehicle heading, pitch, and roll information at a fairly high update rate. Providing this information requires the use of multiple sensors, possibly including one or more Global Positioning System (GPS) receivers augmented with some correction service, inertial measurement units (IMU), and dead reckoning sensors (wheel speeds, transmission gear and speeds, throttle, brake, and steering wheel position) provided on the vehicle. To account for sensor errors, noise, and the different update rates of each sensor, a filtering is applied to generate the required state measurements.

The reasons for fusing these sensors are:

- Accuracy: IMU integration can lead to the unbounded growth of position error, even with the smallest amount of error or bias in its measurements. This gives rise to the need for an augmentation of the measurements by external sources to periodically correct the errors. GPS can provide this, since it provides a bounded measurement error with accuracy estimates.
- Data availability: GPS is a line-of-sight radio navigation system, and therefore GPS measurements are subject to signal outages, interference, and jamming, whereas an IMU is a self-contained, nonjammable system that is completely independent of the surrounding environment, and hence virtually immune to external disturbances. Therefore, an IMU can continuously provide navigation information when GPS experiences short-term loss of its signals. Similarly, dead reckoning sensors like odometers are internal to the vehicle.

Vehicle Localization is an example of vehicle internal state sensing and estimation. It is one of the tasks that sensor fusion techniques are mostly involved. KF/EKF/UKF/PF based techniques are widely used to both fuse the GPS data and DR data, and iteratively estimate vehicle position [8, 42, 43, 45].

### 2.5.2. Map Matching

Map matching algorithms use inputs generated from positioning technologies (such as GPS or GPS integrated with DR) and supplement this with data from a high resolution spatial road network map to provide an enhanced positioning output. The general purpose of a map matching algorithm is to identify the correct road segment on which the vehicle is travelling and to determine the vehicle location on that segment. Map matching not only enables the physical location of the vehicle to be identified but also improves the positioning accuracy if good spatial road network data are available. This means that the determination of a vehicle location on a particular road identified by a map matching algorithm largely depends on the quality of spatial road map used in the algorithm.

Procedures for map matching vary from those using simple search techniques, to those using more advanced techniques such as the use of an Extended Kalman Filter, fuzzy logic, and belief theory [46]. Approaches for map matching algorithms in the literature can be categorised into four groups: geometric (techniques like point to point, point to curve, curve to curve matching), topological (making use of also the topological features of the roads like connectivity, contiguity of the links, road turn, road curvature, etc), probabilistic, and other advanced techniques. An extensive survey of current map matching algorithms can be found in [40, 41].

### 2.5.3. Steering Control and Lane Following

One of the key goals of an automated vehicle is the ability to perform automatic steering control. To be able to control steering, a measure of the vehicle's orientation and position with respect to the road must be available to the controller. The most commonly used technique is vision-based lane marker detection, radar-based offset signal measurement, and the magnetic nail-based local position sensing. Vision- and radar-based systems provide an offset signal at a preview distance ahead of the vehicle that contains relative orientation information. The vision system directly processes the image of the road and detects lane markers. Therefore, it does not require any modifications to current highway infrastructures. Hough Transform [47] is one of the most widely used vision based lane marker detection algorithms. For lane tracking purposes mostly KF or EKF based techniques have been used [48], recently some particle filters based techniques have also been proposed [49–51]. A survey of various techniques for video based lane estimation and tracking can be found in [52].

### 2.5.4. Adaptive Cruise Control

Adaptive cruise control (ACC) systems have been recently introduced as a technological improvement over existing cruise controllers on ground vehicles. ACC systems regulate the vehicle speed to follow the driver's set point if there is no vehicle or any obstacles in sight. When a slower vehicle is observed ahead, the ACC controlled vehicle will follow the vehicle ahead at a safe distance by adjusting its relative

speed. ACC systems are capable of maintaining a vehicle's position relative to the leading vehicle including in congested traffic and even in city traffic by using stop-and-go features while maintaining a safe distance between leading and following vehicles autonomously [18, 53].

### 2.5.5. Traffic Sign Recognition

Reliable traffic-sign detection is currently one of the most important tasks in automotive vision industry. The aim here is to inform the driver of the current speed or other types of restrictions by automatic detection and recognition of roadside signs and warn the driver in case of any violations or even react autonomously. It represents a significant challenge due to common variations in weather and lighting conditions in conjunction with the obvious on-vehicle constraints. Automatic road sign recognition can be divided into two stages: initially detection of candidate signs within the image and recognition (i.e. verification or classification) of the type of sign present. Several shape-based or color-based approaches have been used for detection such as various generalizations of classical Hough transform as in [54,55], genetic algorithms [56], template-based matching [57], an AdaBoost based framework [58], feature matching [59] and a direct Support Vector Machine based approach [60]. Recognition is usually performed by a machine learning based classification algorithm. Commonly this is an Artificial Neural Network (ANN) [54–56] or in some more recent works, Support Vector Machines (SVM) [60]. Tracking is a third stage sometimes employed for tracking candidates among successive image frames. For tracking purposes, mostly various types of Kalman filters have been proposed [61], more recent work also use particle filter based techniques [62].

# 3. MULTICORE/MANY-CORE COMPUTING

Multicore architectures can be classified in a number of ways. Most distinguishing attributes are the application class, power/performance, processing elements, memory system, and accelerators/integrated peripherals. In recent years, there has been a wide of range of multicore architectures produced for the commercial market. They have targeted every market segment from embedded to general purpose desktop and server realms.

First group of processors are general purpose multicores such as Intel Core, AMD Opteron, and Sun SPARC T4 [63, 64]. The microarchitecture of their cores is traditional and based on a powerful conventional uniprocessor. They employ a modest number of identical copies of these cores with large caches. These chips are intended for applications found in the desktop and server markets in which power is not an overriding concern.

The second group of processpors are targeted mobile and embedded applications such as Intel Atom and Arm Cortex. They too have identical general-purpose cores that are well suited to control dominated applications. Power is an overriding concern for these chips because many are intended to run from batteries.

The next set of architectures are more specialized and are targeted to high-performance, massively parallel computing such as NVIDIA GPUs, AMD GPUs, and IBM Cell. These architectures target high performance in their application domain and, for the most part, employ significant numbers of cores—for graphics processing units, this number is in the hundreds. They are also called many-core processors. Although discontinued, The IBM Cell Broadband Engine implements a heterogeneous architecture with a modest number of very specialized data processing engines [65, 66]. These designs are generally very high power ranging from 100 W to 180 W.

Another set of multicore architectures are specialized for specific application domains such as Tilera T64. They exhibit the most variety. Most of them target data dominated application domains such as wireless baseband, and audio/visual codecs where simple parallelism can often be exploited. Accordingly, they support high computation rates. Many feature interconnection networks that are tuned to the needs of their intended application domain.

A complete survey of recent multicore processors and review of their common attributes can be found in [67]. In the remaining of the chapter, manycore and multicore processor architectures and their respective programming models that are used in this work are described in summary.

## 3.1. Manycore Computing

Graphics Processing Units (GPUs) have been progressively and rapidly advancing from being specialized fixed function architectures to being highly programmable and incredible powerful parallel computing devices which are referred to as many-core or massively parallel processors. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphics APIs. This corresponds a transformation from general purpose GPU (GPGPU) processing, where graphics hardware is used to perform computations for tasks other than graphics, to the more recent trend of *GPU Computing*, where GPU architectures and programming tools have been developed that have created a parallel programming environment that is no longer based on the graphics processing pipeline, but still exploits the parallel architecture of the GPU. In fact, GPU Computing has transformed the GPGPU concept into the simple mapping of parallelizable algorithms onto SIMD format for the GPU. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory single instruction multiple data (SIMD) multicore processors. The following sections make a brief presentation of the main features of NVIDIA's CUDA architecture based on the summary in [68]. More detailed description can be found in [69–72].

### 3.1.1. GPU Computing Architecture

To address different market segments, GPU architectures scale the number of processor cores and memories to implement different products for each segment while using the same scalable architecture and software. NVIDIA's scalable GPU computing architecture varies the number of streaming multiprocessors to scale computing performance, and varies the number of DRAM memories to scale memory bandwidth and capacity.

Each multithreaded streaming multiprocessor provides sufficient threads, processor cores, and shared memory to execute one or more CUDA thread blocks. The parallel processor cores within a streaming multiprocessor execute instructions for parallel threads. Multiple streaming multiprocessors provide coarse-grained scalable data and task parallelism to execute multiple coarse grained thread blocks (possibly running different kernels) in parallel. Multithreading and parallel-pipelined processor cores within each streaming multiprocessor implement fine-grained data and thread-level parallelism to execute hundreds of fine-grained threads in parallel. Application programs using the CUDA model thus scale transparently to small and large GPUs with different numbers of streaming multiprocessors and processor cores.

### 3.1.2. Fermi Computing Architecture

To illustrate GPU computing architecture, Figure 3.1 shows the third-generation Fermi computing architecture configured with 16 streaming multiprocessors, each with 32 CUDA processor cores, for a total of 512 cores. The GigaThread work scheduler distributes CUDA thread blocks to streaming multiprocessors with available capacity, dynamically balancing the computing workload across the GPU, and running multiple kernel tasks in parallel when appropriate. The multithreaded streaming multiprocessors schedule and execute CUDA thread blocks and individual threads. Each streaming multiprocessor executes up to 1,536 concurrent threads to help cover long latency loads from DRAM memory. As each thread block completes executing its kernel program and releases its streaming multiprocessor resources, the work scheduler assigns a new

thread block to that streaming multiprocessor.

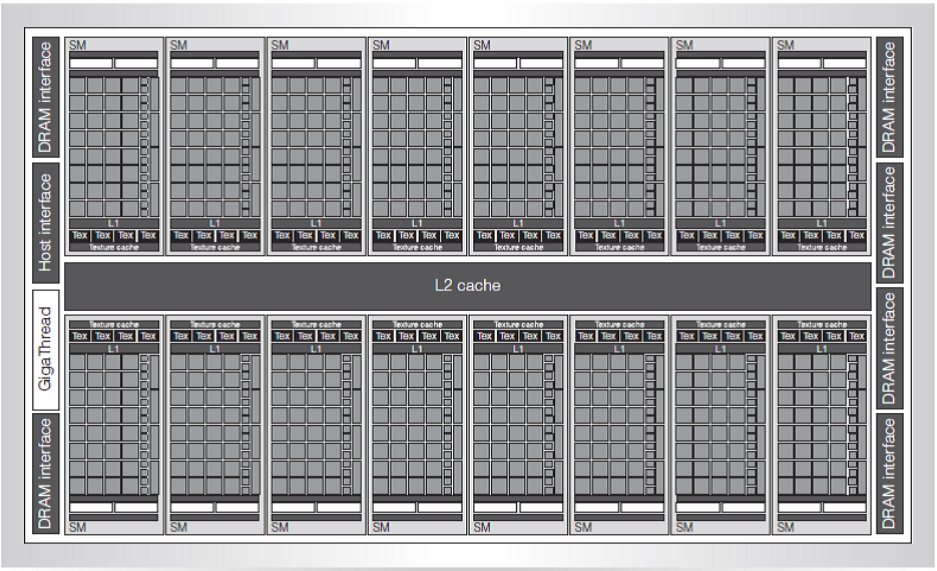



Figure 3.1. Fermi GPU Computing Architecture [68].

The PCIe host interface connects the GPU and its DRAM memory with the host CPU and system memory. The CPU+GPU coprocessing and data transfers use the bidirectional PCIe interface. The streaming multiprocessor threads access system memory via the PCIe interface, and CPU threads access GPU DRAM memory via PCIe.

The GPU architecture balances its parallel computing power with parallel DRAM memory controllers designed for high memory bandwidth. The Fermi GPU in Figure 3.1 has six high-speed GDDR5 DRAM interfaces, each 64 bits wide. Its 40-bit addresses handle up to 1 Tbyte of address space for GPU DRAM and CPU system memory for large-scale computing.

### 3.1.3. Streaming Multiprocessor

The Fermi streaming multiprocessor introduces several architectural features that deliver higher performance, improve its programmability, and broaden its applicability. As Figure 3.2 shows, the streaming multiprocessor execution units include 32 CUDA

processor cores, 16 load/store units, and four special function units (SFUs). It has a 64-Kbyte configurable shared memory/L1cache, 128-Kbyte register file, instruction cache, and two multithreaded warp schedulers and instruction dispatch units.

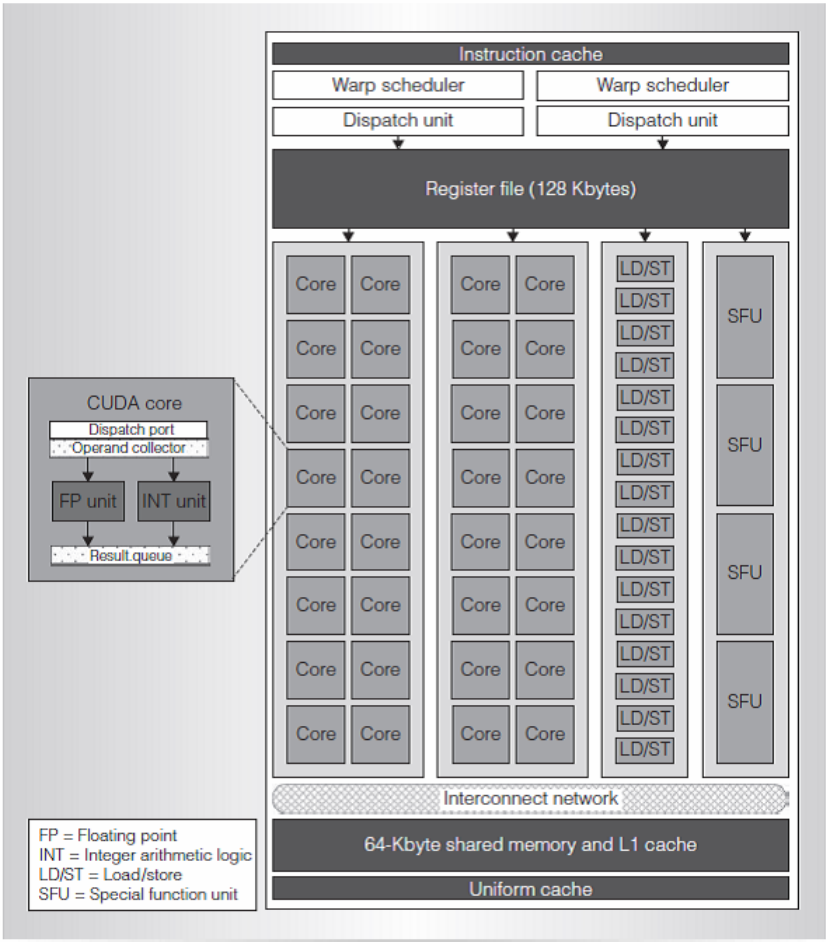



Figure 3.2. Fermi Streaming Multiprocessor [68].

The streaming multiprocessor implements zero-overhead multithreading and thread scheduling for up to 1,536 concurrent threads. To efficiently manage and execute this many individual threads, the multiprocessor employs the single-instruction multiple thread (SIMT) architecture introduced in the first unified computing GPU. The SIMT instruction logic creates, manages, schedules, and executes concurrent threads in groups of 32 parallel threads called warps. A CUDA thread block comprises one or more warps. Each Fermi streaming multiprocessor has two warp schedulers and two dispatch units that each select a warp and issue an instruction from the warp to 16 CUDA cores, 16 load/store units, or four SFUs. Because warps execute independently, the streaming

multiprocessor can issue two warp instructions to appropriate sets of CUDA cores, load/store units, and SFUs.

### 3.1.4. CUDA Programming Model

CUDA is a hardware and software coprocessing architecture for parallel computing that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. Because most languages were designed for one sequential thread, CUDA preserves this model and extends it with a minimalist set of abstractions for expressing parallelism. This lets the programmer focus on the important issues of parallelism—how to design efficient parallel algorithms—using a familiar language.

By design, CUDA enables the development of highly scalable parallel programs that can run across tens of thousands of concurrent threads and hundreds of processor cores. A compiled CUDA program executes on any size GPU, automatically using more parallelism on GPUs with more processor cores and threads.

A CUDA program is organized into a host program, consisting of one or more sequential threads running on a host CPU, and one or more parallel kernels suitable for execution on a parallel computing GPU. A kernel executes a sequential program on a set of lightweight parallel threads. As Figure 3.3 shows, the programmer or compiler organizes these threads into a grid of thread blocks. The threads comprising a thread block can synchronize with each other via barriers and communicate via a high-speed, per-block shared memory.

Threads from different blocks in the same grid can coordinate via atomic operations in a global memory space shared by all threads. Sequentially dependent kernel grids can synchronize via global barriers and coordinate via global shared memory. CUDA requires that thread blocks be independent, which provides scalability to GPUs with different numbers of processor cores and threads. Thread blocks implement coarse-grained scalable data parallelism, while the lightweight threads comprising each thread

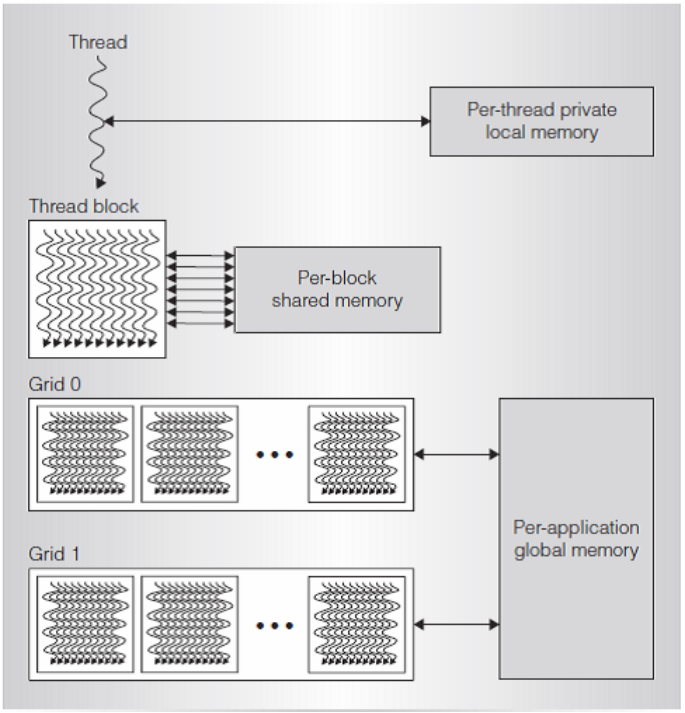



Figure 3.3. The CUDA Thread Hierarchy with Corresponding Memory Spaces [68].

block provide fine-grained data parallelism. Thread blocks executing different kernels implement coarse-grained task parallelism. Threads executing different paths implement fine-grained thread-level parallelism. Details of the CUDA programming model are available in the programming guide [73].

## 3.2. Multicore Computing

### 3.2.1. INTEL® Core i7 Architecture

Intel Core i7 is a family of several Intel desktop x86-64 processors, the first processors released using the Intel Nehalem microarchitecture and the successor to the Intel Core 2 family. The Intel Core i7 is a high-performance general-purpose processor in all respects. It attempts to do everything well. This comes at the cost of a high (140 W) maximum power dissipation. It is implemented with up to eight four-issue out-of-order, two-way symmetric multithreading (SMT) cores, as seen in Figure 3.4. These cores contain many complex enhancements to extract as much performance out

of a single thread as possible. Each core also contains a 128-bits SIMD unit to take advantage of some data parallelism. In keeping with most Intel processors, it supports the CISC x86 ISA. This design allows it to do many things well, but lower power more specialized designs can compete favorably in particular application domains. The memory system is typical of that found in a general purpose multicore machine with just a few cores. It uses a fully coherent memory system and has large standard caches. The coherence is broadcast based, which is sufficient because of the limited number of cores. These characteristics come together to create a chip that is good at a wide variety of applications provided power is not a constraint.

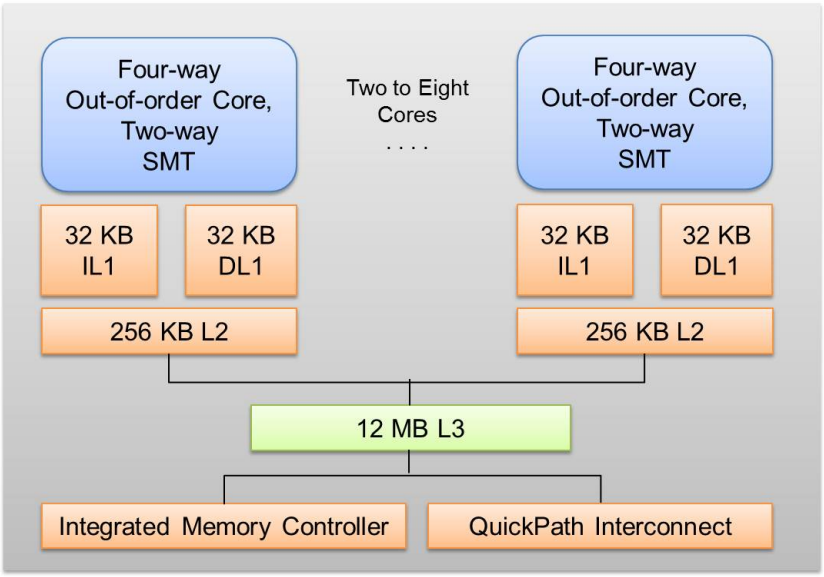



Figure 3.4. Intel Core i7 Architecture Block Diagram.

### 3.2.2. OpenMP Programming Model

There are a number of alternatives for parallel programming on multicore CPUs such as OpenMP, POSIX threads (pthreads) [74], raw native threads, some custom APIs like Intel® TBB (Threading Building Blocks) [75].

We considered to use OpenMP in this work. OpenMP (Open specifications for MultiProcessing) is a shared-memory application programming interface (API) originally developed for SMPs (shared memory parallel processors) to provide portable,

user-friendly, and efficient approach for parallel programming. This model assumes, as its name implies, that programs will be executed on one or more processors or cores that share some or all of the available memory. Its specification was defined by the *OpenMP Architecture Review Board* (ARB).

It is flexible for creating parallel code and well suited for programming multicore CPUs. It is a compiler-supported approach. Compared to other parallel programming approaches, OpenMP directives make it easy to specify parallel loop execution, to create teams of threads, to specify how to share work among the members of a team, to synchronize threads, and to specify whether or not data is to be shared. For our applications we have seen that this would be sufficient.

Although they expose the control of parallelism at its lowest level, especially for raw threads, programming with other approaches is usually much more complex, and the resulting code is likely to differ substentially from a sequential program. For most of them, the programmer must declare threading structures, create and terminate the threads individually, compute the loop bounds for each thread. If interactions occur within loop iterations, the amount of thread-specific code can increase substantially.

