DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Hardware Acceleration in the Context of Motion Control for Autonomous Vehicles

JELIN LESLIN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Hardware Acceleration in the Context of Motion Control for Autonomous Systems

JELIN LESLIN

Master's Programme, Embedded Systems, 120 credits
Date: November 16, 2020
Supervisor: Masoumeh (Azin) Ebrahimi
Examiner: Johnny Öberg
Industry Supervisor: Markhusson Christoffer &
Harikrishnan Girishkasthuri
School of Electrical Engineering and Computer Science
Host company: Volvo Trucks
Swedish title: Hårdvaruacceleration i samband med rörelsekontroll för autonoma system

Abstract

State estimation filters are computationally intensive blocks used to calculate uncertain/unknown state values from noisy/not available sensor inputs in any autonomous systems. The inputs to the actuators depend on these filter's output and thus the scheduling of filter has to be at very small time intervals. The aim of this thesis is to investigate the possibility of using hardware accelerators to perform this computation. To make a comparative study, 3 filters that predicts 4, 8 and 16 state information was developed and implemented in Arm real time and application purpose CPU, NVIDIA Quadro and Turing GPU, and Xilinx FPGA programmable logic. The execution, memory transfer time, and the total developement time to realise the logic in CPU, GPU and FPGA is discussed. The CUDA developement environment was used for the GPU implementation and Vivado HLS with SDSoc environment was used for the FPGA implementation. The thesis concludes that a hardware accelerator is needed if the filter estimates 16 or more state information even if the processor is entirely dedicated for the computation of filter logic. Otherwise, for a 4 and 8 state filter the processor shows similar performance as an accelerator. However, in a real time environment the processor is the brain of the system, so it has to give instructions to many other functions parallelly. In such an environment, the instruction and data caches of the processor will be disturbed and there will be a fluctuation in the execution time of the filter for every iteration. For this, the best and worst case processor timings are calculated and discussed.

Keywords: Hardware acceleration, Computation offloading, State estimation filter, Autonomous systems, FPGA, GPU.

Sammanfattning

Tillståndsberäkningsfilter är beräkningsintensiva block som används för att beräkna osäkra / okända tillståndsvärden från bullriga / ej tillgängliga sensoringångar i autonoma system. Ingångarna till manöverdonen beror på filterens utgång och därför måste schemaläggningen av filtret ske med mycket små tidsintervall. Syftet med denna avhandling är att undersöka möjligheten att använda hårdvaruacceleratorer för att utföra denna beräkning. För att göra en jämförande studie utvecklades och implementerades 3 filter som förutsäger information om 4, 8 och 16 tillstånd i realtid med applikationsändamålen CPU, NVIDIA Quadro och Turing GPU, och Xilinx FPGA programmerbar logik. Exekvering, minnesöverföringstid och den totala utvecklingstiden för att förverkliga logiken i båda hårdvarorna diskuteras. CUDAs utvecklingsmiljö användes för GPU-implementeringen och Vivado HLS med SDSoc-miljö använ-des för FPGA-implementering. Avhandlingen drar slutsatsen att en hårdvaru-accelerator behövs om filtret uppskattar information om mer än 16 tillstånd även om processorn är helt dedikerad för beräkning av filterlogik. För 4 och 8 tillståndsfilter, visar processorn liknande prestanda som en accelerator. Men i realtid är processorn hjärnan i systemet; så den måste ge instruktioner till många andra funktioner parallellt. I en sådan miljö kommer processorns instruktioner och datacacher att störas och det kommer att bli en fluktuation i exekveringstiden för filtret för varje iteration. För detta beräknas och diskuteras de bästa och värsta fallstiderna.

Nyckelord: Hårdvaruacceleration, beräkningsavlastning, tillståndsskattningsfilter, autonoma system, FPGA, GPU.

Acknowledgments

This thesis work was carried out in Volvo AB, Gothenburg. I would like to thank my industrial supervisors Christoffer Markhusson and Harikrishnan Girishkasthuri for the daily discussions, guidance and support throughout the project. My supervisor Dr. Masoumeh (Azin) Ebrahimi and examiner Dr. Johnny Öberg who helped me frame the research question and made sure it is moving in correct direction. I am also grateful to the team at Volvo for all the support related to data, software and hardware access.

Stockholm, November 2020
Jelin Leslin

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

ACP	Accelerator Coherency Port
API	Application Programming Interface
APU	Application Processing Unit
AXI	Advanced eXtensible Interface
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High level Synthesis
IP	Intellectual Property
LUT	Look Up Table
MPI	Message Passing Interface
MPSoC	Multi-Processor System-on-Chip
OpenCL	Open Computing Language
PCI	Peripheral Component Interconnect

PL	Programmable Logic
PS	Processing System
RPU	Real time Processing Unit
RTL	Register-transfer level
SDSoC	Software Defined System on Chip
SLAM	Simultaneous Localisation and Mapping
SM	Streaming multiprocessor
SOC	System on Chip
TCM	Tightly Coupled Memory
VMM	Vehicle Motion Management

Chapter 1

Introduction

This chapter gives an overall view of the research domain the thesis focuses on and an active problem within the domain that has been addressed. The motivation to choose this topic, problem definition, the engineering approach to solve the problem and the end goals during the initial stage of the project along with the report outline is discussed here.

1.1 Motivation and Importance

Present-day vehicles are equipped with accurate digital sensors and camera devices for making driving easy. Recent developments in machine learning and its footprint in almost all industry is something no one can disagree. Integrating these features can improve the fuel consumption, safety, and potentially break the co_2 curve of transport industry. According to [3], an autonomous vehicle is defined as a vehicle that can interpret and adapt to their surrounding through a combination of sensor tools and artificial intelligence to solve certain predetermined tasks. Recent studies from Harvard Business School has showed that the autonomous vehicle market share is predicted to be as high as 87.2% in 2045 [4]. Foreseeing all the advantages, trucking companies have started investing in the research and development of autonomous trucks. Companies are making joint ventures with chip manufacturers and hardware suppliers, to make hardwares and application-specific Application Programming Interfaces (APIs) that can efficiently run their computationally intensive algorithms.

Inside an autonomous vehicle, a vehicle automation module plans the vehicle motion and decides where the vehicle should go. A vehicle control module controls the vehicle actuators like breaks, steering, and powertrain, to make the vehicle go as requested. A key to control the vehicle is to know the states of the vehicle like the current speed, acceleration, and position. The automation module uses sensors like cameras, radars, and GPS to create a model of the vehicle environment and the vehicle itself. The control module uses wheel speed sensors and Inertial Measurement Units (IMUs) to estimate the motion of the vehicle. To realize the estimation, a state estimation filter is used. The filter is basically controlling the driving dynamics of the vehicle and is computationally intensive.

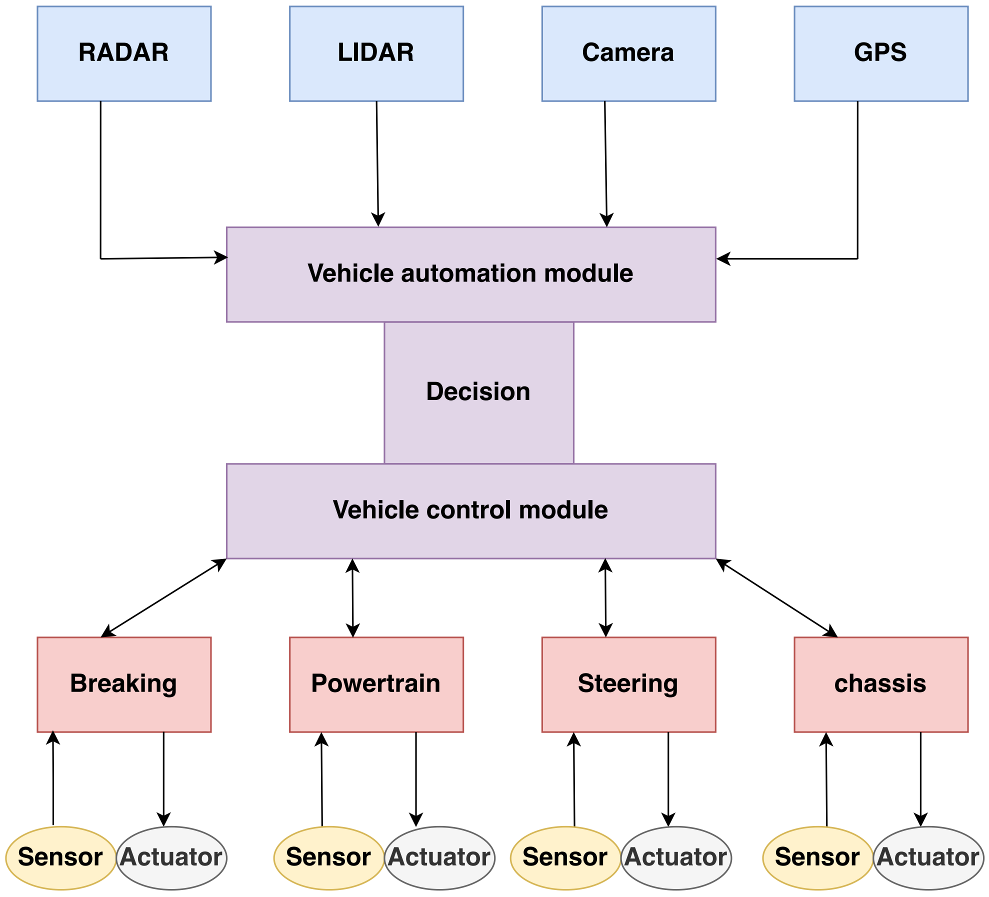


Figure 1.1 – Abstract diagram for sensor control

1.2 Problem statement

The motion estimation block of the vehicle motion management system is a computationally intensive part because of the presence of an information update filter in its real-time environment that estimates the rear axle state of the vehicle periodically. The base algorithm for this information update filter is the Extended kalman filter [5]. Even though the flow of this filter is sequential, the matrix operations can be parallelized and this raises the question of why not offload this computationally intensive filter to an application-specific hardware such as a FPGA/GPU.

- If the parallelization with matrix operations is not effective then what is the threshold/maximum state estimation filter where hardware acceleration must be considered for better performance?
- The Effects of the caches in the memory system of a multi application processor may drastically decrease its performance in a real-time environment. What is the best and worst-case processor execution time of the filter when caches are involved ? This is very important if the control algorithm is categorized for safety and time-critical applications.

1.3 Goal

The goal of this work is to develop a 4, 8 and 16 state estimation filter, offload the computations from the processor to a FPGA or a GPU, then compare the performance of three versions of the filter in these two accelerators and make a study on when acceleration will be necessary by analysing the best and worst case performance of the software running on the processor, with and without accelerator, using the following engineering flow.

- Program the Processor in C.
- Program the GPU with Compute Unified Device Architecture (CUDA) Application Programming Interface (API)s using the same C code in processor.
- Program the FPGA using Vivado high level synthesis in Software Defined System on Chip (SDSoC) environment using the same C code in processor.

1.4 Purpose

State estimation filter is the computationally intensive unit in motion estimation algorithm that controls the vehicle motion in an autonomous truck. The filter is a Kalman filter. Similar works on kalman filter have been implemented before, but they deal with higher dimension matrices to predict large number of states, where a high speedup between accelerator and processor is obvious. This thesis will deal with small matrix filters. Automotive, robotic, and space industries who are working with autonomous/self-driving machines will have insights from this project if they are using a smaller dimension state estimation filter in their model to estimate unknown values from the available sensor inputs that give control inputs to their actuators.

1.5 Delimitations

The scope of this thesis is only to identify possible hardware acceleration options for the state estimation filter in the motion control block by experimenting different matrix sizes. It is outside the scope of this thesis to look for any options in other blocks in the vehicle motion management model. The hardware version of the algorithms implemented in this project could be made more efficient by saving more resources in the programmable logic of the System on Chip (SOC) if the logics were implemented by a hand written Register-transfer level (RTL) code. But the scope of the thesis was to benefit Volvo and Volvo was not concerned about the resource utilization but only the execution time, so High level synthesis tools were used to code the Field Programmable Gate Array (FPGA).

1.6 Thesis outline

This thesis is divided into four main parts. Chapter 2, is a description of the background of the Kalman filter and its importance with previous implementations. Chapter 3 explains the hardware and software programming platforms used to implement the algorithm. Chapter 4 describes the engineering flow of how the algorithm was implemented in the hardware and the optimisation strategies employed to speed up the execution time with results. Chapter 5 is the conclusion with a summary and discussion of which hardware should be used under what circumstances along with available future work options from this project.

Chapter 2

Background study

This chapter will serve as a frame of reference for the study. It reviews primarily literature regarding vehicle motion management and its role in autonomous trucks. The role of the state estimation filter inside the vehicle motion management block and its mathematical derivation is also discussed here. Finally, a note on the evolution of hardware acceleration and its need here is presented.

2.1 Autonomous systems

An autonomous system is designed to operate without human intervention. For autonomous trucks, the job is to go from source to destination all by itself. This involves four abstract level algorithms that does the decision as a human would do. The perception algorithms act as the ears and nose of a truck, this involves working with inputs from lidar, radar, and other sensors that analyze the surrounding environment of the truck using image and signal processing techniques. Once the environment is known, the truck has to locate its position, Localization algorithms perform the job of finding the ego-position relative to a reference frame in an environment [6]. This step is needed to integrate the truck with the navigation system. The localization algorithms are usually run in parallel with mapping algorithms to check and update its position and are referred to as Simultaneous Localisation and Mapping (SLAM).

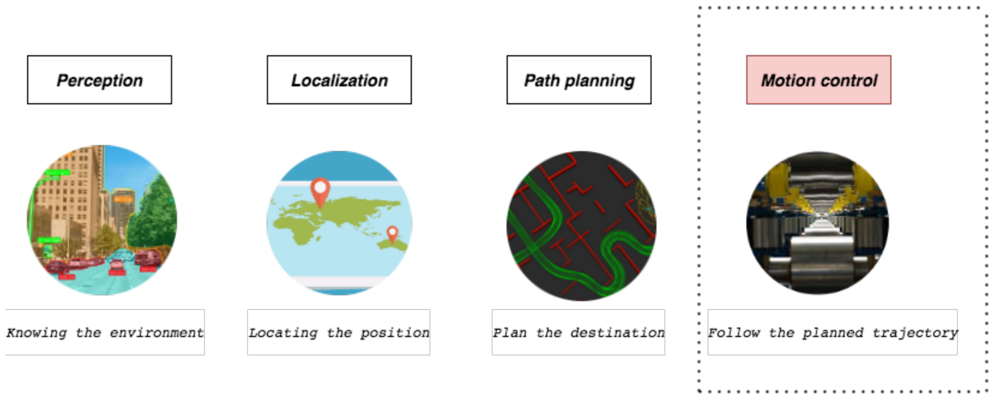


Figure 2.1 – Autonomous Systems

Once the environment and positioning are known, a path planning algorithm is employed to navigate the truck to its desired destination without hitting any obstacles. Robotic path planning algorithms like rapidly exploring random trees and graph search logics are generally used for this part. The last part is the motion control block which makes sure the truck is following the planned path by providing inputs to the actuators. Basically, the motion control block drives the truck by estimating/knowing its current state and gives feedback to other algorithms about how efficient the actuators can be used in any given instant. Even a minute error here can lead to an accident and all these four algorithms operate together in a synchronized manner and update at regular time intervals.

