Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Software Support for Parallel ADAS Applications on Pre-development Version of the Aurix TC4

Bc. Lukáš Bielesch

Supervisor: Ing. Radek Olexa
Subfield: Cybernetics and Robotics
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Bielesch Lukáš**	Personal ID number:	**465960**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Measurement**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Software support for parallel ADAS applications on pre-development version of the Aurix TC4 microcontroller

Master’s thesis title in Czech:

Softwarová podpora pro paralelní ADAS aplikace na vývojovém vzorku mikrokontroléru Aurix TC4

Guidelines:

1. Familiarise with pre-development samples of Infineon AURIX™ TC49A MCU and its new Parallel Processing Unit (PPU).
2. Develop basic software support (boot code, IRQ handling, …) for the Infineon AURIX™ TC49A microcontroller.
3. Explore the standalone computing cluster with vector processing capabilities (PPU). Demonstrate the advantages/disadvantages of offloading the tasks from TriCore to the vector processor.
4. Provide integration of the PPU capabilities with other tools such as MATLAB/SIMULINK with MetaWare toolbox, the Neural Network Software Development Kit (NN SDK) etc. The aim is to speed up the development process of PPU applications.
5. Develop a set of automotive-related applications demonstrating effective application vectorisation. The applications will use both the basic software and the PPU support. The applications will work on the virtual model of the CPU and the actual pre-development hardware samples.

Bibliography / sources:

[1] AURIX™ TC49x User manual V0.73. Infineon Technologies AG. 01-12-2021
[2] Effective Vector Programming Guide MetaWare for AURIX™ TC4x. Version 4222-013. Synopsys. September 2021
[3] Machine Learning and Embedded Computing in Advanced Driver Assistance Systems (ADAS). September 25, 2019. ISBN 978-3-03921-376-4 [online]. Available from WWW: https://www.mdpi.com/books/pdfview/book/1573

Name and workplace of master’s thesis supervisor:

Ing. Radek Olexa   HighTec EDV-Systeme GmbH

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **10.02.2022**   Deadline for master's thesis submission: ___________

Assignment valid until:
by the end of summer semester 2022/2023

Ing. Radek Olexa Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to express my sincere gratitude to HighTec EDV Prague headed by my supervisor Ing. Radek Olex, for the willingness and time he gave me during the development of this thesis as well as the opportunity to work with state-of-the-art technologies. Secondly, I would like to thank my colleagues whose expertise greatly facilitated the process of understanding the automotive AURIX architecture. Special thanks go to my brother Marek and my parents for their unconditional support throughout my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Signature:

In Prague, May 20, 2022

Abstract

To meet the combination of performance, ASIL safety standards and cost-effectiveness, many new architectures in the automotive industry are adopting a heterogeneous design combining scalar, and vector DSP processors. The aim of this thesis is to create a set of tools and functions to support the new TC4xx family of microcontrollers adopting the heterogeneous design. Furthermore, high-level tools are analyzed and used to enable accelerated development processes.

The scope of this thesis further includes an implementation of basic software support consisted of multi-core booting, initialization of essential peripherals, easy-to-use registration of interrupt service routines and printing to UART with multi-core synchronization. Further it contains, the inter-processor communication allowing data exchange between individual cores, and thus offloading computationally intensive operations on a parallel computing unit.

To demonstrate the capabilities of the vector processor, two experiment applications were developed - a Kalman filter-based application to estimate the speed of the lead vehicle and a convolutional neural network trained to recognize the drowsy driver.

The correct functionality was verified by running the programs in a simulation environment and subsequently on the first prototype of the evaluation board.

Keywords: Aurix TC4xx, Pre-development, BSP, SIMD, ADAS, Model-based design, Embedded code generation, Kalman filter, Convolutional neural network

Supervisor: Ing. Radek Olexa

Abstrakt

V mnohých nových architektúrach v automobilovom priemysle sa využíva heterogénny dizajn kombinujúci skalárne a vektorové DSP procesory s cieľom splniť výkonové kritériá, bezpečnostné normy ASIL a kompetitívnu cenu. Cieľom tejto práce je vytvoriť súbor nástrojov a funkcií na podporu novej rodiny mikrokontrolérov TC4xx s heterogénnym dizajnom. Okrem toho sa analyzujú a využívajú vysokoúrovňové nástroje, ktoré umožňujú zrýchliť vývojové procesy.

Táto práca ďalej zahŕňa implementáciu základnej softvérovej podpory pozostávajúcej z bootovania viacerých jadier, inicializácie základných periférií, ľahko použiteľnej registrácie funkcií obsluhy prerušenia a výpisu do UART so synchronizáciou viacerých jadier. Ďalej implementuje medziprocesorovú komunikáciu umožňujúcu výmenu údajov medzi jednotlivými jadrami, a tým aj odľahčenie výpočtovo náročných operácií na paralelnej jednotke.

Na demonštráciu možností vektorového procesora boli vyvinuté dve aplikácie - algoritmus založený na Kalmanovom filtri na odhad rýchlosti vedúceho vozidla a konvolučná neurónová sieť natrénovaná na rozpoznávanie ospalého vodiča. Správna funkcionalita bola overená spustením programov v simulačnom prostredí a následne na prvom prototype evaluačnej dosky.

Klíčová slova: Aurix TC4xx, Pre-development, Základná softvérová podpora, Paralelné výpočty, Asistenčné systémy vodiča, Generovanie kódu, Kalmanov filter, Konvolučná neurónová sieť

Contents

Figures

Tables

Chapter 1

Introduction

The pressure to improve driver safety through neural network-based algorithms in driver assistance systems is forcing automotive microcontroller manufacturers to create new architectures satisfying the latest safety and security standards while delivering adequate performance to utilize complex algorithms.

This is also the case with Infineon's new TC4xx family of microchips, which has adopted a heterogeneous architecture consisting of TriCore safety microcontrollers and an ARC71 Parallel Processing Unit optimized for compute-intensive operations. Currently, TC4xx is in an early development phase where support is being gradually built up. A simulation environment is provided to speed up SW development before the hardware availability and can be used to perform the initial tests. The same code is reused on hardware later on.

This thesis focuses on developing a set of software tools and features allowing to boot the platform, provide basic run-time capabilities, establish communication between computing modules, and explore the capabilities of PPU using parallel algorithms. Another objective is to use modern high-level approaches to model complex algorithms and generate optimized code for the PPU unit, replacing the hand-written code.

After the analysis of TC4xx architecture in Chapter 2, requirements for the essential software tools are listed in Chapter 3. Their correct functionality is verified by a multi-core application that properly initializes all cores and utilizes interrupts for LED flashing.

The following Chapter 4 describes en early development process of the TC49x device, analyses the simulation environment, and depicts encountered problems and their workarounds for the first hardware samples.

The processing capabilities of a Parallel Processing Unit unit for offloading computationally intensive operations and algorithms are provided in Chapter 5. It also builds support for booting PPU and notifying TriCores using inter-core interrupts. Chapter 6 specifies an inter-processor communication between scalar and vector cores using shared memory and software mailboxes and creates an application demonstrating the use of a vector processor for offloading tasks. Following on from this, several experiments are carried out to demonstrate the advantages and disadvantages of using a vector processor within the TC4xx architecture in Chapter 7.

Last but not least, the possibilities of speeding up development with the use of high-level tools that enable code generation are studied in Chapter 8. The following Chapter 10 specifies the requirements of modern ADAS on embedded devices and compares frequently used architectures. A model-based approach using Matlab Simulink is applied to model and deploy the estimation algorithm using a Kalman filter. The neural network mapping tool is used to tailor CNN for the target embedded implementation.

1.1 Motivation

This thesis was created in cooperation with HighTec EDV company, a supplier of Toolchain for AURIX Tricore architecture. This cooperation allowed us to use the prototype of the development board, draft versions of other development tools, and documents currently unavailable to the general public. The following projects have been developed and documented in details and are available to HighTec's customers as a support for TC49x compiler:

- tc49x-bsp-example [7]
- tc49x-uart-example [9]
- tc49x-ppu-base-example [8]

Chapter 2
Aurix TC4xx

In 1999, the first TriCore microcontroller family called AUDO was launched. Its focus was real-time performance and functional safety features. It was used in many automotive applications like braking systems, electric power steering systems or airbags.

The following TC2xx family has adopted a multicore architecture which significantly increased its performance. Thanks to this design, several safety applications can run simultaneously on one platform, which subsequently speeds up achieving the ASIL certification necessary for critical systems in automotive.

Another TC3xx family has increased the number of cores to 6 and extended its peripheral set of Signal Processing Units, Gigabit Ethernet and additional CAN FD and LIN interfaces. Together with a higher temperature range and 40 nm flash technology [3], the use case has further expanded to domain control and data fusion applications.

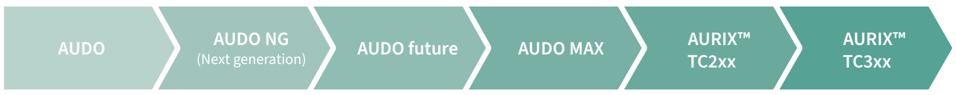


Figure 2.1: Evolution of TriCore families prior to TC4xx [3]

To further increase the safety and comfort of drivers, a single ECU has to now provide even more computing resources to integrate multiple functionalities [20]. The latest driver assistance systems based on cameras and LIDARs like automatic lane-following, drowsy driver detection or intelligent cruise control require a higher speed of communication peripherals and greater throughput of the image-processing algorithms. Some even adopted solutions based on deep learning algorithms that are especially resource-intensive and can not be effectively executed on multi-purpose CPUs. These reasons have led to the emergence of a new TC4xx family. It is built on top of the previous family but offers better performance thanks to increased frequency up to 400MHz and larger memory units. The essential functional blocks are shown in Figure 2.2.

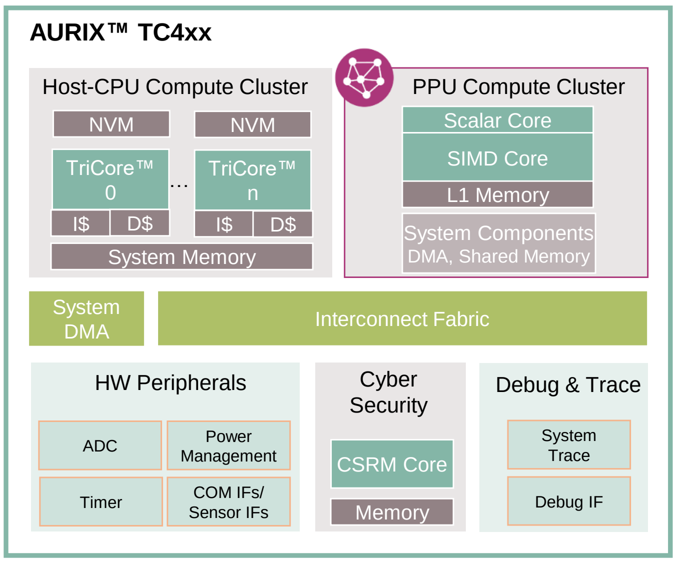


Figure 2.2: Functional blocks of TC4x [24]

Compared to the previous version, TC4xx architecture has several new features and offers many more peripherals, but in this thesis, only the most important ones are mentioned:

2.1 Compliance with Safety Requirements

Functional safety is described in detail in the 2018 ISO 26262 standard and defines automotive safety integrity levels in the automotive industry. There are four levels, ranging from ASIL-A with the lowest requirements up to ASIL-D with the highest demands to suppress potential risks. The Aurix architecture is designed to create applications with the highest level of ASIL-D certification. It uses two-level memory access protection, so different processes can access only memory blocks allocated in advance. Another safety feature is a resource management unit. It implements a protection mechanism for peripheral access.

2.2 Virtualization

As the computational power of each computing unit increases, one CPU must serve multiple applications. For the certification mentioned above, it is necessary to satisfy hard real-time constraints and ensure that the applications do not interfere with each other and do not use similar hardware resources. When several processes run on one CPU, their execution must be planned, and they take turns in the running. Every such exchange is accompanied by a context switch when current values of essential registers are saved for later use. This might bring a significant overhead in software virtualisation and decrease the real-time capabilities.

In the case of TC4xx, hardware-based virtualisation solves this issue. It is the process of isolating applications running on shared hardware using virtual machines. Virtual machines act as isolation containers and can be represented by a bare-metal application, task or even fully operating RTOS. There are three levels of VMs, and each has a dedicated resource set. In total, up to 8 virtual machines can run on one core:

- Hypervisor - runs and schedules other VMs.
- Real-time virtual machine - has its own resource set. Context switches are done in hardware, which brings little overhead and, therefore, it is suitable for time-critical applications.
- Up to 6 other VMs - Share one resource set. A hypervisor must handle the context switches, so they are used for lower priority tasks.