The OpenMP is comprised of three primary API components that influence run-time behavior: compiler directives, runtime library routines and environment variables. It is a portable programming model having specifications for C/C++ and Fortran and already available on many systems (Linux, Windows, IBM, etc.). The API is designed to permit an incremental approach to parallelizing an existing code without requiring a major reorganization of the original sequential code. Thus, it could be possible to have both sequential and parallel versions of an application in one source code via conditional compilation.

OpenMP is based on so-called *fork-join* execution model as illutrated in Figure 3.5. Under this approach, execution is started by a single thread called “master thread”. When a parallel region is encountered, the master thread spawns a set of threads called a team (*fork*). The set of instructions enclosed in a parallel region is executed. At the

end of the parallel region all the threads synchronize and terminate leaving only the master thread (*join*). Parallel regions and other OpenMP constructs are defined by means of compiler directives. OpenMP implementation sorts out the low-level details of actually creating independent threads to execute the code and assign work to them according to the strategy specified by the programmer.

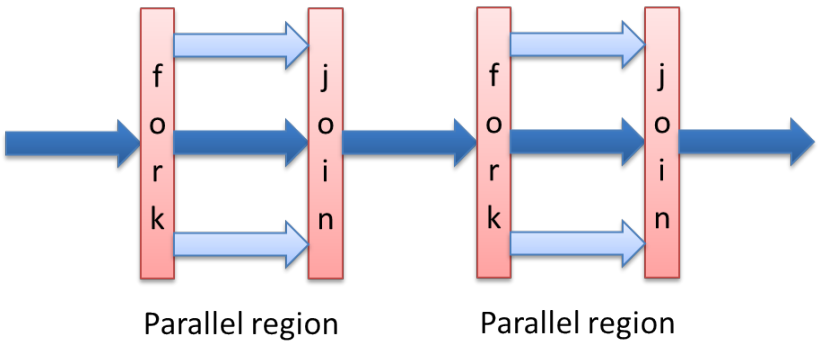



Figure 3.5. OpenMP Execution Model.

The `parallel` one is the main OpenMP construct and identifies a block of code that will be executed by multiple threads in parallel. The number of threads depends on setting of the `NUM_THREADS`, use of the `omp_set_num_threads()` library function or setting of the `OMP_NUM_THREADS` environment variable. Although it could be dynamic, the implementation default is usually the number of cores on a node. The master is a member of the team and has thread number 0. Starting from the beginning of the region, the code is duplicated and all threads will execute that code. There is an implied barrier at the end of a parallel section. Figure 3.6 shows an example `parallel` and `for` constructs.

```
omp_set_num_threads(...)
#pragma omp parallel shared (...) private(...)
{
   #pragma omp for schedule(...)
      for-loop
} /* end of parallel region */
```

Figure 3.6. Example OpenMP parallel construct.

The other constructs of OpenMP include data scoping constructs explicitly defining how variables should be scoped in parallel regions such as `shared`, `private`, worksharing constructs dividing the execution of the enclosed code region among the members of the team that encounter it such as `for`, `sections`, `single` and synchronization clauses such as `critical`, `master`, `barrier`, and `atomic`. More detailed information about OpenMP can be found in [76]. The official specifications can be obtained from [77].

# 4. LOCALIZATION AND MAP MATCHING

Most of the ADAS and autonomous vehicle applications fuse different type of sensors for a better sensing of the environment and the vehicle's own state [21]. Localization and map matching are among the fundamental tasks of ADAS and autonomous vehicle applications [40]. We consider a particle filter based localization and map matching algorithm proposed in [43] where GPS (Global Positioning System) and odometer data is fused with digital map information as an additional sensor. Figure 4.1 shows our configuration of vehicle localization sensor fusion system.

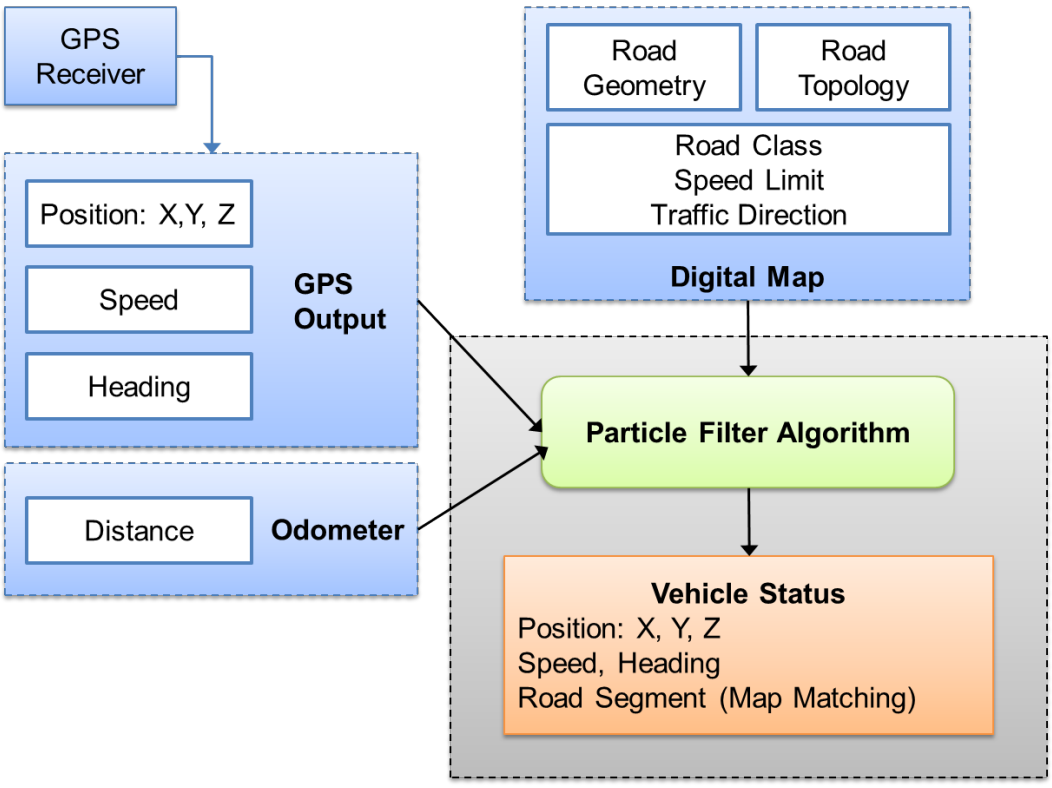



Figure 4.1. Vehicle localization fusion system.

Particle filters are among the principal tools for the on-line estimation of the state of a non-linear dynamic system [32,33]. Particle filtering has been applied widely in applications in tracking, navigation, detection and video-based object recognition [34,78]. Although, in general, particle filtering methods yield improved results compared to

other Bayesian filters, such as Kalman filters, it is difficult to achieve real time performance as the algorithm is computationally intensive [58]. This has been a prohibitive factor for real-time implementations for many applications of particle filtering.

Recently, particle filters are proposed in the context of localization and map matching. We consider a multi-hypothesis localization and map matching algorithm where map topology information is used in terms of routeability as the likelihood calculation in the particle filter to increase map matching performance, at same time further increasing the computational cost of the algorithm.

The algorithm makes use of digital maps as a prior in particle filtering algorithm. Two main features of maps are used to bias the particles in more towards the possible areas. First feature of the map is the function of the area of a particle. The probability of being in a certain location is calculated by checking if the location is on a road or not on a road segment. If on a road segment, type of road is checked. Probability of being in a certain location is calculated in conjunction with the speed of the vehicle. Second feature of the map is the topology. Given a previous location of a particle probability of travelling to another location on a certain road segment is calculated. If the new location is topologically not possible less probability is assigned.

## 4.1. Related Work

With the rise of the massive parallelization made possible by GPUs and multicore CPUs, researchers have worked on parallel versions of particle filters in the past decade. These algorithms differ in the specific parallelization, the number of particles they can handle, and the degree of communication between the computing units. This section examines a number of attempts to implement distributed and parallel particle filters [79].

The first work dealing with parallelism in particle filters is [80]. In this work, the particles are partitioned into several subsets, each partition is assigned to a separate processor. Sampling, weight calculations, and resampling are performed independently

and locally for each subset. The weighted sum of all the particles is considered as the estimation. A local estimate and a local sum of weights are calculated for each subset first, then these local estimates are centrally gathered and combined into a global estimate. It is claimed that local resampling is comparable with global resampling, in terms of estimation error. The authors present performances in terms of execution time, with 4000 and 32000 particles, on several parallel supercomputers where number of processing elements ranging from 2 to 32.

Three methods are proposed to implement parallel particle filters in the work [81]: (i) Global Distributed Particle Filter (GDPF), (ii) Local Distributed Particle Filter (LDPF), and (iii) Compressed Distributed Particle Filter (CDPF). In GDPF, only the sampling and weight calculation steps run in parallel on different processing elements, while resampling is performed centrally. For the resampling step, all particle data is transferred to a central processor and the new particles are sent back to each processing element. The central processor calculates the global estimate from the particle data. In LDPF, resampling is also performed locally on each processor without any communication with other processors. Aggregated particle data is sent to a central unit in order to calculate the global estimate similar to the algorithm in [80]. With CDPF, resampling and the calculation of the global estimate are performed centrally, but only a small representative subset of the particles of each processor are sent to the central processor. It is concluded that LDPF provides both better estimation and performance with a number of simulations where number of particles ranging from 50 to 5000 and number of processors from 1 to 10.

Two parallel particle filter algorithms are proposed in [82]: (i) Resampling with Proportional Allocation (RPA), and (ii) Resampling with Non-proportional Allocation (RNA). The sampling and weight calculation steps are performed in parallel in both algorithms. In the RPA method the resampling step involves centralized communication, whereas RNA method performs it completely locally. Various particle exchange methods are proposed to improve the performance of local resampling step. The number of particles that are exchanged among the processing elements is a significant fraction of the total particles (at least 25% of all particles for each core). For both algorithms

(RPA and RNA) the estimate is calculated as the weighted average of all particles from all processing units. It is concluded that RPA provides a better estimation, while RNA has a simpler design. In a later work [83], the authors compares a standard Particle Filter with a Gaussian Particle Filter on an FPGA. The results indicate that the Gaussian Particle Filter, while being faster than a standard particle filter, is equally accurate for (near-)Gaussian problems.

Some of the previously presented algorithms (GDPF, RNA, RPA, Gaussian particle filter) are compared using an OpenMP implementation on a 4-core CPU for a bearings only tracking problem in [84]. The Gaussian particle filter outperforms (in terms of accuracy over computational time) all other algorithms, since the state estimation problem is Gaussian. The other particle filter algorithms (GDPF, RNA, RPA) exhibit similar estimation accuracies. The number of particles in the comparison is limited to 10K particles. In terms of execution time performance, both RNA and the Gaussian Particle Filter achieve near linear speedups with respect to the number of cores for a large number of particles, while GDPF and RPA achieve only sub-linear speedup.

Another parallel particle filter implementation is presented in [85], where the authors exploit GPU specific hardware features. In this paper, first, a parallel approach for sampling and weight calculations is proposed and then, the resampling step is performed using a specific hardware feature of GPUs called the rasterizer. In practice, this step is close to the RNA algorithm in [82] but, since pseudo-random numbers are generated on the host CPU and subsequently transferred to the GPU, the performance of the filter is severely damaged. In fact, about 85% of the total execution time is spent on generating pseudo-random numbers and transferring them to the GPU. This makes the implementation not suitable for real-time state estimation in complex problems.

The work in [86] describes a GPU implementation which consists of parallel sampling, parallel weight calculations, and resampling performed locally on the different computing units. For the sampling stage, the finite-redraw importance-maximizing (FRIM) method is used, which checks the weight of the drawn particle and redraws

until a particle with a reasonable weight is constructed. The FRIM method is known to reduce the required total number of particles, but to limit the iterations, a fixed number for maximum number of redraws has to be imposed. The generation of pseudo-random numbers is performed on the host CPU, as in [85], and then copied to the GPU. This makes the implementation quite limited. In fact, with the use of a low performance laptop-GPU, the experiments go up to 4K particles with execution times around 200 ms in the best case. It is not clear how the estimate is calculated from the weighted particles and whether it is executed on the GPU or on the host CPU.

Another recent study [87] presents a GPU-accelerated particle filter for visual tracking. The particle filter is partially executed on the GPU for accelerating the image processing steps of the application. The sampling, estimation and resampling steps are performed on the host CPU and only the weights are calculated on the GPU. They use a standard SIR (Sampling Importance Resampling) particle filter implementation. They are able to run experiments only up to 4000 particles achieving around 40 frames per second.

In one of the recent studies, Ferreria, Lobo and Dias [88] present real-time implementation of a Bayesian framework using CUDA including a particle filter specialized for real-time robot vision. Goodrum *et al.* [89] presents a CUDA and OpenMP implementation of particle filter for a single target video tracking application. Another study by Ulman [90] presents a CUDA implementation of particle filter optimized for tracking naval vessels. Although these projects utilize the general particle filter algorithm, they differ significantly in their calculation of the likelihood phase. This variety also influence the approach used in parallelization.

A number of methods for software and hardware implementations of particle filtering have been proposed in the literature. Special architectures [91], field-programmable gate arrays (FPGAs) [92], and SIMD processor arrays [93], cluster of computers [94] have been utilized for various types of problems. Many of the graphics processing unit (GPU) implementations are focused on low-level stream processing or OpenGL [85, 95, 96].

Although emerging multicore processors and GPUs are good candidates for parallel particle filter implementations in embedded applications, multicore implementations, especially using the GPU computing concept and the platforms and tools such as NVIDIA's CUDA architecture, are still very recent and few.

## 4.2. Particle Filter

Particle Filters, also known as Sequential Monte Carlo (SMC) methods, are iterative methods that track a number of possible state estimates, so-called particles, across time and gauge their probability by comparing them to measurements.

We are considering a dynamic system with state $x_t$ at a given time $t$. As there can be uncertainty in the state information, we model the initial system state by its probability distribution $p(X_0)$, where $X_0$ is a random variable describing the state at time $t = 0$. The *system model* is a Markov process of the first order. Thus, $p(X_t|X_{t-1})$ denotes the probability distribution of the system's current state given the system's previous state. We assume that the system state can only be tracked by measurements $y_t$, which may be influenced by noise. The relation between measurements and system states is described by the *measurement model*.The distribution $p(Y_t = y_t|X_t)$ describes the probability of the current measurement given the system's current state.

The *sampling importance resampling* (SIR) algorithm is one of the most widely used sequential Monte Carlo methods, which allow system state estimates to be computed on-line while the state changes as it is the case for tracking algorithms. A SIR filter usually manages a fixed number of possible system state hypotheses $x_t^i$, also called particles. Ideally, these particles approximate the distribution of the system state $p(X_t)$. The SIR algorithm has distinct stages iterated over discrete time steps. Figure 4.2 graphically represents one iteration. The individual stages are:

### 4.2.1. Sampling (Prediction)

To follow the state during subsequent iterations, the system model is used to obtain a possible new state for every particle $x_t^i$ based on its last state $x_{t-1}^i$ where $u_{t-1}$ is measured inputs and $v_{t-1}$ unmeasured forces or faults:

$$x_t^i = Ax_{t-1}^i + Bu_{t-1}^i + v_{t-1}^i, i = 1, ..., N \tag{4.1}$$

Formally, this corresponds to drawing or *sampling* the new particles from the distribution $p$. Now, the set of particles $x_t^i$ forms a prediction of the distribution of $X_t$.

### 4.2.2. Importance (Update)

The measurement model is evaluated for every particle and the current measurements to determine the *likelihood* that the current measurement $y_t$ matches the predicted state $x_t^i$ of the particle. The resulting likelihood is assigned as a weight $w_t^i$ to the particle and indicates the relative quality of the state estimation:

$$w_t^i = w_{t-1}^i p(y_t|x_t^i), i = 1, ..., N \tag{4.2}$$

At this point, when the particles are weighted, a state estimation can easily be obtained via various techniques, such as using the highest-weighted (highest-probability) sample, or using the weighted sum of the particles to get a mean-equivalent, or using the average of particles within some distance from the best particle. As an approximation we can take the following after normalizing the weights to

$$w_t^i = \frac{w_t^i}{\sum_{i=1}^N w_t^i} \tag{4.3}$$

$$\hat{x_t} \approx \sum_{i=1}^{N} w_t^i x_t^i \tag{4.4}$$

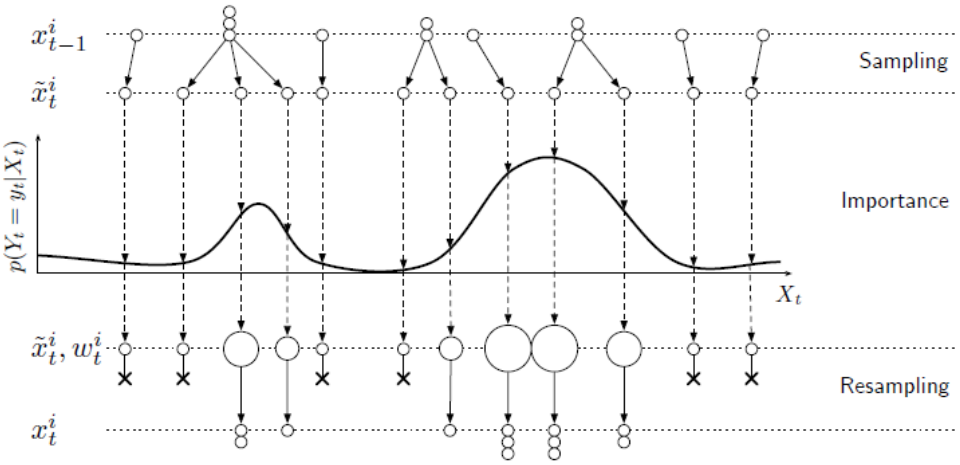



Figure 4.2. Illustration of one iteration of sampling importance resampling algorithm.

### 4.2.3. Resampling

If the number of effective samples fall below a certain value, resampling is required. Particles with comparatively high weights are duplicated and particles with low weights are eliminated. The distribution of the resulting particles $x_t^i$ approximates the distribution of the weighted particles before resampling. This can be done by calculating the number of effective particles $N_{eff}$ as follows:

$$N_{eff} = \frac{1}{\sum_i (w_t^i)^2} \tag{4.5}$$

When $N_{eff}$ is lower than a certain threshold resampling is done. The threshold value for $N_{eff}$ can be set to a value depending on $N$. Effective sample size (ESS) is another metric to decide if resampling is required. After resampling all the weights are set to the same value.

The basic flow of SIR particle filter algorithm is shown in Figure 4.3.

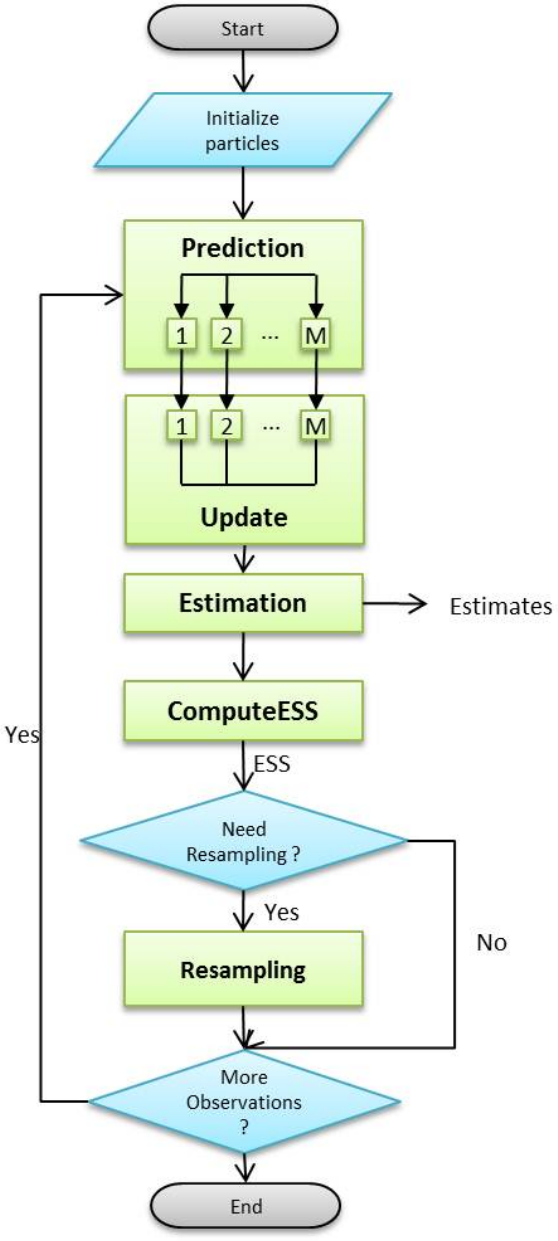



Figure 4.3. Particle Filter Algorithm.

## 4.3. Particle Filter based Localization and Map Matching

For the vehicle localization problem, state is represented as a four-dimensional vector $x$=[$Lon$, $Lat$, $\Theta$, $L$] where $Lon$, $Lat$, $\Theta$ and $L$ stand for position, orientation and link or road segment on the map database, respectively.

Basically, the new location of the vehicle is predicted using the odometer data in the prediction stage and corrected by the GPS measurements and a map based likelihood function in the weight update stage.

The operations performed in the main stages of the particle filter are summarized in the following sections.

### 4.3.1. Prediction

The data coming from the odometer is used to measure vehicle displacement. The new location ($Lon$, $Lat$) of the vehicle is randomly calculated for each particle in the range of this displacement. If the predicted location is on a road segment, it is also stored in the state of the particle as the predicted road segment ($L$). This stage requires a high number of random number generations for the calculation of the new values of each state variable.