2.2 Vehicle motion management

The Vehicle Motion Management (VMM) function performs the motion control function. It makes sure the vehicle can support the speed and direction as demanded by the previously set values and this requires communication with many subsystems in the vehicle on the run. In most autonomous driving architectures the VMM function is separated from the main architecture and is designed to compute a set of high level requests from the main architecture. By doing this, the main function will be relieved from the responsibility of having inherent and detailed knowledge of the actuators and vehicle dynamics sensors. Another reason for doing this is the possibility to isolate algorithms with different requirements on system safety levels. An abstract block diagram of the VMM model developed for simulation and testing by Volvo is shown in figure 2.2. The tasks in the primary control unit calculate the force and coordinate the actuators. The motion estimation estimates the motion of the truck by estimating the different velocity and acceleration of the truck using a state estimation filter. The capability unit estimates the vehicle capability and aggregation level. The vehicle diagnostics unit monitors and controls the health of the vehicle. Even though the scheduling budget for the entire VMM function is 10 ms, a safety deadline of 7 ms is marked as a limit for this block. The state estimation filter is full of matrix operations, which is why this filter was chosen for hardware acceleration.

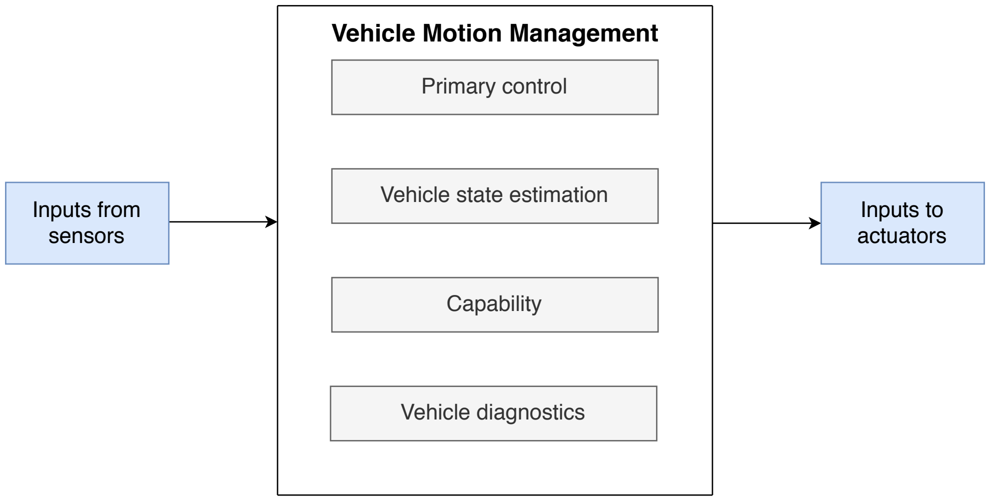


Figure 2.2 – Vehicle motion control

In the Vehicle motion MATLAB model, the state estimation filter estimate longitudinal velocity, rate of change of longitudinal velocity, yaw rate and yaw acceleration, in a four state vector output. The system and noise matrices are aligned in different dimensions to give a four vector output. But in future, eight or 16 states would be considered for more precise state estimation with more parameters.

2.3 State estimation filter

The VMM algorithm is scheduled to happen every 10 ms i.e., every 10 ms new sets of inputs are given to actuators of the truck. Exact state information of the truck like velocity, position, and angle at any instant is needed to compute the control inputs to its actuators at that instant. The actuators will work based on the control inputs and move the vehicle through its planned path safely. This unknown/uncertain state information is calculated using the state estimation filter. For this purpose, three classes of state estimation filters are generally used in almost all applications.

- Kalman filter [5] -Performs well in linear systems.
- Extended Kalman filter [7]-Performs good in mildly non-linear systems but may diverge in highly nonlinear systems.
- Particle filter [8] -Performs good in highly non-linear systems but may suffer in high-dimensional space environments.

The choice of selecting which filter to use also depends on the system model and the application of the filter's output vectors. For example, the filter can be used in sensor fusion to correct the uncertain values. In a perception environment, the Lidar may perform with very good accuracy, but it will fail badly in bad weather. A Radar may not be as accurate as Lidar, but it performs well in bad weather. So these two uncertain values can be fed to the filter to predict the correctness of inputs. In this project, the Extended Kalman filter is not used to predict the correctness of inputs but rather to predict an unknown value from known sensor inputs. This decision was made by Volvo and the project started with a 4 state estimation filter that is used to estimate the longitudinal velocity of the truck using the Extended Kalman filter algorithm.

Figure 2.3 is a simplified version of the filter in VMM MATLAB model. The vehicle parameters are vectors of different sizes. Some parameters like wheelbase, distance from imu to rear axle are constant and parameters like the angle of wheel to road are dynamic depending on the speed of the vehicle. Noise covariance vectors include process and noise covariance vectors, sensor inputs are the parameters with which state estimation has to be done (for this project 13 sensor inputs were used), the initialization vector has the initialization information when the filter resets itself and calculates the information vectors on the run without depending on previous output. This happens when the logic is true for the if-clause branching in figure 2.4. The output of the filter is a vector predicting the unknown state values.

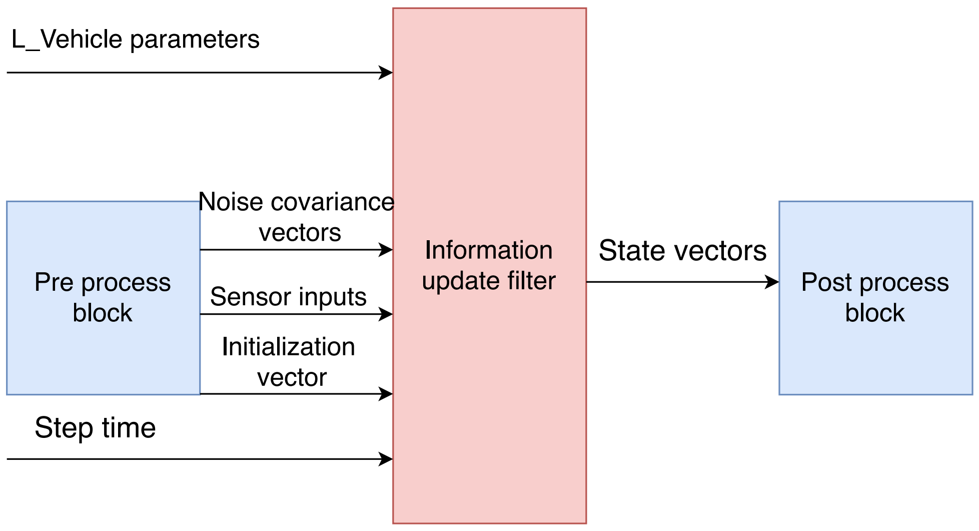


Figure 2.3 – Input and output interface

The filter equations are formed from time domain equations, based on Newton's second law :

$x_t = F.x_k + G.v_t$

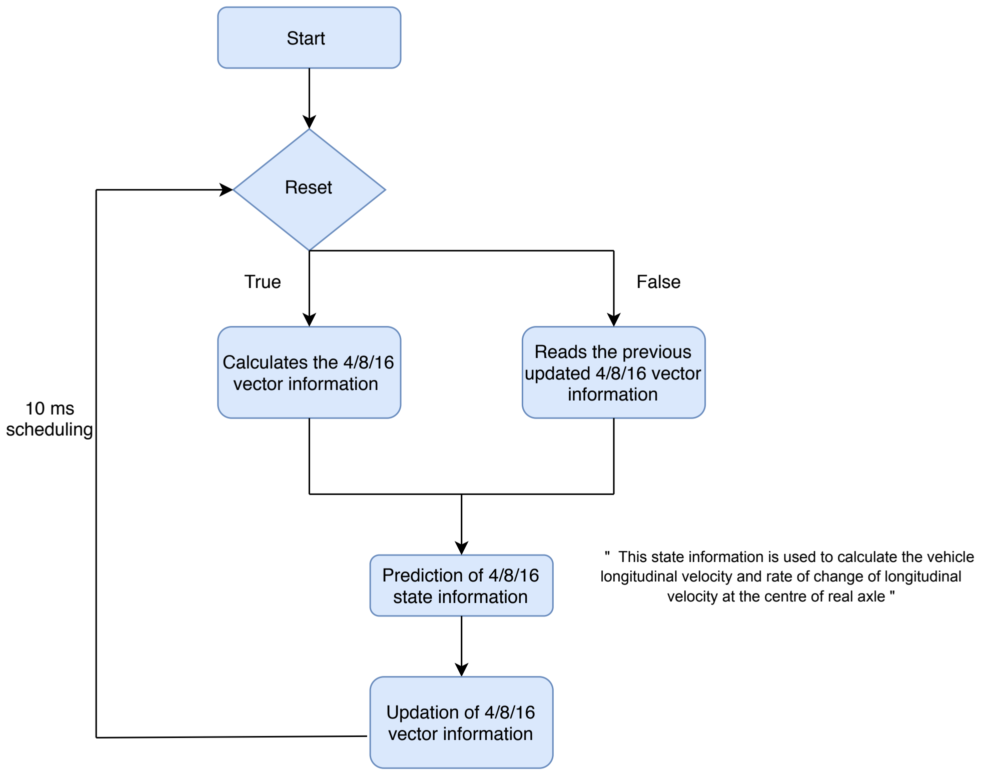
where, x_t is the position and v_t is the velocity at different time instant, F is system matrix (representation of the linear system in a matrix format), Q is the noise matrix (a matrix to represent that the system state changes over time) and G is system state relation to the noise matrix. The relational matrix G is derived from F and Q. Variables F, G, and Q are constant matrices so their time indices are dropped. This equation is used to calculate the state of the vehicle. Equations (2.1)-(2.7) are used to compute the N state prediction vector (the predicted states of the vehicle at a given instant and N represents the number of states being predicted), (2.8) and (2.9) gives the measurement update (N state information vector and (N*N) information matrix to the next iteration of filter). The variable "i" represents intermediate (N*1) information vectors at different time intervals that are used to predict the final state vector and the variable "I" represents intermediate (N*N) information matrices at different time intervals that are used to predict the final co-variance matrix "P". The subscript letter k refers to the sampling instant for each variable. For an N state estimation filter,

The (N*1) information vector is calculated from (N*N) system matrix and previously sampled information vector:

$$i_h = F^{-1}.i_{k|k} \tag{2.1}$$

(N*N) information matrix is calculated from (N*N) system matrix and previously sampled information matrix:

$$I_h = F^{-T}.I_{k|k}.F^{-1} \tag{2.2}$$



Start
Reset
True
False
Calculates the 4/8/16 vector information
Reads the previous updated 4/8/16 vector information
10 ms scheduling
Prediction of 4/8/16 state information
" This state information is used to calculate the vehicle longitudinal velocity and rate of change of longitudinal velocity at the centre of real axle "
Updation of 4/8/16 vector information

Figure 2.4 – Flow chart of State estimation filter

(N*N) intermediate matrix X for predicting new information vector and matrix in current sampling interval:

$$X = I_h.G.(G^T.I_h.G. + Q^{-1})^{-1}.G^T \tag{2.3}$$

Predict the (N*1) information vector for current sampling interval:

$$i_{(k+1)|k} = i_h - X.i_h \tag{2.4}$$

Predict the (N*N) information matrix for current sampling interval:

$$I_{k+1|k} = (F.I^{-1}_{(k|k)}.F^T + G.Q.G^T)^{-1} = I_h - XI_h \tag{2.5}$$

Calculate the estimated (N*N) covariance matrix:

$$P_{k+1|k} = I_{k+1|k}^{-1} \tag{2.6}$$

Calculate the state output in the form of a (N*1) vector:

$$x_{k+1|k} = P_{k+1|k}.i_{k+1|k} \tag{2.7}$$

Compute/update the actual (N*1) information vector from the measured sensor values:

$$i_{k+1|k+1} = i_{k+1|k} + H_k^T.R(y - h.(x_{k+1|k}) + H_k.x_{k+1|k}) \tag{2.8}$$

Compute/update the actual (N*N) information matrix from the measured sensor values:

$$I_{k+1|k+1} = I_{k+1|k} + H_k^T.R.H_k \tag{2.9}$$

The nine equations here are mapped to the blocks of computation in figure 2.5. For a four state estimation, the measurement and update consumes more percentage of the total execution time, because the inverse operation calculations are fast and the multiplications in update sections are 13 dimensional (13 sensor inputs) computations. But for 8 and 16 dimensional estimation the inverse operations in the prediction itself take more time.

Matrix formation: Matrices F and Q are formed based on the sampling time interval T, linear matrix H is formed from the measured sensor input values. When the size of matrix F and Q are changed, G is altered accordingly. From here on, all the matrix operation dimensions change correspondingly and the number of predicted states increases/decreases. In the measurement and update part, matrix H has to be changed according to the format (13*N) where N is the number of states predicted. The matrix formats used when programming a 4 state filter is shown below.

$$F = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Q = \begin{bmatrix} T/2 & 0 \\ 0 & 0 \\ 0 & T/2 \\ 0 & T \end{bmatrix}$$