In summary, the concept of hardware virtualisation enables the combination of several apps on one MCU. Thanks to two-layer memory protection and resource allocation mechanisms, the applications are independent of each other from a hardware point of view. The VMs can be developed using different tools according to their safety requirements. They might run on top of different software stacks (AUTOSAR), as Figure 2.3 shows. Existing monolithic applications could be split into smaller specialised ones, increasing their security and making the necessary certification process more manageable.

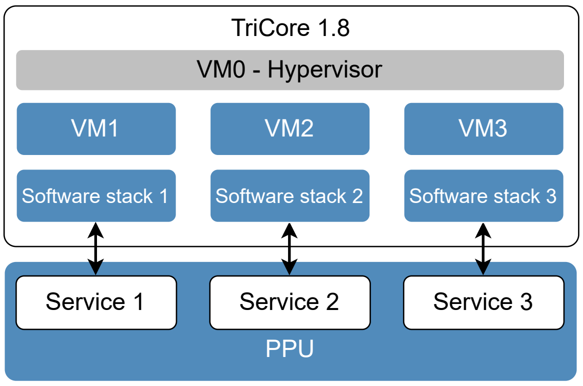


Figure 2.3: Virtualization in TC4xx

Hypervisor

A Hypervisor is a combination of hardware and software that creates and runs virtual machines. It must be able to configure and allocate the available memory and peripheral resources requested by VMs. It is also responsible for their scheduling based on selected policy, i.e. Round Robin or Priority scheduling. However, the primary responsibility is to configure and monitor isolation between applications (freedom-from-interference), which is also crucial from the safety and security point of view. Some functionalities like interrupt forwarding or context switching pervade inside the hardware.

2.3 Cyber-security Cluster

Even in safety-critical applications, MCUs use sensor data from different domains. If the cross-domain communication is not secured, it is susceptible to potential manipulation. A cyber-security cluster is a combination of modules and functions that support safety and security standards defined by ISO 26262 [28]. It contains hardware accelerators for different hashing algorithms and AES symmetric cryptographic operations. It is also capable of accelerating higher-level secure protocols like IPSec.

2.4 Parallel Processing Unit

Existing safety-critical MCUs cannot deliver the required performance to execute computationally intensive algorithms like image-processing or sensor fusion, which are inherent in ADAS applications. Automotive uC vendors (including Infineon) are increasingly adopting heterogeneous architectures that incorporate a high-performance processor specialised in matrix and vector applications. The parallel processing unit is a vector processor based on DesignWare EV71 from Synopsys. The Vector DSP Unit is its central part and allows SIMD (Single Instruction Multiple Data) computations based on 512-bit wide vector registers designed for high-performance parallel programming. The programmable CNN engine is also a part of the PPU and is optimised to compute Multiply-Accumulate (MAC) operations, which is the basis for inferencing neural networks. Its functionality is further described in Chapter 5.

2.5 Summary

The AURIX TC4xx family of microcontrollers was created to meet the increased requirements for performance, safety, and individual parts' ability to work independently of each other. The heterogeneous architecture consists of scalar computational units focused on functional safety, a cyber-security cluster to secure cross-domain communication and a parallel unit designed for computing complex algorithms and inferencing neural networks.

Chapter 3

Basic Software Support

TC4xx is at an early stage of development. From the compiler vendor's side [21], it is crucial to create and provide basic software support for the customers as a basis for further development.

A board support package (BSP) is an essential code and other tools required to run a computer hardware device. In addition to the previous, a BSP can contain directives and compilation parameters.

The term BSP is mainly used in the context of real-time operating systems. The reason for using this term is that this thesis will serve as a basis for developing a certified real-time operating system PxRos on the TC4xx architecture.

Software support must perform all necessary initialisations and, at the same time, be easy to use. The following is a more detailed list of expected functionalities:

- Initialisation of all the necessary things (stack pointer, trap vectors, vector table, ...).
- Initialisation system and peripheral clocks to a maximum frequency according to [26].
- Easy interrupts handler registration.
- Implementation of trap handlers for better error tracking
- All the memory access protection features, as well as safety features (Secure Core), are turned off. They are used in the later stages of a development cycle.
- Virtualisation has to be turned off. The functionality will be used primarily with the real-time operating system.
- Shared-code implementation - all the TriCores execute the same code.

Based on the previous requirements, the basic software support has been created and consists of the components shown in Figure 3.1.

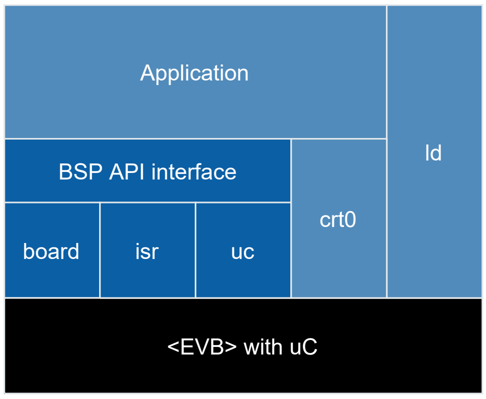


Figure 3.1: Components of BSP [7]

3.1 *C* run-time Initialization

C run-time initialisation is a set of execution startup routines that performs any initialisation required before calling the main function. It is usually written in assembly language because function calling generally requires a stack to be initialised, which is not the case at the beginning of run-time. Because the stack is decreasing in Aurix architecture, it is initialised simply by setting the corresponding register to the end address of the memory segment allocated to it in the linker script (section 3.3)

CSA Initialization

The state of a task or function is defined by its context. It is a set of values of address registers. When a function call occurs or when an interrupt is triggered, the context of the currently running task of function must be saved for later use. Context save areas is a special feature for Aurix™architecture [24]. It is a linked list of saved contexts and must be initialised prior to any function call or interrupt.

Global Variables Initialisation

Global initialised, and uninitialised data are stored in the volatile memory during run-time. The following script shows examples of both variable types:

```
int var_init = 7;
int var_uninit;

void main(void)
{
   static int var_static_init = 9;
   static int var_static_uninit;
}
```

Volatile memory maintains its data only when the device is powered. Due to this reason, global initialises data are loaded into the flash memory. At the beginning of run-time, they must be copied to the RAM at the beginning of the program. Linker script (described in 3.3) groups them together and creates *copy tables*. One entry of the copy table contains:

- Source address – start address of the table stored in a flash,
- Destination base address – start address in target memory,
- Memory size that should be copied.

In the case of uninitialised data, the initial value should be 0. It is sufficient only to clear the target memory. Linker script also creates *clear tables*, where one entry contains the destination base address and memory size that should be cleared.

3.2 BSP

The basic set of functions consists of three sub-modules as shown in Figure 3.1. This ensures modularity for different versions of the microcontroller or evaluation board.

Uc Sub-module

The functions in BSP uC sub-module are tied up to a particular uC version. In this case, there is just one TC49x step A version.

The sub-module contains functions for initialising the system and peripheral clocks. It also allows to disable the watchdog and register a vector or trap table. It also enables to work with the system timer, which is later used to generate interrupts. The complete table of created functions can be found in Appendix C.1.

ISR Sub-module

Interrupts are used to respond to asynchronous requests from a specific part of the microcontroller that needs to be serviced.

A trap is a form of interrupt that occurs due to an exception or illegal access. Because the traps are always active, trap handlers help users identify the issue in the program.

The interrupt sub-module creates interrupt and trap tables and registers a dummy handler for all the interrupts. Thanks to initialised RAM handler tables, it also creates an easy-to-use interface to register an interrupt handler. A complete table of functions is available in Appendix C.2.

The main reason for ISR being a separate sub-module is that RTOS' micro-kernels handle interrupts themselves. In such a case, the module is not used.

Board Sub-module

Some functionality of the microcontroller depends on the evaluation board used. The functions of this sub-module take into account the mapping of peripherals to hardware pins and allow the initialisation and use of LEDs or external crystal oscillator. A complete table of functions is available in Appendix C.3

3.3 Linker Script

The compilation process in C consists of 4 steps: pre-processing, compilation, assembly and linking.

The linker takes output of the assembler - one or more objects or archive files and combines them into an output file (an executable or another object file).

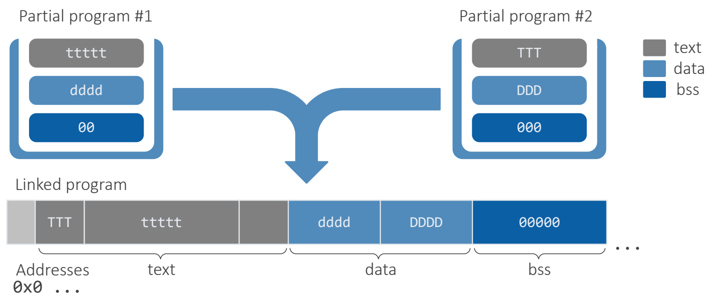


Figure 3.2: Linking of an executable file [21]

In the process of linking, a linker script is run. It must be aware of the memory regions, their sizes and the requirements for accessing them. The data are grouped according to their type into sections:

- .text - an executable section that stores code,
- .data - global variables with non-zero initial values,
- .rodata - constants that can be read-only,
- .bss - uninitialized global data with default 0 value.

The sections are then placed in memory so that each Core can access its data efficiently. As was already mentioned, the initialised data must be copied into RAM at the beginning of the run-time. For this reason, the sections have two addresses associated with them:

- The virtual memory address, where the program expects to find the section at run-time.

- The load memory address, where the loader places the code.

The linker script creates clear tables and copy tables from these addresses and section sizes, which are then provided to the C run-time routine to initialise the data correctly.

3.4 Example Application

A simple multi-core application was created to demonstrate the correct functionality of the developed tools. The timer of each Core periodically generates interrupts. When an interrupt is serviced, the LED on the evaluation board flashes, as shown in Figure 3.3.

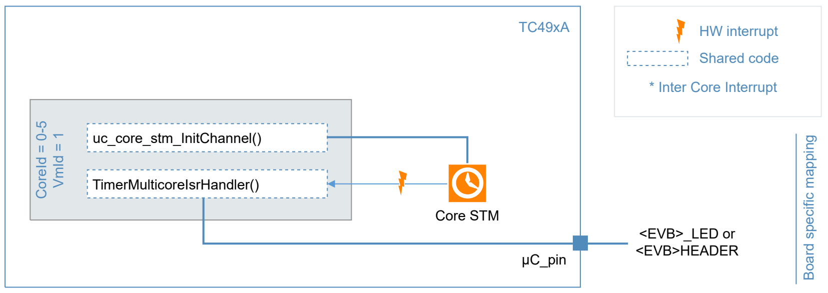


Figure 3.3: Application view of the example [7]

Execution Flow

After power-on reset, only TriCore0 starts its execution. Other cores, including PPU, are kept in reset.

Certain functions, such as starting other CPUs or initialising peripherals, are executed only by the reset Core(Figure 3.4). Because the shared-code approach has been used, the execution of some functions is conditional on the identification number.

At first, the stack and context-saving units are initialised, which are necessary for function calls and correct interrupt handling. Then, the *pre-init* function is executed, which registers vector and error tables, enables access to peripherals and disables the watchdog. These functions require the highest access rights of the hypervisor. In the last function before *main*, the clock and used peripherals are initialised, including the LEDs on the development kit.

After entering *main* function, the interrupt service routine is registered to handle interrupts triggered by a core timer. In the end, the interrupts are globally enabled, and the LEDs blink indefinitely at frequencies proportional to their ID.

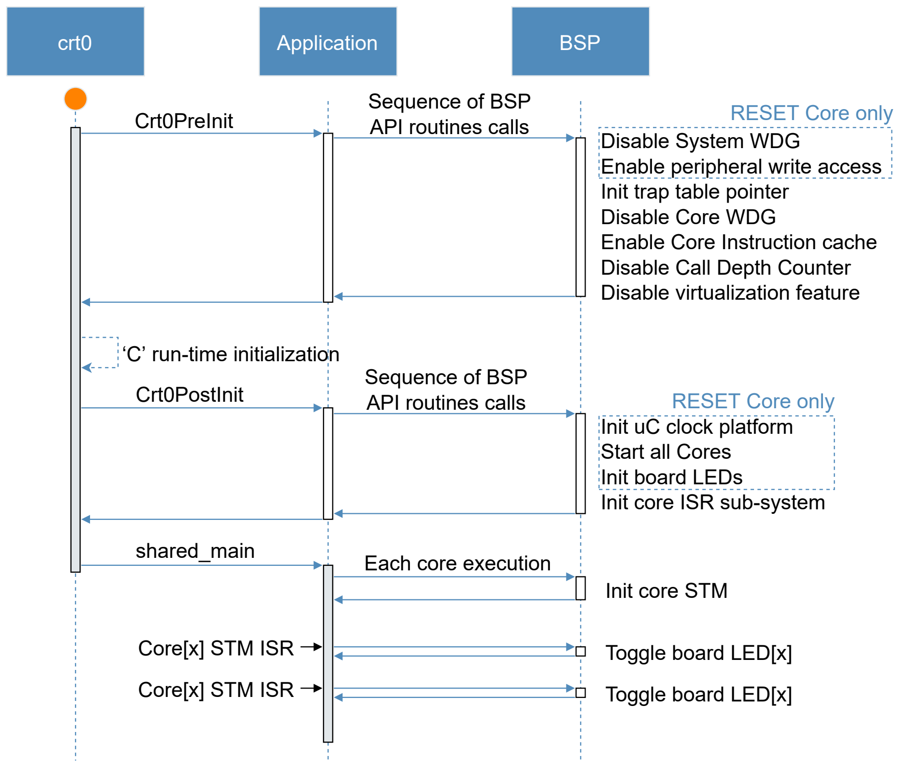


Figure 3.4: Execution flow of the example [7]

3.5 UART Example

Despite various debugging tools, writing to a terminal application via UART is still a frequently used technique. For this reason, BSP was extended with this functionality.