### 4.3.2. Weight Update

Weights are updated using the GPS readings first. The likelihood function is designed so that the particles that are within the error range of the GPS reading get higher weights. Then the weights are augmented with the map data by multiplying them with the probabilities derived from the map:

$$w_t^i = w_{t-1}^i \times p(zone) \times p(topology) \tag{4.6}$$

Two types of map attributes are used in the likelihood calculation. The first feature is the type of area where the particle resides on the map (road segment, building, parking area, etc.). The probability of being in a certain type of zone or road class (e.g. motorway, major road, local road, residential road, etc.) is calculated based on the speed of the vehicle (e.g., for a vehicle at a speed of 120 km/h, the relative probability of being on a motorway is chosen to be higher than being on a residential road). The second feature of the map is the topology. Given the previous location of

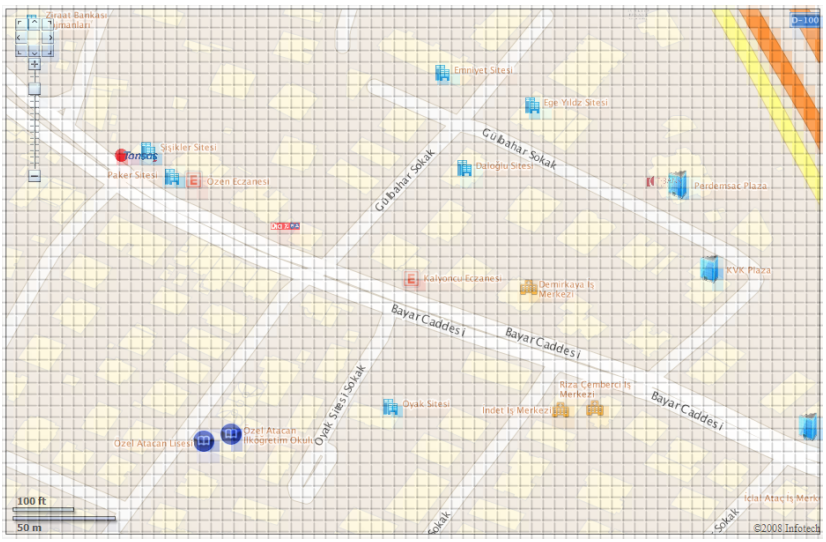
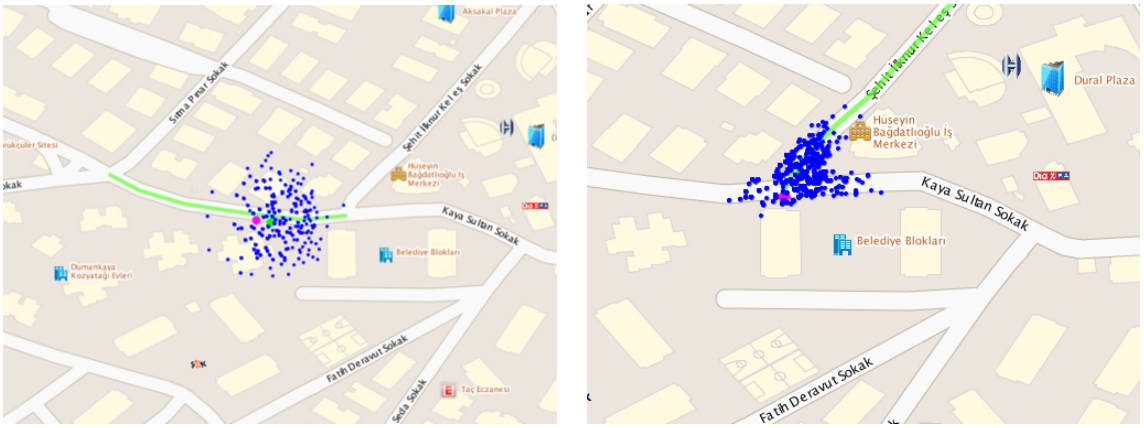



Figure 4.4. Zone Map.

a particle, the probability of travelling to a new location on a certain road segment is calculated using the map topology. Possible reachable roads are searched in the road network in forward and backward directions for the distance travelled measured from the odometer. If the predicted location of the particle is found to be reachable, a high probability is assigned, otherwise a low probability is assigned (e.g., due to the connectivity, direction of traffic flow, turn restrictions, etc.).

Behavior of the particles shows how particle filter can produce multiple hypotheses for edge selection. When map information is not used particles are scattered depending on the precision of the GPS information. When the map information is utilized particles are concentrated on possible edges. An example of particle behavior is illustrated in Figure 4.5.




(a) Behavior without using map information: particles are scattered

(b) Behavior with using map information: particles are concentrated on possible edges

Figure 4.5. Particle behavior with and without using map information.

Road network is made of nodes and edges. Edges are made of several vertices. Edges of the network can be unidirectional or bidirectional i.e. allowing traffic flow in each direction. Searching reachable roads in the network requires traversing all through the vertices. The complexity of traversal depends on the distance travelled and the topology of the network in the neighborhood.

### 4.3.3. Estimation

The location component of the system state is calculated as the weighted mean of each particle's location information. Map matching is achieved by selecting the road segment with highest weight as the matched link on the map. Figure 4.6 shows the estimated positions and road segments on the map. Purple dots show measured location, green dots show estimated location and green lines show matched road segment.

The flow of our particle filter algorithm for localization and map matching is shown in Figure 4.7.


Figure 4.6. Display of estimated positions and road segments.

## 4.4. Paralel Implementations

Before attempting parallel implementations, we first characterized the execution profile of the particle filter algorithm for different number of particles using a sequential implementation (a single threaded implementation in C). The percentages of three distinct parts of the algorithm were examined and critical function blocks in terms of execution time were identified. Figure 4.8 shows the execution time breakdown for different number of particles ranging from 256 to 128K. We see that the prediction and update sections dominate the execution time by a large margin. Therefore, those sections were selected as the first targets of parallelization in both platforms.

Particle filters heavily use random number generation. Our particle filter implementation uses the Mersenne-Twister random number generation algorithm. An existing implementation of Mersenne-Twister algorithm has been adopted for both the CPU and GPU platforms. Each thread is provided with its own random number generator instance.

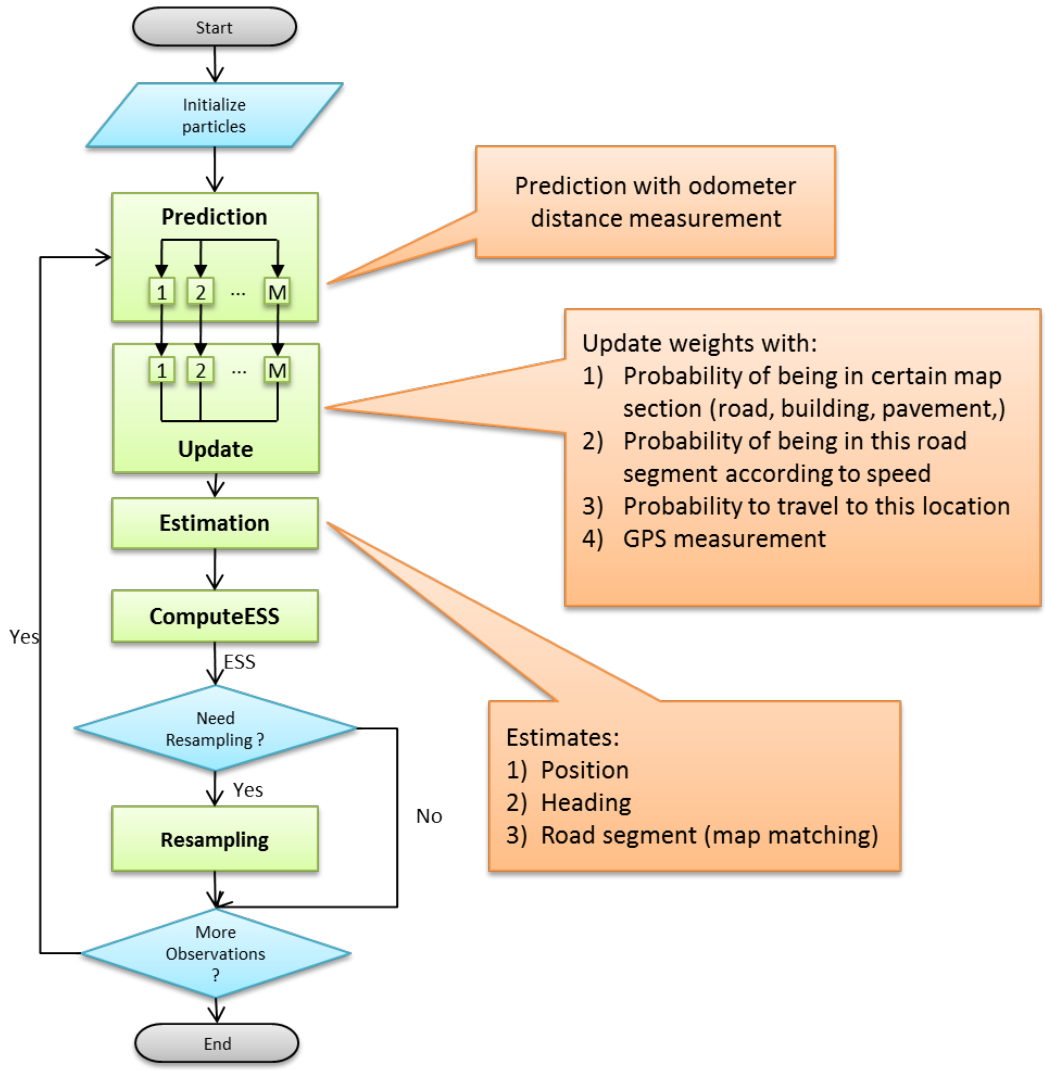
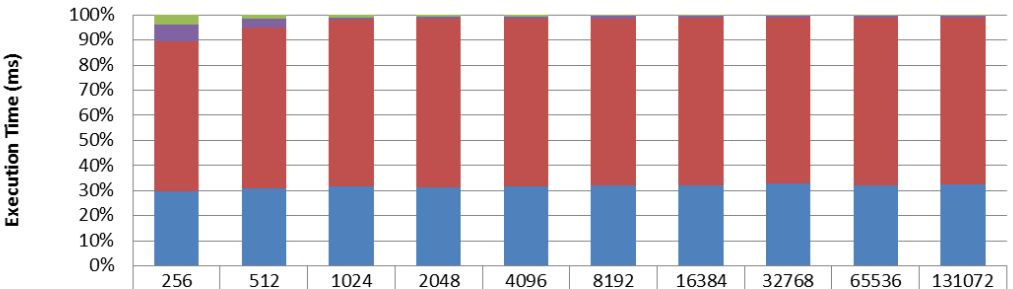



Figure 4.7. Particle Filter Localization and Map Matching.




| | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 131072 |
|---|---|---|---|---|---|---|---|---|---|---|
| Resampling | 0.06 | 0.05 | 0.05 | 0.08 | 0.13 | 0.21 | 0.37 | 0.72 | 1.30 | 2.77 |
| Estimation | 0.09 | 0.084 | 0.04 | 0.08 | 0.164 | 0.33 | 0.67 | 1.38 | 2.72 | 5.61 |
| Update | 0.85 | 1.69 | 3.38 | 6.7 | 13.36 | 26.52 | 54.24 | 110.39 | 219 | 434 |
| Prediction | 0.42 | 0.81 | 1.6 | 3.15 | 6.35 | 12.74 | 26.26 | 54.96 | 105 | 213 |

Figure 4.8. Execution Time profile of Particle Filter Localization and Map Matching.

### 4.4.1. Multicore (OpenMP) Implementation

We used OpenMP (Open Multi-Processing) programming model [76, 77] for the parallelization of the *predict* and *update* sections of the particle filter on a multicore CPU. OpenMP is an application programming interface that provides shared memory parallel programming on many architectures. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior.

Since mainly the same operations are repeated for all the particles in a loop for both the *predict* and *update* sections and the particles can be processed independently of each other, the iterations (effectively the particles) have been distributed among the cores. Each core therefore performs the prediction and update steps on a subset of particles.

The `for` construct is used to parallize the loops. It specifies the iterations of the loop immediately following it must be executed in parallel by the team. The `schedule` OpenMP work-sharing clause in the `for` construct specifies how the cycles of the loop are assigned to threads. The static scheduling mechanism of OpenMP is used for the *predict* part where the operations are the same for each particle. The `static` clause causes loop iterations to be divided into pieces of size *chunk_size* and

then statically assigned to threads in a round-robin fashion in the order of the thread number. It is suitable for regular workloads. Dynamic scheduling has been employed for the *update* part, in order to have a better workload distribution among the cores since the complexity of map based operations for each particle in the update step can be different. The `dynamic` clause causes loop iterations to be divided into pieces of size *chunk_size* and then dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another until there are no more chunks to work on.The last chunk may have fewer iterations than *chunk_size*. The pseudo code for multicore CPU implementation of prediction and update parts are shown in Figure 4.9 and Figure 4.10, respectively.

```
// Set the number of threads
omp_set_num_threads(NUM_THREADS)
// Loop for each particle
#pragma omp parallel for private(i, particles) schedule(static, 1)
for i = 1 → NUM_PARTICLES do
  // Compute predicted location and heading
  particles[i].longitude += pOdometer * drawGaussian_normalized(mtState, i);
  particles[i].latitude += pOdometer * drawGaussian_normalized(mtState, i);
  particles[i].heading += TRANSITION_MODEL_STD_H * drawGaussian_normalized(mtState, i);
end for
```

Figure 4.9. Pseudo code for multicore CPU Implementation of Prediction.

### 4.4.2. GPU (CUDA) Implementation

In our GPU implementation, we used the CUDA programming model. This actually represents a hybrid (CPU+GPU) implementation of particle filter. We implemented most of the main steps of the filter in C using CUDA Toolkit 3.2 [73]. The *Prediction*, *Update*, *Estimation*, and *ComputeESS* parts were implemented as parallel kernels to run on GPU (device), where resampling part is run on CPU (host). The

```
// Measurements Input
// from GPS: pLongitude, pLatitude, pHeading, pSpeed, pHDop, pVDop
// from Odometer: pOdometer
// Set the number of threads
omp_set_num_threads(NUM_THREADS)
// Loop for each particle
#pragma omp parallel for private(i, particle, pSpeed, pLatitude, pLongitude, pHeading, pOdometer,. . .) schedule(dynamic, CHUNKSIZE)
for i = 1 → NUM_PARTICLES do
  // Find the current road with predicted location
  roadId = getRoadIdWithCoor(particle[i].latitude, particle[i].longitude)
  // Probability being on this road based on speed
  if roadId! = 0 then
    // On the road
    roadProb = getLocProbability(roadId, pSpeed)
  else
    // Not on the road
    if pSpeed > 0 then
      roadProb = SMALL_ZONE_PROB
    else
      roadProb = MID_ZONE_PROB
    end if
  end if
  // Probability to travel to this location
  // Find possible roads from previous location in forward direction
  roads = findNextRoadInfo(pOdometer, DIR_FORWARD)
  if currentRoadisinpossibleroads then
    routeProb = HIGH_PROB
  end if
```

```
    // Find possible roads from previous location in backward direction
    roads = findNextRoadInfo(pOdometer, DIR_BACKWARD)
    if currentRoadisinpossibleroads then
      routeProb = HIGH_PROB
    end if
    // update particle weight using errors
    particle[i].weight *= normalPDF(particle[i].longitude - pLongitude, pHDop)
    particle[i].weight *= normalPDF(particle[i].latitude - pLatitude, pVDop)
    particle[i].weight *= normalPDF(particle[i].heading - pHeading, pHError)
    // update particle weight using probabilities
    particle[i].weight *= roadProb * routeProb
end for
```

Figure 4.10. Pseudo code for multicore CPU Implementation of Update.

CUDA implementation flow is illustrated in Figure 4.11.

Since the prediction and update parts of a particle filter work on particles independently, a separate thread is created for each particle on the GPU for the *predict* and *update* kernels. This is accomplished by using the appropriate execution configuration parameters, namely grid size and block size, when the kernels are launched. Each thread determines which particle it should process via built-in variables, namely the thread block index `blockIdx.x`, the thread index within its block `threadIdx.x`, and the total number of threads per block (block size) `blockDim.x`. The pseudo code for GPU implementation of *predict* is shown in Figure 4.12.

The states of particles are stored in the global device memory, and during initialization both host and device memory are allocated for particles, and the initial particle data are copied to the device. The global memory is used to pass on data from one kernel to the next. Map data (both road network and grid based zone map) is also transferred to the device memory during initialization.

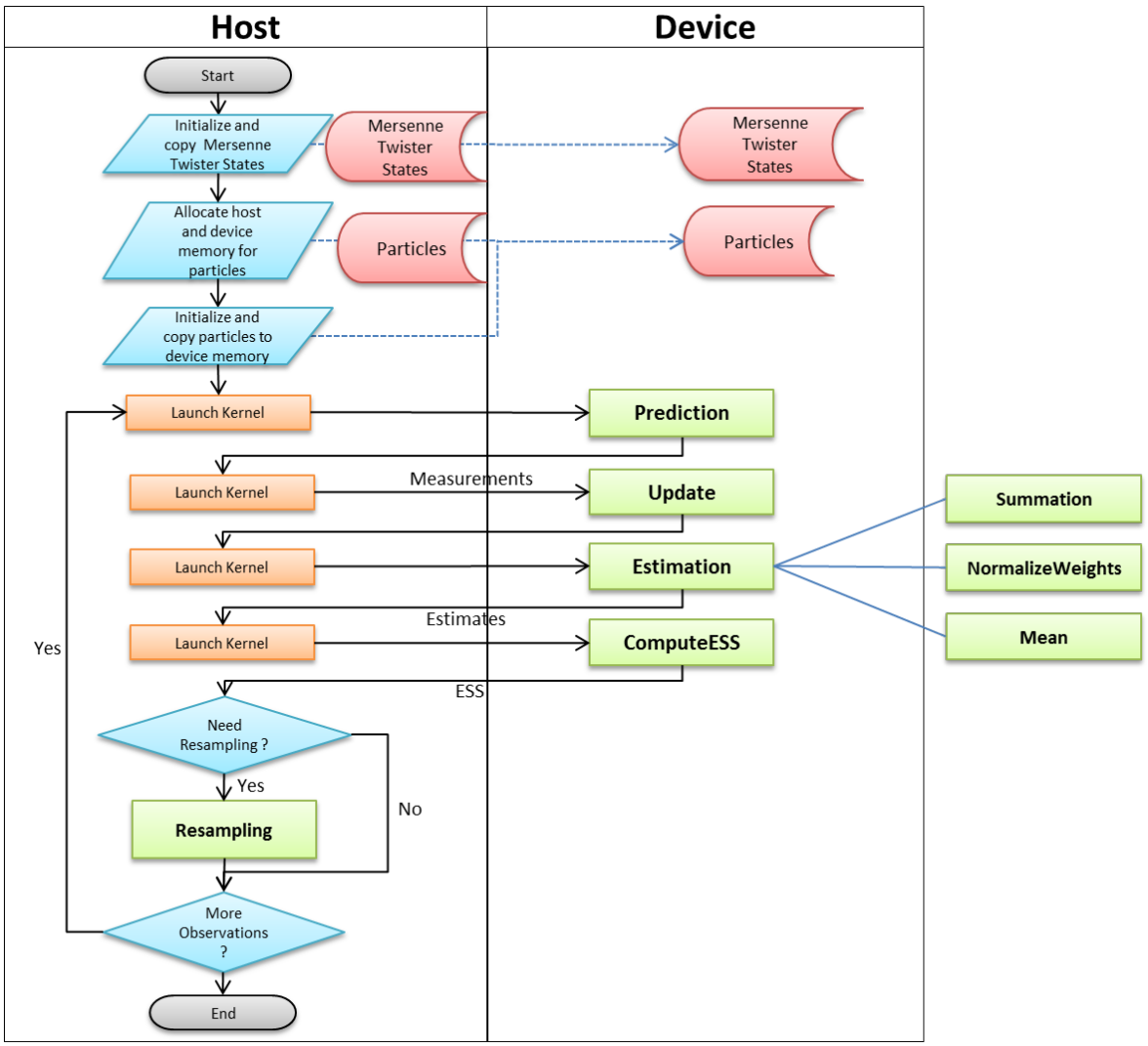



Figure 4.11. CUDA implementation of particle filter localization and map matching.

The same Mersenne-Twister random number generator code has been used in the device and each thread is enabled to use its own random number generator instance. Initial Twister states for the maximum possible number of threads are created on the host and transferred to the device memory at the initialization.

```
__global__ Predict Kernel(particles, mtState, pOdometer)
tid = blockIdx.x × blockDim.x + threadIdx.x;
particles[tid].longitude += pOdometer* drawGaussian_normalized(mtState, tid);
particles[tid].latitude += pOdometer* drawGaussian_normalized(mtState, tid);
particles[tid].heading += TRANSITION_MODEL_STD_H * drawGaus-
sian_normalized(mtState, tid);
```

Figure 4.12. Pseudo code for GPU Implementation of Predict.

The *update* kernel is implemented by creating a separate thread for each particle on the GPU similar to the *predict* kernel. The measurement values are passed as parameters at the kernel launch for each iteration. The pseudo code for GPU implementation of *update* is shown in Figure 4.13.

The *estimation* part consists of the summation and normalization of the weights and the calculating weighted mean of the state variables. This part is divided into three separate kernels: *summation*, *normalizeWeights* and *mean* kernels as shown in in Figure 4.11.

The division of the processing workload into separate kernels was necessary due to the fact that the only way to enforce synchronization between all concurrent CUDA threads in a grid is to wait for all kernels running on that grid to exit (as opposed to all threads in a block which can be synchronized by `syncthreads()` instruction) – this is only possible at CUDA stream level.

For the *summation* kernel, three different implementations were tested. In the first one, global atomicAdd operations are used. In the second one, each block calcu-

```
__global__ Update Kernel(particles, mtState, pOdometer, pSpeed, pLongitude,
pLatitude, pHDop, pHError)
tid = blockIdx.x × blockDim.x + threadIdx.x;
// Find the current road with predicted location
roadId = getRoadIdWithCoor(particle[tid].latitude, particle[tid].longitude)
// Probability being on this road based on speed
if roadId! = 0 then
  // On the road
  roadProb = getLocProbability(roadId, pSpeed)
else
  // Not on the road
  if pSpeed > 0 then
    roadProb = SMALL_ZONE_PROB
  else
    roadProb = MID_ZONE_PROB
  end if
end if
// Probability to travel to this location
// Find possible roads from previous location in forward direction
roads = findNextRoadInfo(pOdometer, DIR_FORWARD)
if currentRoadisinpossibleroads then
  routeProb = HIGH_PROB
end if
// Find possible roads from previous location in backward direction
roads = findNextRoadInfo(pOdometer, DIR_BACKWARD)
if currentRoadisinpossibleroads then
  routeProb = HIGH_PROB
end if
```

```
// update particle weight using errors
particle[tid].weight *= normalPDF(particle[tid].longitude - pLongitude, pHDop)
particle[tid].weight *= normalPDF(particle[tid].latitude - pLatitude, pVDop)
particle[tid].weight *= normalPDF(particle[tid].heading - pHeading, pHError)
// update particle weight using probabilities
particle[tid].weight *= roadProb * routeProb
```

Figure 4.13. Pseudo code for GPU Implementation of Update.

lates a partial sum using atomicAdd operations on shared memory and then adds this partial sum to the global sum by using a global atomicAdd operation at the end. In the third implementation, the parallel reduction technique is used to calculate the partial sums within each block and these partial sums are added to the global sum by using global atomicAdd operations [97]. The parallel reduction version was used for the final implementation since its performance was found to be superior. The parallel reduction technique is illustrated in Figure 4.14. The pseudo code for GPU implementation of summation is shown in Figure 4.15.