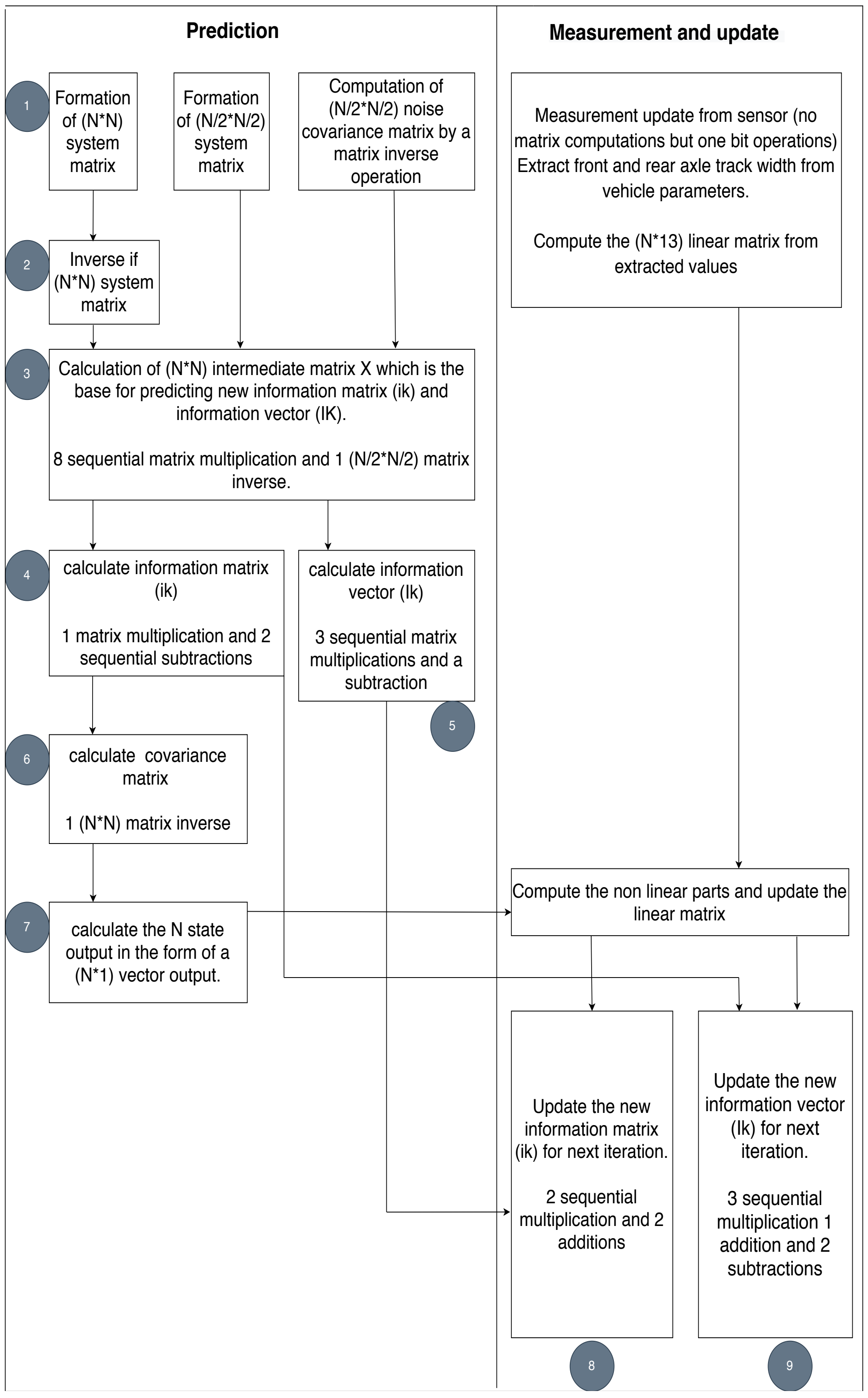

Prediction
Measurement and update
1
Formation of (N*N) system matrix
Formation of (N/2*N/2) system matrix
Computation of (N/2*N/2) noise covariance matrix by a matrix inverse operation
Measurement update from sensor (no matrix computations but one bit operations) Extract front and rear axle track width from vehicle parameters.
Compute the (N*13) linear matrix from extracted values
2
Inverse if (N*N) system matrix
3
Calculation of (N*N) intermediate matrix X which is the base for predicting new information matrix (ik) and information vector (IK).
8 sequential matrix multiplication and 1 (N/2*N/2) matrix inverse.
4
calculate information matrix (ik)
1 matrix multiplication and 2 sequential subtractions
calculate information vector (Ik)
3 sequential matrix multiplications and a subtraction
5
6
calculate covariance matrix
1 (N*N) matrix inverse
Compute the non linear parts and update the linear matrix
7
calculate the N state output in the form of a (N*1) vector output.
Update the new information matrix (ik) for next iteration.
2 sequential multiplication and 2 additions
Update the new information vector (Ik) for next iteration.
3 sequential multiplication 1 addition and 2 subtractions
8
9

Figure 2.5 – Computations in state estimation filter

$$H = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & sensorinput \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & sensorinput & 1 \\ 1 & 0 & sensorinput & 1 \\ 1 & 0 & sensorinput & 0 \\ 1 & 0 & sensorinput & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

2.4 Hardware acceleration

The end of Moore's law has brought a new era in the field of computer architectures with the evolution of application-specific hardware accelerators. All hardware providers are focusing on developing hardwares in an integrated development environment, with abstract level APIs to provide a balance of energy-efficient, high performance, safety-critical, and fast time to market. The three components that are used for computations are FPGA, GPU, and CPU.

Offloading parts of an application to pre-programmed hardware for increasing the efficiency of the overall application is referred to as hardware-software co-design. In FPGAs and reconfigurable devices the logic is put in the form of hardware structures, whereas in GPUs the logic is cross-compiled to execute in different GPUs in a different engineering flow. The parts that are partitioned to execute with FPGA/GPU are the hardware parts and the other computations are the software parts. Technically, the GPU is also a software part that is executing single instruction multiple data on the run but in this context, it is used for accelerating the overall performance. In most cases, hardware acceleration is used to free the CPU from computational load. A SOC will have all the hardwares embedded in it. When the chip is powered on, all the hardwares become active. Efficient load balancing is possible only when the performance of all functions in the overall application is studied with their best and worst-case execution time and this balance is very important when running an autonomous system that involves many different tasks in parallel. The outcome of this thesis will help the task scheduler to understand when offloading a state estimation filter will be efficient.

2.5 Related study

Many researchers have worked with hardware implementation of state estimation filters. In [9], the authors have implemented a similar Extended kalman filter for state estimation in mobile robots. The inputs to the filter directly comes from image sensors and hence the filter has to do more preprocessing. The preprocessing is put in a pipeline stage before the prediction step of the filter. The engineering flow begins with implementing a C code for the overall design and checking the quality of design in a high end intel PC processor, then porting the algorithm to an Intel SOC with FPGA (Nios-II Fast Core 50 MHz) and finally offload only the matrix operations to the programmable logic in the SOC. They concluded the research by saying a speedup of 4x is achieved with FPGA acceleration for a 27 feature estimation filter. But, the application requirements here is different from our need. They are running the filter continuously for 5000 iterations and they deal directly with image pixels which is not the use case for Vehicle motion control.

Reference [10] proposes a Parallel Architecture of the filter for Radar Tracking Applications is evaluated for implementation in FPGA, GPU and multicore CPU. A speed up of 39x is achieved with FPGA and 31x is achieved with GPU. A similar filter for multi target tracking is used in [11] where the authors designed a hardware architecture with just 6 multipliers, 2 dividers, 9 adders, 5 subtracters, one control unit, some registers and multiplexers for pipeline and control but they have a precision loss of 8.9% for one iteration and the size of matrices are not discussed in the paper, instead it focuses on the area and power to achieve a reasonable accuracy. The whole design has a gate count of 33,424, total LUT count of 3653 and 2210 number of occupied slices. However, the filters designed for radars are employed in tracking systems to measure the angular co-ordinates and most of them involve high dimensional matrix operations. For an autonomous truck its a different level of non-linearity and less matrix operations.

Fonseca et al. in [12] has designed a filter with similar equations like this project and focus on estimation of velocity. The authors have implemented the functionality in FPGA in a LABVIEW environment and they conclude by saying FPGA can give 3x speedup but they fail when the dimensions go high because of resource scarcity.

A 7398x speedup of Kalman filter with 18 matrix operations is shown in GPU as compared to a single thread CPU by Huang et al in [13] . But this filter has 4500 state dimensions and is used for a different application. A speedup of this factor is possible only when matrix operations are in high dimensions because all the operations of a 4500 state estimation will be 4500*4500 matrix/vector operations. In this project, the matrix dimensions are small and task level parallelism is not possible so speedup is no close to the theoretically possible but there is going to be a GPU in the autonomous truck/system and if it is idle, the question is how efficient it is to offload the filter to GPU so the CPU can do any other job parallelly is the question.

In [14] Lin et al. proposes a different approach to speed up a smaller dimension kalman filter in GPU by parallelizing different iterations of the entire filter itself and giving a maximum likelihood output. For example, the filter should not operate for 9 iterations but collect the sensor observations in an allocated GPU memory and at the 10th iteration, the filter is operated on all 10 combined sensor observations, thereby parallelizing the entire filter itself. In Volvo's model, a set of sensor observations comes to the filter every 10 ms, and the filter is expected to give output to the next stage before it receives its next set input. If this [14] method is implemented then the entire model has to be altered so it is not considered. However, this is an efficient proposal to speed up smaller dimension filters and it should be considered if there is a scope to alter the scheduling algorithms with a trade-off in the quality of state estimation.

Chapter 3

Hardware Platforms

This chapter includes a detailed account of the hardwares (previously presented as an abstract in Section 2) that has been used for testing and running the application explained in Section 4. The content to expect from this section can be broadly split up into the following;

1. A brief description of the SOC with ARM embedded processors, Programmable logic and GPU architectures.
2. A note on the APIs, software tools and board support packages used to program these hardwares.

3.1 Processor and FPGA

The processor and FPGA used for this project are embedded in a single board designed by Xilinx as shown in figure 3.1 specific for automotive applications which includes image processing, motion control, advance driver assistance system (ADAS), and edge computing. A simplified version of the Multi-Processor System-on-Chip (MPSoC) is shown in figure 3.2 architecture. The MPSoC has a quad-core application processor, two real-time processors, FPGA programmable logics and an arm mali GPU. It is capable of running multiple applications parallelly. For example, motion control could be run in the real-time processor when an IOT edge computation could be done in the four A 53 processors. Basically the application processor is the high-performance processor. The real-time processor can be used for safety-critical and timing specific applications.

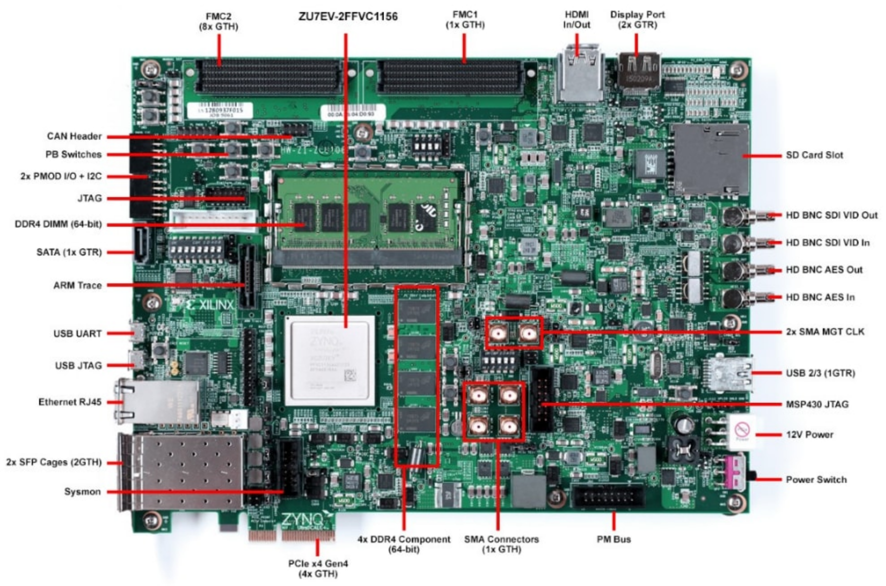


Figure 3.1 – Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [1]

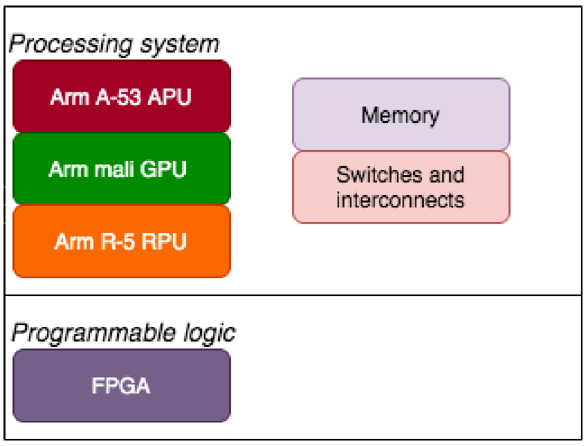


Figure 3.2 – Simplified SOC architecture

The board comes with an integrated development environment toolbox called SDSoC which eases the Intellectual Property (IP) integration work with the processor in an HW/SW co-design. This is done by hardware-specific pragmas that generate the Advanced eXtensible Interface (AXI) bus interfaces between processors and programmable logic. The SDSoC is a natural fit for design teams consisting of software and hardware engineers because it enables to take advantage of programmable logic device in a software development environment [15]. Xilinx has advised the users to port their application on a decision tree basis as shown in figure 3.3 to the hardware on the board.

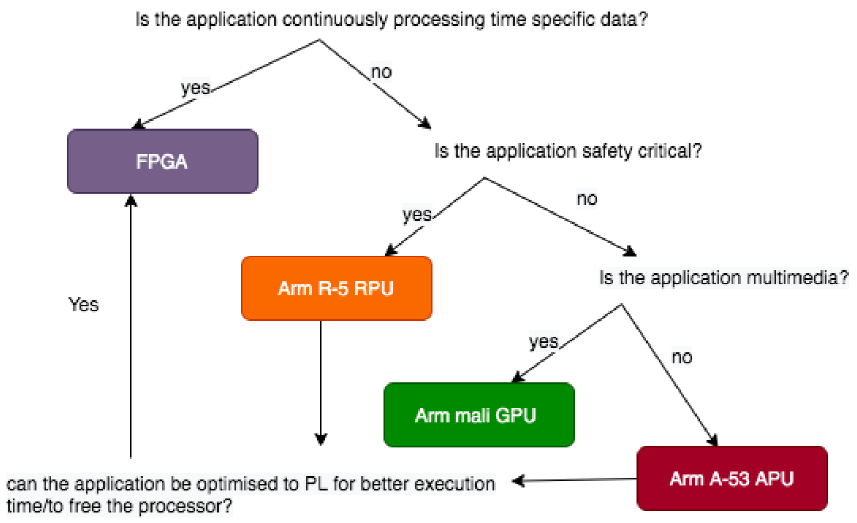

Is the application continuously processing time specific data?
yes
no
FPGA
Is the application safety critical?
yes
no
Arm R-5 RPU
Is the application multimedia?
Yes
yes
no
Arm mali GPU
Arm A-53 APU
can the application be optimised to PL for better execution time/to free the processor?

Figure 3.3 – Decision tree for application porting

For this project the HW/SW co-design of the filter itself was not investigated because the research question was about offloading the filter computations from the CPU. So, the full filter computation was ported to both processor and FPGA separately without any partition and the timing performance was analysed.

The SOC operates in 4 modes of power [16]: LPD-Low Power Domain when Application Processing Unit (APU) and GPU is powered off and the SOC is operated with RPU and its on chip memory; FPD-Full Power Domain when all hardwares in the SOC are running along with external peripherals including display port and Peripheral Component Interconnect (PCI) express; PLPD-Programmable Logic Power Domain when only the PL with its interfaces are operating along with on chip memory and serial interfaces; BPD-Battery Power Domain when the SOC is powered by an external battery to keep the real time clock with external crystal oscilloscope running with a battery backed block RAM to maintain time. This is used when the device is off. In this project, all the modes were tried and there is no impact/change on the observed results because the hardwares are operating in its full potential if it is switched on. However, there was a difference in the number of cycles for data communication when different configurations of AXI bus communication were used. The different configurations are shown in table 3.2 in section 3.1.3.

3.1.1 Quad core Arm cortex-A53 processor

The APU acts as the general-purpose CPU in the MPSoC architecture. It has four ARM A-53 cores each operating at 1.5 GHz. Each core has its own 32 KB L1 instruction and data caches. A common L2 cache of 1 MB is available for all 4 cores. Access to L2 cache is provided by a Snoop control unit inside the APU. The cores are developed for running embedded applications with NEON and floating-point computing units which brings acceleration benefits for DSP and media applications.

A dedicated memory management unit is also assigned for each core to translate virtual memory to physical memory addresses. To enable HW/SW co-design, the APU has an Accelerator Coherency Port (ACP) that brings in data from the programmable logic not just faster but also coherently.