The following features are required from this module:

- use it calling *printf* function which supports commonly used format, length and precision specifiers
- multi-core functionality
- possibility to redirect the output to another peripheral

Asclin Module

The Asclin is a hardware peripheral present on Aurix evaluation boards. It could be configured for ASC (UART), LIN, and Master SPI applications [22]. Asclin module utilises a set of functions for transmission and reception of data via the UART interface in the polling mode.

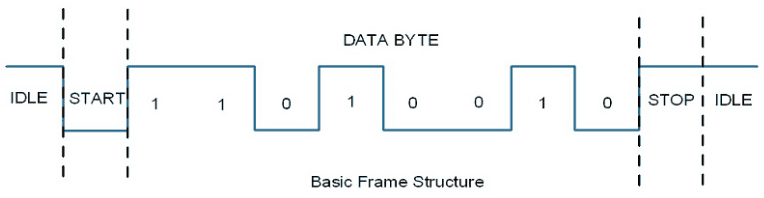


Figure 3.5: UART data frame [51]

For successful communication, both sides must have equally defined frame parameters and communication speed parameters - Baudrate. Asclin module sets the communication speed to:

```
Baudrate = 115200 Bd/s
```

and the frame is composed similar to the Figure 3.5:

1 start bit	8 data bits	1 stop bit

The complete list of functions is in Appendix C.6.

Mutex Module

In the multi-core application, it is necessary to control access to the shared resource (UART). Mutex module provides simplified functionality of handling exclusive access to the shared resource. The mutex is defined as a structure consisting of:

- State of the mutex (locked/free),
- Owner of the mutex (core ID).

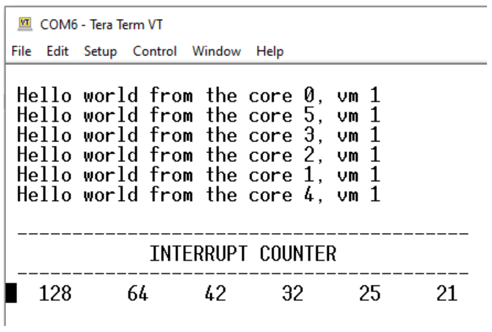
The mutex-locking function uses `CMPSWAP` atomic instruction, which conditionally swaps a source register with a memory word. The execution of an atomic instruction forces the completion of all data accesses semantically ahead of the instruction, which ensures that two TriCores can not change the mutex state simultaneously [24]. The complete list of functions is in Appendix C.4.

Printf Module

Toolchain Standard Library is not suitable for multi-core applications when running a shared code. For this reason it was necessary to create a module that consists of re-entrant functions, is independent of other libraries and at the same time is compatible with standard functions. Printf module is an implementation of C's formatted printing family of functions (C.5) with the primary use case in embedded systems based on [33]. It supports all standard specifiers and flags and all width and precision sub-specifiers (shown in Appendix D). The module does not depend on other packages. It only requires *putchar* function to be defined by a peripheral, where the output is redirected. In the initial phase, it was redirected to the simulation output and after releasing the hardware prototype into the UART channel.

3.5.1 Application

The application is based on a basic example and adds UART functionality. Each TriCore sends a "Hello world" message via UART at the beginning of the run. In the *main* function, reset Core periodically sends current interrupt count as Figure 3.6 shows.



```
Hello world from the core 0, vm 1
Hello world from the core 5, vm 1
Hello world from the core 3, vm 1
Hello world from the core 2, vm 1
Hello world from the core 1, vm 1
Hello world from the core 4, vm 1

-----------------------------------------
            INTERRUPT COUNTER
-----------------------------------------
  128     64     42     32     25     21
```

Figure 3.6: Terminal window view of the application output [9]

3.6 Summary

In this chapter, the basic building blocks of the BSP were described and implemented.

- A set of functions dependent on the evaluation board and microcontroller used.
- A linker script that correctly places the data into different memory blocks and generates executables.
- A multi-core application that demonstrates proper booting sequence, interrupt handler registration and interrupt handling itself.

To simplify debugging, the functionality has been extended by the possibility of writing to a UART channel with implemented shared resource protection for multi-core applications.

Chapter 4

Preliminary Phase of Development

The initial development phase of an embedded MCU takes place before the official release. During this period, support from the compiler and debugger providers and uC vendor is gradually built up. This stage, however, comes with several challenges:

- Toolchains might contain bugs, which are progressively fixed by frequent patch releases based on the feedback of other developers.
- The supporting documents are in draft versions until the design is finalised.
- As TC4xx is a heterogeneous architecture, several parties have to participate in the development, making the whole process more complex.
- In the initial phase, development boards were not available, so the virtual prototype TC49x VKD was used [25]

  A virtual prototype is an executable software model that runs on a host system. It emulates the hardware, including CPU instruction sets, memory maps, registers, and interrupts, at a sufficient level that can be tailored for software development. From a software perspective, it is binary-compatible with the hardware being emulated, allowing users to run unmodified binary images of the entire software stack. It is a complete functional representation of the target system on which to develop software [36].

Advantages of Virtual Prototyping

Using a virtual model in the early stages of development brings several advantages. Firstly, the system can be validated before the hardware design is completed. This way, many serious flaws or bugs can be found, and the design can be adapted, which reduces the number of iterations of hardware prototypes before the official release.

Nowadays, traditional development methods, where software is developed only after the completion of chip design, do not meet product development schedules. An instruction-accurate virtual prototype can be used instead. It means that unmodified binary can ideally be executed on the actual hardware. In reality, it was necessary to make specific changes, but the example was functional within days of silicon availability.

The simulation gives complete visibility into the memory and registers and gives the possibility to print messages to the output easily. Moreover, warnings and errors are reported without the need to check uC registers so that the program can be easily tracked.

Disadvantages of Virtual Prototyping

Using a virtual development model brings with it potential problems. The quality of the simulator is determined by how detailed the processor is modelled, which brings a trade-off between speed and precision. Cycle-accurate models require excessive computational power and take a long time to model. On the other hand, they can be used for benchmarks, which is not the case with VDK.

Another problem is that virtual models cannot describe the real behaviour of hardware peripherals. An example is the frequency stabilisation of a crystal oscillator or communication peripherals that python scripts must emulate.

4.0.1 First HW Sample

After releasing the first hardware prototype, it was necessary to add functionality such as pin mapping. It had to be taken into account that not all peripherals were fully functional, and some more complex communication peripherals like Gigabit ethernet could not be used. The use of the first hardware prototypes equally brings advantages and disadvantages.

On the one hand, it is possible to benchmark the applications, and programs can be executed using actual peripherals, which is not possible by using a virtual prototype.

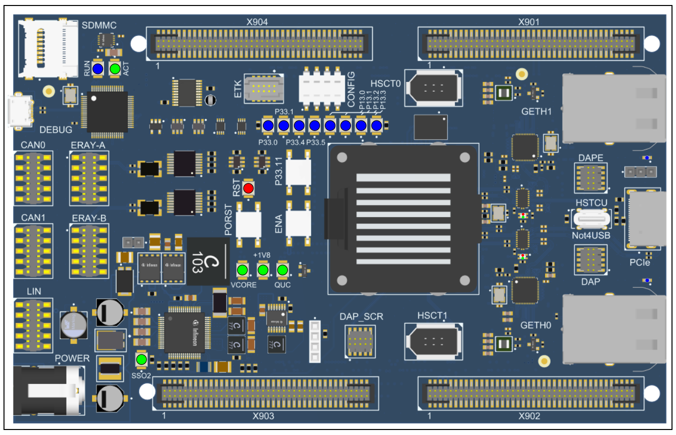


Figure 4.1: Triboard TC499A COM [27]

On the other hand, the first hardware samples almost certainly contain bugs. In the case of simulators, the error can be quickly corrected, but with real hardware, possible workarounds have to be found until the next batch is released.

The use of a heterogeneous architecture only makes these problems worse. An embedded-application binary interface (EABI) specifies standard conventions for file formats and data types. Conflicts in EABIs of different architectures cause inter-processor communication to be restricted only to data types interpreted similarly by both platforms. Moreover, incompatibility between different toolchains makes it impossible to fuse projects of distinct architectures into one executable.

4.1 Summary

The initial phase of development is an integral part of the uC cycle, in which support is being built, and bugs are being removed. Communication between partners, flexibility in fixing bugs and providing patch versions and workarounds are the most important in this phase. Although the simulation platform is not fully compatible with real hardware, it can speed up development thanks to good visibility and debugging capabilities. After the release of the evaluation boards, the code can be quickly adapted, and the necessary tests can be performed.

Chapter 5

Parallel Processing Unit

A parallel processing unit is a processor optimized for high-performance embedded signal processing or vision applications. It is a part of the TC4xx to meet the performance requirements of automotive real-time applications.

As seen in the following figure, the PPU combines Synopsys DesignWare ARC EV71 and infrastructure components that help to link essential signals (clock, interrupts) to other architecture parts.

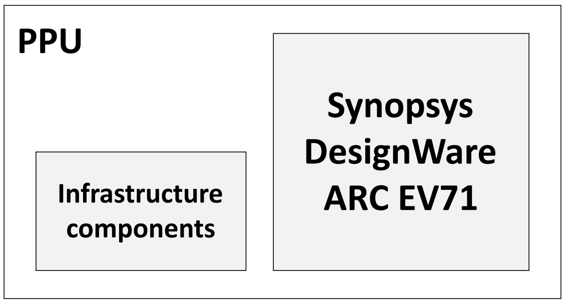


Figure 5.1: PPU Components [23]

It integrates a 32-bit scalar core, 512-bit vector DSP and a deep neural-network accelerator.

The critical feature of DSP is the possibility to execute Single Instruction, Multiple Data (SIMD) instructions. In one instruction, the same operation is done on all vector elements (or between two vectors) concurrently. This way, the execution of applications with a high level of data parallelism can be significantly accelerated compared to scalar processors. DNN accelerator is a part of the PPU, adapted for executing neural networks that are increasingly being used in automotive applications.

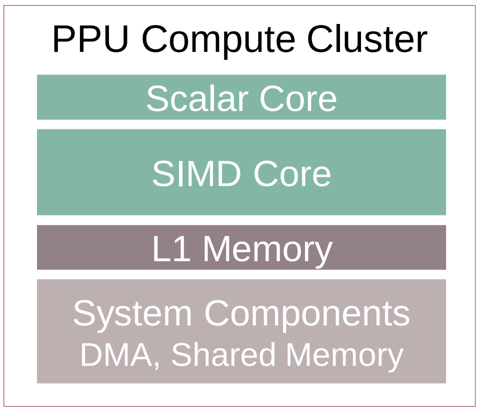


Figure 5.2: Functional blocks of PPU [24]

5.1 Scalar Core

The Scalar Unit fetches and decodes instructions.

Thanks to Very Long Instruction Word Architecture, it can concurrently execute the instructions in different processor parts. Scalar, branch and jump instructions are executed by the scalar unit. When a vector instruction is identified, it is sent to the Vector DSP unit and executed in one out of three execution slots. Therefore, it can execute three vector instructions and one scalar instruction each cycle. Another functionality of scalar core is inter-core interrupt handling. This makes it possible to notify TriCores with minimum delay.

5.2 Vector DSP

Vector DSP unit is closely coupled with scalar core and has a size of 512 bits [37]. Depending on the element size, it can contain 16 32-bit elements, 32 16-bit elements or 64 8-bit elements.

Vector instructions are single-instruction, multiple-data instructions that operate on vector registers in different possible scenarios [39]:

- Vector-Vector Instructions - operate on vector registers (in the vector register file) as source and destination operands
- Vector-Scalar Instructions - The value in the scalar register is duplicated N times for a vector length of N elements, and then used as the second source operand in a vector operation.
- Vector-immediate Instructions - The immediate value is duplicated N times to the vector width of N elements and then used as the second operand in a vector operation
- Vector Reduction Instructions - The vector elements are reduced to

a single element based on the operation (sum/minimum/maximum of adjacent elements)

The Table 5.1 shows examples of vector instructions together with equivalent execution by a scalar core.