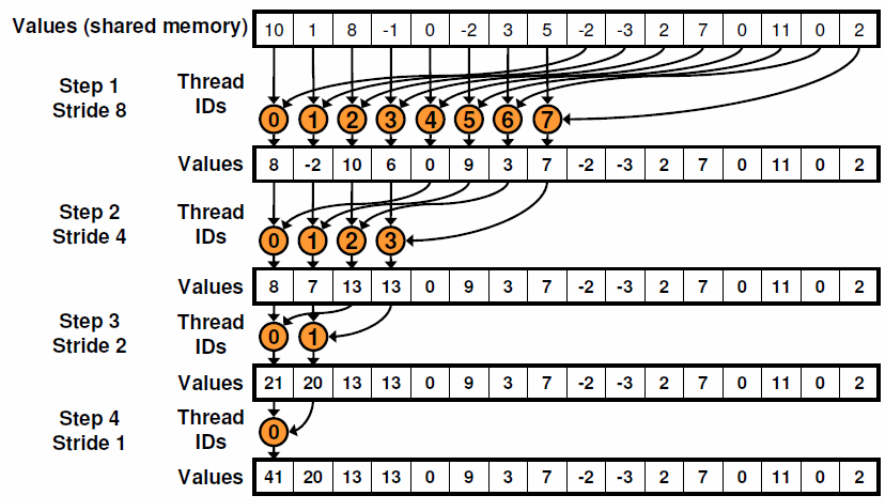



Figure 4.14. Parallel Reduction – Sequential Addressing.

The *normalizeWeight* kernel is implemented similar to the *predict* and *update*

```
__shared__ float sdata[256]
// Load shared memory
unsigned int tid= threadIdx.x;
unsigned int i= blockIdx.x * blockDim.x + threadIdx.x;
sdata[tid] = particles[i].W;
__synchthreads();
// do reduction in shared memory
for s = blockDim.x/2 → s > 0; s >>= 1 do
  if tid < s then
    sdata[tid]+ = sdata[tid + s];
  end if
  __synchthreads();
  // write result for this block to global memory
  if tid = 0 then
    atomicAdd(&data[0], sdata[0]);
  end if
end for
```

Figure 4.15. Pseudo code for GPU Implementation of Summation.

kernels. Each thread adjusts its weight independently by using the sum value which is passed to it as a parameter at the kernel launch. The *mean* kernel and the *computeESS* kernel also use the parallel reduction technique similar to the summation kernel. After the estimation is completed, the estimated state variables are transferred to the host.

The amount of data transfers between the host and device has been kept very small for the iterations where resampling is not required. Resampling only occurs if effective particle size (ESS) is lower than a certain threshold. If resampling is required, the current weights of the particles are transferred to the host, and the surviving particles are calculated on the host by the resampling process. Then surviving particles are duplicated according to their respective weights

# 5. TRAFFIC SIGN RECOGNITION

Traffic sign recognition (TSR) is one of the key components of Advanced Driver Assistance Systems (ADAS) and has been worked on for a long time in intelligent vehicles domain [98–100]. Although the appearance of the traffic signs was originally designed to be easily distinguishable from natural objects, the reliable, automated recognition of traffic signs, especially under adverse environmental conditions, remains as a complex task. Recent approaches on TSR tend to use a scheme of three stages:(i) localization/detection of sign candidates, (ii) classification of the candidates and (iii) tracking of the traffic sign candidates over time [101]. Many algorithms available in the literature generally differentiate in using different methods in the detection and classification stages [58, 59, 102, 103]. Figure 5.1 shows the structure of a traffic recognition system.

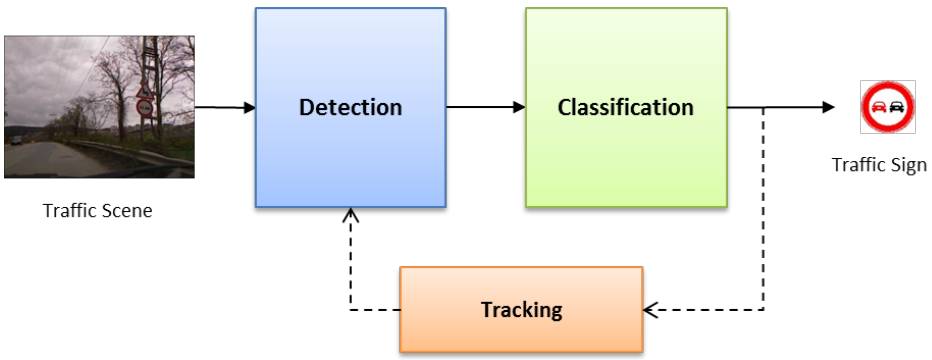



Figure 5.1. Structure of Traffic Sign Recognition System.

Detection: Many sign recognition systems use color segmentation for a preliminary reduction of the search space and apply algorithms like color thresholding, or the Bayesian classification of the color. Another commonly used approach for the detection of traffic signs is the search of their distinctive shapes, which can be easily differentiated from natural objects due to their artificial appearance. An intuitive approach on shape detection can be pattern matching. One of the most favorite algorithms applied to shape detection is the Hough transform and its derivatives. Local features represent the third major concept to detect road signs within traffic scenes. Common local

feature detectors are Scale Invariant Feature Transform (SIFT) [104, 105] and Speeded Up Robust Feature (SURF) [106].

Classification: There exist different ways to recognize the detected sign candidates. One of them is to apply methods of OCR/ODR (optical character/digit recognition) systems. Another approach uses pictogram-based classification of the traffic signs by template matching or cross correlation. Other classifiers like Multi-Layer Perceptron and Support Vector Machines are also widely used.

Tracking: To increase the robustness of TSR, many systems rely on tracking to maintain a road sign candidate over time. In this way, misinterpretations of the candidates can be reduced. The most common tracker adapted to the TSR problem is the Kalman filter and its modifications.

## 5.1. Related Work

Most of the work done in this field so far has been strictly bounded by available computing capacity. However, recent developments in multicore and many-core architectures present a research challenge, also in this area, to meet real-time performance requirements with a parallel processing model. There are a very limited number of studies in the literature for parallel implementations of traffic sign recognition. [107] and [108] describe the detection and classification of traffic signs on an application-specific multicore processor, respectively. For detection process, proposed approach is based on Viola-Jones algorithm. A multilayer perceptron net with a feed forward topology is used for classification. Two separate networks for speed limit signs and their abolishments as well as a general network for both categories were trained and used. A real-time template-based approach for the recognition of speed limit signs using GPU computing is described in [109]. A feature-based speed limit sign detection system using a GPU is described in [110]. The studies cover only speed limit signs and do not include map integration.

In another recent study, a GPU implementation of a road sign detector based

on Particle Swarm Optimization is presented [111]. [112] complements the detection described in [111] with a classification based on Learning Vector Quantization and Multi-layer Perceptron implemented on GPU using CUDA. It is reported that the system was tested over two real sequences taken from a camera mounted on-board a car and was able to correctly detect and classify around 70% of the signs at 17.5 fps.

There are also some attempts to enhance the performance of visible light cameras for sign recognition using infrared cameras [113, 114]. They are limited to a subset of traffic signs and use expensive hardware, and the recognition rate is lower than the proposed system.

## 5.2. Kinect Sensor

"Kinect for Xbox 360", or simply "Kinect" (originally known by the code name Project Natal), is a "controller-free gaming and entertainment experience" by Microsoft for the Xbox 360 video game platform [115]. Based around a webcam-style add-on peripheral for the Xbox 360 console, it enables users to control and interact with the Xbox 360 without the need to touch a game controller, through a natural user interface using gestures and spoken commands.

The Kinect sensor is a horizontal bar connected to a small base with a motorized pivot and is designed to be positioned lengthwise above or below the video display as seen in the following figure. The device features an RGB camera, depth sensor and multi-array microphone running proprietary software which provides full-body 3D motion capture, facial recognition and voice recognition capabilities.


Figure 5.2. Kinect Sensor.

Kinect has become very popular in a short time since its launch on November 4, 2010 (in North America) not only to play games, but also in robotics research for depth sensing and 3D vision with its relatively low price around $150 [116–119]. Other potential uses can be HCI, educational use, surveillance, motion capture, people/object tracking, 3D scanning, etc. However, we did not encounter an application of Kinect in intelligent vehicles research yet.

Kinect is actually a hardware which works by projecting an infrared laser pattern onto nearby objects. A dedicated IR sensor picks up on the laser to determine distance for each pixel, and that information is then mapped onto an image from a standard RGB camera. It is possible to end up with an RGBD image, where each pixel has both a color and a distance, which it can be used to map out body positions, gestures, motion, or even generate 3D maps.

The Kinect device has two cameras and one laser-based IR projector. Figure 5.3 shows their placement on the device. Each lens is associated with a camera or a projector.

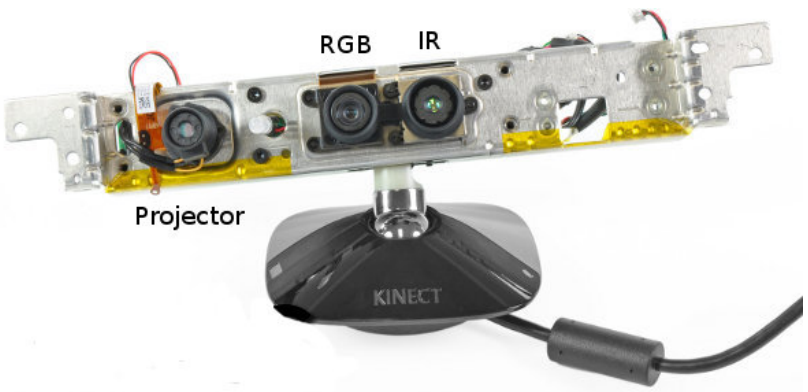
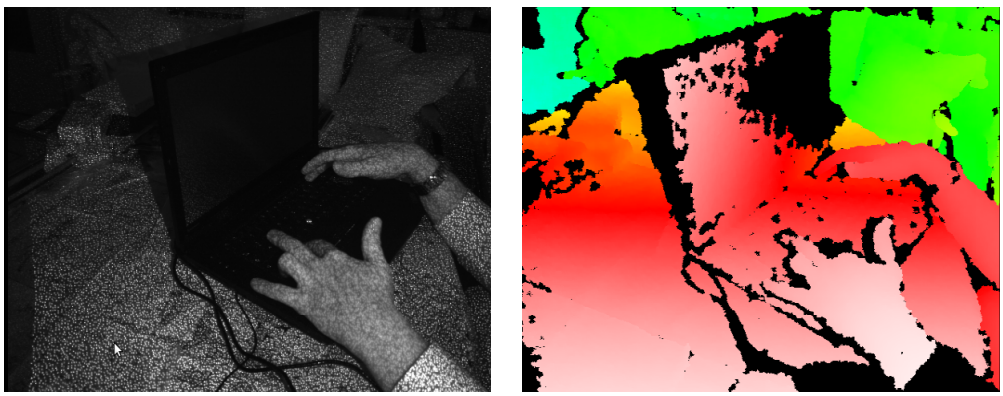



Figure 5.3. Inside of Kinect Sensor.

The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions. Kinect sensor outputs video at a frame rate of 30 Hz. The RGB video stream uses 8-bit VGA resolution (640×480 pixels) with a Bayer color filter, while the monochrome

depth sensing video stream is in VGA resolution (640×480 pixels) with 11-bit depth, which provides 2,048 levels of sensitivity. The Kinect sensor has a practical ranging limit of 1.2–3.5 m distance when used with the Xbox software although the sensor can maintain tracking through an extended range of approximately 0.7–6 m. The sensor has an angular field of view of 57° horizontally and 43° vertically, while the motorized pivot is capable of tilting the sensor up to 27° either up or down. The horizontal field of the Kinect sensor at the minimum viewing distance of 0.8 m is therefore 87 cm, and the vertical field is 63 cm, resulting in a resolution of just over 1.3 mm per pixel.

The IR camera and the IR projector form a stereo pair with a baseline of approximately 7.5 cm. The IR projector sends out a fixed pattern of light and dark speckles. Depth is calculated by triangulation against a known pattern from the projector. The pattern is memorized at a known depth. For a new image, each pixel in the IR image is compared to with the memorized pattern at that pixel and gives an offset from the known depth, in terms of pixels (i.e. disparity). The infrared image in Figure 5.4a shows the laser grid Kinect uses to calculate depth. The depth map shown in Figure 5.4b is visualized using color gradients from white (near) to blue (far).



(a) Infrared Image (b) Depth Image

Figure 5.4. Depth sensing with Kinect sensor.

## 5.3. Sign Recognition Algorithm

We propose a generic template-based approach which can be applied to a wide range of traffic signs and the parallel implementation on a multicore CPU and GPU platform. Our approach uses a new sensor (Kinect) which provides both color and infrared images of the traffic scene, which enhances the detection stage, and we also propose using digital map information to augment template matching in the classification stage in order to increase the robustness of the recognition and to contribute to real-time performance.

In the area of sign recognition with map fusion, the localization and map matching step is generally ignored and assumed as perfect. We provide a complete system, including the map matching and localization. We use a particle filter-based matching and localization algorithm proposed in [43] where GPS (Global Positioning System) and odometer data is fused with the topology of the digital map data as an additional sensor. The algorithm also generates a probabilistic measure for the correctness of the map matching. This measure is taken into account while using the digital map for sign recognition. Figure 5.5 shows the sign recognition fusion system.

The proposed traffic sign recognition algorithm is implemented based on a template matching pipeline. The Kinect camera's depth image output is used to determine candidate regions on the RGB image. Template matching is employed for classification. A special color segmentation or thresholding scheme is applied to candidate regions and the templates in the database to use only four colors (red, blue, black and white). A distance is calculated between the candidate region in the source image and different sizes of template images in the template database based on a difference function. The template having the minimum distance is denoted as the matched or recognized sign. Figure 5.6 summarizes the design of the algorithm.

We preferred a template-based approach, since feature based approaches such as SIFT and SURF, work best when objects have some complexity however traffic signs have simple shapes and constant color regions and do not have any texture in addition

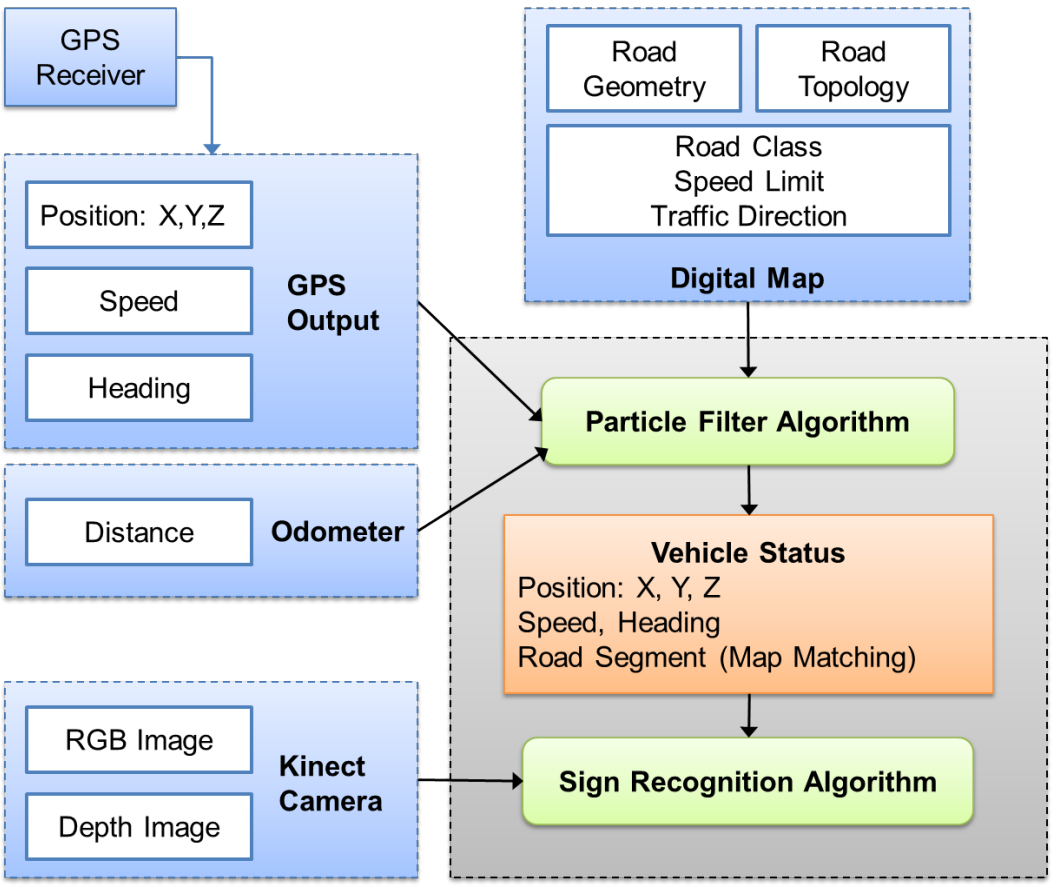



Figure 5.5. Sign Recognition Fusion.

they usually appear small in the camera images. SIFT often cannot extract enough distinct features from these signs and same number of matches would be returned by different templates.

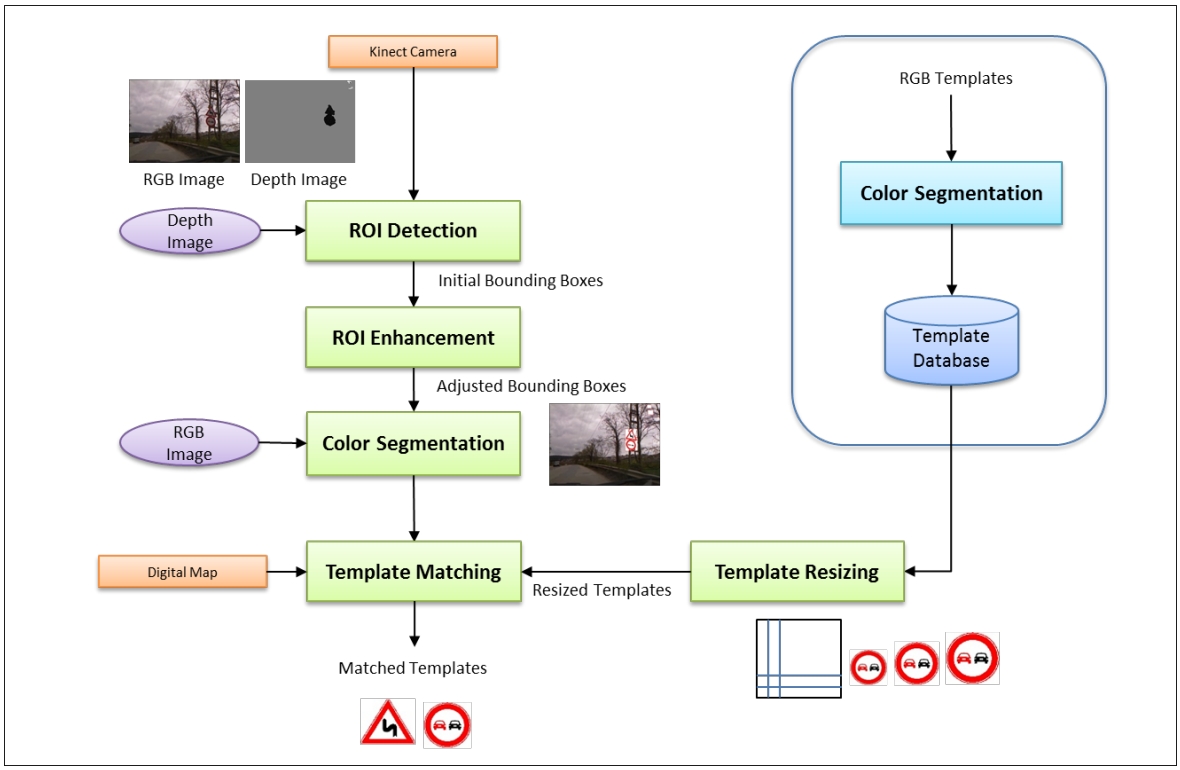



Figure 5.6. Sign Recognition Algorithm.

The following sections describe the stages of the sign recognition flow in detail including the contents of the template database.

### 5.3.1. Template Database and Map-based probabilities

A template database is created from the sign images. Each sign template has two versions, one with a white background, the other with a black background. When sign recognition is carried out under low light conditions including at night, templates with black background are needed. Sample sign templates with white and black bacgrounds are shown in Figure 5.7. Templates are also converted to four colors by use of color segmentation. Only black, white, red and blue are preserved in the image. We used an automatic resizing function by employing bilinear interpolation according to the size

of the region of interest found in the scene.

(a) White background (b) Black background

Figure 5.7. Sample templates with white and black backgrounds.

Some attributes of a digital map database can be used to detect speed limits and candidate traffic signs. Each road segment in a digital road map database is classified into groups called "functional road classes". Each functional road class is associated with a speed limit. Functional road classes and their descriptions in an industry standard digital map database are listed in Table 5.1. An example of functional road classification on a digital map for an area in Istanbul is shown visually in Figure 5.8.

Table 5.1. Functional road classes in a digital map database.

| FRC | Description |
|---|---|
| 1 | Motorway |
| 2 | Major roads of high importance |
| 3 | Other major roads |
| 4 | Secondary roads |
| 5.1 | Local connecting roads |
| 5.2 | Local roads of high importance |
| 6 | Residential connecting roads |
| 7 | Residential roads |
| 8 | Other roads, usually not suitable for vehicle traffic, e.g. stairs, narrow walkways |

The other attributes of a digital map database that can be used to help sign detection include manoeuvres (turn restrictions, permitted, prohibited or priority ma-

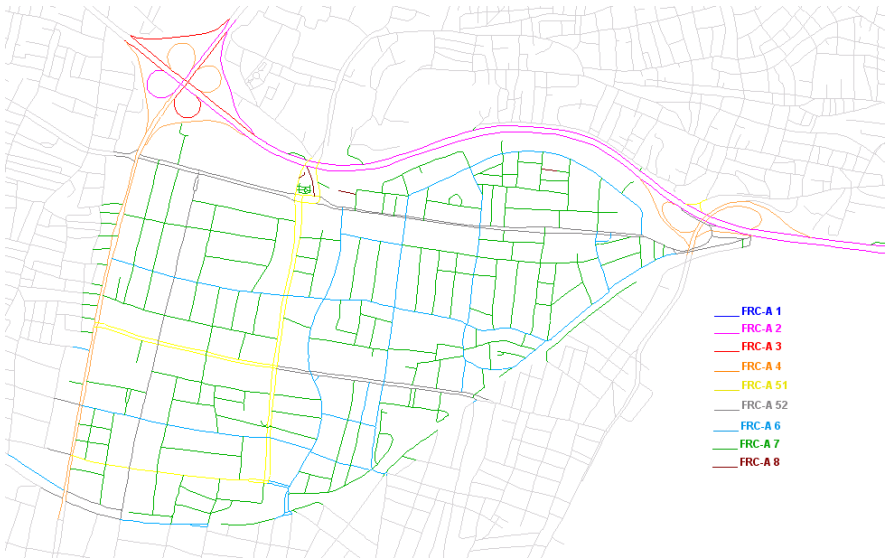



Figure 5.8. Road classification on a digital map.

noeuvres), map topology (geometry of a road segment, road curvature, existence of bends, junctions), and direction of traffic flow (one-way roads). Points of Interest (POI) information also important for detection of some of the road signs like school crossings, hospitals, etc.