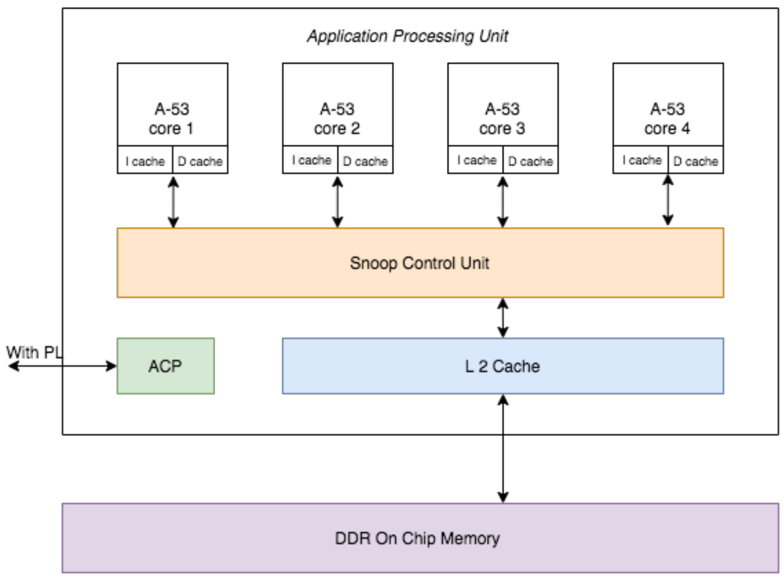


Figure 3.4 – Application Processing Unit

3.1.2 Dual core Arm cortex-R5 processor

The Real time Processing Unit (RPU) has two ARM R-5 Processors cores, each operating at 600 MHz. This is not designed for running intensive algorithms instead safety-critical applications are scheduled to run here because it has a dedicated 128KB Tightly Coupled Memory (TCM) for each processor. This TCM stores instructions and data from the Double Data Rate (DDR) DRAM one time for any dedicated application and makes sure the processor does not starve from cache problems. In addition to TCM, each core also has a 32KB instruction and data cache like the processors in the APU. A dedicated on-chip memory of size 256 KB is available in the RPU to perform memory access without the DDR DRAM which the APU will generally be accessing. The interrupt controllers are inside the RPU but in the APU they are placed outside. Also, the peripheral ports interfacing each core in the RPU is designed to give a low latency output compared to the APU. The SDSoC environment did not provide a direct connection between the RPU and the programmable logic block but it is possible by to write a driver in linux to interface with one of the RPU's connection peripherals. An abstract diagram of the RPU block inside the SOC is shown in figure 3.5

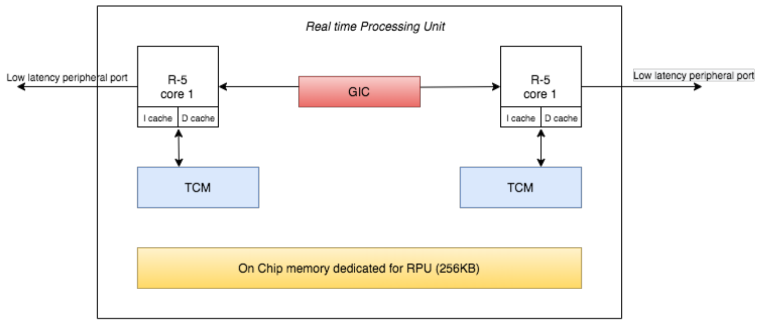

Real time Processing Unit
Low latency peripheral port
R-5 core 1
I cache
D cache
GIC
R-5 core 1
I cache
D cache
Low latency peripheral port
TCM
TCM
On Chip memory dedicated for RPU (256KB)

Figure 3.5 – Real Time Processing Unit

3.1.3 Xilinx Programmable logic

The programmable logic is also called fabric accelerator which is designed for optimal performance/watt, with faster transistor ON/OFF switching speeds and lower power leakage when operating at lower power. In addition to the hardware itself, the dedicated compiler Vivado has functionalities to adjust clock skew during the hardware design. It also has extra pipeline analysis to find the sequential loops and parallelize them automatically [17].

The Programmable Logic (PL) block in SOC has storage and signal processing blocks as shown in table 3.1 to perform computations and mathe-matical operations, Programmable Input-Output ports to communicate with external interfaces ranging from simple one pin logic to high-speed ethernet serial protocols. A communication can be established between PL and Processing System (PS) or directly between PL and any other external interface. This helps us to create many IPs of different functionality in the PL block if we manage to design resource efficient IPs. Another advantage of having so much resources is to have many IPs of same functionality but operating at different clock frequency. This architecture can be used for energy efficient overall application.

Resource	Available
Flip Flops	548120
LUTs	274080
DSPs	2520
BRAM	912

Table 3.1 – Resources available for PL in ZCU102

Flip-flops and look up tables are basic building blocks called configurable logic blocks (CLB) of an FPGA. Flip-flops act as memory storage elements /registers and Look up tables (LUTs) act as truth tables to produce the correct output for a pre defined input sets. In most Xilinx devices, block RAM is added to the FPGA fabric as dual port RAM modules to store more data for computations because the flip-flops can only store one bit at a time. In addition to this, DSP blocks are integrated with the PL block because the floating point operations are slow with flip-flops and LUTs. The use of DSP can accelerate the signal processing applications which use floating point operations.

The PL and PS communicate with the ARM AXI bus interface [1]. The ACP in figure 3.4 provides a connection from the L2 cache in the APU to the PL and if there is a L2 cache miss, then the memory transfer latency will increase. This is why it is advisable to store the memory in on chip PL memory. The AXI busses are configured based on the power mode, data size and cache coherency requirements and this is discussed in [1], chapter 15. Table 3.2. shows different AXI configurations that are described in detail in the technical reference manual [1]; all versions were used for this thesis and is important when considering energy utilisation of the SOC. The PL cannot be connected to the RPU in the SDSoC environment, but can be configured by writing a seperate driver API everytime the slave initiates a read/write request in the bus.

Bus name	Description	Master	Slave
S_AXI_HPC0_FPD S_AXI_HPC1_FPD	High Performance cache-Coherent Ports	PL	PS
S_AXI_HP0_FPD S_AXI_HP1_FPD	High Performance Ports for the FPD	PL	PS
S_AXI_LPD	LPD AXI port	PL	PS
M_AXI_HPM0_LPD	High performance Master port for LPD	PS	PL
M_AXI_HPM0_FPD M_AXI_HPM1_FPD	High performance Master port for FPD	PS	PL

Table 3.2 – AXI configuration provided by SDSoC to communicate APU with PL

3.2 GPU

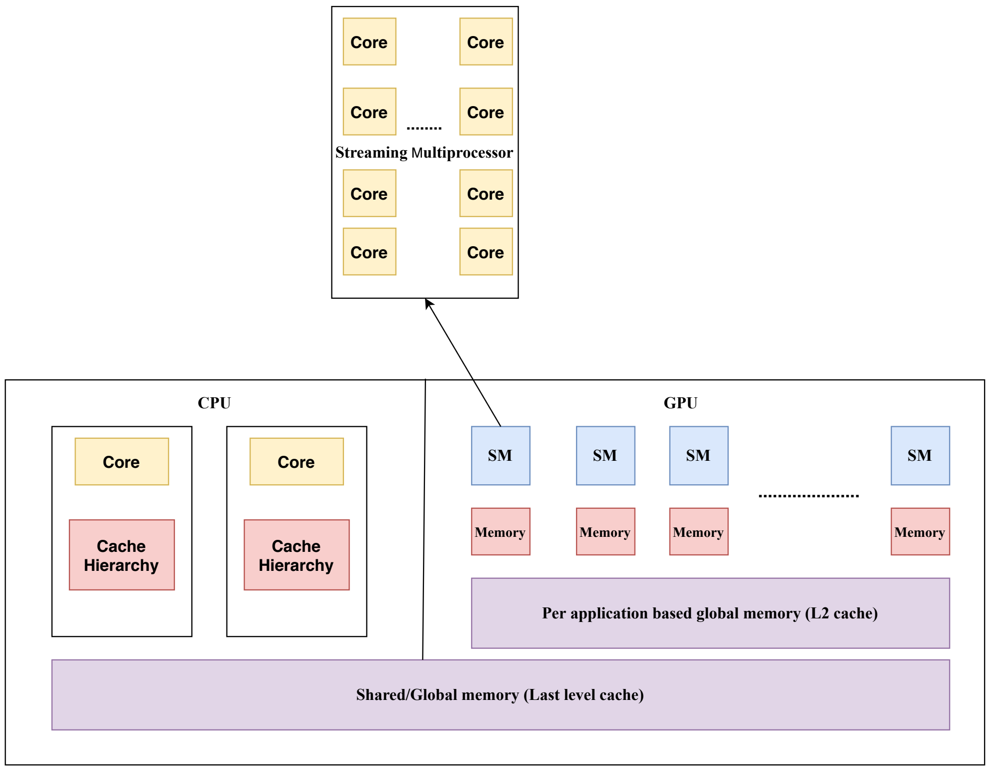


Figure 3.6 – GPU Architecture

The use of GPU for computation is a major breakthrough in research and product development for both academia and industry. It provides a platform for massive parallelization of a single instruction by enabling multiple processors to run in parallel, thereby reducing the execution time of an application. Unlike FPGA a big advantage of GPU is the reuse of hardware for different applications. The functionality can be developed in different host environments and the cross-compiled instructions are stored in on-chip memory and are used only when needed. Whereas, in an FPGA the hardware resources have to be dedicated to a particular application. A simple architecture diagram of a CPU-GPU embedded architecture is shown in figure 3.6. A GPU generally shares memory with the CPU and depends on the CPU for control instructions. Once it receives the instruction from the CPU it splits the instruction and data to streaming multiprocessor blocks. Each streaming multiprocessor block has a number of dedicated cores and a common memory for all its cores to perform the same instruction on different data. These streaming multiprocessors are referred to as CUDA cores in NVIDIA's proprietary parallel computing program model CUDA [18].

.

A CUDA is basically an API that NVIDIA has developed to program its GPU. This API provides software engineers with a friendly design approach to program the GPU. A programmer does not have to worry about the internal GPU architectures of Streaming multiprocessor (SM) and the number of cores in each SM but can program with an understanding of blocks and threads [19]. These blocks and threads are allocated to cores by warp scheduling which is taken care of by the CUDA API. The cores execute the threads in a group of 32. This 32, is a constant number for CUDA programming and is called the warp size. This is why the allocation of threads by a programmer should always be in multiples of 32. This unique feature in the CUDA programming model helps make CUDA programs portable. The same code can be ported to any NVIDIA GPU device and the CUDA API ports the logic in the best possible way according to the device architecture. CUDA itself is not a programming language but is written mostly in C/C++ along with the CUDA API functions and the file is saved with a .cu extension.

Initially, the assignment was planned to be conducted with the GPU in Drive Px2. The Drive Px2 hardware is a SOC environment which has ARM CPUs and pascal GPUs, everything embedded and is currently being used by the Research and Development team at Volvo. But Nvidia recently announced a similar machine, NVIDIA DRIVE AGX Pegasus which has a Turing

GPU architecture with faster memories and processing capabilities than the Pascal architecture. Turing is an advanced version of the Volta architecture - A detailed study with micro benchmarking on the performance, memory management and latency cycles with CUDA applications is described in [20] for all NVIDIA GPU architectures and this motivation was convincing to port the filter in a Turing GPU and compare its performance with Pascal. For this, a Quadro T1000 (Turing) GPU was considered in this project.

3.2.1 NVIDIA DRIVE PX2

The NVIDIA Drive Px2 is an autonomous driving specific AI supercomputing machine which has two Tegra architecture SOCs with 12 ARM-57 CPU cores and two discrete pascal GPUs. The device comes with a driveworks SDK which is capable of running all perception, localization, planning and visualization algorithms that an autonomous vehicle needs to operate. It has all the interfaces from HDMI cables to CAN bus ports that a car/truck will need to connect. It is used to develop and train applications, validate the trained algorithms in the car/truck in a real world environment, and prototype it in comparison with the engine control unit. It does not have a programmable logic but the discrete pascal GPU is used as a hardware accelerator for computationally intensive blocks of the application. Both the GPUs have a dedicated memory subsystem available. NVIDIA has eased the work of users by integrating the Linux and AUTOSAR environments in this supercomputer in addition to the CUDA libraries that other NVIDIA devices have.

For this project, the scope is to find/compare if offloading the filter to the hardware accelerator (discrete Pascal GPU) would be beneficial if the VMM algorithm is chosen to run in Drive PX2. So, All features of this supercomputer were not explored. Instead the filter was made to run with CUDA in the discrete GPU with the help of Linux in the on-chip Drive Px2.

3.2.2 NVIDIA Quadro

NVIDIA DRIVE AGX Pegasus is an advanced version of drive Px2, which may be considered for development in Volvo soon. This supercomputer is using the Turing architecture GPU. So, the Quadro T1000 GPU that is based on Turing architecture is used in this project to study the performance between Pascal and Turing architectures.

The Turning is an advanced version of Pascal with independent scheduling and instruction cache memory. Tensor cores and Ray tracing cores are introduced in the Turing architecture for deep learning applications but they are not used exclusively in this project. Each core inside the SM block in figure 3.6 has its own register and L0 instruction cache memory, whereas in Pascal, the cores inside an SM block have to share a common instruction cache. Figure 3.7 shows the architectural difference in memory and cache system between the Pascal and the Turing GPUs. Keeping the shared memory coupled with L1 caches allows the applications to decide if it can store the instruction in shared memory or if it should go one level down in the cache [21] during the run-time. Table 3.3 shows the difference in speed and memory between GPUs used.

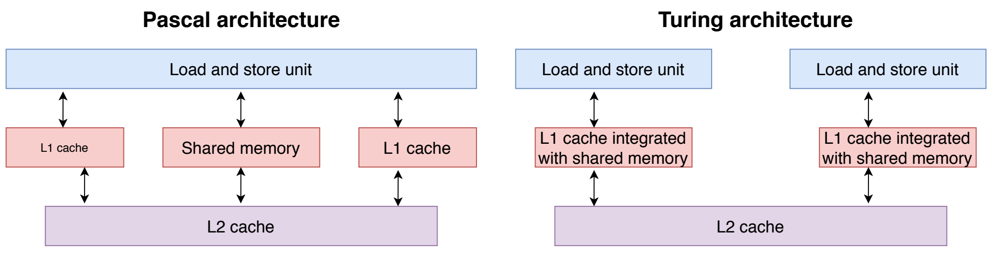


Figure 3.7 – Pascal vs Turing memory and cache difference

Specification	Drive PX2	Quadro T 1000
Core clock rate	1290000 KHz	172500 KHz
memory clock rate	3003000 KHz	400100 KHz
Total SMs	9	14
Shared memory per SM	98304 bytes	65536 bytes

Table 3.3 – Specifications of the GPUs used

Chapter 4

Implementation and Results

This chapter contains the methods used to explore the hypothesis and address the problems (previously presented as an abstract in section 1.2). The content to expect from this section can be broadly split up into the following;

1. MATLAB version of filter and its performance in intel i7 processor running at 2.8G Ghz.
2. The use of MATLAB parallel computing toolbox to access GPU with a matlab script.
3. Implementation of the filter in two different embedded processors.
4. Implementation of the filter in two different GPUs with CUDA programming model.
5. Implementation of the filter in FPGA using Vivado High level synthesis.

4.1 MATLAB

4.1.1 Filter design and MATLAB profiling

The filter was designed by Volvo Autonomous system using the Extended Kalman filter algorithm as discussed in section 2.3 and this is a four state filter. For performance analysis and research purpose, an eight state and 16 state filter was developed with the same algorithm and design flow as shown in figure 2.5. MATLAB has an interactive profiler that gives a full picture of the execution times of each line in the script. A test script was written for all three filters and the profiler was executed after blocking all other activities of the CPU.