Instruction type	**Example**	**Serial execution**
Vector-Vector	`vvadd vr0, vr1, vr2`	`for (i=0; i<N; i++)` `vr0[i]=vr1[i]+vr2[i]`
Vector-Scalar	`vvadd vr0, vr1, r1`	`for (i=0; i<N; i++)` `vr0[i] = vr1[i] + r1`
Vector-immediate	`vvadd vr0, vr1, 1`	`for (i=0; i<N; i++)` `vr0[i] = vr1[i] + 1`

Table 5.1: Examples of vector instructions

Vector instructions execute the same operation on all vector elements. For correct execution, all vector elements must have the same data type. Furthermore, these instructions could be applied only to data stored in the vector memory, which serves as Level 1 (L1) data memory for vectors in the system memory hierarchy.

Due to this reason, algorithms generally employ a pipeline consisting of three tasks:

- Bring input data from system memory into VCCM.
- Process the data using vector instructions.
- Bring output data from VCCM to system memory.

This pipeline could be optimized using the Streaming Transfer Unit, which can simultaneously bring the data in and out of the vector memory during processing.

Static data used only by vector processor should be marked with the *_vccm* keyword so that Metaware compiler [39] places them in VCCM.

The Vector DSP has a floating-point unit extension that supports additional math functions like:

- Reciprocal square root
- Sine function
- Cosine function
- Logarithm to the base 2
- Exponentiation with base 2

This feature is used in high-level Model-Based design (described in Chapter 8), where a code-replacement library is used to generate highly optimized code for the PPU.

5.3 DNN Accelerator

A DNN accelerator employs a specialized architecture for the execution of neural networks, which require fast memory access and high performance. According to [12], inferences in CNNs are primarily dominated by multiply-accumulate operations that implement the networks' convolution operations. In some cases, MACs can represent more than 95% of the total arithmetic operations included in the CNN task.

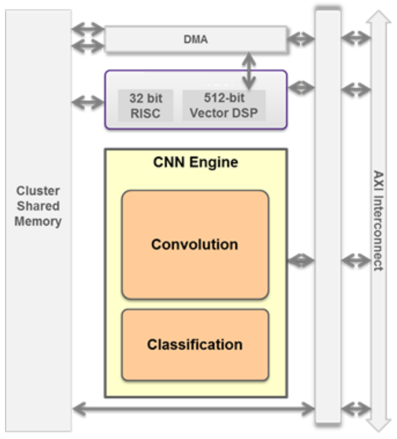


Figure 5.3: PPU CNN engine [15]

The neural network models are trained in floating-point arithmetic. On the other hand, embedded devices have limited resources, and float operations are energy-intensive. For this reason, they mostly use fixed-point arithmetic, i.e. 16-bit, which slightly decreases their accuracy but enables them to be executed efficiently on DNN accelerators. Chapter 8 describes this technique in detail.

5.4 PPU-booting Example

This project aimed to extend the functionality of the BSP example by booting a parallel unit and generating notification inter-core interrupts, as is shown in Figure 5.5.

After a power-on reset, TriCore0 is the only core that starts execution. Other cores, including PPU, are held in reset. For the proper functioning of the parallel unit, it is necessary to execute the series of steps shown in Figure 5.4 to boot it and initialize its data:

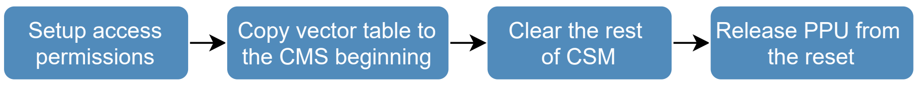

Setup access permissions
Copy vector table to the CMS beginning
Clear the rest of CSM
Release PPU from the reset

Figure 5.4: PPU boot sequence

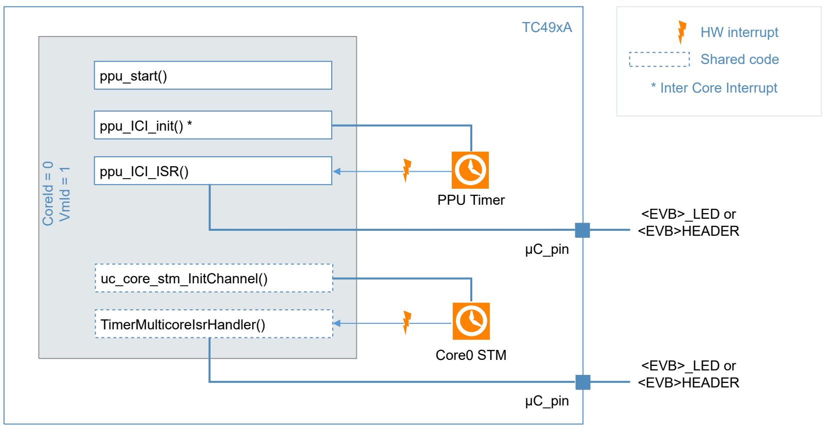


Figure 5.5: Application view of PPU booting example

In the first step, giving the parallel unit the right to access some memory blocks is necessary. The parallel processing unit runs on top of the Synopsys Processor Execution Environment [45], which implements all the initial run-time routines. Execution starts from the Shared Memory Cluster block, a volatile memory block that retains its data only while the device is powered. For this reason, its vector table is placed in flash memory and must be copied to the correct location before the execution of main function starts. In addition, the rest of the cluster-shared memory block must be deleted to prevent possible data inconsistency. Finally, the PPU is released from reset by clearing the corresponding uC register, and the TriCore must register a routine for an inter-core interrupt, which is periodically triggered.

In the second step of the booting sequence (Figure C.7), the running Tricore must know the address and size of the PPU vector table to copy it into the target memory. However, it is located in another project, and the memory address and size are resolved by Metaware linker script [40]. A possible solution would be to link the two projects together. However, this is a problem because TriCore and the PPU use incompatible compilers. The only solution to this problem is to define these parameters statically in the TriCore project:

```
#define VECTOR_TABLE_LMA    (0xB0680000) // lmu7_noncached
#define VECTOR_TABLE_SIZE   (0x190u)
```

Moreover, every change in the PPU project has to be applied to the other one.

Chapter 6

Inter-processor Communication

In Chapter 5, PPU booting and triggering of inter-core interrupt in TriCore was resolved. For successful task offloading, the devices must be able to exchange the data and synchronize their action. This mechanism is called Inter-processor communication.

The most important aspect of defining IPC is the memory layout. TC4xx uses one shared address space with several memory units that are accessible from all processors. In this case, local volatile memory for general purpose usage (LMU RAM) is dedicated to information exchange. Because all processors can read from and write to the shared memory, mutually exclusive access to avoid data inconsistencies and race conditions must be implemented. Chapter 3 covered mechanisms which can be employed to address this issue.

6.1 IPC using software mailboxes

Communication via global data is hard to track and error-prone since protection mechanisms are application dependent. To suppress this problem as much as possible, data structures - mailboxes and data exchange functions can be used.

In this implementation, a mailbox is a data structure used for Inter-processor communication. It contains all the necessary information for safe data exchange:

- ID of the requested service
- *data ready* flag, which notifies PPU that the request is pending
- execution status of the request
- number of arguments
- pointers to data arguments

Each TriCore uses its mailbox, and, together, they are statically allocated in the shared memory. They must be accessible by both platforms using the same physical addresses. As can be seen in Figure 6.3, there is also a PPU mailbox which stores its current status and a data structure - mutex, which is used to provide exclusive access to the memory block. In the context of TC4xx architecture, the PPU acts as a producer - it can compute several tasks, also referred to as kernels. These kernels are parts of the application that will benefit from its vector DSP capabilities.

Request from TriCore

Communication is done by remote procedure calls. When a TriCore wants to use the services of the PPU, it has to lock the shared memory at first. If another core uses the memory, it must wait until the resource is unlocked. It then updates its mailbox with the requested function, the number of input parameters and the pointers to the input parameters. Finally, the *data ready* flag is updated, and access to the memory is passed to the PPU. While the call is being processed, the client is blocked. An inter-core interrupt signals the TriCore that the kernel has been executed, and output data of executed function can be accessed.

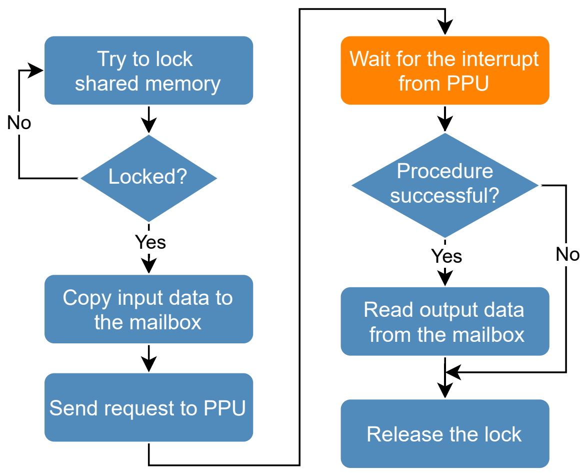

Try to lock shared memory
No
Locked?
Yes
Copy input data to the mailbox
Send request to PPU
Wait for the interrupt from PPU
Procedure successful?
Yes
No
Read output data from the mailbox
Release the lock

Figure 6.1: Remote procedure call [8]

Request Processing on PPU Side

After the initialization is done, the parallel unit updates its mailbox and gradually goes through the mailboxes until it hits the data-ready flag. In the case of the TC4xx microcontroller, only TriCore units can use the atomic function for exclusive memory access and therefore, the *data ready* variable in each mailbox is used for inter-core synchronization. The input data is then moved to vector memory, where most of the calculations are performed. In the end, the associated inter-core interrupt is generated, and the whole process is repeated.

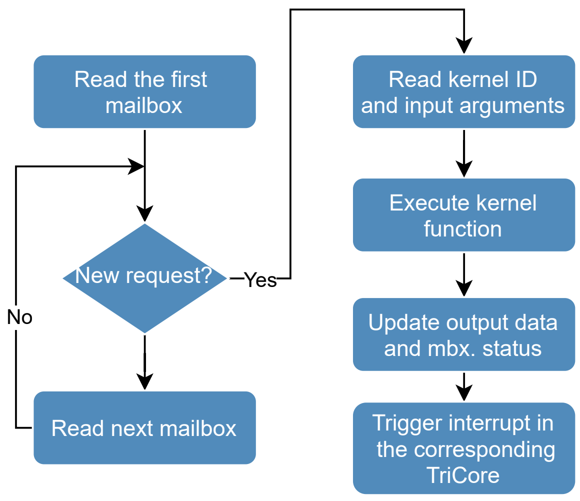

Read the first mailbox
New request?
Yes
No
Read next mailbox
Read kernel ID and input arguments
Execute kernel function
Update output data and mbx. status
Trigger interrupt in the corresponding TriCore

Figure 6.2: Remote call processing [8]

6.2 Example Using IPC

First of all, a section for shared memory had to be created in the linker script of both projects. Then mailboxes and mutex were statically placed in it to ensure exclusive access.

According to the previous description of the remote procedure call, the following functions were implemented:

- TriCore request
- PPU request processing

Finally, algorithms commonly used in automotive applications, which are at the same time well parallelizable, must have been developed.

Examples of such applications are:

- fast Fourier transform, which is widely used in signal processing i.e. in Radar Signal Processing
- The Kalman-based algorithm which is used in sensor fusion algorithms as well as in estimation algorithms.
- Neural network graphs which are increasingly used in vision-based driver assistant systems.

The first two algorithms have been implemented using a specialized Vector DSP library [43] and a model-based approach (described in Chapter 8). Since the neural network model could not be deployed on the evaluation board, this kernel was replaced by a matrix operation from the Linear Algebra library [44]. The final project has the structure shown in Figure 6.3.

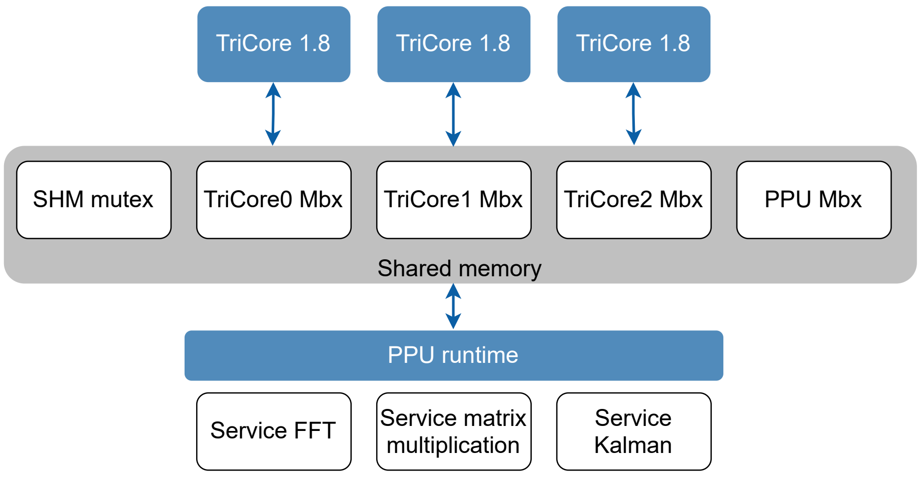


Figure 6.3: IPC setup between TriCores and PPU [8]

6.3 Potential Improvements

Dispatcher[47] and Bare-metal Low-lever Driver[46] are static libraries to communicate between a TriCore processors and the PPU. Dispatcher is designed for the PPU and runs on top of Synopsys run-time environment [45] while the second one is designed for TriCores. Together, they work on a similar principle as the implemented inter-processor communication. All communication with a host goes through a shared memory channel. As a doorbell mechanism, cross-core interrupts are used. They offer more extensive functionality in comparison to the implemented version.