The localization and map matching algorithm determines the vehicle location and the map segment. By use of the matched segment, we can calculate map based probabilities for each sign in the database, considering different map contexts for various sign classes. Table 5.2 summarizes the sign classes and their respective map based context.

### 5.3.2. ROI Detection with Kinect Sensor

The Kinect camera's depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions. When used in outdoor environment, we end up with a very effective function of Kinect; it detects reflective surfaces (signs in traffic, most of the time) and thus makes region of interest (ROI) detection very easy and robust. Figure 5.9a and

Table 5.2. Traffic signs and their respective map context.

| Sign Class | Signs | Map context |
|---|---|---|
| Speed Signs |  | Road Class, Speed Limit |
| Manoeuvres |  | Manoeuvres (restrictions), One way information and map topology |
| Bends |  | Map topology, existence of a bend in the driving direction is checked |
| Junctions |  | Map topology, existence of a junction and type of junction is checked |
| School |  | POI, existence of a school is checked |
| Parking |  | Road Class |

5.9b show the RGB image and the depth image coming from Kinect camera for the same scene. The depth image shows the region of interest in the RGB image.

(a) Input RGB Image

(b) Depth Image

Figure 5.9. ROI detection with Kinect sensor.

The image frame captured from Kinect camera contains the pixel depth information. To be able to display the image, these depth values are converted to pixel colors. While creating the range image from depth data, the initial bounding boxes are created by following the neighboring pixels within a given pixel tolerance. After creating initial bounding boxes, overlapping bounding boxes are merged into one box

aiming to create just one box for each sign.

### 5.3.3. ROI Enhancement

As seen in the Figure 5.10a, the initial bounding boxes (green) are not perfect. There are several reasons for this. The IR camera and the RGB camera of the Kinect sensor have different fields of view and focal lengths. The RGB camera has a wider field of view. This is why, when objects get closer to the image edge, the difference in pixel locations increases. The two cameras are separated from each other by 2.5 cm. Sometimes the pixel image coming from the IR camera does not cover the whole sign. Sometimes the two signs are so close (their distance is smaller than the pixel tolerance when calculating the bounding boxes) that only one bounding box is found for two signs. There may be some small bounding boxes caused by reflections coming from other surfaces. An algorithm has been developed to overcome these errors with the following rules:

- Bounding boxes which are too small are deleted.
- A formula to correct difference between RGB and range image is applied. By use of this formula, bounding boxes are moved based on their distances from center of the image.
- Bounding boxes are either split or enlarged based on their aspect ratios.
- Some buffer is added to cover missing pixels in the sign edges.

The initial and enhanced ROIs are shown in Figure 5.10b in green and yellow rectangles, respectively.

### 5.3.4. Color Segmentation

For a better matching, color segmentation is applied to the region of interests first. All the colors in the image are segmented in four colors: red, blue, white and black. An example of color segmentation can be seen in Figure 5.11.

(a) Initial ROIs (b) Enhanced ROIs

Figure 5.10. ROI Enhancement.

Figure 5.11. Color Segmentation.

### 5.3.5. Template Matching

After color segmentation, the templates are matched against the region of interests by computing the sum of the differences between pixel color values. For each region of interest, templates are resized based on the size of the bounding box before matching. Several template sizes with different aspect ratios are tried. Starting from corner of the region of interest, the difference between the template and the region of interest is calculated. The difference value is normalized according to the template size. The template with the lowest difference value is selected as the match.

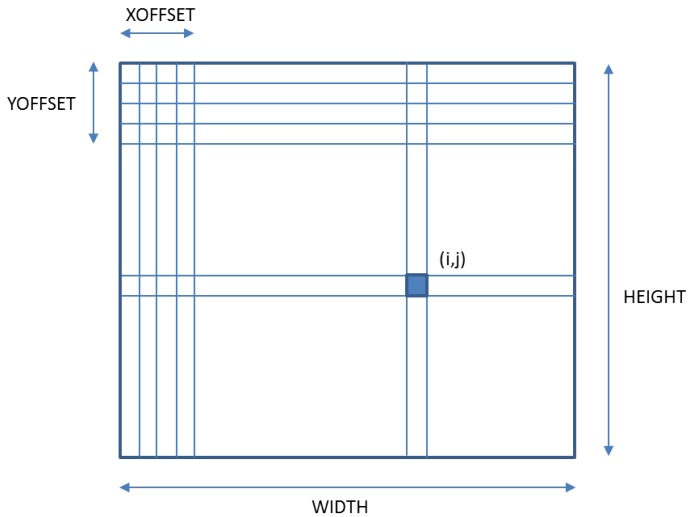


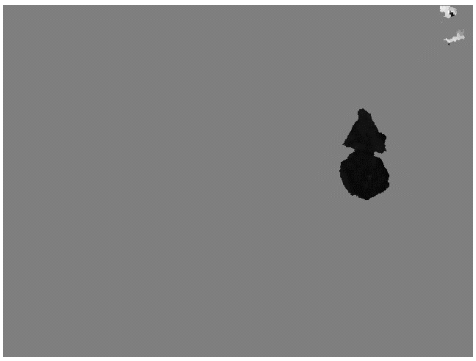

Figure 5.12. Sum of Differences Computation.

Figure 5.13 shows a successful matching example with input RGB image, depth image, color segmented region of interests and recognized templates.

### 5.3.6. Map Fusion

Template matching generates a likelihood measure for each sign. This measure is the distance between the template image and the camera image. Since we successfully detect the location of the sign on the camera image, the sign with the lowest distance value can be selected as the matched sign most of the time. But some of the signs are very similar to each other. Also, even if we find the location of the sign successfully,

(a) Input RGB Image



(b) Depth Image

(c) Color Segmented ROIs

(d) Recognized Templates

Figure 5.13. Successful matching example.

the camera image may not be clear. As a result of this, the algorithm returns very close likelihood results. When we fuse this information with the probabilities coming from the map, the correct sign can be selected. The recognition performance of our algorithm increases radically. Table 5.3 shows two examples of template matching results, with and without map fusion.

Table 5.3. Sign recognition with map fusion.

| Image | Template Matching without Map Fusion | | Template Matching with Map Fusion | |
|---|---|---|---|---|
|  |  |  |  |  |
| | 0.70% | 0.67% | 0.52% | 0.90% |
|  |  |  |  |  |
| | 0.78% | 0.77% | 0.66% | 0.97% |

## 5.4. Parallel Implementations

The execution time profile of the sequential implementation of sign recognition shows that the template matching process has the highest computational cost by more than 98 percent of the total execution time. This has been chosen as the target for parallelization.

Although the number of templates to be matched are decreased in some cases by augmenting map information, the matching process for each video frame involves the following parameters:

- $r$ number of ROIs detected in the frame
- $n$ number of templates in the template database
- $m$ number of different sizes for each template to be used for matching
- $s$ number of different starting positions for matching in each ROI
- $w$ width of template in pixels
- $h$ height of template in pixels

Assuming ($x$,$y$) denotes the starting search image coordinates and ($i$,$j$) denotes the template image coordinates, the time required for the matching process for each frame can be defined as the following:

$$t = r \times n \times m \times s \times \sum_{i=0}^{h} \sum_{j=0}^{w} Diff(x+i, y+j, i, j) \tag{5.1}$$

This process must be handled in real time in vehicle environment, this means that it must be completed within nearly 100 ms for a video sequence at a rate of 10 frames per second. This time requirement might be shorter in the case that there are other applications in the vehicle like localization and map matching to run simultaneously using the same resources. This can only be achieved by a parallel implementation.

Three parallel implementations have been developed for multicore CPU, single GPU and multi GPU architectures. For all cases, the detection stage is performed on the host sequentially, which can be performed very fast with the help of the Kinect camera. Details of each parallel implementation are described in the following sections.

### 5.4.1. Multicore (OpenMP) Implementation

The multicore CPU implementation is performed using the OpenMP programming model. The matching operations for each template are distributed among the multiple CPU threads. The number of threads is determined by the maximum number of cores in the system. For each region of interest, the work is distributed on a tem-

plates basis. Matching operations for each template are handled in parallel by different CPU threads. The `for` construct is used to parallize the loop. The `critical` OpenMP synchronization construct is used for updating global sum of differences (SAD) values. The `critical` directive specifies a region of code that must be executed by only one thread at a time. The pseudo code for multicore CPU implementation is shown in Figure 5.16.

### 5.4.2. GPU (CUDA) Implementation

Data parallel nature of template matching is suitable for GPU computing. Pixel level difference calculations for a template are performed in parallel on the device. Furthermore, depending on the size of the region of interests, multiple templates are matched in parallel on the device by employing concurrent kernels. This is actually a hybrid implementation employing both multicore CPU and GPU.

The GPU implementation is performed using CUDA. The pixel level matching operations are designed to run on GPU in parallel. A GPU kernel (*matching* kernel) has been implemented to perform the matching of a region of interest to a resized template and produce the sum of differences (SAD) values. A separate GPU thread is created for each pixel operation when the kernel is launched. Initially, all memory allocations are done for RGB images, resized templates and SAD values on both host and device. For each video frame, detection is performed on the host and region of interests are found. If at least one region of interest is found in the video frame, the RGB image is copied to the device memory. Each region of interest found in the frame is matched against different sizes and starting positions of all the templates by calling the *matching* kernel. Resizing is done on the host, each template is resized based on the size of region of interest and resized templates are copied to the device memory before launching the *matching* kernel. Since the RGB image and the resized templates are already in the device, the kernel is then called with only the corner positions of the region of interest, the template number and the size of the template. The flow of the implementation is shown in Figure 5.15.

```
// Loop for each sign location
for r = 1 → numROIs do
  // Set the number of threads
  omp_set_num_threads(NUM_THREADS)
  #pragma omp parallel for private(tempNo)
  for tempNo = 1 → numTemplates do
    // Try different sizes
    for z = 0 → NUM_ZCHANGE do
      for v = 0 → NUM_VCHANGE do
        resizeTemplate(tempno, bb.minx, bb.miny, bb.maxx, bb.maxy, z, v);
        // Try different starting positions
        // Loop through the search image within defined bbox
        for y = bb.miny → bb.miny + NUM_YOFFSET do
          for x = bb.minx → bb.minx + NUM_XOFFSET do
            // Loop through the template image
            // compare pixel color values and apply penalties
            SAD ⇐ doMatching();
          end for
        end for
      end for
    end for
    #pragma omp critical
    if SAD < minSAD then
      minSAD ⇐ SAD;
      matchedSign ⇐ tempno;
    end if
  end for
end for
```

Figure 5.14. Pseudo code for multicore CPU Implementation of Template Matching Algorithm.

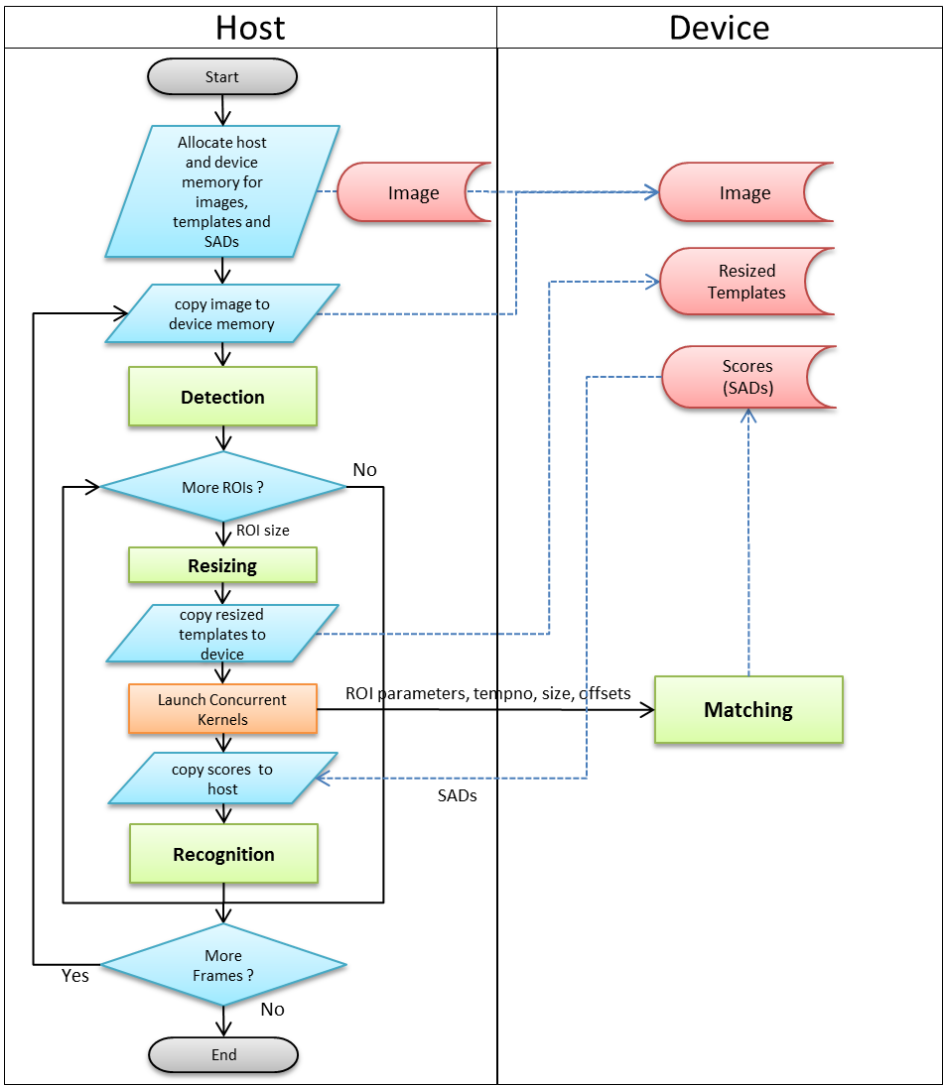



Figure 5.15. CUDA Implementation of Sign Recognition Algorithm.

Since the number of pixels in region of interests are relatively small (e.g. $49 \times 48$ (2352 pixels), $69 \times 50$ (3450 pixels), $44 \times 40$ (1760 pixels)) compared to whole images ($640 \times 480$), to be able to achieve maximum occupancy of GPU cores, the *matching* kernel is designed to compute SAD values for all different starting positions ($4 \times 5$) of a resized template each time it is launched. So each launch of the *matching* kernel performs 20 matching operations in parallel at the template level in addition to the pixel level parallelism (e.g., for a $44 \times 40$ pixels region of interest, 35200 threads are created instead of 1760, corresponding to 138 blocks instead of 7 blocks, respectively). The pseudo code for GPU implementation of matching is shown in Figure 5.16.

To further increase the efficiency, kernels for each different size of the same template are launched concurrently using different streams. Concurrent kernels is a scheduling convenience allowing different streams of the same context to run simultaneously. It enables to increase the efficiency if there are inefficient low block count kernels, mostly by reducing idle streaming multiprocessor count while kernels are finishing up. The maximum number of concurrent kernels that can be executed on a Fermi GPU is 16. The number of different sizes ($4 \times 4$) to be matched for each template is also 16 in our implementation. This enables the matching of all different sizes of a template to be launched concurrently.

SAD values are accumulated in the global memory by using AtomicAdd operations. For each template, after calling the kernels for all variations, the SAD values are copied back from the device to host, and for each region of interest, the SAD values are processed to determine the result of recognition. The pseudo code for implementation of *matching* kernel is shown in Figure 5.17.

### 5.4.3. Multi-GPU Implementation

In order to increase the frame rate, a multi GPU solution has also been implemented and tested on two GPUs. This is also a hybrid implementation employing both multicore CPU and GPUs and can be used with any number of GPUs.

```
// Loop for each sign location
m ⇐ NUM_ZCHANGE × NUM_VCHANGE;
s ⇐ NUM_XOFFSET × NUM_YOFFSET;
for r = 1 → numROIs do
  colorSegmentation(vgaImage, bb.minx, bb.miny, bb.height, bb.width);
  resizeTemplates(bb.width, bb.height);
  copyDatatoDevice(vgaImage, resizedTemplates);
  for tempNo = 1 → numTemplates do
    Adjust penalties based on background color;
    Copy Initial SAD values to device;
    // Loop for each size variation
    for i = 0 → m do
      z ⇐ i % NUM_ZCHANGE;
      v ⇐ i / NUM_ZCHANGE;
      grid((s × (width − z) × (height − v) / BLOCK_SIZE), 1, 1);
      threads(BLOCK_SIZE);
      MatchingKernel <<< grid, threads, i >>> (tempNo, z, v, minx, ...)
    end for
    cudaThreadSynchronize();
    Copy final SAD values from device;
    for i = 0 → s × m do
      if SAD[i] < minSAD then
        minSAD ⇐ SAD[i];
        matchedSign ⇐ tempNo;
      end if
    end for
  end for
  return matchedSign;
end for
```

Figure 5.16. Pseudo code for GPU Implementation of Matching.

__**global**__ MatchingKernel (tempno, vgaImage, resizedTemplates, minx, miny, height, width, SAD, z, v, whitePenalty, blackPenalty)

$tid \Leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`;

// Find pixel locations for vga image and template image based on thread Id

$pixX \Leftarrow tid \,\%\, width$;

$pixY \Leftarrow (tid \,/\, width) \,\%\, height$;

$x \Leftarrow ((tid \,/\, width) \,/\, height) \,\%\, NUM\_XOFFSET$;

$y \Leftarrow (((tid \,/\, width) \,/\, height) \,/\, NUM\_XOFFSET)$;

$tnx \Leftarrow (z \times NUM\_VCHANGE) + v$;

$inx \Leftarrow (((((x \times NUM\_YOFFSET) + y) \times NUM\_ZCHANGE) + z) \times NUM\_VCHANGE) + v$;

// Locate pointers to pixel locations on vga and resized template image

// Get pixel values in Red, Green and Blue for both vga and template images

// Make comparisons and apply penalties accordingly

// If colors do not match increase difference

$SAD \Leftarrow 0$;

**if** *color* <> *template color* **then**

  $SAD += defaultPenalty$; // 20

**else if** $color = white$ **then**

  $SAD += whitePenalty$; // 10 for white background, 3 for black background

**else if** $color = black$ **then**

  $SAD += blackPenalty$; // 3 for white background, 10 for black background

**else if** $color = red$ **then**

  $SAD += redPenalty$; // 2

**else if** $color = blue$ **then**

  $SAD += bluePenalty$; // 1

**end if**

// Add partial SAD to global SAD for this particular matching

**atomicAdd**$(SAD[inx], SAD/(width \times height))$;

Figure 5.17. Pseudo code for GPU Implementation of Matching Kernel.

Five CPU threads are used in the implementation. The *detection* thread gets the depth, and RGB image frames, perform the detection phase, for each region of interest found in the depth image, resizes the templates based on the size of region of interest and puts the related data including region of interest boundary, a reference to resized templates into a queue to be passed to a GPU to perform the matching. The *dispatcher* thread keeps track of the availability of GPUs and determines the target GPU that will process the next region of interest data and assigns the device number to the data slot in the queue. Each GPU has to be controlled by one CPU thread in multi GPU programming with CUDA Toolkit 3.2. Two *matching* threads are responsible for controlling the GPUs, including sending the required data (i.e. RGB image, ROI boundary, resized templates) to the device, launching the *matching* kernels concurrently, receiving the SAD values from the device and storing them into the results queue. The *recognition* thread processes the SAD values and determines the sign recognized for each region of interest. The implementation details are depicted in Figure 5.18.

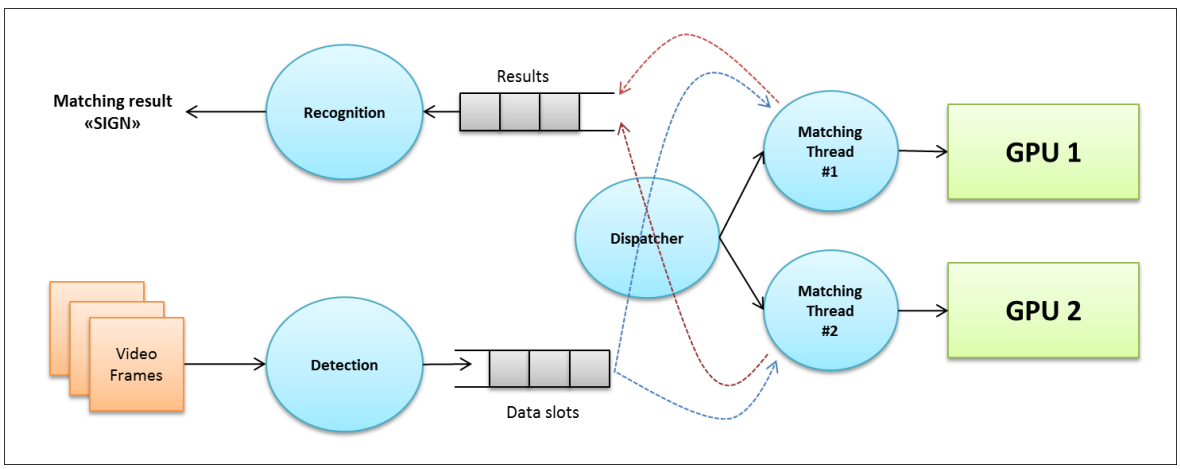



Figure 5.18. Multi-GPU Implementation of Sign Recognition Algorithm.

# 6. EXPERIMENTAL RESULTS

## 6.1. Test Platform

Performance of parallel implementations have been tested using real video, GPS and odometer data captured by a test vehicle equipped with a Kinect camera, GPS and odometer in test routes comprising various road (highways, urban traffic, etc.) and lighting conditions (night/day, sunny/cloudy). Parallelization tests were performed on our test platform, a dual processor HP® Z800 workstation having two Intel® Xeon® 5660 6-core processors running at 2.80 GHz and two NVIDIA® GeForce GTX580 graphics processing units. The operating system is 64-bit Microsoft® Windows® 7 Professional Edition. Test platform is shown in Figure 6.1.

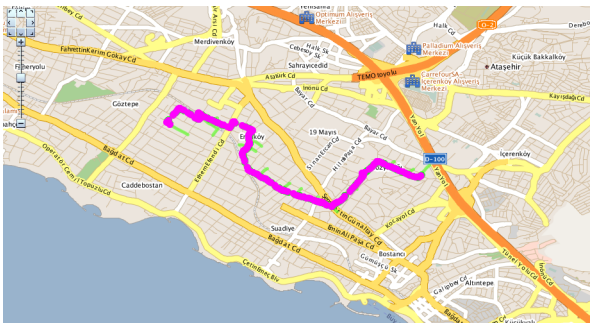
The GTX580 GPU has NVIDIA's new generation CUDA compute architecture called Fermi and has 16 streaming multiprocessors, each having 32 streaming processors or processing cores, and thus in total has 512 processing cores. Hence, it is capable of running 512 threads simultaneously. Each core runs at 1.544GHz. Each streaming multiprocessor has 64KB configurable L1 cache. All cores shares a 768MB L2 unified cache and a 1512MB global memory.