The model now has input datas coming from the test script and the lines of code being profiled by the profiler. The Profiling report showed the inverse function used is the bottleneck in this filter. There are four occurrences of the inverse function and they are easy to implement in MATLAB because of the inbuilt library functions. The MATLAB profiling of the filter however is not taken into consideration when benchmarking the execution time because this script is a high-level script running on an intel i7 processor which is operating at 3GHz frequency. This frequency results in high power consumption, which is not feasible in any embedded processor and is out of scope for this project too. MATLAB is a scientific computing tool used to model algorithms in the initial stages of design. So, the profiler information is used to find bottlenecks of the algorithm and as reference timing numbers.

4.1.2 GPU computing toolbox

To port the filter functionality to a GPU, MATLAB has a parallel computing toolbox that enables multi-core processing without Message Passing Interface (MPI) programming and has dedicated libraries with which the GPU in the computer can be accessed without CUDA/C. This enables high-level programmers to perform computationally intensive parts of their Matlab script in a GPU making use of massive parallelization threads without spending time in writing in a low-level language like C/C++, with only slight modifications in their code. However, this is not very accurate in terms of timing as implementing functionality to the GPU with CUDA because the CUDA supported library APIs have much faster data transfers. But, as a hardware engineer working with hardware accelerators there are a lot of options to explore. This toolbox helps in better understanding and validating the hypothesis about the range of parallelization a logic can explore before jumping into implementing application-specific hardwares. The toolbox also helps engineers to explore parallelism not only inside an algorithm/block but also possible task-level parallelism in their Simulink models. The toolbox is supported by an interactive parallel code profiler to profile individual operations.

Complex MATLAB functions like inverse and pseudoinverse are widely used in big data processing, batch processing cloud computing, and robotics algorithms. These functions are implemented in MATLAB using inbuilt library codes. They are very important and challenging for any engineer who is aiming to speed up the execution time of his/her algorithm in hardware. In this thesis, the toolbox was used for initial profiling of the information filter update algorithm as the first step of GPU profiling. Similar to malloc in a processor or a Cuda malloc in a GPU the Matlab parallel computing toolbox has a function gpuArray() which creates memory space in the GPU and transfers data from the processor to the GPU. Once the data is passed, the computations happen in the GPU. The bottleneck in any GPU application is the efficient use of data transfer bandwidth and it is the same here. Transferring one data using the gpuArray function for a filter like application is not as efficient as transferring 1000 data in a single gpuArray call for a convolution neural network application. This will make the communication time larger than the computation time. But, the computation time can be analyzed separately with an inbuilt interactive profiler and compared with the processor computation.

There is not much task-level parallelism in the filter because it is very sequential and so parallelization can be achieved in operations only. For-loops are the area where parallelism can be explored and this is evident from the profiling results. As mentioned earlier, the communication time is large than the computation time and the performance with MATLAB script itself is not very efficient as implemented in CUDA because the CUDA libraries make faster data transfers and the .m script is a high-level programming script that has many layers before it communicates with the hardware and they are only iterpreted, not compiled.

4.2 Processor programming

The filter functionality is tested in the MATLAB toolboxes for 100 different input sets from the CAN bus where the outputs for the corresponding input sets were recorded. The next step is to program the processors, i.e., the APU and RPU as discussed in section 3.1.1. For this, three C codes were written in MATLAB for the three filters and was debugged using the Visual studio compiler on a host PC. This C code is used as base for porting the program to APU, RPU, FPGA and GPU. When many software teams are involved in developing a product, it is better to benchmark the results separately in all possible hardwares. So, in the final stage of the project it will be easy to schedule the functions in the best possible way by giving priority to safety critical functions. Otherwise, the processing units may potentially interfere [22] with each other and not perform as expected. For this reason, the functionality of the filter functionality was implemented in both the APU and RPU and both timings were considered.

As shown in figure 2.5, the filter is filled with matrix arithmetic operations which can be implemented by simple for loops, but the inverse operation becomes tricky. There are two N*N and two (N/2,N/2) inverse matrix operations for one iteration of an N state filter. In MATLAB, the inverse matrix operations are performed by an inbuilt library function. To solve the matrix inverse operation four methods were used: 1. Inversion by analytic formula; 2. Inversion by Cholesky decomposition matrix; 3. Inversion by bubble sorting; and 4. Inversion by the Gauss-Jordan method. However, only two were considered for profiling because those had the best case execution times, and were suitable for implementation on the selected hardware. The bubble sort algorithm was not easy to port in FPGA, because the High level Synthesis (HLS) tool did not support recursive functions so this was eliminated from the implementation. Comparing Gauss-Jordan and Cholesky decomposition, the former was better in terms of fewer number of cycles for computation in all the 100 test inputs. The implementations in a related study [13] used Cholesky decomposition because it uses LU decomposition, which is parallelizable, but the results proved different for this project, because of the small matrix sizes. The Cholesky method might perform better for higher dimensional matrix inverse operations. On an average, the A-53 took 10 microseconds to execute an eight matrix inverse operation in Gauss-Jordan method, but 17 microseconds for the Cholesky decomposition. Algorithm wise, Cholesky is more parallelizable than Gauss-Jordan but the best case functional code in CPU is the base for programming the accelerators so Gauss-Jordan implementation was chosen.

MATLAB uses a persistent variable to store the previous information of the filter and this is very important because if the filter reset is not enabled then the previous information vectors will be used for the prediction block operations. In C, this was implemented by using declaring a global variable for the information vectors which gets updated in every iteration of the filter.

- By formula:
  This analytic method presented in [23] was used to calculate the matrix inversion of the small matrix. A 2*2 matrix can be solved using this formula but as the size increases, the number of elements for computation increases and it becomes difficult to solve the inverse operation by just swapping the matrix position. There are two 2*2 matrix inverse operations in a four state filter and they are solved by

using the formula.

$$\mathrm{M}^{-1} = \begin{bmatrix} a & b \\ b & z \end{bmatrix}^{-1} = \frac{1}{az - b^2} \begin{bmatrix} z & -b \\ -b & a \end{bmatrix}.$$

Even to solve a 4*4 matrix inverse, the above formula was used but by calculating the determinant and adjoint of the M matrix instead of swapping the numbers. So, for the four state filter, the inverse was implemented without much complexity and with less number of instruction cycles.

- By Gauss-Jordan method:

```
Data: N*N square matrix
initialization;
read the N*N matrix;
Initialize an N*N unit matrix;
Form the augmented matrix from two N*N matrices ;
Apply Gauss elimination on the augmented matrix ;
while Gauss elimination do
    Switch any 2 rows;
    Multiply each element of row by a integer;
    Add 2 rows together;
end
Result: Inverse of N*N square matrix
```

Algorithm 1: Gaus -Jordan matrix inverse using Gauss elimination

The Gauss elimination [24] is performed by doing the three points in the while-loop till the other half of the augmented matrix becomes a unit matrix. For a larger dimensional state estimation, 80% of the total latency is with the prediction task. This is because of four inverse matrix computations that occur in the prediction task, and they cannot be paralallized at task level because they are sequential but the inverse operation itself can be parallelized.

4.2.1 APU Profiling

The processor can be profiled in both Linux and a standalone environment, a detailed description of the core itself is in the technical manual of the cortex-a53 mpcore processor [25]. As shown in figure 3.4, the APU has four uniform A-53 cores with separate level 1 cache and shared level 2 cache system. The application was profiled in standalone environment. The level 1 cache has separate instruction and data memory space with a cache line length of 64 bytes each. The level 1 instruction cache is 2 way set associative, the (level 1) data cache is four way set associative. This is because it is an application specific processor where data will be in more demand than instructions. The best case execution time would be when the filter has a dedicated core with the cache subsystem all for itself, which is practically not possible when running the entire autonomous system software model.

These four cores are the brain of the entire system. It has to give instructions, manage data and take decisions for the overall system. For analysing a worst case scenario, ARM allows the programmer to disable the cache memory and run the application in a bare-metal environment. The cache pipeline architecture is different for different ARM processors. For A-53, disabling the instruction/data cache will disable its access to both the levels of cache i.e, the processor has to go all the way down to the RAM memory for fetching instructions every time if instruction cache is disabled, likewise for data with data cache disabled. The C code is compiled for a standalone bare metal platform with a clock counter header file to profile the execution time of the filter. The executable binary file is copied in an SD card, then the application is executed through a terminal window of SDSoC. The caches were disabled by using a Xilinx defined header file "xil_cache.h" that has built in codes to eliminate the cache use when the following functions are called:

- Xil_ICacheDisable()- Disables the entire Instruction cache ;
- Xil_DCacheDisable()- Disables the entire Data cache ;
- Xil_L1CacheDisable()- Disables the entire Level 1 cache ;
- Xil_L2CacheDisable()- Disables the entire Level 2 cache ;

All the mentioned functions were used inside the C code and the execution time was profiled for the three filters based on the number of CPU cycles the core runs for producing the result of one iteration. The execution time was also profiled in a PetaLinux environment by using the "unistd.h" header file to measure the number of clock periods the core operates for one iteration. The obtained results are shown in figure 4.1. Detailed analysis of the functionality can be visualized in real time by connecting a UART and JTAG from the ultrascale board to the host computer with SDSoC software kit by using printf statements wherever a visualisation is needed.

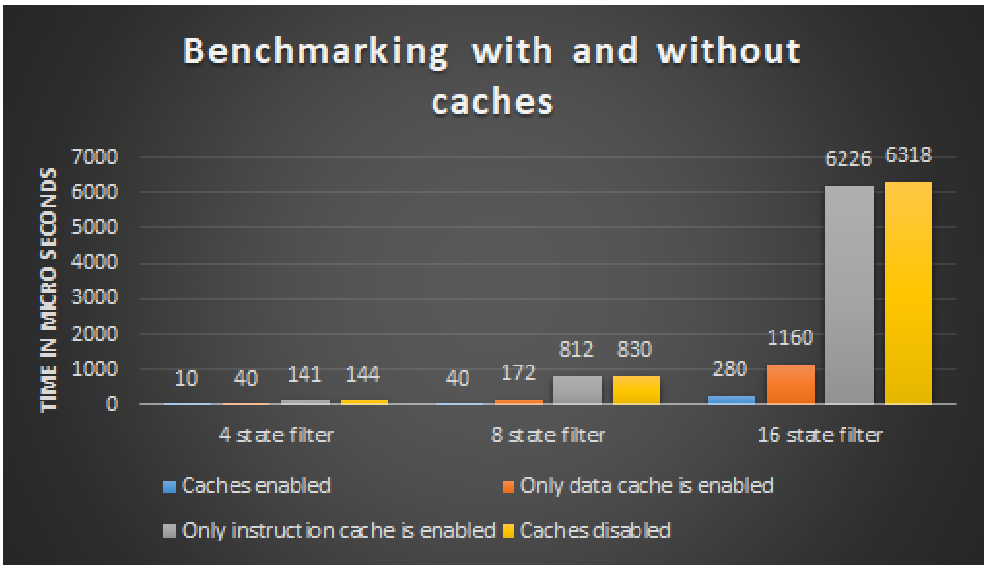


Figure 4.1 – A-53 Best and worst case execution time

The numbers in figure 4.1 give another interesting insight saying that data caches are more important for this project than instruction cache because the data dependencies during the run time of the filter is higher than the instruction dependency. A difference of 20 times is seen between the best and worst case scenarios. The numbers in real time depends on other applications running parallelly on the same core. For example, if a deep learning neural network algorithm is running in parallel with this filter then it will cause many data cache misses, because both involve operating huge amount of datas for the same instructions, thereby rewriting the intermediate datas that the filter would need and the core has to go down to the memory for that data, thereby conceding more latency cycles.

4.2.2 RPU Profiling

The RPU is operating at a lower frequency than the APU but is designed to meet real time deadlines. It has faster pipeline stages and memory transfer cycles to external interfaces. As shown in figure 3.5, it has two cores with separate instruction and data caches. In addition to this, both cores have a dedicated TCM memory to store data and instructions near to the processor itself. Even-though they are influenced by other applications, the data and instructions are readily available in the TCM. There is no best or worst case execution times here. When compared to the APU profiling in figure 4.1, the execution time is 68 microseconds for the four state filter, 296 for the eight state and 1885 for the 16 state filter. This is six times worse than the best case of the APU but it is six times better than the worst case of the APU. The timings of the RPU is constant. So, the integration engineer can confidently schedule the filter application by knowing its predictable execution time in a time critical embedded system environment.

4.3 CUDA Programming

Initially GPUs were used to increase the graphical output of a system by providing assistance to the CPU. Their integrated circuits provided a fixed rendering pipeline giving access to do step by step tasks [26]. As they became more competent, programmable stages were added allowing them to execute programs in itself. The programs are executed in the form of kernels. CUDA and Open Computing Language (OpenCL) are the two main toolkits used to program a GPU. OpenCL is a wider and more general purpose platform whereas CUDA is NVIDIA specific and more efficient in terms of developement, performance and execution. A short note on the terms used for CUDA programming from the book [27] is discussed below.

- Kernel: It is similar to a function in C/C++. The kernel function call invokes the GPU to perform the computation. The CUDA program will run in the CPU if the function is defined without the kernel keyword. In this project 37 kernels are used for the four state filter. For the other two filters, 39 kernels were launched.
- Grid: A grid refers to a collection of blocks, it is purely for a programmers understanding. It can be in 1, 2 or 3D. For this project, all the kernel launches were of 2D grid size.
- Block: A block is a group of threads of uniform size, this is also for a programmer's point of view. The group is a multiple of the warp size as mentioned in section 3.2. Each block will be executed in a single SM during CUDA runtime, and each thread in a block can be accessed.

- Thread: It is the smallest operation that can happen in a GPU. For example, in this project, a kernel launched for 16*16 addition will have 16 threads operating parallely inside 16 cores in one cycle, inside a SM. Sequential/parallel execution of threads happen in a block.

  The details of the kernels in each filter is given in section 4.3.5.

4.3.1 Matrix operation parallelization

The filter is a sequential execution of functions but the functions are full of matrix operations. Parallelizing the addition, multiplication, and subtraction matrix operation is the first strategy to reduce the computation time. This is an efficient way of programming the GPU, because when a GPU is switched on, all the cores become active and consume power even if it does not get any instruction to execute. This is why core utilization is important in GPU programming and maximum core utilization can be achieved by doing maximum number of floating point computations in one cycle. For a CPU to compute the Z in (1), it will require 16 floating-point addition operation cycles, but a GPU will perform the 16 operations in one cycle if the programmer launches 32 threads in one block (threads in the block are multiples of 32-warp size). For larger matrix dimensions the speedup will be even more.

$$Z = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

$$\mathrm{Z} = \begin{bmatrix} A_{11}+B_{11} & A_{12}+B_{12} & A_{13}+B_{13} & A_{14}+B_{14} \\ A_{21}+B_{21} & A_{22}+B_{22} & A_{23}+B_{23} & A_{24}+B_{24} \\ A_{31}+B_{31} & A_{32}+B_{32} & A_{33}+B_{33} & A_{34}+B_{34} \\ A_{41}+B_{41} & A_{42}+B_{42} & A_{43}+B_{43} & A_{44}+B_{44} \end{bmatrix} ..(1)$$

From the device query, the maximum amount of threads that can be allocated to one SM is 2048 for both GPUs. This means, only 2048 threads will be executed at a given time no matter how many threads we allocate. A parallelisation to explore here is to overlap the kernel execution and data transfer. This is possible by assigning more threads to the SM and make it wait. At a given moment only 2048 threads can execute in one SM, so we can use this time to transfer data from the CPU for other computations in parallel. The thread-block balancing should also be noted when performing this parallelisation. Launching more threads in one block will make the SM wait and is a bad idea so the strategy is to launch small kernels in many blocks so many SMs will be running parallely.