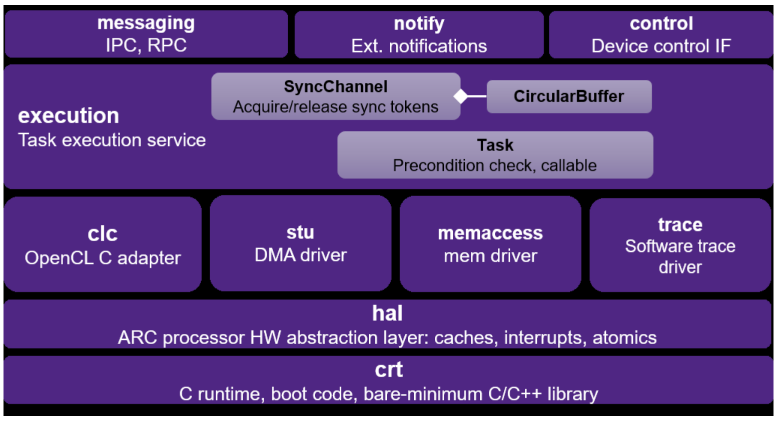


Figure 6.4: Components of SPEED runtime [45]

The primary improvement is that the communication takes place on multiple priorities: Normal and Emergency. This reduces the chance that a high priority emergency task will be waiting at the expense of lower priority ones.

In addition, a Notification buffer is implemented, where the PPU reports all critical information about execution and possible errors. In remote procedure calls, callers generally do not know whether the call was actually invoked. These problems can be avoided if a notification system is implemented.

A parallel unit can execute a periodic task and thus address the Producer-Consumer problem. In it, the PPU periodically produces items and stores them in shared memory. On the other hand, the consumer checks the availability of items and eventually waits for the producer to create them. In this way, it can deterministically schedule the execution of the PPU and thus achieve lower latency than with asynchronous item requests.

Because the current version does not support the development board, so it was necessary to implement a custom inter-processor communication.

6.4 Summary

The goal of this Chapter was to implement inter-core communication between vector and scalar processors to allow offloading of computationally intensive algorithms to the parallel processor. Thanks to the memory layout, it was possible to use shared memory block for data exchange. To avoid multiple accesses, a statically allocated structure - mailbox was defined for each scalar core, and the atomic function `CMPSWAP` was used for exclusive memory access.

The offloading process was implemented via remote-procedure calls. The scalar processor must first gain memory access, then shares data with the vector processor, and waits until the task is executed. In the implementation process, it was found that some data are interpreted differently due to different application binary interfaces of AURIX and ARC architectures. In order to avoid possible errors, the data exchange was limited to specific data types only. The created application captures a typical example of using a vector processor to compute:

- A Fast Fourier transform implemented in [43].
- An estimation algorithm based on the Kalman filter implemented in Chapter 10.
- Matrix multiplication linear algebra operation implemented in [44].

Chapter 7

Application Vectorization

A parallel processing unit is used to improve the application performance offloaded by scalar TriCores. It uses parallel computing to achieve speed-up, in which several calculations or processes are carried out simultaneously.

I particular, two types of parallelism are used:

- Data-level parallelism
- Instruction-Level Parallelism

Instruction-level Parallelism

Instruction-level parallelism is the simultaneous execution of a batch of instructions. DesignWare EV71 has a Very Long Instruction Word Architecture with three parallel vector data paths. Since they are all scheduled simultaneously, three vector and one scalar instructions could be executed each clock cycle, which might significantly speed up the task. Most of the parallelisation is carried out by a Metaware compiler[39], but there are some general guidelines which might improve the compiler's performance:

- use of unconditional processing loops with a high level of data parallelism as [38] suggests,
- use the supportive macros for how many times a loop is expected to execute ,(`__builtin_assume()`)
- use the supportive macros for loop unrolling (`#pragma clang loop unroll()`) [48]. Loop unrolling replaces the loop with an enumerated sequence of loop iterations which eliminates the loop control overhead.

On the other hand, this architecture also has a potential problem, where a stall (cache miss) in one of the issued slots will stall the execution of the entire processor. In this case, it is even more critical to avoid cache misses than for scalar processors. Because the impact of the Instruction-level parallelism is very application dependent, it will not be evaluated in this chapter.

Data-level Parallelism

Data-level parallelism occurs when instructions from a single stream operate concurrently on multiple data elements. The parallel processing unit has 512 bit-wide register, which means that the same operation could be performed on 16 single-precision floating-point variables.

As was stated in Chapter 5, PPU's Vector DSP unit has three parallel vector data paths, which means that it is capable of executing three vector instructions and one scalar instruction each cycle.

The improvement of the task execution time can be evaluated by the term *speed-up*, which is the gain in speed made by parallel execution compared to sequential execution of the task.

$$s = \frac{T_{DSP}}{T_{seq}} \tag{7.1}$$

In an ideal case, the PPU can theoretically execute 49 single-instruction floating-point addition operations in one cycle, whereas the scalar processor requires 49 clock cycles. This task was perfectly vectorizable with speedup:

$$s = \frac{3 \cdot 16 + 1}{1} = 49 \tag{7.2}$$

Amdahl law (7.3) can be used to estimate the speed-up of a more complex task because it considers the theoretical speed-up of the PPU s, as well as the proportion of the execution time p that might be the subject of a speed-up.

$$S_{latency} = \frac{1}{(1-p) + \frac{p}{s}}, \tag{7.3}$$

where $S_{latency}$ is the theoretical speedup of the execution of the whole task.

The Equation 7.3 shows that the theoretical speed-up of the execution of the whole task is always limited by the part of the task that cannot be parallelised. Even if the theoretical speed-up of the PPU was approaching infinity $s \to \infty$, the resulting *speed-up* would have been bounded by:

$$S_{latency} \leq \frac{1}{(1-p)} \tag{7.4}$$

If, for example, 90% of the program can be parallelised, the speed-up will never exceed ten, as Figure 7.1 shows.

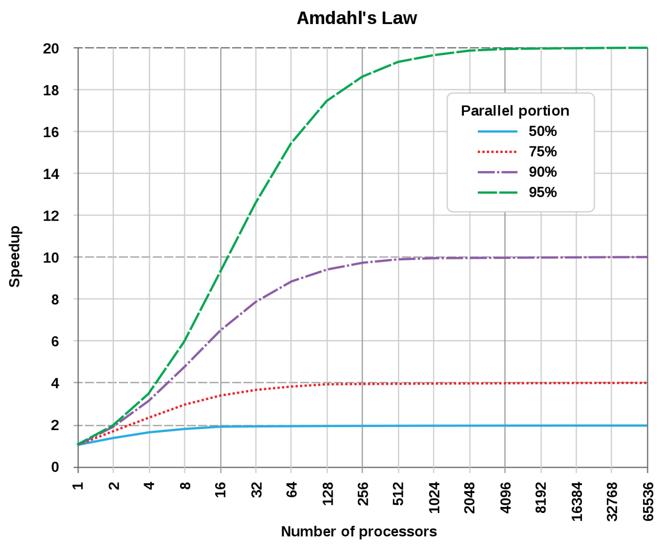


Figure 7.1: Visualization of Amdahl Law [4]

On this basis, it is possible to conclude that offloading tasks on SIMD architectures is not always advantageous. Some algorithms are serial by their nature and cannot be efficiently vectorised.

Common examples of such algorithms such as recursive or serial algorithms, stream parsing, control code or state machines [38] are likely to perform better in a scalar implementation.

On the other hand, algorithms for image processing, matrix linear algebra operations or Kalman filter generally have little or no conditional jumps, a high level of data parallelism, and no dependencies between loop iterations.

Moreover, if their dimensions are multiples of the SIMD width, the maximum speed-up could be reached using SIMD programming.

7.1 Application Size

As mentioned above, linear algebra matrix operations could generally be effectively vectorised using SIMD. But there is another important aspect, the size of the processed dataset.

PPU has 512 bit-wide vector registers, which are 16 single precision floating numbers. In the experiment performed, the matrix multiplication time of two square matrices on a scalar and a vector processor was done. The dimension of the matrices was gradually increased up to 99. Figure 8, comparing the execution time, shows that the vector processor significantly outperformed the scalar processor. In addition, there was a significant improvement for scalar multiples of 16 and 8 and 4 because the performance of vector processor was maximally used.

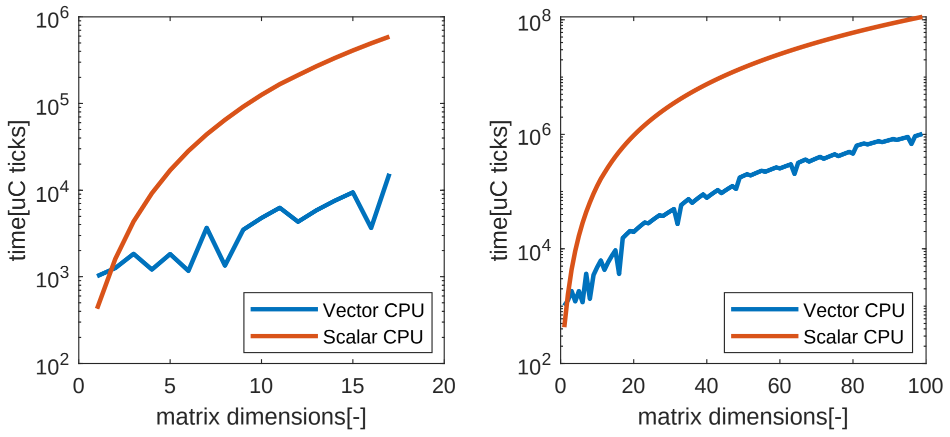


Figure 7.2: Computation time of matrix multiplication

7.2 Impact of Data Transfers

Offloading tasks to the vector processor brings overhead into the execution. Before submitting, the TriCore has to prepare all the parameters into a given data structure and then notify PPU that the data is ready. The Vector processor accepts the task, but it cannot process the data because SIMD instructions could be executed only with the data located in the Vector memory. To execute the kernel using vector instructions, it hast to perform these steps:

- Copy data into VCCM
- Process the data on the vector DSP
- Copy the result into the system memory

Finally, the host processor needs to handle the notification when the task completes.

In another experiment, the same matrix multiplication was performed with the difference that the calculation time was compared with the total execution time of the task. From Figure 7.3, it can be observed that the overhead associated with the data transfer significantly increases the total time.

The overhead could be further reduced by fully exploiting the functionality Streaming Transfer Unit, which enables the stream of the data in or out of VCCM simultaneously with processing. This functionality is implemented in PPU Dispatcher [47] which runs on top of Synopsys Processor Execution Environment for DSP [45]. Unfortunately, it is currently running only in a Virtual environment and was not used in our case.

From this, it is possible to conclude that if the matrix dimensions are significantly lower, they do not take full advantage of SIMD possibilities, and overhead with task offloading might cause the scalar code to be more effective.

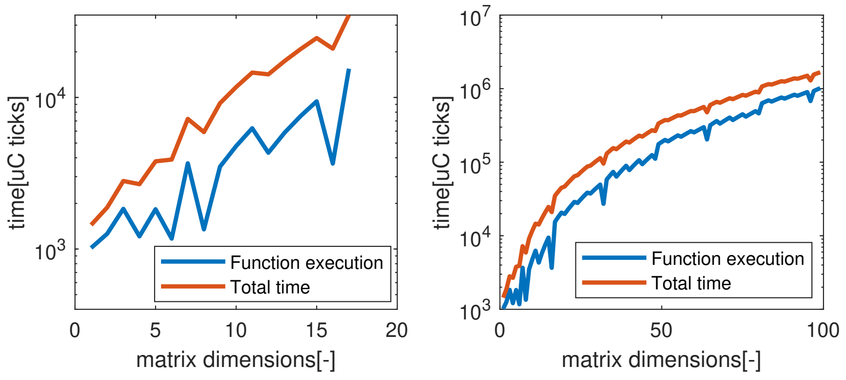


Figure 7.3: Effect of data transfer in task offloading

7.3 Effect of Multi-core Application

TC4 microcontroller contains up to 7 TriCores but only one Vector processor. When the virtualisation feature is turned on, up to 50 virtual machines might use the services of the PPU. Depending on the workload of the vector processor, the latency of task completion may be affected.