(a) HP Z800 Workstation (b) NVIDIA GTX580

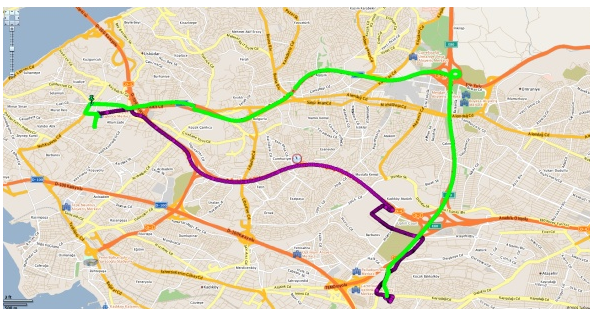
Figure 6.1. Test Platform.

## 6.2. Localization and Map Matching

Performance of parallel implementations have been tested using real GPS and odometer data captured in three different test routes comprising various speed and road conditions (highways, urban traffic, etc.). Figure 6.2a and Figure 6.2b show test routes.



(a) Test route (urban traffic)



(b) Test route (highways)

Figure 6.2. Test routes for localization and map matching.

One six-core CPU and one 512-core GPU were used in the tests. Tests were repeated on each platform for different number of particles ranging from 256 to 128K. For the multicore CPU tests, we ran the OpenMP implementation with 6 threads. For the CPU+GPU tests, the block size (the number of threads in each thread block) was chosen as 256 to keep the multiprocessor occupancy as high as possible (Occupancy is defined as the ratio of the active warps to the maximum number of warps supported on a multiprocessor of a GPU).

Execution times for each test were measured by running the algorithm for the

same test route with around 200 iterations and taking the average elapsed times for each section of particle filter. Single threaded results on the CPU were taken as the baseline for all speedup calculations.

The OpenMP implementation provided approximately a 4.7x speedup with a theoretical maximum increase of 5.4x on a 6-core CPU based on the Amdahl's law since the parallelized sections correspond to 98% of the total execution time. This is mostly due to the parallel overhead of OpenMP such as the time to create, start, and stop threads, the time spent in barriers, and assigning work to threads, etc. We observed similar speedups after the number of particles exceeds 4096.

With the CUDA implementation, we achieved increasing speedups of up to 75x when the number of particles reached 128K. We see that the performance of GPU is better exploited when the number of particles or threads is increased. The relatively low speedups for the smaller number of particles are mainly due to the low occupancy of streaming multiprocessors. Figure 6.3 shows the execution times of sequential, multicore CPU and GPU implementations for different number of particles and Figure 6.4 shows the relative speedups.

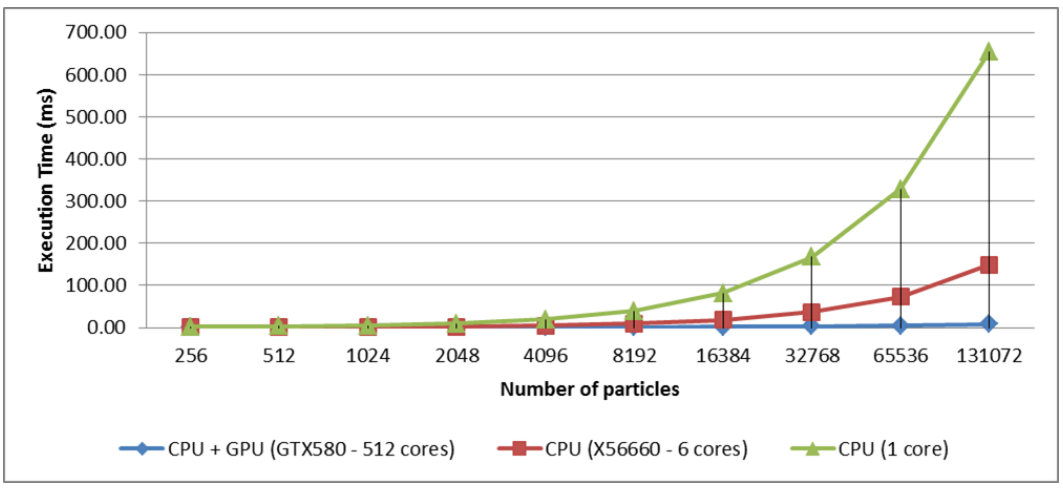



Figure 6.3. Execution time comparison of sequential and parallel implementations for localization and map matching on the multicore CPU and the GPU.

When we examine performance of kernels separately, we see that speedups can

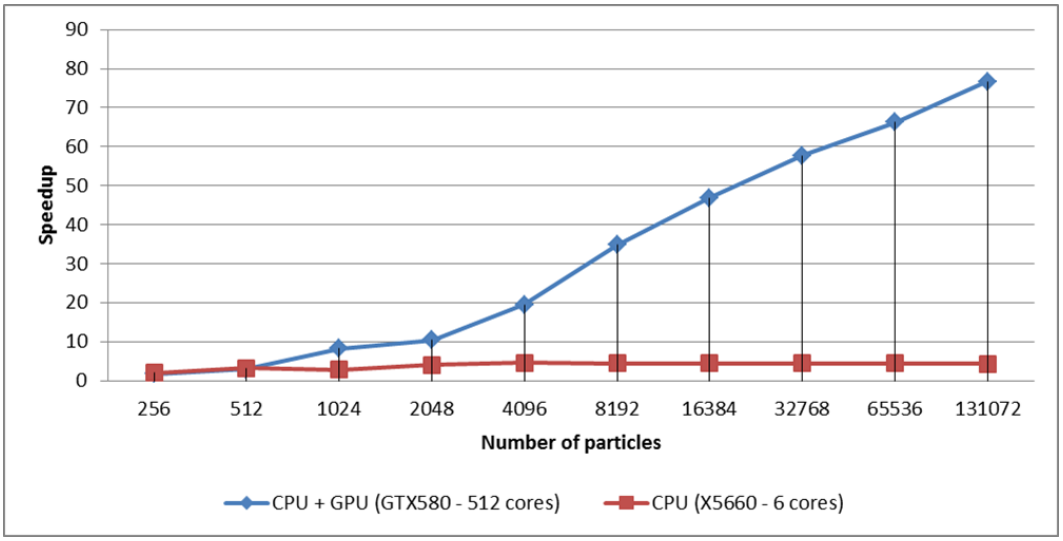



Figure 6.4. Speedup comparison of parallel implementations for localization and map matching on the multicore CPU and the GPU.

be as high as 150x for the *predict* kernel, where there are no data dependencies among threads and operations performed are almost identical for all threads. We see 100x average speedups for the *update* kernel, where we observe the negative effect of branching and divergence on the kernel execution performance since road network is traversed to a new location for some particles which causes different execution paths for threads within the same warp and different execution paths are serialized. We see speedups around 10x for the *estimation* and *computeESS* kernels, where synchronization requirements within blocks and global atomic operations reduce speedups. The sequential implementation of resampling on the CPU is also a speedup limiting factor. However, overall speedups achieved are sufficient for real-time localization and map matching using a high number of particles.

Another point is that the performance of a CPU core where we take the sequential execution time as the baseline for the speedup calculations is generally higher than the performance of a single processing unit of the GPU.

As the results show, CUDA implementation is much faster than the OpenMP implementation for large number of particles. This is mainly due to the computational

power of the GPU architecture in terms of number of cores and speed of floating point operations and the suitability of data-parallel nature of particle filter algorithm to this platform.

We also examined the sensitivity of the localization and map matching performance to the number of particles employed by the particle filter to determine the optimum number of particles to be used on such a platform. The error rate is calculated as the ratio of the number of wrong map matches to the total number of positions on the test routes. Tests show that the error rate decreases significantly until the number of particles exceeds 32K. Figure 6.5 shows the error rates for two different routes.

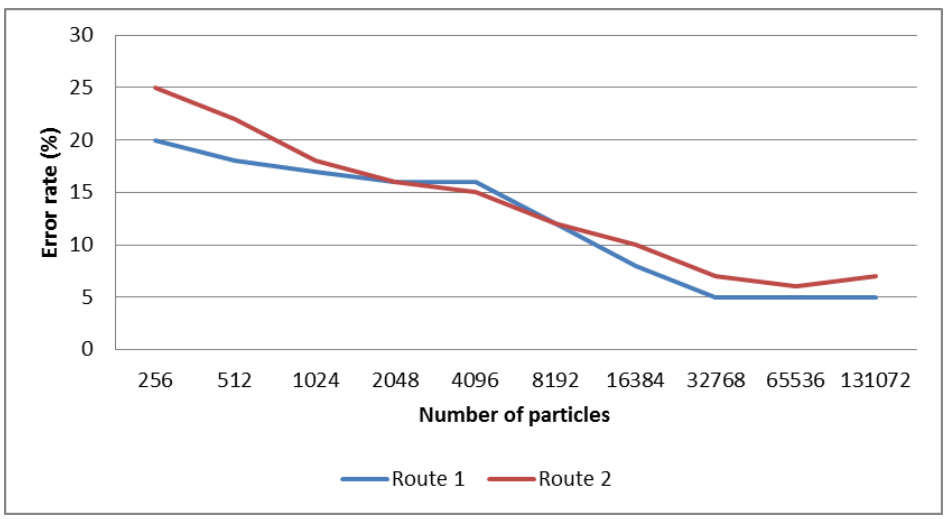



Figure 6.5. Effect of number of particles on the error rate of map matching algorithm.

Finally, we used a map data containing nearly 20,000 road segments in our tests which could be copied to the device memory at once at the initialization. For bigger map data which cannot be kept in device memory as a whole, a double buffering mechanism can be used and the map data can be partially copied to the device memory concurrently with kernel executions as the vehicle travels.

## 6.3. Sign Recognition

To be able to test sign recognition performance of the system, different types of routes have been tried and extensive test data were collected. Following environmental conditions have been tested:

- Route types: Main Roads, rural roads, urban roads.
- Lighting conditions: Night/Day, Sunny/Cloudy.

Figure 6.6 shows examples of the environmental conditions tested. Video over 90 minutes under various environmental conditions have been recorded with Kinect camera by using Mobile Robot Programming Toolkit's (MRPT) log collection utilities [120]. Both vga video and the range video sequences have been then converted to image frames which were used in testing by using MRPT utilities.

(a) Sunny, Residential Connecting Roads

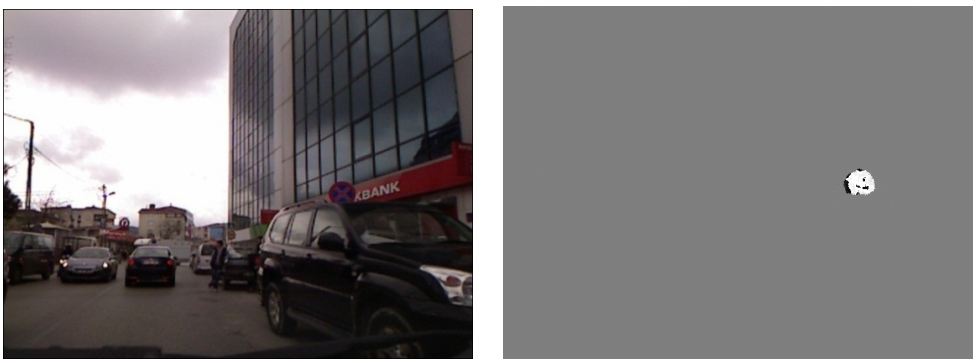
(b) Cloudy, Rural Roads

(c) Night, Main Roads

(d) Night, Residential Streets

Figure 6.6. Test routes and conditions.

Since detection of region of interests are handled by the the Kinect camera, detection is successful even in very bad lighting conditions. We have observed that the system can detect signs that can hardly be seen by human eye. Table 6.1 summarizes success rates of detection and recognition for different route types. We see that map fusion improves recognition performance dramatically especially under night conditions. Figure 6.7 shows an example of a successful recognition with partial occlusion. Figure 6.8 shows an example of a successful recognition with very close signs in cloudy weather. Figure 6.9 and Figure 6.10 show examples of successful recognition at night. Each figure shows original RGB image, depth image, color segmented region of interests on the RGB image and the templates selected at the end of matching process for each case.



(a) Input RGB Image

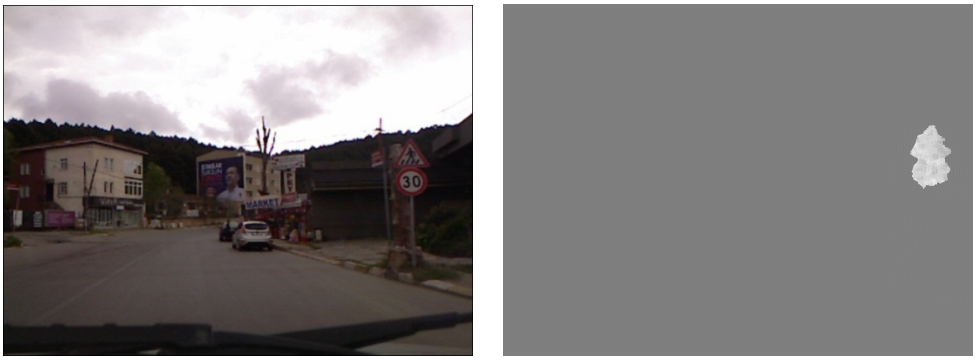
(b) Depth Image

(c) Color Segmented ROIs

(d) Recognized Templates

Figure 6.7. Successful matching with partial occlusion.

Multicore CPU and GPU implementations were tested on the same test platform having two 6-core CPUs (Intel® Xeon X5660) (12 cores in total) and having two 512-



(a) Input RGB Image

(b) Depth Image

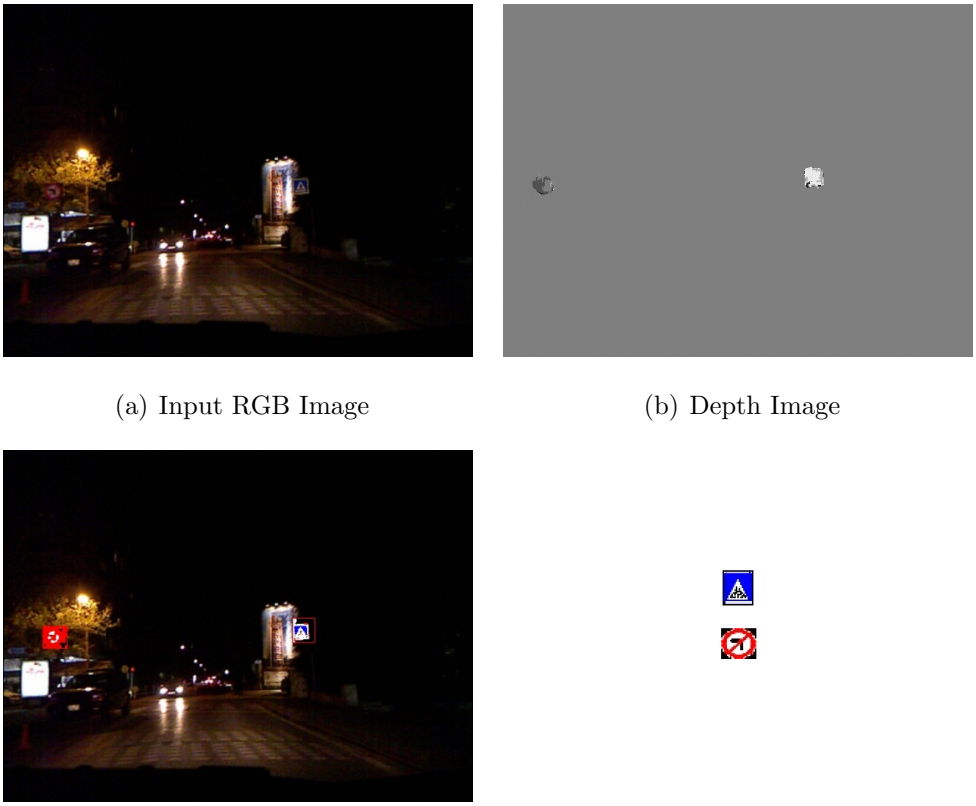
(c) Color Segmented ROIs

(d) Recognized Templates

Figure 6.8. Successful suburban with cloudy weather. Two very close signs.

Table 6.1. Detection rates for traffic signs using Kinect camera.

| Route no | Route type | Detection rate |
|---|---|---|
| 1 | Cloudy, Residential Roads - Urban | 93% |
| 2 | Sunny, Residential Roads - Urban | 89% |
| 3 | Cloudy, Main Roads | 91% |
| 4 | Cloudy, Connecting Roads- Rural | 95% |
| 5 | Night, Main Roads | 94% |
| 6 | Night, Residential Roads | 92% |



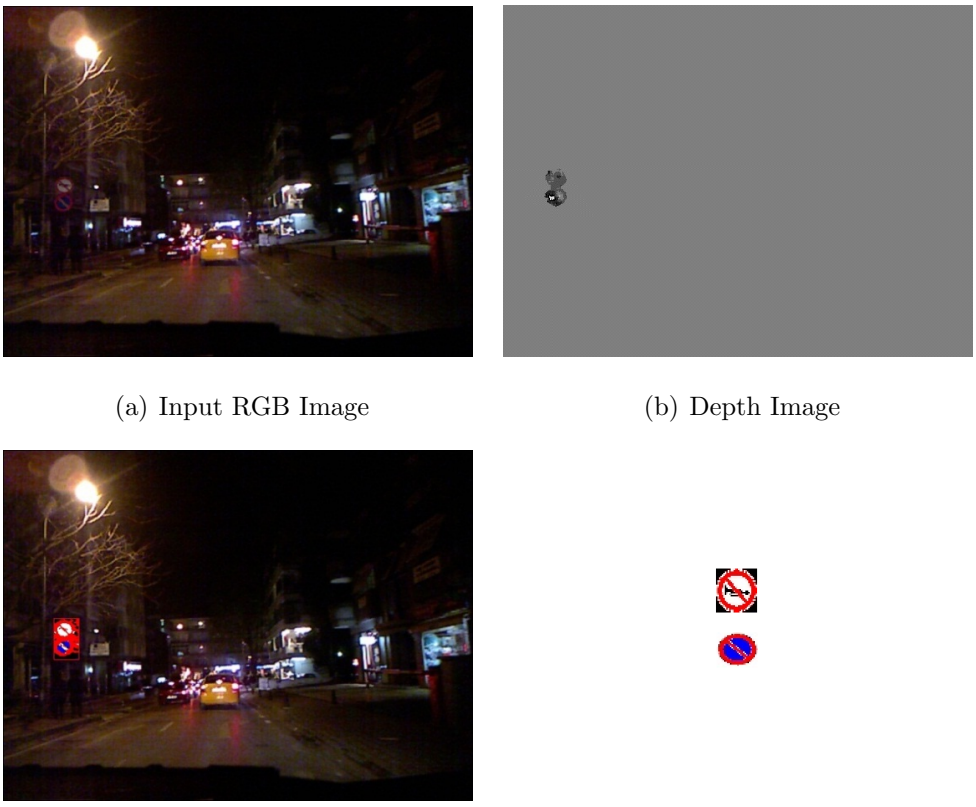
(a) Input RGB Image (b) Depth Image

(c) Color Segmented ROIs (d) Recognized Templates

Figure 6.9. Successful recognition at night conditions.

Table 6.2. Recognition rates for traffic signs using Kinect camera.

| Route no | Route type | Recognition without map fusion | Recognition with map fusion | Improvement |
|---|---|---|---|---|
| 1 | Cloudy, Residential Roads - Urban | 84% | 92% | 9% |
| 2 | Sunny, Residential Roads - Urban | 71% | 85% | 20% |
| 3 | Cloudy, Main Roads | 71% | 86% | 20% |
| 4 | Cloudy, Connecting Roads- Rural | 50% | 83% | 66% |
| 5 | Night, Main Roads | 55% | 88% | 60% |
| 6 | Night, Residential Roads | 40% | 80% | 100% |



(a) Input RGB Image

(b) Depth Image

(c) Color Segmented ROIs

(d) Recognized Templates

Figure 6.10. Successful recognition at night conditions.

core GPUs (NVIDIA® GeForce GTX 580) (1024 cores in total). Real video data captured in vehicle environment has been used for the tests. The average processing time for frames was measured and the execution time of sequential implementation was taken as a reference in the speedup calculations. For each region of interest, 16 (4x4) different starting positions, and for each template, 20 (4x5) different sizes, were used. Tests were performed with a template database having 52 templates. The recognition of each region of interest involved 16,640 matching operations which are the combination of number of starting positions, number of different sizes and number of templates.

Multicore CPU implementation were tested with different numbers of threads, ranging from 1 to 24. Speedups of up to 10.6x were achieved for the multicore CPU implementation. We observed linearly increasing speedups until the number of threads reached the number of cores in the system which is 12. After that point, we observed that the speedups were not improved with the increasing number of threads, but rather stayed in the range between 8.7 and 9.7. The execution time at the maximum speedup was around 250ms corresponding to 4 frames per second. However, the linear speedups show that we can further increase frame rates when we have a higher number of cores in the system.

Speedups up to 18.1x and 35.2x were achieved on a single GPU and multi GPU platforms, respectively. The execution times at the maximum speedups approximately correspond to 7 and 13 frames per second. We have seen that real-time performance is possible up to 13 frames per second by employing multi GPUs. For GPU tests, we used 256 threads as the block size. We observed that 100% occupancy was achieved. We observed that the 88.75% of total GPU time spent on each GPU is for kernel computation and approximately 11.25% is for memory copies, mostly from host to device, as shown in GPU time summary plot in Figure 6.11.

Execution times and speedups for all implementations are shown in Figure 6.13 and Figure 6.12, respectively.

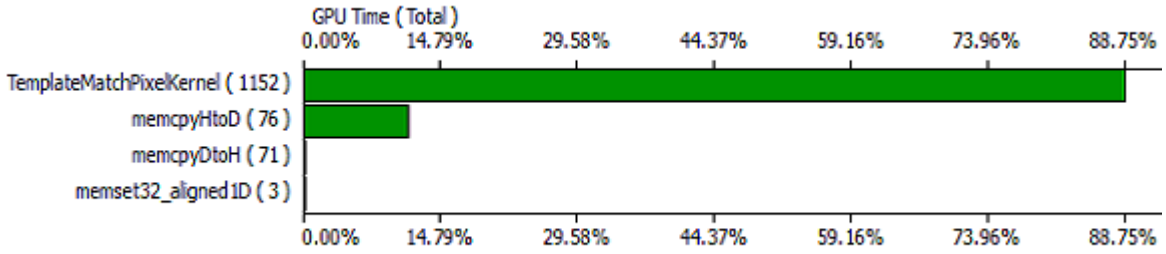



Figure 6.11. GPU time summary plot.