But, in this project the maximum threads that can be made to run in parallel would be 16 in a 16*16 multiplication. The dimensions are simply too small to fill the GPU threads so the full potential of the GPU is not used in this project. Parallelisation strategies for very large matrices and the idea of launching more threads to keep the cores busy is explained by Y.Sun and Y.Tong in [28].

4.3.2 Memory copy minimization

The overall performance of the GPU not only depends on the computation but also on efficient memory transfers. Especially in these type of projects where the GPU is considered. Not because it will be faster than the CPU but because by offloading the work, the CPU can take another load. Inefficient memory transfers can cause a disturbance in the CPU's cache and might result in additional problems.

For this, the data dependency between CPU and GPU should be analyzed. In this project, the best possible execution is to get all instructions and data from the CPU in bulk for one iteration and perform the computation giving back only the final state output to the CPU. But this is not the case in all applications where data is huge, and instructions depends on the output of intermediate results. Figure 4.2 shows the execution timeline of the filter where memory is collected only once from the host. During the execution in GPU the CPU memory is left undisturbed. After the execution of all kernels the output data is given back to the CPU.

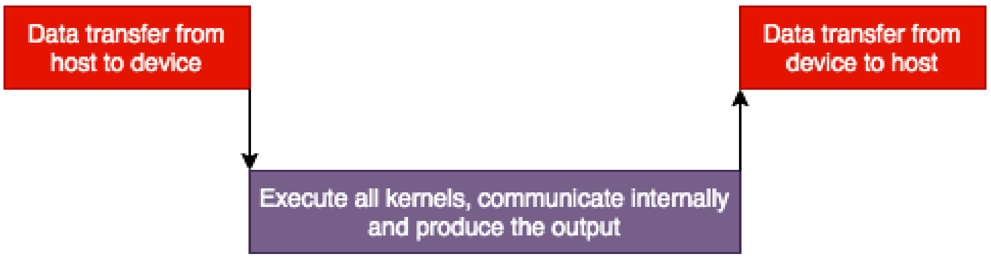

Data transfer from host to device
Data transfer from device to host
Execute all kernels, communicate internally and produce the output

Figure 4.2 – CPU-GPU memory transfer

4.3.3 cuBLAS and cuSOLVER investigation

All hardware vendors have high-level APIs to speed up the basic linear arithmetic and matrix operations. cuBLAS [29] is one such feature from CUDA to run dense linear algebra computations. cuSOLVER is a similar library file used to speed up factorization operations. NVIDIA advises the programmers to use these APIs not only for speed up but also to identify and solve the bugs in later stages of design because it gives a higher layer abstraction for code readability and re-usability.

The cuBLAS library is called during a kernel launch, but before that some memory is allocated for it. This memory allocated in the host should be aligned and non-pageable for maximum performance [30]. The functions (kernels) are performed after the memory allocation and the cuBLAS handle is destroyed after its execution. The inverse and 16 matrix multiplication kernels were launched with cuBLAS. However, the results were very poor (5 times worse) as compared to normal CUDA implementation. When analyzed, it was noted that the cuBLAS and cuSOLVER APIs were performing good but the time taken to initiate this routine is huge. This is not a one time sacrifice in performance like the device setup time which happens only once when a device is initiated but the routine is destroyed after each execution so the initiation has to happen during the next execution of the filter operation. Also, all the scientific works on these libraries only talk about matrix sizes of 1000 and above. So, it was concluded that the use of high-level acceleration APIs will not really help in this kind of project dealing with small matrices.

4.3.4 Discrete and unified memory investigation

There are two types of memory allocation in the CUDA programming model. One where the CPU and GPU share the same memory space, the other where the GPU has its own discrete RAM memory.

- In a discrete GPU, the GPU accesses RAM using via the PCIe bus. The processor here is just another device that is connected to the memory controller. The PCIe device acts as a bus master, taking over that bus and talking to the memory controller directly like the processor does. The discrete GPU also has its own RAM and this RAM is accessible by the processor by acting as a PCIe bus master.
- In an integrated GPU, the GPU can access memory just like the processor. The GPU and processor share a common bus that connects them with the memory controller.

CudaMalloc is the API that assigns memory in the GPU's discrete memory and Cuda-MallocManaged is the API which assigns memory to a memory shared with the CPU. This shared memory version of the memory management was introduced in the CUDA 6 and is called the unified memory architecture. To avoid confusion, the CPU and GPU communicate by synchronization calls when accessing memories in the shared region. In this project, one of the main reasons to use two GPUs is to talk about this parameter. The discrete GPU in NVIDIA DRIVE Px2 cannot access the shared memory because it is located far away from the processor and is communicated via the PCIe bus, but in Quadro T1000 the program runs in a laptop which can utilize the unified memory architecture. From an application perspective, the Tegra SOCs, where the GPU is close to the CPU, will be benefited from this unified memory architecture. Drive Px2 itself has a Tegra SOC, but the integrated GPU in Tegra has only two SMs which is used for multimedia applications. Figure 4.3 shows the timing in Quadro T1000 for memory transfers when the filter was implemented using both memory models where a timing difference by a factor three is obtained.

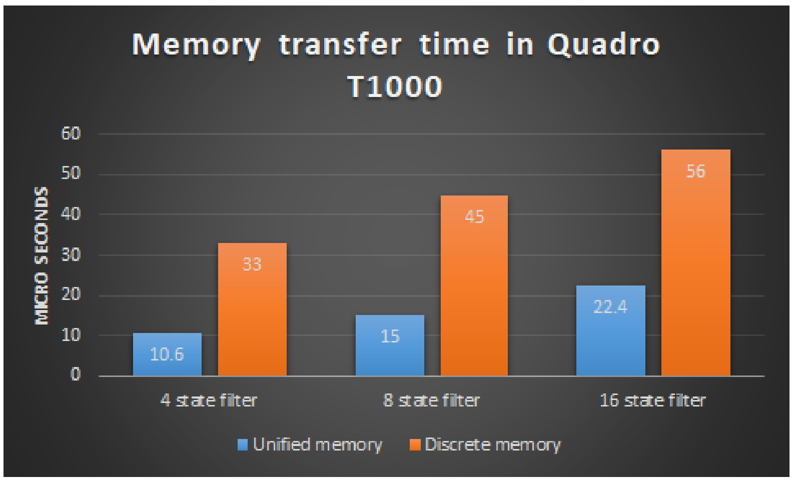


Figure 4.3 – Memory transfer latency for Unified vs Discrete memory model

4.3.5 Profiling results from NVIDA nprof

The results of both GPUs were obtained by nvprof profiling, where memory transfer and computation time in the GPU is separated from CPU API calls. For GPU profiling the CPU API calls were eliminated and only the GPU activity was profiled. The profiler also gives indepth analysis of kernels, blocks and threads that happens during the computing stage. It shows minute timing intervals of each kernel execution and the number of times a kernel is called. For analysis, the kernel concurrency, i.e., the percentage of time when two or more kernels are being executed, and memory copy throughput, i.e., to check if the memory copies are fully utilising the bandwidth of the GPU, was examined and it was noticed that both were very small and limited by the functionality of the filter because of its sequential operations and lesser data requirement.

Task wise split up of computation time : Totally, 37 kernels were launched for the four state filter and 39 kernels for the eight and 16 state filters, where 1 kernel that was used to calculate the variable-Ih was called 4 times at different instances but others only once. Inverse functions were also involved in the computation. Table 4.1 shows the split up of computation time in the two GPUs in microseconds. The computation time in the discrete and unified memory access mode is the same in Quadro T1000, and this is because the computation time of each kernel is independent of the memory access pattern. It only depends on the clock speed of the operating core.

Tasks	4 state filter	8 state filter	16 state filter
	Quadro-PX2	Quadro-PX2	Quadro-PX2
Prediction	47-74	56-83	65-90
Measurement and update	15-17	17-25	21-30
Total	62-91	73-109	86-120

Table 4.1 – Computation time in GPU

Figures 4.4 and 4.5 show the overall activity time of the two GPUs profiled. The numbers inside the colored box indicate the time in microseconds. The timing numbers are low in Quadro T1000 compared to the Drive Px2 because of the cache architecture and faster operating CUDA cores as discussed above. The speedup in the GPU is less than the best-case processor execution time for four and eight state filter but is only 1.9x for best case 16 state filter even though the matrix operations are executed in fewer cycles than the CPU. This is because of smaller kernel launches that are not able to overlap memory transfer and kernel execution. The inability to launch the kernels parallelly is a limitation of the filter, because the operations are sequential, and, each kernel launch will have an overhead if the GPU instruction queue has nothing left to do, this adds more execution time.

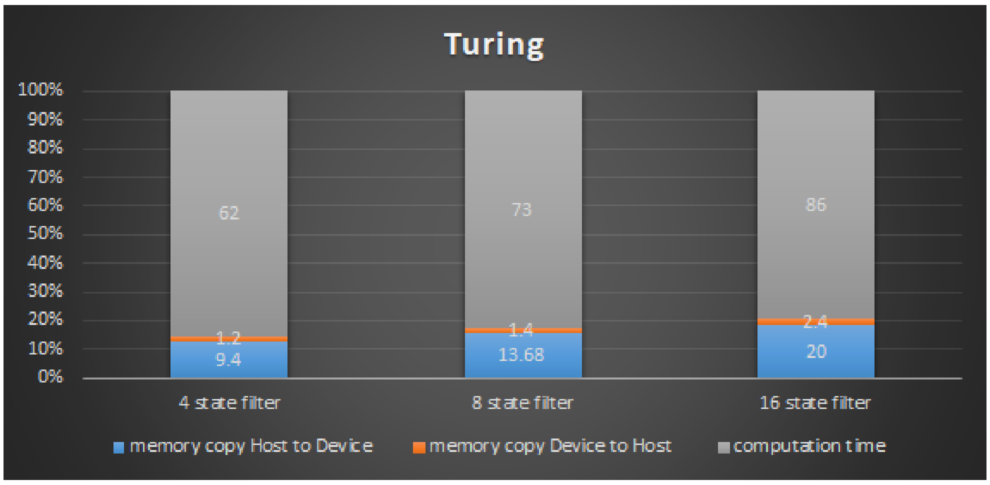


Figure 4.4 – Quadro T1000 profiling results

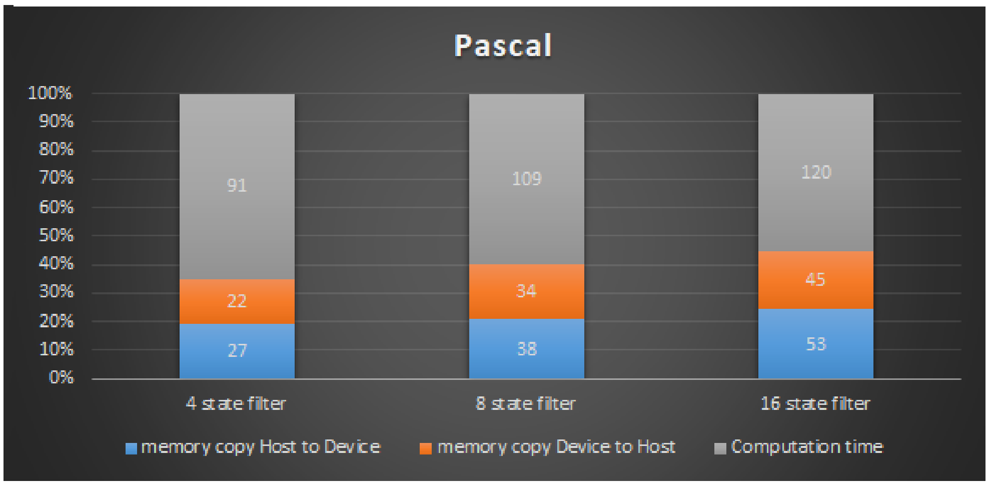


Figure 4.5 – Drive Px2 profiling results

An interesting insight to note from figures 4.4 and 4.5 is the percentage of GPU activity. In Drive Px2 almost 50% of the total GPU activity is spent on memory transfers whereas in Quadro, it is 20% for the same case. The percentage difference is not because of clock cycles but because of the discrete memory architecture in Drive Px2 and it will be the same in Drive AGX even if it uses a Turing GPU because the discrete GPU will be separated from CPU inside the SOC unlike a GPU in a Laptop.

4.4 FPGA Programming

The FPGA was programmed using the same C code used to program the CPU and the GPU, but in a SDSoC environment that give access to the programmable logic. The SDSoC uses the Vivado HLS library APIs to generate RTL code thereby realising the hardware block for the C function. Another way to easily program the FPGA is by using the MATLAB HDL coder toolbox following the steps in [31]. The Simulink model written with a high-level MATLAB script can be directly converted to a hardware using this toolbox. The toolbox also has a code generation workflow to change and validate the design so it becomes easy for the user to build the functionality in hardware.

A drawback with this toolbox is that it does not support some important filter functionalities like inverse, while-loops and diagonal-matrix operations. Writing a different inverse function is a possible option but it will not be the same scale benchmarking of execution time. The optimisations in FPGA and GPU are considered from the functionality that has best case execution time in processors. Moreover, a fixed point converter is employed by the HDL tool to convert the double-precision numbers to single precision before it starts to write the RTL script, this will cause precision difference in the output. However, to explore the toolbox, the prediction block of the four state filter was programmed in FPGA using the MATLAB HDL coder toolbox and it was noticed that the resource utilization of the hardware written by the HDL coder directly from the MATLAB script was more efficient than the C-RTL.

But, for the reasons mentioned above, FPGA programming was done with the base C code using SDSoC. Two types of hardware optimizations are done. One is within the design of the filter IP and the other is to interface the IP with the processor/external device.

4.4.1 IP Kernel

Optimizing the logic of the filter was easy with Pipelining and unrolling library functions provided by the vivado HLS design suite. The real challenge was to optimize the matrix inverse operations which actually took more time to execute than the other filter operations combined.

Loop Pipelining: When a C/C++ loop is read by a processor instruction set it reads and executes it sequentially. The next iteration of the loop will start only after complete execution of the current loop. But when programming an FPGA the next iteration can be started immediately when the data needed for it is ready from the current loop. This way, the number of cycles to Complete a loop can be reduced. This optimization trick is called loop level pipelining. A simple diagram for a three iterative for-loop that does read, compute, and write cycles at 1 cycle latency each is shown in figure 4.6. Vivado HLS also allows the user to manually set the initiation interval. In the given example the initiation interval is 1, i.e., after the first cycle in the current iteration of the loop, the next iteration will begin. However, the pipelining technique is a bottleneck where data dependency (if the data for the next iteration of the loop is not ready) is a factor.

A false data dependency can sometimes delay the execution of pipelining. To solve this, the Read after Write, Write after Read and Write after Write dependency analysis in the loop can be made manually and this is explicitly mentioned to the Vivado compiler by using the dependence pragma library function for any variable inside the loop. Loop pipelining was used in this project wherever unrolling is not possible, i.e., wherever data dependency is an issue. For example, update of the information vector needs 13 loop iterations and each iteration has a loop latency of 12, so no loop can be unrolled because the output of each loop increments the next loop. Pipelining helps to solve the problem in 12*13=156 cycles instead of 12*13*13=2028 cycles. A problem with using pipelining is that it might show problems during the routing stage of the design. This is because the registers and computational unit has to be connected within the mentioned design clock frequency. In this project, the 16 state filter showed negative slack (timing error) in many places where pipelinig was heavily relied on when operating at 200 MHz. So, the 16 state filter design was limited to 100 MHz clock speed.