7.4 Summary

In this chapter, the process of task offloading on a vector processor was described. Based on the analysis and experiments carried out, it is possible to conclude that multiple things have to be taken into account when vectorising tasks:

The algorithm must be well parallelisable to take advantage of the SIMD functionality of the vector processor.

Within the TC4xx architecture, offloading of tasks introduces overhead. For the benefits of vectorising an application to outweigh the disadvantages associated with the necessary data offloading, the application must have sufficient complexity.

When the vector processor is heavily loaded, it may not be beneficial to offload more tasks, as the latency for getting responses can increase in such a situation.

Chapter 8

Model-based Design

In automotive, the demand on processing power is constantly increasing. MCUs must be able to run more complex algorithms to increase driver safety and developers have to adhere to more strict functional safety standards. These factors together with shorter development timelines cause that traditional methods for embedded software development are insufficient to address today's complex tasks while taking into account the safety and real-time requirements, CPU consumption and aggressive development timeline [20].

Model-based design performs verification and validation through testing in the simulation environment. It covers various disciplines, functional behavior, and cost/performance optimization to deploy a product from early concept of design a final validation and verification testing [20].

The centre of the design process is the model of the system in the Simulink environment, which is a graphical extension of the Matlab computing and development tool for modelling and simulating dynamic systems. The Simulink tool is based on an intuitive block diagram environment and enables simulation and analysis of a wide range of engineering systems and tasks.

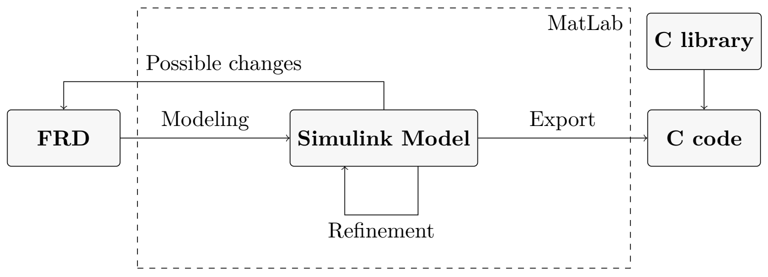


Figure 8.1: A model-based design process based on MatLab Simulink [53]

The whole process consists of four main parts as Figure 8.1 shows:

- At the beginning, the list of requirements for the model is defined.
- Then the modelling phase takes place, in the process of which, the requirements can still be modified.

- As soon as the model is ready, it is verified in the simulator and any shortcomings are refined.
- After the model is tested and verified, it is transferred into C code which is then executed on the ECU.

8.1 Code-replacement library

Embedded Code generation software normally produces ANSI/ISO C code. However, the Code Replacement Library can be used [41]. The Code Replacement Library is a table that describes mapping of Simulink blocks to Vector DSP [43] and Linear Algebra[44] libraries. The code replacement feature replaces function calls shown in Table 8.1 with calls to functions highly optimized for the PPU.

Entry Name	Replacement Function Name
Addition	vdsp_add_f32
Subtraction	vdsp_sub_f32
Element Multiplication	vdsp_mult_f32
Matrix Multiplication	vec_blas_sgemm
Offset	vdsp_offset_f32
Scaling	vdsp_scale_f32
Transposition	vec_aux_strnsps2m
Complex Conjugation	vdsp_cmpl_conj_f32
Absolut Value	vdsp_abs_f32
Cosine Function	vdsp_cos_f32
Sine Function	vdsp_sin_f32
Exponential Function	vdsp_exp_f32
Natural logarithm	vdsp_log_f32
Logarithm with 10 base	vdsp_log10_f32
Reciprocal of the Square Root	vdsp_rsqrt_f32
Square Root	vdsp_sqrt_f32
Tangent	vdsp_tan_f32
Arctangent	vdsp_atan_f32
Maximum	vdsp_max_inter_f32
Minimum	vdsp_min_inter_f32
Saturate	vdsp_sat_f32
Modulo operation	vdsp_mod_f32
Signum Function	vdsp_sign_f32

Table 8.1: Code-replacement functions [41]

Model-based design development method provides an executable model that can be used not only as a functional specification for implementation, but also as a new basis for testing. Thanks to a higher level representation of a model, the generated code can efficiently tailored to the vector processor, but also easily ported to another architecture.

Chapter 9

Neural Network SDK

Machine-learning algorithms for computer vision are evolving rapidly. They are also getting into embedded systems. Many vision ADAS applications are based on multiple-layer convolutional neural networks (CNNs) because they have become superior to traditional algorithms on a variety of image understanding tasks. In contrast to traditional algorithms, deep learning approaches are generalized learning algorithms and have better performance in real-world conditions. Many modern MCUs including TC4 have specialized computing units called deep learning accelerators, that can effectively execute NN graphs.

> The MetaWare Neural Network Software Development Kit (NN SDK) is designed to compile, validate and deploy, high performance machine learning applications on ARC processors. The NN SDK allows compiling and optimizing of Neural Network(NN) models targeted for ARC processors. The NN SDK provides integration capabilities with Machine Learning(ML) frameworks such as TensorFlow and Keras. [42]

the whole process from the creation of the model, through the optimizations to the generation of the code for the PPU is captured in the Figure 9.1

Neural Network

Neural network is a computing system consisting of interconnected nodes - artificial neurons. Each node has input, output and an associated weight and threshold. When it one or more inputs are received, they are scaled, sumed and passed through the activation function to the output of the neuron. Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer as Figure 9.2 shows. This structure reflects the behavior of the human brain, allowing computer programs to learn to recognize patterns and solve common problems in the fields of AI, machine learning, and deep learning. The model must be trained before it is used. Because there is a significant workload asymmetry between initial training and subsequent inference, the training is done on a host device like PC.

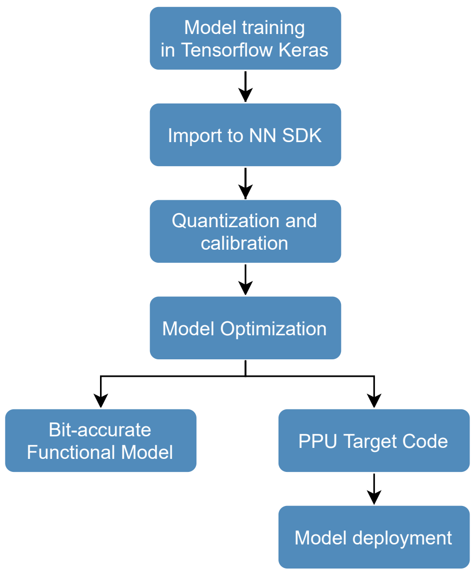

Model training in Tensorflow Keras
Import to NN SDK
Quantization and calibration
Model Optimization
Bit-accurate Functional Model
PPU Target Code
Model deployment

Figure 9.1: NNSDK Workflow

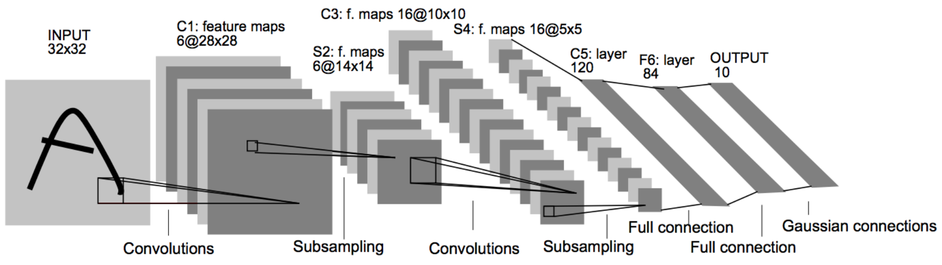


Figure 9.2: Convolutional neural network architecture [11]

Model Preparation

After the model is created, the weights of individual neurons usually have a random value. For accurate object recognition, all the parameters must be trained. This process requires a large dataset of input data (images) as well as the expected output data (object on the picture). The model is fed with the data and the output is compared with its expected value - ground truth. This error is then used to update the weights with the criterion of its minimization. This method is based on an automatic differentiation algorithm and is called back propagation. The process is repeated for many iterations until it converges to its minimum.

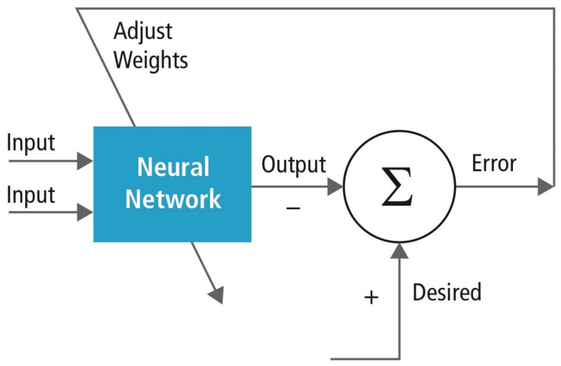


Figure 9.3: Neural network training process [14]

Model Import into NN SDK

In the first stage of, the trained NN model is parsed and translated it to an internal High Level Intermediate Representation (HLIR). However, several conditions must be met for the import to proceed correctly. The model is built and trained using TensorFlow or Keras compatible framework and must consist of supported layers shown in the Table 9.1.

Group	Layers/Operators
Activation	RELU Sigmoid Tanh Softmax
Convolution	1D 2D
Pooling	Max Avg
Fully connected	Inner Dot Product
Recurrent	LSTM Vanilla RNN

Table 9.1: Layers supported by NNSDK

Model Quantization

These resource-intensive requirements of neural networks are particularly challenging in embedded platforms [18].

We therefore must

While neural networks are often trained on computers using 32bit floating-point numbers, in embedded systems, integer operations are simpler than floating-point operations. This process is called quantization and it significantly simplifies computational complexity and increases computational speed. Furthermore, it is also possible to reduce the number of bits used to represent these numbers, which would reduce the memory requirements of the model. Approximation of single precision floating point value and backward transformation for $fx16$ quantization type are performed by the following formula:

$$scale = 2^n \tag{9.1}$$
$$x_{f_x} = Round(x_{f_{p32}} \cdot scale) \tag{9.2}$$
$$x_{f_{p32}} = \frac{x_{f_x}}{scale} \tag{9.3}$$

where n is the number of fractional bits, x_{f_x} is a fixed point value and $x_{f_{p32}}$ is a single precision floating point value.

The process of quantization, however, leads to quantization errors. LSB determines the precision of the binary fixed-point representation:

$$LSB = \frac{1}{2^n} = \frac{1}{scale} \tag{9.4}$$

Setting a correct scale for a given NN layer is a trade-off between precision and possible overflow. Lower value of scale as well as LSB means a finer granularity and thus decreasing of quantization error. On the other hand, when a number is too large to be stored, clipping occurs and the result could be highly skewed.

To correcly quantize each layer, the MetaWare NN quantizer needs to determine the range of values of the blobs. This process is referred to as calibration and is carried out by running the original graph on several input objects. To achieve the smallest quantization error, the data subset must sufficiently reflect the reality. Otherwise, there may be a systematic increase in error rate when using a small scale, or random fatal errors when the value is too high. The number n of files be used for calibration should be adapted to the size and type of the NN graph.

Model Optimization

After quantizing the NN model, multiple transformations are invoked to optimize it for efficient execution on resource-constrained devices like PPU. First of all, the components that are relevant to training but not to network feed-forwarding are removed One of them are dropout layers which suppress over-fitting by randomly dropping out nodes during training but in the inference, they have no practical utilization.

In the learning stage, images from dataset are processed in sub-sets called mini-batches. This approach often reduces the risk of getting stuck at a local minimum but it creates another dimension which increases memory requirements. Contrary to that, in the inference phase, the images are processed individually so the extra dimension could be removed.

Reducing the size of trained model is another option for optimization. Figure 9.4 of the trained CNN model from Section 10.2 shows how the coefficients of a particular layer are activated during the inference of an image. We may observe, that several filters have zero values and do not contribute to the functionality. These filters as well as the scaling layers with scaling factor of 1 could be pruned, which reduces the number of coefficients stored in memory and therefore cutting down bandwidth. The can boost the occurrence of such sparse matrices when we bias the training to maximize sparseness as [55] suggests.

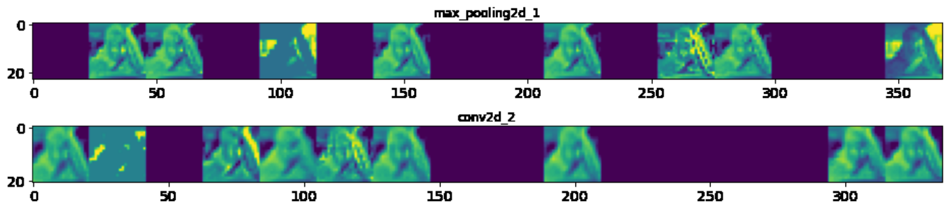


Figure 9.4: Activation of CNN layer

Code Generation and Model Deployment

In the final step, the source code of the NN graph is generated directly for target hardware or as a Bit-accurate Functional Model. BFM is a bit-true representation of the NN model, which means that it produces bit-accurate outputs for the same input as the model running on the PPU. It could be executed on the host machine(PC) to find out, how the accuracy changed after model quantization. If the result is insufficient, the quantization has to be re-executed and provided by a more versatile sub-set of data for calibration. When using the device target, the graph is optimized for the PPU device and runs within a simulation. In this stage, NN model might exceed hardware capabilities of PPU. In such case, the whole architecture has to be simplified and the code generation process is repeated.