## 6.4. CUDA Optimizations

Getting the best performance from CUDA architectures generally requires a number of optimizations starting from high priority optimizations that present substantial improvements thru lower priority optimizations [121]. This section summarizes the optimizations performed throughout the CUDA implementations in this study.

Memory optimizations are the most important area for performance. The goal is to maximize the use of the hardware by maximizing bandwidth. Bandwidth is best served by using as much fast memory and as little slow-access memory as possible. The peak bandwidth between the device memory and the GPU is much higher than the peak bandwidth between host memory and device memory. Hence, for best overall application performance, it is important to minimize data transfer between the host and the device [121].

For the particle filter implementation, all the particle information and the random number generators are kept in the device memory and only the sensor measuements and estimation results are transferred between the host and the device throughout the iterations. When resampling is required, only the current particle weights are transferred to the host and surviving particle indexes are transferred back to the device. Data have been kept on the device as long as possible. For the traffic sign recognition implementation, only the region of interests found in each frame are transferred to the

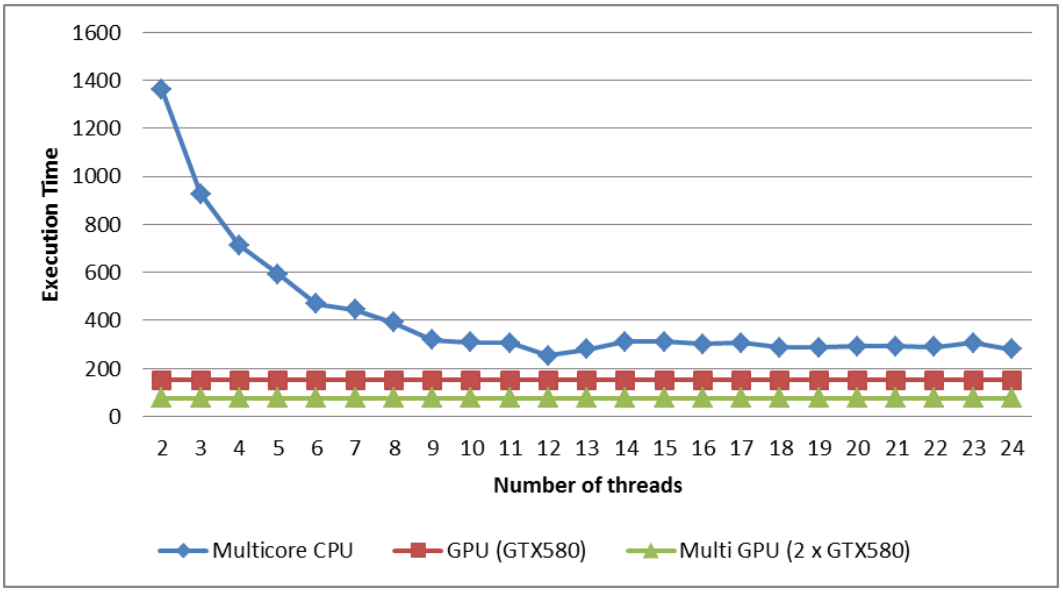



Figure 6.12. Execution time comparison of sequential and parallel implementations for sign recognition on the multicore CPU and the GPU.

device rather than the whole frame to minimize the data transfers between the host and the device.

Because of the overhead associated with each transfer, batching many small transfers into one larger transfer performs significantly better than making each transfer separately. For the particle filter implementation, the estimation results for various state variables are stored into a data structure and this data structure is transferred to the host instead of transferring each variable separately.

Higher bandwidth between the host and the device is achieved when using *page-locked* (or *pinned*) memory. Pinned memory is memory allocated using the *cudaMallocHost* function, which prevents the memory by being swapped out and provides improved transfer speeds. The image frames are stored in pinned memory for the traffic sign recognition implementation to improve transfer speeds.

CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include

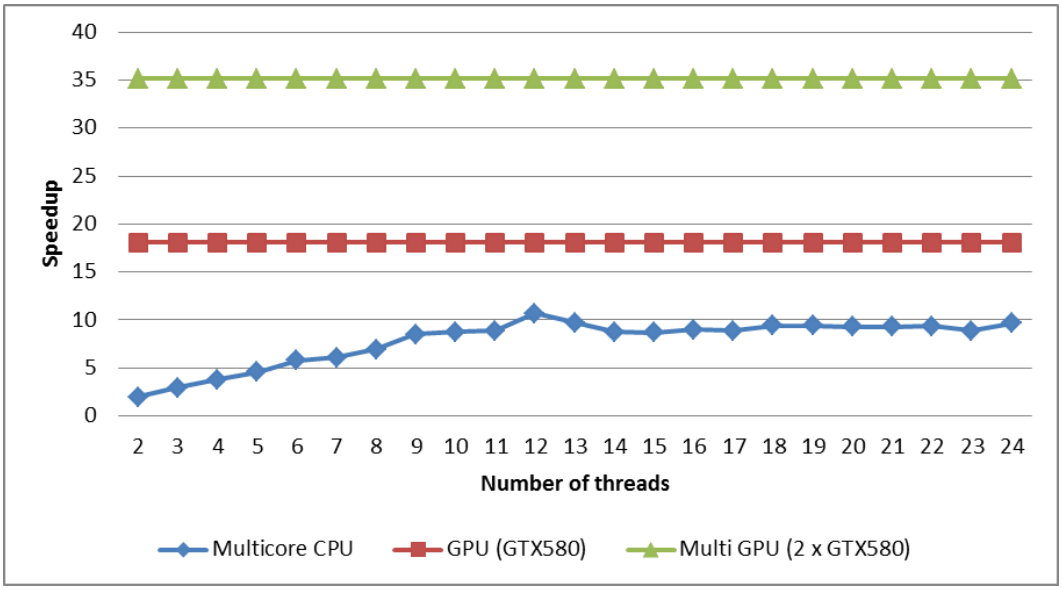



Figure 6.13. Speedup comparison of parallel implementations for sign recognition on the multicore CPU and the GPU.

global, local, shared, texture, and registers. Of these different memory spaces, global and texture memory are the most plentiful. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory.

Because it is on-chip, shared memory is much faster than local and global memory. In fact, uncached shared memory latency is roughly 100x lower than global memory latency—provided there are no bank conflicts between the threads. It is recommended to use shared memory to avoid redundant transfers from global memory. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.

For all CUDA implementations, shared memory has been used as much as possible. Especially, the use of shared memory is exploited in the parallel reduction used for summation kernel of the particle filter implementation. The partial sums are calculated in shared memory and accumulated to the global sum stored in the global memory.

One of the most important performance considerations in programming for the CUDA architecture is coalescing global memory accesses. Global memory loads and

stores by threads of a warp (for devices of compute capability 2.x) are coalesced by the device into as few as one transaction when certain access requirements are met. The particles are kept in sequence in the global memory for the particle filter implementation. The sequential addressing is used in parallel reduction for the implementation of summation kernel. This way, it was ensured that global memory accesses are coalesced into minimum number of transactions.

Another key to good performance is to keep the multiprocessors on the device as busy as possible and to make sure the CUDA application is exploiting all the available resources on the GPU. Thread instructions are executed sequentially in CUDA, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metric related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is *occupancy*. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [121]. Maximum number of possible active warps per SM for Fermi architecture is 48 (1536 threads).

One of several factors that determine occupancy is register availability. Register storage enables threads to keep local variables nearby for low-latency access. However, the set of registers (known as the register file) is a limited commodity that all threads resident on a multiprocessor must share. Registers are allocated to an entire block all at once. So, if each thread block uses many registers, the number of thread blocks that can be resident on a multiprocessor is reduced, thereby lowering the occupancy of the multiprocessor [121].

For purposes of calculating occupancy, the number of registers used by each thread is one of the key factors. For example, devices with compute capability 2.x have 32K 32-bit registers per multiprocessor and can have a maximum of 1536 simultaneous threads resident (48 warps $\times$ 32 threads per warp). This means that in one of these devices, for a multiprocessor to have 100% occupancy, each thread can use at most 20 registers. For the particle filter and sign recognition implementations, the number of

registers to store local variables for each kernel was chosen carefully not to reduce the occupancy. 100% occupancy is achieved.

Device memory allocation and de-allocation via *cudaMalloc* and *cudaFree* are expensive operations, so device memory should be reused and/or sub-allocated by the application wherever possible to minimize the impact of allocations on overall performance. For both particle filter and sign recognition implementations, device memory allocations have been made at the beginning of computations and the allocated memory areas have been reused throughout the iterations without requiring further allocations/deallocations.

The only way to utilize all multiprocessors in a device of compute capability 1.x is to launch a single kernel with at least as many thread blocks as there are multiprocessors in the device. Applications have more flexibility on devices of compute capability 2.x, since these devices can execute multiple kernels concurrently and therefore allow applications to also fill the device with several smaller kernel launches as opposed to a single larger one. This is done using CUDA streams.

On Fermi hardware it is best to interleave kernel launches to multiple streams rather than to launch all kernels to one stream, then the next stream, etc. This is because the hardware can immediately launch kernels to different streams if there are sufficient resources, whereas if subsequent launches are to the same stream there is often delay introduced, reducing concurrency.

For the sign recognition implementation, to maximize parallel execution at a higher level, concurrent execution on the device is explicitly exposed through streams. Kernels for each different size of the same template are launched concurrently using different streams. Concurrent kernels enable to increase the efficiency if there are inefficient low block count kernels, mostly by reducing idle streaming multiprocessor count while kernels are finishing up. The maximum number of concurrent kernels that can be executed on a Fermi GPU is 16. The number of different sizes ($4 \times 4$) to be matched for each template is also 16 in our implementation. This enables the matching

of all different sizes of a template to be launched concurrently.

Devices of compute capability 2.x come with an L1/L2 cache hierarchy that is used to cache local and global memory accesses. Programmers have some control over L1 caching. The same on-chip memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call.Experimentation has been performed to find out the best combination for a given kernel: 16 KB or 48 KB of L1 cache (and vice versa for shared memory) with or without global memory caching in L1 and with more or less local memory usage. The experimentation showed 16 KB L1 cache and 48 KB shared memory combination to be more effective.

The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing. The block size for both particle filter and sign recognition implementations is chosen to be 256.

Integer division and modulo operations are particularly costly and should be avoided or replaced with bitwise operations whenever possible. Shift operations have been used to avoid expensive division and modulo calculations.

Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition has been written so as to minimize the number of divergent warps.

Parallel reduction has been used for the summation kernel in the particle filter implementation. Parallel reduction is a common and important data parallel primitive. Although it is quite easy to implement in CUDA, it needs several optimizations to achieve better performance. These steps also demonstrate important optimization

strategies in CUDA. Parallel reduction is a tree based approach used within each thread block. It needs to be able to use multiple thread blocks to process very large arrays. Each block reduces a portion of the array. Partial results need to be communicated between thread blocks. Since CUDA has no global synchronization, the solution is to decompose the computation into multiple kernel invocations. Kernel launch serves a global synchronization point. It has negligible hardware overhead and low software overhead. In the case of reductions, code for all levels is the same, it means recursive kernel invocations.

In general, the optimization goal is to reach peak gpu performance by choosing the right metric: either GFLOP/s for compute-bound kernels or bandwidth for memory-bound kernels. Reductions have very low arithmetic intensity. Therefore, the optimization goal should be bandwidth.

In kernel implementation, each thread loads one element from global memory to shared memory, reduction is done in shared memory in a loop and the result for the block is written to global memory. When interleaved addressing is used during reduction, it causes highly divergent warps which are very inefficient. When divergent branch in the loop is replaced with strided index and non-divergent branch, we get better performance with approximately 2.3x speedup, but in this case we face with a new problem which is shared memory bank conflicts. When we use sequential addressing by replacing strided indexing in the loop with a reversed loop and thread ID based indexing as shown in Figure 4.15, we end up with conflict free implementation which results in another speedup by approximately 2x. The cumulative speedup is 4.7x.

# 7. CONCLUSION

In this thesis, we aimed to address the real-time performance requirements of ADAS and intelligent vehicle applications on emerging multicore and manycore architectures. We have surveyed the current generation of advanced driver assistance systems. We have examined multiple ways in which such systems gather and analyze data in order to help the driver, increase traffic safety, and autonomous behaviour. State estimation and sensor fusion algorithms play an important role in ADAS applications. Vehicle localization through fusion of GPS and dead reckoning sensors like odometer is one of the fundamental tasks in intelligent vehicles. It is also one of the representative tasks for vehicle internal state sensing. First, we have focused on computationally intensive state estimation algorithms, namely particle filters, which is known to be a successful state estimation tool for nonlinear systems, but its computational complexity has often been a prohibitory factor for it to be employed in real-time applications. We proposed a parallel particle filter based vehicle localization and map matching algorithm which fuses GPS, odometer and digital maps. We developed parallel versions of the algorithm both on multicore CPUs using OpenMP and manycore GPUs using CUDA programming model. We conducted tests with real data collected from a vehicle, equipped with a GPS and an odometer, on a parallel system having two 6-core CPUs and two 512-cores GPUs. We examined the execution times and parallel speedups. We also examined the effect of number of particles on the success rate of the algorithm to determine the optimum number of particles to be used on such a platform.

In the second part of the research, we focused on a traffic sign recognition application which is also one of the fundamental camera based ADAS applications and constitute a good representative of vehicle external environment sensing. We proposed a parallel traffic sign recognition algorithm which employs Kinect sensor for image acquisition and sign detection. We used a template matching based algorithm which is a simple but also a computationally complex algorithm for the recognition phase. We also employed digital map fusion which improves the success rate of the recognition. We developed parallel versions of the algorithm both on multicore CPUs using

OpenMP and manycore GPUs using CUDA programming model. We conducted tests with real data collected via a vehicle, equipped with a Kinect sensor together with a GPS and an odometer, on a parallel system having two 6-core CPUs and two 512-cores GPUs. Test data consists of various weather and lighting conditions. We examined the execution times and parallel speedups. We also performed and tested a multi GPU implementation using CUDA.

In summary, we introduced a real-time traffic sign recognition system employing digital map fusion, and we derived and tested parallel implementations and conducted performance analysis on emerging multicore CPUs and GPUs. We presented experimental results and performance analysis on state-of-the-art 6-core CPUs and 512-core GPUs having the new generation Fermi CUDA architecture. Test results show that up to 75 times speedups can be achieved for particle filter based localization and map matching on GPU over sequential implementation, and real-time performance is possible in the case of high computational cost of using map topology information. We observed that increasing the number of particles up to a certain level drastically decreases the error rate. We showed that success of localization and map matching can be increased by employing a high number of particles where real-time performance can be achieved only by parallelization.

We have achieved up to 35.2x speedups over implementations on sequential systems for our template matching based recognition algorithm which corresponds to 13 frames per second. The speedups achieved for our sign recognition system show that the template matching based recognition approach with map augmentation, which is a simple but computationally intensive technique, can be used with real-time performance in the vehicle environment. We observed detection rates over 90% using the Kinect sensor and recognition rates over 80% for various road, weather and lighting conditions. Test results show that the system performs very well even in poor lighting and night conditions. Frame rates can be further increased by prefiltering the templates to be matched via map augmentation. The proposed system is unique since it is not limited to certain sign types, can be used for recognition of wide range of traffic signs, can be used in any lighting conditions, utilizes the Kinect sensor to achieve a

good price/performance, and runs on commercially available parallel hardware.

All components used in the system are commercially available. A standard GPS receiver, odometer input which is available in all vehicles, navigable digital map database, Kinect sensor and multicore CPU and GPUs are utilized resulting in a very good performance. Resulting system can be utilized in ADAS domain.

As the results show, CUDA implementation is much faster than the OpenMP implementation both for particle filter based localization and for sign recognition algorithms. This is mainly due to the computational power of the GPU architecture in terms of number of cores and speed of floating point operations and the suitability of data-parallel nature of both particle filter algorithm and template matching based sign recognition algorithm to this platform. On the other hand, scalability is one of the most important chracteristics of a parallel implementation. Although speedups for multicore CPU implementations are not so high due to the number of cores in the currently available CPUs, the linear or near-linear speedups achieved show that we have a good scalability and we can expect the speedups to increase linearly as the number of cores in multicore CPUs increase in the future.

We can expect to achieve similar performances for other ADAS or autonomous vehicle tasks such as road estimation, lane following, and object tracking where a particle filter is employed. Also, we can expect similar performances for other camera based ADAS applications where low level vision and image processing operations are involved.

Programmability is another important aspect of a parallel system. When we evaluate the implementations from the programming practices point of view, we have seen that the OpenMP actually provides an easy to use programming model to the programmer. The parallelism can be introduced incrementally, by using extensions to the sequential code. It is quite possible to maintain one source code for both sequential and parallel versions of the same program managed by conditional compilation. On the other hand, CUDA programming model is also easy to use, but requires significant

reorganization of the sequential code. This is due to the fact that some part of the code has to be implemented as kernels to execute on the GPU, whereas some part of it to run on the host. However, we have seen that it is also possible to maintain one source code for both sequential and parallel versions of the same program managed by conditional compilation and careful organization.

Given the technological advancement in automobile industry, it can be expected that in the future, more and more ADAS will become standard in new vehicles. For increased safety, all those systems will be combined and the amount of data gathered for analysis will increase. Information will also be shared among vehicles via vehicular communication systems, to provide even more data to vehicles in order to produce more accurate warnings and reactions. Further in the future, self-driving cars will become dominant drivers, thereby eliminating human error in driving completely.

Our target architecture is a combination of a multicore CPU and a many-core graphics processing unit (GPU), which can take place in a production vehicle environment as a unified computing platform in the near future. We can conclude that the high computational power, energy efficiency and programmability of emerging general purpose multicore/manycore processors make them a good candidate for a unified vehicle computing platform to host advanced driving assistance systems (ADAS) and autonomous vehicle applications by replacing specialized hardware and/or software platforms for each application. On the other hand, it is possible to meet the real-time performance requirements of those applications on such a platform. Parallelization and using parallel programming techniques will be the key method to speed up applications on multicore and manycore architectures.

Future work may include investigating the co-scheduling of other tasks that can run simultaneously on the same platform with sign recognition and localization while delivering required throughput and minimal affordable latency.

As an another future work, test results on hypotetical homogeneous & heterogeneous multicore architectures can be examined and a heterogeneous multicore archi-

tecture to best fit to the domain can be proposed.

New features of the architectures can be incorporated to the design and implementation of the algorithms to further increase the performance of the applications. Especially, each new release of GPU architectures, apart from the significantly increased number of cores, comes with more advanced features to further increase the performance of the parallel implementations. For instance, NVIDIA's new Kepler architecture's dynamic parallelism feature adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU [122]. Similarly, Hyper-Q feature of Kepler enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times. Hyper-Q increases the total number of connections (work queues) between the host and the GK110 GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi) [122].

# REFERENCES


1. Gietelink, O., J. Ploeg, B. D. Schutter, and M. Verhaegen, "Development of Advanced Driver Assistance Systems with Vehicle Hardware-in-the-loop Simulations", *Vehicle System Dynamics: International Journal of Vehicle Mechanics and Mobility*, Vol. 44, No. 7, pp. 569–590, 2006.

2. Bishop, R., "A Survey of Intelligent Vehicle Applications Worldwide", *IEEE Intelligent Vehicles Symposium (IV), 2000*, pp. 25–30, 2000.

3. Shaout, A., "Advanced Driver Assistance Systems - Past, Present and Future", *Seventh International Computer Engineering Conference (ICENCO), 2011*, pp. 72-82, 2011.

4. Yannis, G., and C. Antoniou, "State-of-the-art on Advanced Driving Assistance Systems", *Workshop on the Role of Advanced Driver Assistance Systems on Traffic Safety and Efficiency*, Athens, 2000.

5. Bishop, R., *Intelligent Vehicle Technology and Trends*, Artech House, Norwood, 2005.

6. Cheng, H., N. Zheng, X. Zhang, J. Qin, and H. Van de Wetering, "Interactive Road Situation Analysis for Driver Assistance and Safety Warning Systems: Framework and Algorithms", *IEEE Transactions on Intelligent Transportation Systems*, Vol. 8, No. 1, pp. 157–167, 2007.

7. Zheng, N., S. Tang, H. Cheng, Q. Li, G. Lai, and F. W. Wang, "Toward Intelligent Driver Assistance and Safety Warning System", *IEEE Intelligent Systems*, Vol. 19, No. 2, pp. 8–11, 2004.

8. Buehler, M., K. Lagnemma, and S.Singh, *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*, Springer, Berlin, 2010.

9. Rouff, C., and M. Hinchey, *Experience from the DARPA Urban Challenge*, Springer, London, 2012.

10. S. Thrun et al., "Stanley: The Robot that Won the DARPA Grand Challenge", *Journal of Field Robotics Special Issue on the DARPA Grand Challenge 2005, Part 2*, Vol. 23, No. 9, pp. 661-692, 2006.

11. C. Urmson et al., "Autonomous Driving in Urban Environments: Boss and the Urban Challenge", *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, Vol. 25, No. 8, pp. 425-466, 2008.

12. Geiger, A., M. Lauer, F. Moosmann, B. Ranft, H. Rapp, C. Stiller, and J. Ziegler, "Team AnnieWAY's Entry to the 2011 Grand Cooperative Driving Challenge", *IEEE Transactions on Intelligent Transportation Systems*, No. 99, pp. 1-10, 2012.

13. Leonard, J., J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, and S. Karaman, "A Perception-driven Autonomous Urban Vehicle", *Journal of Field Robotics - Special Issue on the 2007 DARPA Urban Challenge, Part III*, Vol. 25, No. 10, pp. 163–230, 2009.

14. Bacha, A., C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, and D. Anderson, "Odin: Team Victortango's Entry in the DARPA Urban Challenge", *Journal of Field Robotics*, Vol. 25, No. 8, pp. 467–492, 2008.

15. Bohren, J., T. Foote, J. Keller, A. Kushleyev, D. Lee, A. Stewart, P. Vernaza, J. Derenick, J. Spletzer, and B. Satterfield, "Little Ben: The Ben Franklin Racing Team's Entry in the 2007 DARPA Urban Challenge", *Journal of Field Robotics*, Vol. 25, No. 9, pp. 231–255, 2009.

16. Montemerlo, M., J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, and B. Huhnke, "Junior: the Stanford Entry in the Urban Challenge", *Journal of Field Robotics*, Vol. 25, No. 9, pp. 569–597,

2008.

17. Guizzo, E., "How Google's Self-Driving Car Works", *IEEE Spectrum Blog*, 2011, `http://spectrum.ieee.org/automaton/robotics/artificial-intelligence-/how-google-self-driving-car-works`, accessed at October 2012.

18. Ozguner, U., T. Acarman, and K. Redmill, *Autonomous Ground Vehicles*, Artech House, Norwood, 2011.

19. Cheng, H., *Autonomous Intelligent Vehicles: Theory, Algorithms, and Implementation (Advances in Computer Vision and Pattern Recognition)*, Springer, London, 2011.