Similar to loop level pipelining, task-level pipelining can also be performed in vivado HLS. In this project, this was not efficient because in the first task, the prediction is very sequential, and only from its final output the next task measurement could be started. The measurement task's final output was needed for the update task to execute. So, task level pipelining did not help with this filter. Function dataflow is an option to explore task level parallelism by bringing in all functions together. However, it did not show any positive results because theoretically, without the input data a task could not begin. The vivado compiler showed error when using this option.

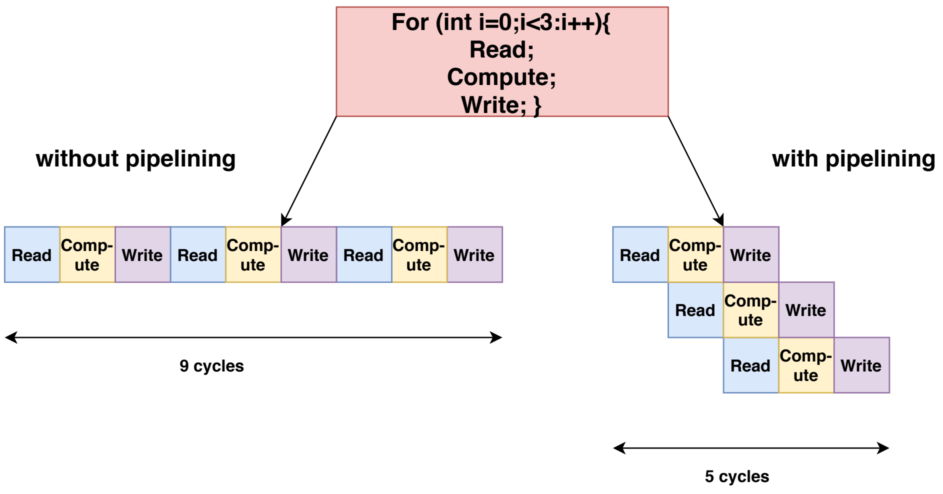


Figure 4.6 – Loop pipelining

Loop unrolling: Similar to loop pipelining, loop unrolling reduces the latency of a loop. This library function creates multiple copies of the loop body and executes them parallelly in one iteration instead of doing multiple iterations. Figure 4.7 shows how a for-loop that goes for three iterations of three cycles each can be made to run in three cycles in one iteration itself. However, the bottleneck here is usage of resources. In an unrolled loop only one multiplier is required but in the rolled version of the loop, three multipliers are required. It is the same with flip flops, and look up tables.

Data dependency is also a factor to be considered when using the loop unrolling technique. Vivado HLS provides the user to define a factor when doing loop unrolling. For a 100 iteration loop, instead of unrolling the full loop the user can choose to unroll it by a factor of two thereby saving 50% more resources than it would be for full unrolling but at the cost of extra latency cycles. Also, it should be noted that sequential loop will consume more power when performing the extra cycles and unrolling the loops may increase glitch power. The trade off is between power, area and time when doing this optimisation. For this project, power and area were not a concern but the priority was execution time, so unrolling was performed for all possible loops. Loop unrolling alone gave 24 times speed up for matrix multiplication operations in the eight state filer design. Because, a matrix multiplication has three loops each loop operates for eight cycles but when unrolling these three loops the multiplication happens in one cycle. Matrix operations were parallelized with this operation but the Gauss-Jordan inverse operation could not be solved by this trick because it had a data dependency issue from previous for loop.

Resource utilization : For this project, Volvo was not particular about the usage of resources in the PL but the focus was only on decreasing the execution time. However, using a high end SOC device with only an information update filter in the PL is not a good choice if the IP for the filter alone occupies more than 60% of the available resources in the PL. To solve this bottleneck Vivado HLS allows the user to reuse the same hardware core when executing the same function again instead of creating another instance of the same core by the usage of allocation pragma.

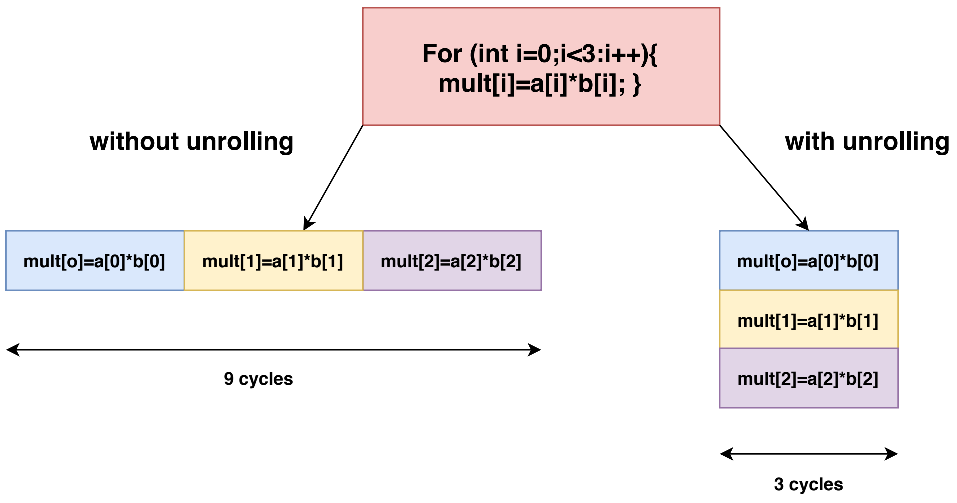


Figure 4.7 – Loop unrolling

It also allows the programmer to specify which resource a particular function/task should utilise by using the HLS resource pragma. For example, the programmer may choose to store data in block RAM instead of Look Up Table (LUT)s because he/she feels the LUTs are already being used by many functions but the block RAM is not utilised. In addition to this, array optimisation libraries were used to split an array of data to small blocks, so they can be accessed directly based on their need for computations. Another technique used is to merge small arrays and put it in a block RAM which has high memory capacity. Resource reuse is also employed in this project. The inverse matrix operation hardwares were reused because they were executed sequentially. One problem when reusing hardwares would be timing violation if the design has to go a long path backward to reuse the resource, but this was not an issue for this project as the design did not show timing error for 100 and 200 MHz for four and eight state filters.

Tasks	4 state filter	8 state filter	16 state filter
	Before-After	Before-After	Before-After
Prediction	7740-1039	53735-3350	392061-7333
Measurement and update	10911-321	21071-476	41391-681
Total	18651-1360	74752-3826	433452-8014

Table 4.2 – Total hardware latency before kernel optimisation - after kernel optimisation at 100 MHz

The total time taken for the execution of the logic in the IP is calculated by multiplying the latency observed in table 4.2 and the estimated clock frequency obtained from the Vivado design report. The latency cycle for the four state filter is fixed, but the latency cyles for eight and 16 state filters might be little less than the numbers shown here and it depends on the input. This is because, for the inverse functions the number of iterations in each for loop will be based on the input matrix and the maximum limit the for-loop is counted when the latency is calculated.

- Four state filter : 1360 * 7.885 nanoseconds = 10.7 microseconds.
- Eight state filter : 3826 * 7.919 nanoseconds = 30.2 microseconds.
- 16 state filter : 8014 * 8.014 nanoseconds = 64.2 microseconds.

Resources	4 state filter	8 state filter	16 state filter
	Before-After	Before-After	Before-After
DSP	333-603	336-628	336-1113
BRAM	44-140	58-378	62-786
LUT	41881-69526	45736-91463	48136-167455
FF	27381-98146	28682-111509	29031-218238

Table 4.3 – Total hardware resource utilisation before kernel optimisation - after kernel optimisation at 100 Mhz

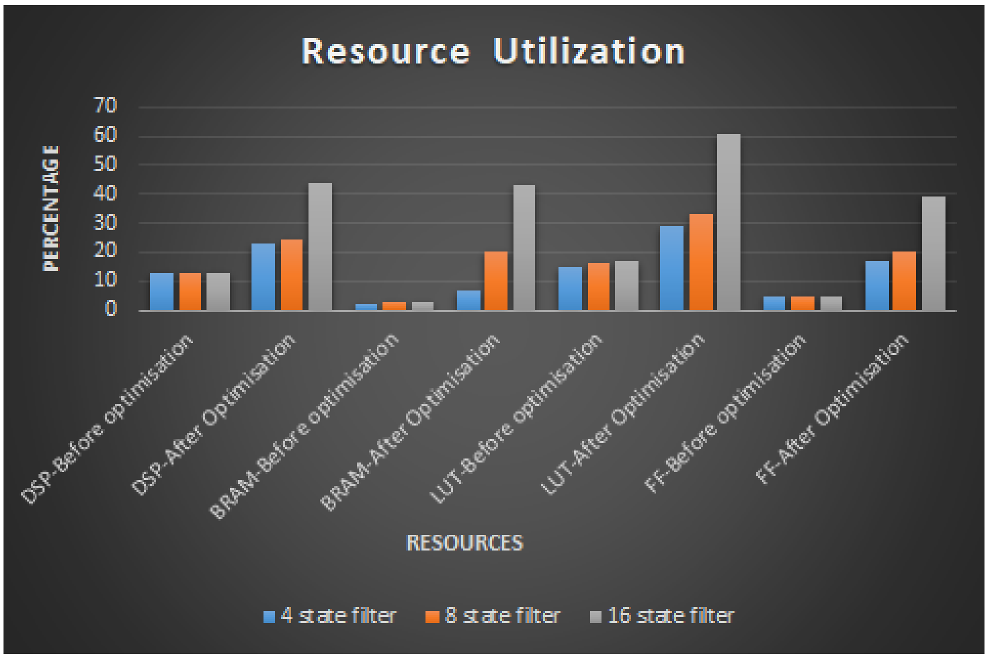


Figure 4.8 – Graph showing the percentage of resource utilization in the programmable logic

4.4.2 External interface and data transfer

On top of Vivado HLS, the SDSoC API provides library functions to integrate the designed IP in the PL with external interfaces. Generally, in MPSoC the data for computation comes from the processor and after computation in PL the output goes back to the processor. This process burdens the CPU with data mover cycles, this affects CPU scheduling. To solve this bottleneck, a zero copy library function in the SDSoC copies the data directly from the on-chip shared memory to the PL using AXI bus. After the computations in the PL, the output can be sent to the processor if it needs the output every interval or back to the shared memory so the processor can access it whenever it needs the log data with a single read and write.

The data access pattern library function lets the programmer decide how data should be accessed and this is based on the application. Some data are better when accessed in block/random, some are better when accessed in a first in first out basis like the FIR filter. In this project, data was accessed as a block of arrays because the filter computation can start only when all data is available at the input ports.

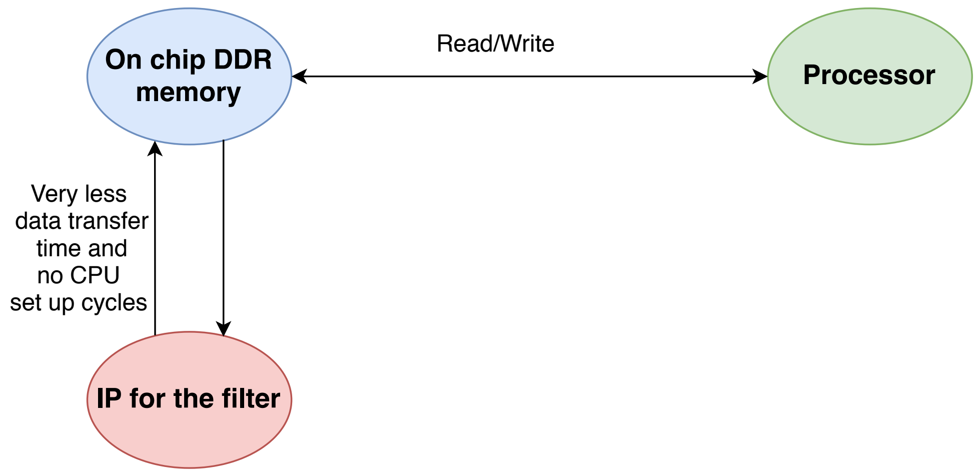


Figure 4.9 – External interface with IP

Data transfer to/from IP	4 state filter	8 state filter	16 state filter
Through processor	2952	4452	5489
Directly from DDR DRAM	854	1292	1800

Table 4.4 – Data transfer cycles to and from FPGA

4.4.3 Synthesis results

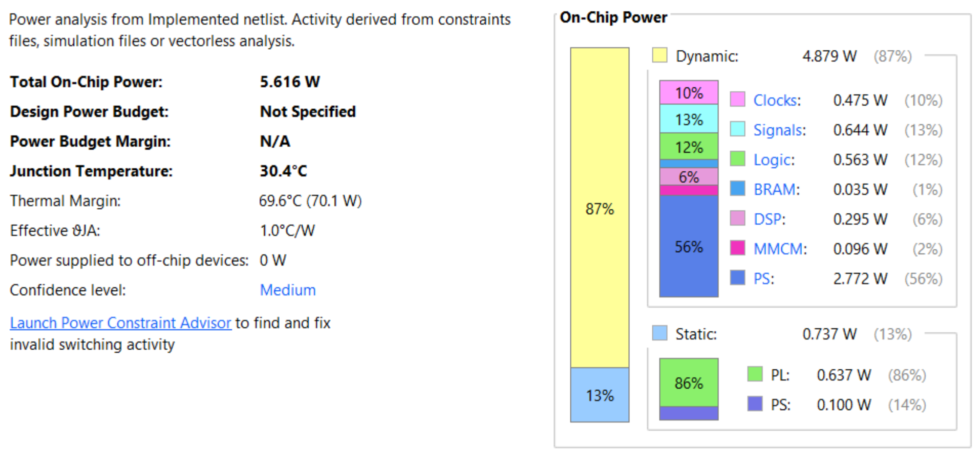
The IP design for three filters were verified using the Vivado IP design toolbox, where timing and power analysis were performed.

Timing: The timing analysis tool checks all the timing paths from input to output to detect any worst path that can cause setup and hold time violations. Slack is the difference between the restricted and the analysed value. The total negative slack [32] for set up time (maximum delay analysis for set up recovery and data check), hold time (for checks related to minimum delay analysis which includes hold, delay and data check) and pulse width pin switching checks (minimum width of high and low pulses of a clock) is 0 for all IPs thus the design has passed user specific timing constraints, i.e., all the signals reach the computational units in correct time without any delay. Table 4.5 shows the positive slack for the design. This positive slack is an indication that the design is working but can be improved, and it is better to have some small margin positive slack instead of 0.

Slack	Four state filter	Eight state filter	16 state filter
worst setup slack	0.477 ns	1.6 ns	0.118 ns
worst hold slack	0.010 ns	0.009 ns	0.010 ns
worst pulse width slack	1.0 ns	2.0 ns	2.0 ns

Table 4.5 – Table showing design timing details for the three filter IPs

Power: Power is a concern not only for the energy efficiency but also for identifying thermal bottlenecks. For an IP, power analysis helps to estimate the life of the hardware before being put to the system. A GPU and Processor will execute different set of instructions during its lifetime but the FPGA unit solves the same problem as the hardware is embedded in it. So, power analysis is more important for FPGA than for the other two hardwares. The Power analysis is performed by the Vivado power estimator through all stages of the flow. Screen shots of the power analysis generated by Vivado post-routing (after all logic implementation and routing has been performed) for the three IPs is shown in figures 4.10, 4.11 and 4.12. It helps in identifying the high power consuming resources in the hardware module. It also gives the total power consumption, estimated power consumption on each supply rail, estimated power breakdown between static and dynamic power, and the device junction temperature at which the generated IP can operate [2]. For this analysis, the airflow, room temperature were all set at the default Vivado suggested parameter values.