9.0.1 Summary

The popularity and feasibility of real-world neural network deployments are growing rapidly. The CNN mapping tool is a necessary tool to enable the neural networks to run on the PPU. It uses different optimizing methods to save the memory and processing resources. Another possibility to optimize the size of the model is to maximize the sparsity of the individual layers as [55] mentions. Nevertheless, a deep understanding of the target system is required to fully exploit its capabilities. Because deep learning algorithms are rapidly evolving, the limited range of possible layers, activation functions and supported frameworks can be limiting.

Chapter 10

ADAS Applications

According to the August 2016 Traffic Safety Facts [50], “The Nation lost 35,092 people in crashes on U.S. roadways during 2015.” An analysis revealed that about 94% of those accidents were caused by human error, and the rest by the environment and mechanical failures [49].

To mitigate the impact of human error during driving, the vehicles use Advanced Driver Assistance Systems that actively or passively improve the safety of passengers. Vehicles equipped with ADAS systems have a variety of sensors (Figure 10.1) that sense the surrounding environment and process the information from them in real-time. In case of danger, the system is able to alert the driver and even take control to avoid an accident.

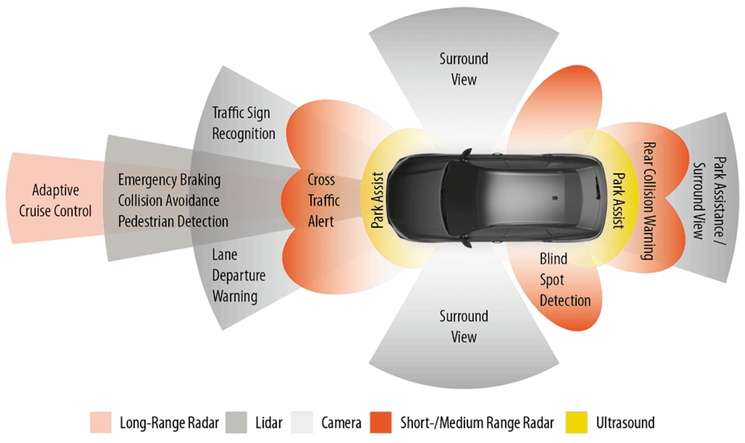


Figure 10.1: Different types of ADAS systems [35]

An ADAS system consists of three layers:

- Perception layer - A suite of sensors (Figure 10.1) accompanied by a fusion unit combining the measured data to accurately estimate the state of a vehicle within the environment.

- Decision layer - A powerful computing unit capable of processing sensory data from the Perception layer and choosing the appropriate actions in real-time, which are then passed to actuators of an Action layer.
- Action layer - A series of actuators and devices that can warn the driver or automatically take actions to resolve risks of an imminent accident.

The rest of this chapter aims at the decision layer of ADAS, which the TC4 is part of. It further compares different hardware architectures and describes the development of two applications using the previously described model-based approach.

10.1 Architectures for ADAS

As previously mentioned, the decision layer of ADAS is represented by the computing units that process the data from sensors and create outputs to assist the driver or take direct action to avoid a possible accident. In order to handle increasingly more complex tasks, like object recognition from high-resolution images, the computing power and speed of ADAS platforms steadily increase. However, there are still challenges in the development and operation of ADAS.

Real-time Capabilities

According to [19], the average reaction time of a human driver alert is 700 ms. Considering that the ADAS platforms are hard real-time systems, they could be beneficial only if they are multiple times faster than the driver. Their reaction time is determined by frame rate and processing latency. The frame rate is the frequency with which the processor gathers sensory data. The processing latency is the execution time of the prediction algorithm. Even complex camera-based applications require images to be gathered and processed within this time bound, so that the car can react quickly to changes.

Safety

Platforms for ADAS computing are safety-critical systems. They must comply with a certain level of ASIL, a risk classification system defined by the ISO 26262 standard [28]. For example, the steering or engine management systems require the highest ASIL D, whereas the vision ADAS require ASIL B certification. The safety restrictions also apply to the software layer, specifically virtual sensors. Virtual sensors provide indirect measurements based on sensory data. Many of them are based on neural networks, which inherently suffer from errors listed in [54]. This is considered a malfunction similar to physical sensors. Therefore redundant support mechanisms must be utilized to mitigate these effects. In addition, automotive standard ISO-26262 [28] defines a long list of costly and time-consuming requirements that have to be fulfilled for the certification process.

Cost-effectiveness

Another important prerequisite for ADAS platforms is cost-effectiveness. Because the highest performing devices are often too expensive, different types of parallel computers with better performance/cost ratios have recently been used. According to [16], these platforms must fully exploit the capacity of the instruction sets, caches, and parallelism to achieve the required performance.

10.1.1 Multi-core Design

Problems with obtaining an extra performance in single-core processors and disproportionate expenses for the increasing raw clock rate have led to the development of multi-cores in a single physical package.

In theory, a homogeneous multi-core processor consisting of n similar processors should speed up the execution n-times. However, the limited communication bandwidth causes a performance bottleneck called the *memory wall*. Its practical consequence is that adding additional cores does not linearly increase the performance. Due to this reason, almost all platforms for ADAS have adopted a heterogeneous architecture.
Another common sign is the presence of a shared memory, which significantly decreases the cost of inter-core data transfers but requires coherency protocols for data consistency checks. As [31] states, a heterogeneous multi-core processor usually comprises general-purpose core(s) and one or more of the following specialized units:

- Graphical Processing Units
- Field Programmable Gate Arrays
- Digital Signal Processors with neural network accelerators

10.1.2 GPU

Graphic processing units are highly parallel instruction-based platforms initially developed to perform graphical calculations. They provide high-performance computational power and are suitable for a wide range of algorithms. One of the disadvantages of GPUs is that the software running on top of these platforms is difficult to analyze, which might be problematic in safety-critical applications. When we compare it with previously mentioned platforms, they are generally more expensive and power-consuming.

10.1.3 FPGA

A Field Programmable Gate Array is an integrated circuit that can change its hardware layout according to program requirements at run-time. This is a completely different approach to CPUs or GPUs, where the programs are compiled to fit the hardware layout. On the one hand, FPGAs provide massive throughput due to their intrinsic parallelism and are energy efficient.

On the other hand, they are not suitable for the serial processing and require more time and knowledge to design and program.

10.1.4 Vector DSP with NN Engine

This setup is also used by the TC4 microcontroller. If the modules are efficiently interconnected, the programming tasks can be assigned to processing units according to their use-case:

- Most of the scalar code, including the data collection and safety-critical code, is done by TriCores.
- The vector DSP unit computes algorithms with a high level of parallelism, such as matrix operation or Kalman filtering. Some DSPs (including PPU) offer programmable safety features, making them suitable even for applications with higher safety requirements.
- Neural-network accelerators can efficiently and frequently compute multiply–accumulated operations, which is a baseline for the inference of CNNs. This architecture has safety features and sufficiently high performance for many driver assistance applications while having lower price and power consumption than GPUs.

There are also some issues when distinct cores have different architectures. During the development, these appeared:

- Several different development tools are required for development.
- Differences in EABIs might cause mistakes in data interpretation in inter-processor communication leading to nondeterministic bugs that are difficult to find.
- Incompatibility between toolchains causes that projects could not be linked together, and subsequently, any change in the project would have to be applied to both platforms.
- When multiple cores use the vector processor, the task completes, on average, in a longer time and with a more significant dispersion of the task's execution time.

Considering all the positive and negative aspects of heterogeneous architectures, the best lower-cost energy-efficient setup is the combination of multi-purpose core(s), DSP(s) and NN accelerator(s). On the other hand, each platform provides a different power/performance tradeoff and hence, the best platform may change across different applications.

10.2 Driver Monitoring

Recently, driver monitoring has become crucial since it was shown that automation aids could lead to driver distraction and thus high reaction times in case of emergency. Previous research on autonomous driving showed that the driver's transition to dependence on automation comes after approximately 15 minutes.

The task is to distinguish whether the driver is paying full attention to the driving. Several approaches were implemented to detect the symptoms of driver drowsiness.

According to [5], one of the most reliable ways of estimating fatigue is by using electroencephalograms. Unfortunately, the majority of drivers reject it because it requires using attached electrodes on their heads.

The most currently used fatigue detection systems are based on the processing of driver images by a camera placed in the car.

There are algorithms like **Percentage of Eye Closure (PERCLOS)** or **Histogram of Oriented Gradients (HOG)** [32]. Still, recently, CNN-based solutions achieved better accuracy according to [29]. Because of this reason and the presence of a CNN engine on PPU, a convolutional neural network was used to detect driver yawning.

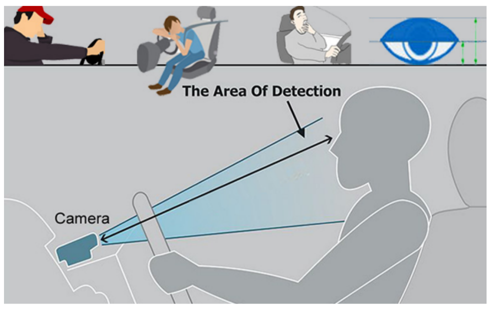


Figure 10.2: Camera-based driver monitoring [17]

This example doesn't aim at reaching the highest possible accuracy in drowsiness detection but rather testing the development process from CNN model learning to generating the source code [42] and deploying it on the PPU.

10.2.1 Neural Network Architecture

A convolutional neural network (CNN) is a neural network architecture designed to find patterns in spatial structures. CNNs are used particularly for image classification and object recognition tasks, which is also this use case.

Considering the supported frameworks listed in [42], eventually, the deep neural network was implemented in Python 3 with the help of Keras, an interface for the TensorFlow library.

The model structure was inspired by the approach in [1]. The input of the neural network is a grey-scale image with 100x100 resolution and values in the range $(0, 1)$. The final output has the same length as the number of detected classes (yawn/no yawn). Because the network structure was too complex for an embedded device, it was gradually simplified until the number of trainable parameters decreased below 10000.

The final structure (Appendix A) consists of several convolutional layers followed by max pooling layers. Subsequently, the output is flattened, and two dense layers are applied. Additionally, the dropout layers were exploited to reduce overfitting and improve generalization by randomly inhibiting nodes during training. Finally, a Softmax activation layer outputs values representing the probability that the detected object belongs to the corresponding class.

10.2.2 Dataset

At first, the Driver drowsiness dataset from [30] was used, which contains two categories: *yawn* and *no yawn*. Figure 10.3 displays a few samples of each category.

Because the dataset contains only 1235 train images and 215 test images, it was necessary to enlarge it manually. The same setup as in [10] was used, which enables creating a dataset using a web camera. The user configures how many images of a specific category should be gathered. Subsequently, the camera window opens, and data gathering begins. Throughout the process, the script provides information on how many remaining images will be collected.

Data augmentation is an important technique to reduce the overfitting on small datasets. Keras function *ImageDataGenerator* allows rotating, shifting, flipping or zooming the existing data samples. In this particular application, applying image distortion does not change its class, contrary to e.g. gesture detection, where the flip of a *left* gesture would result in a *right* gesture.

Eventually, rotation of images in a range of ± 12 degrees, width and height shift by 20% of the corresponding dimension, zoom in the range of 1 ± 0.15 and horizontal flip were utilized. Using these techniques, the final size of the dataset was approximately doubled.

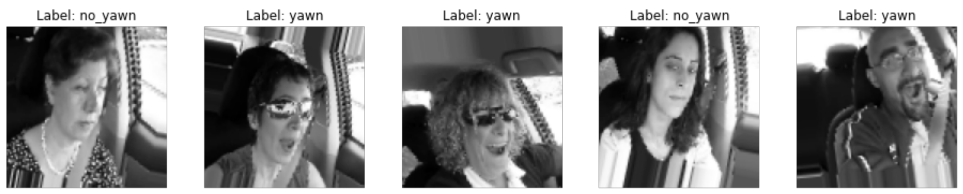


Figure 10.3: Samples from driver drowsiness dataset

10.2.3 Training and Testing

Our proposed neural network was trained using Adam Optimizer with default parameters for 20 epochs with a batch size of 128.

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. The loss function is checked at the end of every epoch, and if it didn't improve in the last X number of epochs, the learning rate is decreased.

Early Stopping monitors the model's performance at the end of every epoch during training on a held-out validation set and terminates the training according to the validation performance.