20. Eskandarian, A., *Handbook of Intelligent Vehicles*, Springer, London, 2012.

21. Redmill, K., J. I. Martin, and U. Ozguner, "Sensing and Sensor Fusion for the 2005 Desert Buckeyes DARPA Grand Challenge Offroad Autonomous Vehicle", *IEEE Intelligent Vehicles Symposium (IV), 2006*, pp. 528-533, 2006.

22. Muller, J. "Think How Much Smarter Your Car Will Be In A Few Years", *Forbes Autos Blog*, 2012, `http://www.forbes.com/sites/joannmuller-/2012/12/24/think-how-much-smarter-your-car-will-be-in-a-fewyears/`, accessed at December 2012.

23. Skog, I. and P. Handel, "In-Car Positioning and Navigation Technologies - A Survey", *IEEE Transactions on Intelligent Transportation Systems, 2009*, Vol. 10, No. 1, pp. 4-21, 2009.

24. Groves, P. D., *Principles of GNSS, Inertial, and Multi-Sensor Integrated Navigation Systems (GNSS Technology and Applications)*, Artech Print on Demand, Boston, 2008.

25. Grewal, M. S., L. R. Weill, and P. A. Andrews, *Global Positioning Systems,*

*Inertial Navigation, and Integration*, Wiley, Hoboken, 2007.

26. Hegarty, C., and E. Chatre, "Evolution of the Global Navigation Satellite System (GNSS)", *Proceedings of the IEEE*, Vol. 96, No. 12, pp. 1902–1917, 2008.

27. Kaplan, E., and C. Hegarty, *Understanding GPS: Principles and Applications*, Artech House, Norwood, 2006.

28. Chen., Z., "Bayesian filtering: From Kalman Filters to Particle Filters, and Beyond", Adaptive Syst. Lab., McMaster Univ., Hamilton, Canada, 2003, `http://soma.crl.mcmaster.ca/ zhechen/download/ieee_bayesian.ps`, accessed at October 2010.

29. Kalman, R. E., "A New Approach to Linear Filtering and Prediction Problems", *Transactions of the ASME – Journal of Basic Engineering*, No. 82 (Series D), pp. 35-45, 1960.

30. Welch, G. and G. Bishop, *An Introduction to the Kalman Filter*, Technical Report, University of North Carolina at Chapel Hill, Department of Computer Science, NC, USA, 1995.

31. Grewal, M., and A. Andrews, *Kalman Filtering: Theory and Practice using MATLAB*, Wiley, New York, 2008.

32. Gordon, N. J., D. J. Salmond, and A. F. M. Smith, "A Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation", *IEE Proceedings on Radar and Signal Processing*, Vol. 140, No. 2, pp. 107-113, 1993.

33. Doucet, A., N. Freitas, and N. Gordon, *Sequential Monte Carlo Methods in Practice*, Springer, New York, 2001.

34. Fox, D., S. Thrun, W. Burgard, and F. Dellaert, "Particle Filters for Mobile Robot Localization", 2001.

35. Thrun, S., "Probabilistic Robotics", *Communications of ACM*, Vol. 45, No. 3, pp. 52–57, 2002.

36. Thrun, S., W. Burgard, and D. Fox, *Probabilistic Robots*, MIT Press, Cambridge, 2006.

37. Arulampalam, M. S., S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking", *IEEE Transactions on Signal Processing*, Vol. 50, No. 2, pp. 174–188, 2002.

38. Mitchell, H. B., *Multi-Sensor Data Fusion: An Introduction*, Springer, Berlin, 2010.

39. Herpel, T., C. Lauer, R. German, and J. Salzberger, "Multi-sensor Data Fusion in Automotive Applications", *IEEE International Conference on Sensing Technology*, pp. 206-211, 2008.

40. Quddus, M. A., W. Y. Ochieng, and R. B. Noland, "Current Map-matching Algorithms for Transport Applications: State-of-the-art and Future Research Directions", *IEEE Transportation Research, Part C, Emerging Technologies*, Vol. 15, No.5, pp. 312-328, 2007.

41. Quddus, M. A., *High Integrity Map Matching Algorithms for Advanced Transport Telematics Applications*, Ph.D. Thesis, University of London, 2006.

42. Redmill, K. A., T. Kitajima, and Ü. Özguner, "DGPS/INS Integrated Positioning for Control of Automated Vehicles", *IEEE Intelligent Transportation Systems Conference 2001*, pp. 172–178, 2001.

43. Peker, A. U., T. Acarman, and O. Tosun, "Particle Filter Vehicle Localization and Map Matching using Map Topology", *IEEE Intelligent Vehicles Symposium (IV), 2011*, pp. 248-253, 2011.

44. Par, K. and O. Tosun, "Parallelization of Particle Filter Based Localization and Map Matching Algorithms on Multicore/Manycore Architectures", *IEEE Intelligent Vehicles Symposium (IV), 2011*, pp. 820-826, 2011.

45. Yang, N., W. F. Tian, Z. H. Jin, and C. B. Zhang, "Particle Filter for Sensor Fusion in A Land Vehicle Navigation System", *Measurement Science and Technology*, Vol. 16, No. 3, pp. 677-681, 2005.

46. Jabbour, M., P. Bonnifait, and V. Cherfaoui, "Map Matching Integrity Using Multihypothesis Road Tracking", *Journal of Intelligent Transportation Systems*, Vol. 12, No. 4, pp. 189-201, 2008.

47. Ballard, D. H.,"Generalizing the Hough Transform to Detect Arbitrary Shapes", *Pattern Recognition*, Vol. 13, No. 2, pp. 111–122, 1981.

48. Redmill, K. A., "A Lane Tracking System for Intelligent Vehicle Applications", *IEEE Intelligent Transportation Systems Conference 2001*, pp. 273–279, 2001.

49. Apostoloff, N., and A. Zelinsky, "Robust Vision based Lane Tracking Using Multiple Cues and Particle Filtering", *IEEE Intelligent Vehicles Symposium (IV) 2003*, pp. 558–563, 2003.

50. Peterson, K., J. Ziglar, and P. E. Rybski, "Fast Feature Detection and Stochastic Parameter Estimation of Road Shape Using Multiple LIDAR", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 612-619, 2008.

51. Southall, B., and C. Taylor, "Stochastic Road Shape Estimation", *IEEE International Conference on Computer Vision (ICCV), 2001*, Vol. 1, pp. 205-212, 2001.

52. McCall, J. C., and M. M. Trivedi, "Video-Based Lane Estimation and Tracking for Driver Assistance: Survey, System, and Evaluation", *IEEE Transactions on Intelligent Transportation Systems*, Vol. 7, No. 1, pp. 20-37, 2006.

53. Vahidi, A., and A. Eskandarian, "Research Advances in Intelligent Collision Avoidance and Adaptive Cruise Control", *IEEE Transactions on Inteligent Transportation Systems*, Vol. 4, No. 3, pp. 143–153, 2003.

54. Moutarde F., A. Bargeton, A. Herbin, and L. Chanussot, "Robust on-vehicle Real-time Visual Detection of American and European Speed Limit Signs, with a Modular Traffic Sign Recognition System", *IEEE Intelligent Vehicles Symposium (IV), 2007*, pp. 1122-1126, 2007.

55. Damavandi, Y. B., and K. Mohammadi, "Speed Limit Traffic Sign Detection and Recognition", *IEEE Conference on Cybernetics and Intelligent Systems*, pp. 797 - 802, 2004.

56. Escalara, A., J. M. Armingol, and M. Mata, "Traffic Sign Recognition and Analysis for Intelligent Vehicles", *Image and Vision Computing*, Vol. 21, pp. 247-258, 2003.

57. Torresen, J., J. W. Bakke, and L. Sekanina, "Efficient Recognition of Speed Limit Signs", *IEEE Conference on Intelligent Transportation Systems (IV) 2004*, pp. 652 - 656, 2004.

58. C. Bahlmann et al., "A System for Traffic Sign-detection, Tracking and Recognition using Color, Shape and Motion Information", *IEEE Intelligent Vehicles Symposium (IV), 2005*, pp. 255-260, 2005.

59. Ren, F., J. Huang, R. Jiang, and R. Klette, "General Traffic Sign Recognition by Feature Matching", *Proceedings of the 24th International Conference Image and Vision Computing New Zealand (IVCNZ 2009)*, pp. 409-414, 2009.

60. Maldonado-Bascón, S., S. Lafuente-Arroyo, P. Gil-Jiménez, H. Gómez-Moreno, and F. López-Ferreras,"Road-Sign Detection and Recognition Based on Support Vector Machines", *IEEE Conference on Intelligent Transportation Systems (ITSC) 2007*, pp. 264 - 278, 2007.

61. Fang, C., S. Chen, and C. Fuh, "Road-Sign Detection and Tracking", *IEEE Transactions on Vehicular Technology*, Vol. 52, No. 5, pp. 1329-1341, 2003.

62. Meuter, M., A. Kummert, and S. Muller-Schneiders, "3D Traffic Sign Tracking Using a Particle Filter", *IEEE Intelligent Transportation Systems Conference (ITSC) 2008*, pp. 168-173, 2008.

63. Kongetira, P., K. Aingaran, and K. Olukotun, "Niagara: a 32-way Multithreaded Sparc Processor", *IEEE Micro*, Vol. 25, No. 2, pp. 21-29, 2005.

64. Shah, M., "Sparc T4: Dynamically Threaded Server-on-a-Chip", *IEEE Micro*, Vol. 32, No. 2, pp. 8-19, 2012.

65. Koranne, S., *Practical Computing on the Cell Broadband Engine*, Springer, New York, 2009.

66. Kahle, J. A., M. N. Day, H. P. Hofstee, and C. R. Johns, "Introduction to the Cell Multiprocessor", *IBM Journal of Research and Development*, Vol. 49, No. 4, pp. 589-604, 2005.

67. Blake, G., R. G. Dreslinski, and T. Mudge, "A Survey of Multicore Processors", *IEEE Signal Processing Magazine*, Vol. 26, No. 6, pp. 26-37, 2009.

68. Nickolls, J., and W. J. Dally, "The GPU Computing Era", *IEEE Micro*, Vol. 30, No. 2, pp. 56-69, 2010.

69. Kirk, D. B. and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Waltham, 2010.

70. Sanders, J., and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, Boston, 2010.

71. Farber, R., *CUDA Application Design and Development*, Morgan Kaufmann, Waltham, 2011.

72. Cook, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann, Burlington, 2012.

73. NVIDIA, *CUDA C Programming Guide 3.2*, 2010, `http://developer.-download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Program-ming_Guide.pdf`, accessed at October 2010.

74. Butenhof, D. R., *Programming with POSIX Threads*, Addison-Wesley Professional, Boston, 1997.

75. Reinders, J., *Intel Threading Building Blocks*, O'Reilly Media, Sebastopol, 2007.

76. Chapman, B., G. Jost, and R. van der Pas, *Using OpenMP, Portable Shared Memory Parallel Programming*, The MIT Press, Cambridge, 2007.

77. OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0", 2008, `http://openmp.org/wp/openmp-specifications`, accessed at 2010.

78. Gustafsson, F., F. Gunnarson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P. Nordlund, "Particle Filters for Positioning, Navigation and Tracking", *IEEE Transactions on Signal Processing*, Vol. 50, No. 2, pp. 425-437, 2002.

79. Chitchian, M., A. Simenetto, A. S. van Amesfoort, and T. Keviczky, "Distributed Computation Particle Filters on GPU Architectures for Real-Time Control Applications", *IEEE Transactions on Control Systems Technology*, No. 99, pp. 1, 2013.

80. Brun, O., V. Teuliere, and J. Garcia, "Parallel Particle Filtering", *Journal of Parallel and Distributed Computing*, Vol. 62, No. 5, pp. 1186 – 1202, 2002.

81. Bashi, A. S., V. P. Jilkov, X. R. Li, and H. Chen, "Distributed Implementations of Particle Filters", *Proceedings of the IEEE Conference of Information Fusion,*

*(Cairns, Australia)*, pp. 1164 – 1171, 2003.

82. Bolic, M., P. M. Djuric, and S. Hong, "Resampling Algorithms and Architectures for Distributed Particle Filters", *IEEE Transactions on Signal Processing*, Vol. 53, No. 7, pp. 2442 – 2450, 2005.

83. Bolic, M., A. Athalye, S. Hong, and P. Djuric, "Study of Algorithmic and Architectural Characteristics of Gaussian Particle Filters, *Journal of Signal Processing Systems*, Vol. 61, No. 2, pp. 205 – 218, 2010.

84. Rosen, O., A. Medvedev, and M. Ekman, "Speedup and Tracking Accuracy Evaluation of Parallel Particle Filter Algorithms Implemented on a Multicore Architecture, *Proceedings of the 2010 IEEE International Conference on Control Applications, (Yokohama, Japan)*, pp. 440 – 445, 2010.

85. Hendeby, G., R. Karlsson, and F. Gustafsson, "Particle Filtering: The Need for Speed", *EURASIP Journal on Advances in Signal Processing*, Vol. 2010, 2010.

86. Chao, M. A., C. Y. Chu, C. H. Chao, and A. Y. Wu, "Efficient Parallelized Particle Filter Design on CUDA, *Proceedings of the 2010 IEEE Workshop on Signal Processing Systems, (San Francisco, USA)*, pp. 299 – 304, 2010.

87. Brown, J. and D. Capson, "A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filter, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 18, No. 1, pp. 68 – 80, 2012.

88. Ferreira, J. F., J. Lobo, and J. Dias, "Bayesian Real-time Perception Algorithms on GPU", *Journal of Real-Time Image Processing (Special Issue)*, Vol. 6, No. 3, 2010.

89. Goodrum, M. A., M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron, "Parallelization of Particle Filter Algorithms", *Proc. of 3rd Workshop on Emerging Applications and Many-core Architecture (EAMA)*, 2010.

90. Ulman, J., "Bayesian Particle Filter Tracking with CUDA", 2010, http://csi702.net/csi702/images/Ulman_report_final.pdf, accessed at April 2010.

91. Bolic, M., *Architectures for Efficient Implementation of Particle Filters*, PhD Dissertation, Stony Brook University, 2004.

92. Happe, M., E. Lübbers, and M. Platzner, "A Multithreaded Framework for Sequential Monte Carlo Methods on CPU/FPGA Platforms", *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pp. 380-385, 2009.

93. Mederios, H., J. Park, and A. Kak, "A Parallel Implementation of the Color-based Particle Filter for Object Tracking", *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1-8, 2008.

94. Falcou, J., T. Chateau, J. Serot and J. T. Lapreste, "Real Time Parallel Implementation of a Particle Filter Based Visual Tracking", *CIMCV - Workshop on Computation Intensive Methods for Computer Vision (ECCV), 2006*, pp. 33-40, 2006.

95. Liu, K., L. Tang, S. Li, L. Wang, and W. Liu, "Parallel Particle Filter Algorithm in Face Tracking", *IEEE International Conference on Multimedia and Expo (ICME), 2009*, pp. 1817-1820, 2009.

96. Hendeby, G., J. Hol, R. Karlsson, and F. Gustafsson, "A Graphics Processing Unit Implementation of the Particle Filter", *Proceedings of the 15th European Signal Processing Conference (EUSIPCO)*, pp. 1639-1643, Poland, 2007.

97. Harris, M., "Optimizing Parallel Reduction in CUDA", *NVIDIA CUDA SDK code samples*, 2010, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, accessed at October 2010.

98. Escalara, A., L. E. Moreno, M. A. Salichs, and J. M. Armingol, "Road Traffic Sign Detection and Classification", *IEEE Transactions On Industrial Electronics*, Vol. 44, No. 6, pp. 848-859, 1997.

99. Eichner, M. and T. Breckon, "Integrated Speed Limit Detection and Recognition from Real-time Video", *IEEE International Intelligent Vehicles Symposium (IV) 2008*, pp. 626–631, 2008.

100. Escalera, S., X. Baró, O. Pujol, J. Vitrià, and P. Radeva, "Background on Traffic Sign Detection and Recognition", *Traffic-Sign Recognition Systems. SpringerBriefs in Computer Science*, pp. 5–13. Springer, London, 2011.

101. Hoferlin, B. and K. Zimmermann, "Towards Reliable Traffic Sign Recognition", *IEEE Intelligent Vehicles Symposium, 2009*, pp. 324-329, 2009.

102. Shneier, M., "Road Sign Detection and Recognition", *IEEE Computer Society International Conference on Computer Vision and Pattern Recognition*, 2005.

103. Keller, C. G., C. Sprunk, C. Bahlmann, J. Giebel, G. Baratoff, "Real-time Recognition of U.S. Speed Signs", *IEEE Intelligent Vehicles Symposium (IV) 2008*, pp. 518 - 523, 2008.

104. Lowe, D., "Object Recognition from Local Invariant Features", *Proceedings of the International Conference on Computer Vision*, pp. 1150-1157, 1999.

105. Lowe, D., "Distinctive Image Features from Scale Invariant Keypoints", *International Journal of Computer Vision*, pp. 91-110, 2004.

106. Bay, H., A. Ess, T. Tuytelaars, and L.V. Gool, "SURF: Speeded Up Robust Features", *Computer Vision and Image Understanding (CVIU)*, Vol. 110, No. 3, pp. 346–359, 2008.

107. Ach, R., N. Luth, and A. Techmer, "Real-Time Detection of Traffic Signs on

a Multi-Core Processor", *IEEE Intelligent Vehicles Symposium (IV), 2008*, pp. 307-312, 2008.

108. Ach, R., N. Luth, A. Techmer, and A. Walther, "Classification of Traffic Signs in Real-Time on a Multi-Core Processor", *IEEE Intelligent Vehicles Symposium (IV), 2008*, pp. 313-318, 2008.

109. Ozcelik, P. M., V. Glavtchev, J. M. Ota, and J. D. Owens, "A Template-based Approach for Real-Time Speed-Limit Sign Recognition on an Embedded System Using GPU Computing", *Proceedings of the 32nd DAGM conference on Pattern recognition*, pp. 162-171, 2010.

110. Glavtchev, V., P. M. Ozcelik, J. M. Ota, and J. D. Owens, "Feature-based Speed Limit Sign Detection Using a Graphics Processing Unit", *Intelligent Vehicles Symposium (IV), 2011*, pp. 195-200, 2011.

111. Mussi, L., S. Cagnoni, E. Cardarelli, F. Medici, and P. Porta, "GPU Implementation of a Road Sign Detector based on Particle Swarm Optimization", *Evolutionary Intelligence*, Vol. 3, No. 3-4, pp. 155–169, 2010.

112. Ugolotti, R., Y. S. G. Nashed, and S. Cagnoni, "Real-Time GPU Based Road Sign Detection and Classification", *Lecture Notes in Computer Science*, Vol. 7491, pp. 153–162, Springer, 2012.

113. Liu, W. and K. Maruya, "Detection and Recognition of Traffic Signs in Adverse Conditions", *IEEE Intelligent Vehicles Symposium (IV), 2009*, pp. 335-340, 2009.

114. Yu, T., Y. Moon, J. Chen, H. Fung, H. Ko, and R. Wang, "An Intelligent Night Vision System for Automobiles", *IAPR Conference on Machine Vision Applications (MVA), 2009*, pp. 505-508, 2009.

115. Zhang, Z., "Microsoft Kinect Sensor and Its Effect", *IEEE MultiMedia*, Vol. 19, No. 2, pp. 4-10, 2012.

116. Ackerman, E., "Top 10 Robotic Kinect Hacks", *IEEE Spectrum Automaton Blog*, 2011, `http://spectrum.ieee.org/automaton/robotics/diy/top-10-robotic-kinect-hacks`, accessed at March 2011.

117. Fiala, M. and A. Ufkes, "Visual Odometry Using 3-Dimensional Video Input", *IEEE Canadian Conference on Computer and Robot Vision (CRV) 2011*, pp. 86-93, 2011.

118. Nakamura, T., "Real-time 3-D Object Tracking Using Kinect Sensor", *IEEE International Conference on Robotics and Biomimetics 2011*, pp. 784-788, 2011.

119. Tanaka, M., "Robust Parameter Estimation of Road Condition by Kinect Sensor", *IEEE SICE Annual Conference (SICE) 2012*, pp. 197-202, 2012.

120. *The Mobile Robot Programming Toolkit*, 2009, `http://www.mrpt.org/`, accessed at February 2010.

121. NVIDIA, *CUDA C Best Practices Guide 3.2*, 2010, `http://developer.-download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf`, accessed at October 2010.

122. NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110", 2012, `http://www.nvidia.com/content/PDF/kepler-/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, accessed at December 2012.

# REFERENCES NOT CITED


1. Bertozzi, M., "Artificial Vision in Road Vehicles", *Proceedings of the IEEE*, Vol. 90, No. 7, pp. 1258–1271, 2002.

2. Bertozzi, M., and A. Broggi, "GOLD: a Parallel Real-time Stereo Vision System for Generic Obstacle and Lane Detection", *IEEE Transactions on Image Processing*, Vol. 7, No. 1, pp. 62–81, 2002.

3. Bordawekar, R., "Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU", *IBM Reseach Report RC25033*, 2010.

4. Bradski, G. and A. Kaehler *Learning OpenCV, Computer Vision with the OpenCV Library*, O'Reilly Media, Sebastopol, 2008.

5. Chen, Y., W. Li, J. Li, and T. Wang, "Novel parallel Hough Transform on multi-core processors", *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2008*, pp. 1457–1460, 2008.

6. Keckler, S., K. Olukotun, and H. P. Hofstee, *Multicore Processors and Systems*, Springer, New York, 2009.

7. "IMAPCAR: A 100 GOPS In-Vehicle Vision Processor Based on 128 Ring Connected Four-Way VLIW Processing Elements", *Journal of Signal Processing Systems*, Vol. 62, No. 1, pp. 5-16, 2011.

8. Par, K. and O. Tosun, "Parallelization of Particle Filter Based Localization and Map Matching Algorithms on Multicore/Manycore Architectures", *IEEE Intelligent Vehicles Symposium (IV), 2011*, pp. 820-826, 2011.

9. Par, K. and O. Tosun, "Real-time Traffic Sign Recognition with Map Fusion on

Multicore/Many-core Architectures", *Acta Polytechnica Hungarica*, Vol. 9, No. 2, pp. 231-250, 2012.

10. Ranft, B., T. Schoenwald, and B. Kitt, "Parallel Matching-based Estimation - A Case Study on Three Different Hardware Architectures", *IEEE Intelligent Vehicles Symposium (IV), 2011*, pp. 1060 - 1067, 2011.

11. Szeliski, R., *Computer Vision: Algorithms and Applications*, Springer, London, 2011.

12. Techmer, A., "Application Development of Camera-based Driver Assistance Systems on a Programmable Multi-Processor Architecture", *IEEE Intelligent Vehicles Symposium (IV), 2007*, pp. 1211-1216, 2007.