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

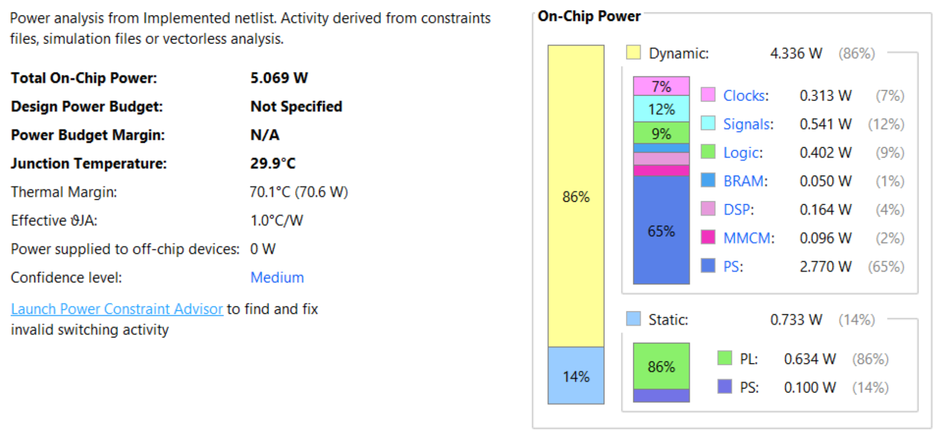
Total On-Chip Power:	**5.616 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**30.4°C**
Thermal Margin:	69.6°C (70.1 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	4.879 W	(87%)	
Clocks:	0.475 W	(10%)	
Signals:	0.644 W	(13%)	
Logic:	0.563 W	(12%)	
BRAM:	0.035 W	(1%)	
DSP:	0.295 W	(6%)	
MMCM:	0.096 W	(2%)	
PS:	2.772 W	(56%)	
Static:	0.737 W	(13%)	
PL:	0.637 W	(86%)	
PS:	0.100 W	(14%)	

Figure 4.10 – Power analysis report for designed four state filter IP from Vivado power estimator [2]



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

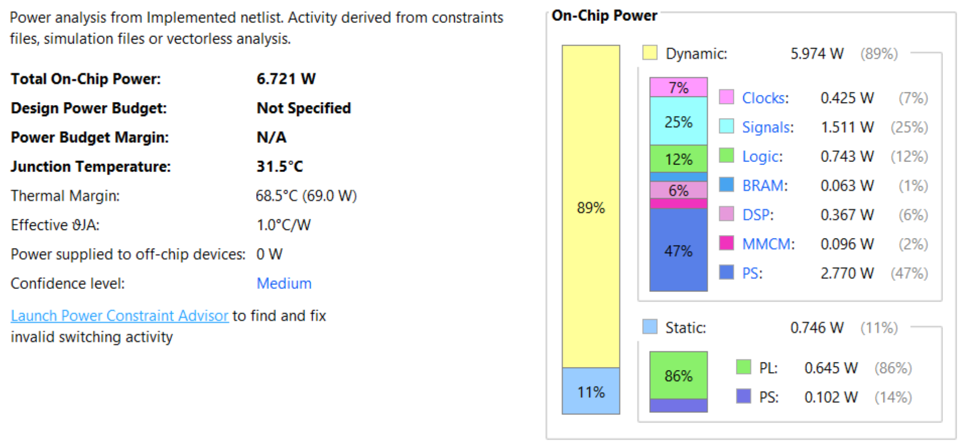
Total On-Chip Power:	**5.069 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**29.9°C**
Thermal Margin:	70.1°C (70.6 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	4.336 W	(86%)
Clocks:	0.313 W	(7%)
Signals:	0.541 W	(12%)
Logic:	0.402 W	(9%)
BRAM:	0.050 W	(1%)
DSP:	0.164 W	(4%)
MMCM:	0.096 W	(2%)
PS:	2.770 W	(65%)
Static:	0.733 W	(14%)
PL:	0.634 W	(86%)
PS:	0.100 W	(14%)

Figure 4.11 – Power analysis report for designed eight state filter IP from Vivado power estimator [2]



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**6.721 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**31.5°C**
Thermal Margin:	68.5°C (69.0 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	5.974 W	(89%)
Clocks:	0.425 W	(7%)
Signals:	1.511 W	(25%)
Logic:	0.743 W	(12%)
BRAM:	0.063 W	(1%)
DSP:	0.367 W	(6%)
MMCM:	0.096 W	(2%)
PS:	2.770 W	(47%)
Static:	0.746 W	(11%)
PL:	0.645 W	(86%)
PS:	0.102 W	(14%)

Figure 4.12 – Power analysis report for designed 16 state filter IP from Vivado power estimator [2]

4.5 Execution time results

A summary of all execution times is presented in the graph 4.13. It shows the overall hardware activity which includes the computation and memory transfer time. Best and worst case is presented for the application processing unit, a real time processing unit will not have any deviation in execution time because it operates from the tightly coupled on chip cache memory. The FPGA IP block will also not show any deviation because it is a prebuilt hardware running on its own clock. The GPU timings here are best case timings because only the CUDA kernel core activities are profiled, deviation might happen based on the non-blocking runtime API calls and this is dependent on the CPU. Even though a unified CPU-GPU memory architecture was profiled and discussed with Quadro T1000 laptop GPU in the previous chapter, for final benchmarking only the discrete GPU profiling is considered because the GPUs used in automotive application does not have that support.

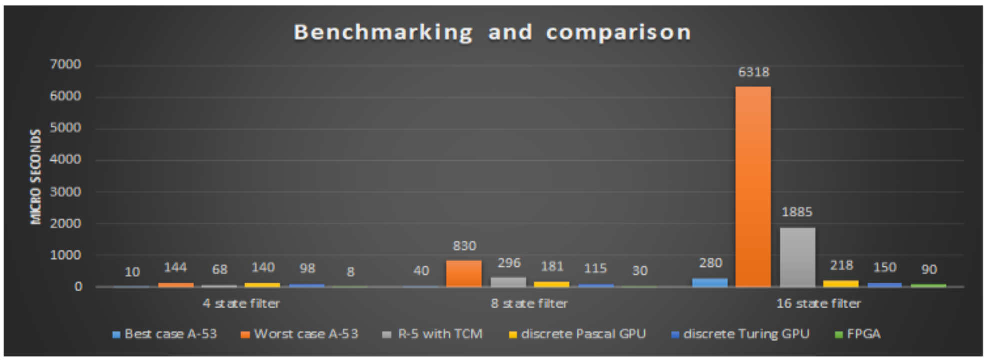


Figure 4.13 – Benchmarking

Chapter 5

Conclusion and future work

This thesis investigated whether it was better to offload a Kalman filter functionality from a processor. The answer to this question depends on the entire application and its safety criticality. This chapter discusses the scenarios when it is better to offload, when it is not so important and when it is a must to offload the filter based on the results obtained from this project.

When APU?
One among the four A-53 cores in the APU should be the considered to run the filter if it executes the task in best case timings (which is better than GPU and almost equal to FPGA for four and eight state filters) in a test environment with all other applications running parallelly and interrupts enabled. From this project, the conclusion is that if data caches are not disturbed then best case execution time or near best case execution time is possible. The Smaller the size of the filter, the less data cache is needed, the more reason to have the filter in the APU itself. Instruction cache disturbance does not affect the performance as bad as data cache disturbance.

When RPU?
The real-time R-5 processor inside the RPU should be considered for running the filter, regardless of its slower performance as compared to other hardwares if the real time safety deadline is an important aspect when scheduling the filter. The overall deadline to run the application is 10 ms, the safety deadline is 7 ms. If the output states predicted by the filter is important for other proceeding blocks to operate, then a delay cannot be tolerated. If the FPGA is running out of resources and the GPU cores are still fully occupied with the algorithms it was designed for (computer vision/deep learning), in this case,

the real-time processor should be a better choice than A-53 whose worst case execution time can cause a scheduling miss. The R-5 is made uninterrupted during its period of execution by placing the instruction set and data to the tightly coupled processor memory instead of using the external memory.

When FPGA?
For a four or eight state filter offloading the filter from processor to FPGA will only relieve the processor of some work because the speedup when comparing best case APU and FPGA is not more than 1.5 times. But for a 16 state filter the FPGA is four times better in computation than the best case execution time of the processor and this best case is definitely not achievable in real time. With so many intermediate data cache access to compute the 16 states, cache misses will increase the latency. So, a 16 state filter will be difficult to schedule in the processor and for this, using the FPGA as accelerator is a better choice than a GPU because it is independent of the processor whereas the GPU has to depend on the processor for its data and instruction set every iteration which will need some memory space in the processor that will cause a delay in the overall performance.

When GPU?
The GPU could be considered as an accelerator even if it is slower than an FPGA if resource utilization in the PL-region of the SOC used is a bottleneck. For example, when implementing the 16 state filter in the FPGA the best case computation time was achieved by utilizing 61% of the lookup tables in the programmable logic. Then the programmable logic cannot be used for any other acceleration. Whereas, a GPU implementation is based on instruction set i.e. the GPU resources can be used by any other application for the times when the filter is not being scheduled, which is not possible in an FPGA whose resources are designed specifically for the filter alone. Another argument to use the GPU would be the development time of the filter. Porting the application to the GPU is easy and fast for a software programmer who coded the application in C/C++. But for porting the logic to an FPGA it requires low level hardware understanding for a efficient/optimised implementation even with high level synthesis tools. It is a necessity if the design faces timing violations.

5.1 Future work

- Finding the performance bottleneck by porting the entire vehicle motion algorithm.
- Use Multithreading to utilize all four A-53 cores in the MPSoc. The L2 cache is a shared memory in the APU and is designed for faster inter-core communication in the SOC with a snoop controller. The performance of the processor should increase, if data is available in the cache. This experiment should be performed after porting the entire algorithm to the SOC.
- To Write a resource efficient RTL script in VHDL/Verilog for the filter design and check how the resource utilisation in the programmable logic could be improved.
- In a rare case, if a singular matrix is operated for inverse function, that iteration of filter would produce uncertain output. To solve this, the MATLAB pseudoinverse function could be implemented instead of the inverse function. This pseudoinverse function is a sparse matrix computation and can be implemented with Xilinx Vitis environment that has a library inbuilt called SVD (Singular Vector Decomposition) computation. The implementation of the pseudoinverse matrix operation itself is a boiling research topic in the FPGA world, even the MATLAB HDL coder toolbox does not have an option for porting the function to FPGA.

References

[1] "Technical reference manual," Xilinx Inc., url:https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[2] "Xilinx power estimator user guide," Xilinx Inc., url:https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug440.pdf.

[3] A. Taeihagh and H. S. M. Lim, "Governing autonomous vehicles: emerging responses for safety, liability, privacy, cybersecurity, and industry risks," *Transport Reviews*, vol. 39, no. 1, pp. 103–128, 2019.

[4] O. B. O. B. Mehdi Nourinejad, "The economics of autonomous vehicles," *Rotman Management Magazine*.

[5] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[6] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis, "A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 829–846, 2018.

[7] M. Hoshiya and E. Saito, "Structural identification by extended kalman filter," *Journal of engineering mechanics*, vol. 110, no. 12, pp. 1757–1770, 1984.

[8] R. Van Der Merwe, A. Doucet, N. De Freitas, and E. A. Wan, "The unscented particle filter," in *Advances in neural information processing systems*, 2001, pp. 584–590.

[9] V. Bonato, R. Peron, D. F. Wolf, J. A. M. de Holanda, E. Marques, and J. M. P. Cardoso, "An fpga implementation for a kalman filter with

application to mobile robotics," in *2007 International Symposium on Industrial Embedded Systems*, 2007, pp. 148–155.

[10] A. Jarrah, A.-K. Al-Tamimi, and T. Albashir, "Optimized parallel implementation of extended kalman filter using fpga," *Journal of Circuits, Systems and Computers*, p. 1850009, 06 2017. doi: 10.1142/S0218126618500093

[11] Z. Merhi, M. Ghantous, M. Elgamel, M. Bayoumi, and A. El-Desouki, "A fully-pipelined parallel architecture for kalman tracking filter," in *2006 International Workshop on Computer Architecture for Machine Perception and Sensing*, 2006. doi: 10.1109/CAMP.2007.4350359 pp. 81–86.

[12] J. Fonseca, R. Oliveira, J. Abreu, E. Ferreira, and M. Machado, "Kalman filter embedded in fpga to improve tracking performance in ballistic rockets," 04 2013. doi: 10.1109/UKSim.2013.149. ISBN 978-1-4673-6421-8 pp. 606–610.

[13] M. Huang, S. Wei, B. Huang, and Y. Chang, "Accelerating the kalman filter on a gpu," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 1016–1020.

[14] Z. Lin, D. Moore, and S. Russell, "Gpu-based parallel kalman filter."

[15] C. Sekar and Hemasunder, "Tutorial t7: Designing with xilinx sdsoc," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, 2017, pp. xl–xli.

[16] "Managing power and performance with the zynq ultrascale+ mpsoc," Xilinx Inc., url:https://www.xilinx.com/support/documentation/white_papers/wp482-zu-pwr-perf.pdf,.

[17] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Ultrascale+ mpsoc and fpga families," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–37.

[18] N. Corporation, "Nvidia cuda compute unified device architecture programming guide," 2007.

[19] J. Nickolls, "Gpu parallel computing architecture and cuda programming model," in *2007 IEEE Hot Chips 19 Symposium (HCS)*. IEEE, 2007, pp. 1–12.

[20] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[21] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "Low overhead instruction latency characterization for nvidia gpgpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–8.

[22] "Isolation methods in zynq ultrascale+ mpsocs," Xilinx Inc., url:https://www.xilinx.com/support/documentation/application_notes/xapp1320-isolation-methods.pdf.

[23] C. Ingemarsson and O. Gustafsson, "On fixed-point implementation of symmetric matrix inversion," in *2015 European Conference on Circuit Theory and Design (ECCTD)*. IEEE, 2015, pp. 1–4.

[24] G. Shapiro, "Gauss elimination for singular matrices," *Mathematics of Computation*, vol. 17, no. 84, pp. 441–445, 1963.

[25] A. Holdings, "Arm cortex-a53 mpcore processor, technical reference manual, 2014."

[26] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 233–248.

[27] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[28] Y. Sun and Y. Tong, "Cuda based fast implementation of very large matrix computation," in *2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2010, pp. 487–491.

[29] C. Nvidia, "Cublas library programming guide," *NVIDIA Corporation. edit*, vol. 1, 2007.

[30] P. Estival and L. Giraud, "Performance and accuracy of the matrix multiplication routines : Cublas on nvidia tesla versus mkl and atlas on intel nehalem," 03 2010.

[31] H. Pant, H. Bourai, G. S. Rana, and S. Yadav, “Conversion of matlab code in vhdl using hdl coder & implementation of code on fpga,” *HCTL Open International Journal of Technology Innovations and Research (IJTIR)*, vol. 14, pp. 1–9, 2015.

[32] “Design analysis and closure techniques,” Xilinx Inc., url:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug906-vivado-design-analysis.pdf.