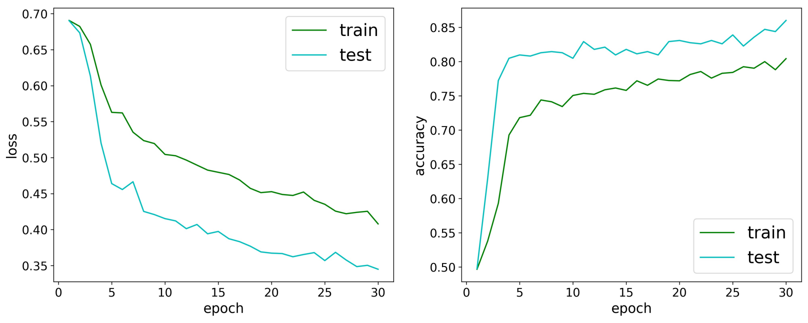


Figure 10.4: NN training process

After several iterations, the test accuracy of 86% was obtained (Figure 10.4)

The confusion matrix in Figure 10.5 gives us detailed information about the classifier's performance. The worst-case scenario, where the image *yawn* was mis-classified as *no-yawn*, occurred only in 5% of cases.

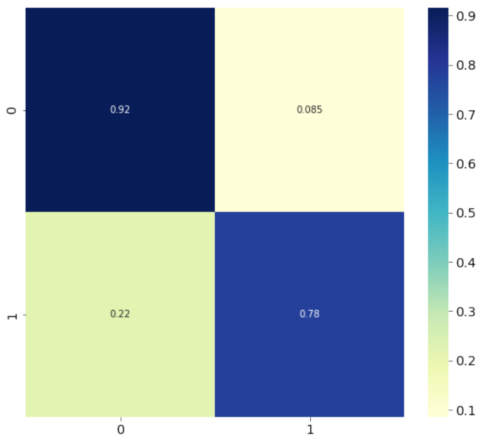


Figure 10.5: Confusion matrix

10.2.4 Running on TC4xx

The source code was created according to 9.1.

At first, the model was checked to verify that it contains only the supported layers shown in Figure 9.1.

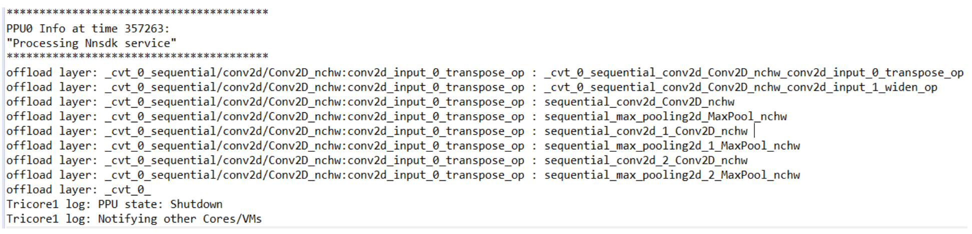
Followingly, the floating-point model was converted into the fixed-point model. This stage was very sensitive in the data selection. The image samples must closely correspond to the images used in reality. Otherwise, the layers could have low granularity, and low resolution or overflow might occur.

In the last step, the model was optimized for the PPU, dropout layers (used only in training) were removed, and the source code was generated.

During these steps, the neural network model was adjusted several times until it met all the NN SDK tool requirements and resource constraints of PPU.

To test the performance of the CNN accelerator, a setup was prepared [10] where the web camera regularly takes an image of the person, processes it into a grayscale image with a resolution of 100x100 pixels and sends it to the microcontroller via ethernet.

Unfortunately, the current version of PPU tools enables running the model only within the simulation environment. Furthermore, it wasn't possible to measure the classification time because the simulation environment is not cycle-accurate.



```
****************************************
PPU0 Info at time 357263:
"Processing Nnsdk service"
****************************************
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : _cvt_0_sequential_conv2d_Conv2D_nchw_conv2d_input_0_transpose_op
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : _cvt_0_sequential_conv2d_Conv2D_nchw_conv2d_input_1_widen_op
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_conv2d_Conv2D_nchw
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_max_pooling2d_MaxPool_nchw
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_conv2d_1_Conv2D_nchw
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_max_pooling2d_1_MaxPool_nchw
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_conv2d_2_Conv2D_nchw
offload layer: _cvt_0_sequential/conv2d/Conv2D_nchw:conv2d_input_0_transpose_op : sequential_max_pooling2d_2_MaxPool_nchw
offload layer: _cvt_0_
Tricore1 log: PPU state: Shutdown
Tricore1 log: Notifying other Cores/VMs
```

Figure 10.6: NN simulation output

10.3 Adaptive Cruise Control

Adaptive cruise control (ACC) is an advanced driver-assistance system for road vehicles that automatically adjusts the vehicle's speed to maintain a safe distance from other vehicles ahead (Figure 10.7). It uses data from sensors like radar or laser sensors, or a camera to predict the actions of other objects in the immediate area.

The aim is to implement the velocity estimation of the lead vehicle based on the Kalman filter algorithm. The sensory data will be artificially generated by Matlab simulation. The main objective is to use and evaluate a model-based approach in Matlab Simulink described in Section 8.

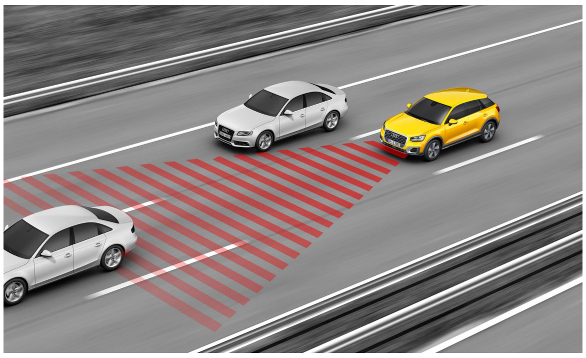


Figure 10.7: A car with active ACC [2]

10.3.1 Kalman Filter

The Kalman filter is one of the most common and significant estimation algorithms. It produces estimates of hidden variables based on inaccurate and uncertain measurements and provides a prediction of the future system state based on past estimations [6]. The inaccuracies are caused by stochastic processes called process and observation noise. When these stochastic processes meet certain requirements mentioned in the following sections, the Kalman filter is the optimal linear filter.

Our derivation is based on [34] and uses the same notation.

A linear discrete-time system can be described as follows:

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{10.1}$$

$$y_k = H_k x_k + v_k \tag{10.2}$$

where F, G and H are matrices describing its dynamics. The system and measurements are affected by a process and observation noise as shown in the Figure 10.8. Assuming that w_k and v_k are uncorrelated and have a normal distribution with zero mean and covariance matrices Q_k and R_k:

$$w_k \sim \mathcal{N}(0, Q_k) \tag{10.3}$$

$$v_k \sim \mathcal{N}(0, R_k) \tag{10.4}$$

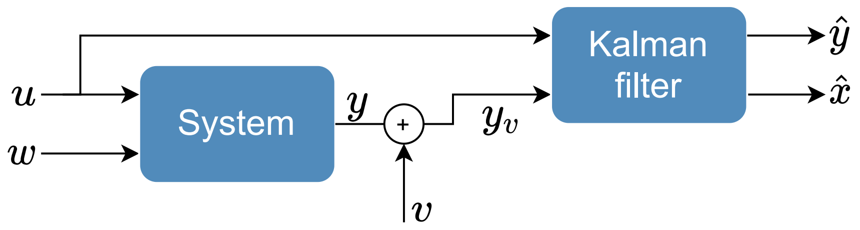


Figure 10.8: Dynamic system with noise

The task is to estimate the hidden system states x_k based on the knowledge of its dynamics and noisy measurements y_k.

From the Equation 10.1, it is evident that x_k is a linear combination of x_0, $\{w_i\}$, and $\{u_i\}$, which are Gaussian random variables. Therefore, x_k itself is a Gaussian random variable and is completely characterized by its mean (expected value) denoted by a hat operator and covariance expressed by matrix P:

$$x_k \sim \mathcal{N}(\hat{x}_k, P_k) \tag{10.5}$$
$$\tag{10.6}$$

Based on the number of available measurements, it can be distinguished:

- A *priori* estimate/covariance matrix with $-$ superscript, which is formed from the measurements before the measurement at time k:

$$\hat{x}_k^- = E[x_k | y_1, \ldots, y_{k-1}] \tag{10.7}$$
$$P_k^- = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \tag{10.8}$$

- A *posteriori* estimate/covariance matrix with $+$ superscript that could be computed after the measurement at time k is available:

$$\hat{x}_k^+ = E[x_k | y_1, \ldots, y_k] \tag{10.9}$$
$$P_k^+ = E[(x_k - \hat{x}_k^+)(x_k - \hat{x}_k^+)^T] \tag{10.10}$$

The relation between priori and posteriori estimates and covariances is shown in the Figure 10.9.

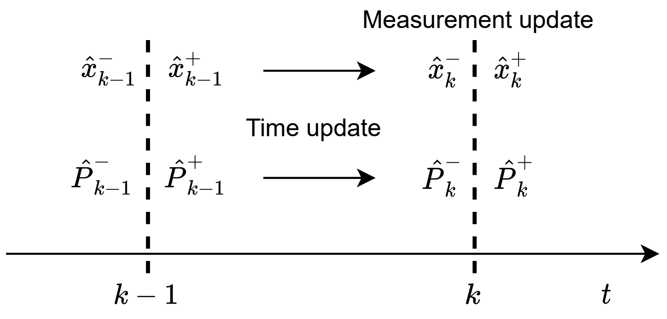


Figure 10.9: Timeline showing state estimates

Kalman filter is based on two steps:

- Time update - computes priori estimate of state x_k
- Data update - improves priori estimate of x_k after the measurement y_k is acquired

The time update step is done every sampling period, whilst the data update is realized only when the measurement data is available.

These steps are used according to the application requirements. For example, if the sampling time is 3-times faster than measurements, 3 time updates will be done before one data update step.

Prior to any measurements, it is necessary to form an initial estimate x_0^+ and its uncertainty P_0^+. In the extreme case, if no information about x_0 is available, then the initial covariance is $P_0^+ = \infty$. On the other hand, if the initial state is known perfectly, then the initial covariance is $P_0^+ = 0$.

10.3.2 Time Update

The time update step is based on the propagation of the state estimate x with time:

$$\hat{x}_k^- = E[x_k] = F_{k-1}\hat{x}_{k-1}^+ + G_{k-1}u_{k-1} \tag{10.11}$$
$$P_k^- = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \tag{10.12}$$

Substituting 10.1 and 10.11 into 10.12, the time-update equation for P can be obtained:

$$\begin{aligned} P_k^- &= E[(F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + w_{k-1})(F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + w_{k-1})^T] \\ &= F_{k-1}P_{k-1}^+F_{k-1}^T + Q_{k-1} \end{aligned} \tag{10.13}$$

10.3.3 Measurement Update

The derivation of measurement update of x and P is based on the solution of the recursive least squares estimation from the Section 3 of [34], where the new estimate/covariance $\hat{x}_k/P_k$ is computed on the basis of the previous estimate/covariance $\hat{x}_{k-1}/P_{k-1}$ and a new noisy measurement y_k:

$$y_k = H_k x + v_k \tag{10.14}$$
$$\hat{x}_k = \hat{x}_{k-1} + K_k(y_k - H_k\hat{x}_{k-1}) \tag{10.15}$$
$$P_k = (I - K_k H_k)P_{k-1}, \tag{10.16}$$

where $\hat{x}_{k-1}$ and P_{k-1} are previous estimates and covariance, and K_k is a matrix called Kalman filter gain.

Because the measurement noise has zero mean and the initial estimate was set to its expected value, the estimator is unbiased for any value of K_k. In order to compute the optimal value of K_k, some other criterion must be taken into account. In this case, the selected optimality criterion is to minimize the sum of the variances of estimation errors

$$J_k = E[(x_1 - \hat{x}_1)^2)] + \cdots + (x_n - \hat{x}_n)^2] \tag{10.17}$$

by setting its derivative with respect to K to zero

$$\frac{\partial J}{\partial K} = 0 \tag{10.18}$$

Substituting 10.15 and 10.16 into 10.17 yields:

$$\frac{\partial J_k}{\partial K_k} = 2(I - K_k H_k)P_{k-1}(-H_k^T) + 2K_k R_k = 0 \tag{10.19}$$

$$K_k = P_{k-1}H_k^T(H_k P_{k-1} H_k^T + R_k + R_k)^{-1} \tag{10.20}$$

The subscript $k-1$ in the least square estimation is equivalent to the *priori* estimate and k corresponds to the *posteriori* estimate.

After applying these changes to the previous equations, the measurement update for $\hat{x}_k$ and P_k can be expressed.

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} = P_k^+ H_k^T R_k^{-1} \tag{10.21}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k \hat{x}_k^-) \tag{10.22}$$

$$P_k^+ = (I - K_k H_k)P_k^- \tag{10.23}$$

The whole algorithm is visualised in Figure 10.10.

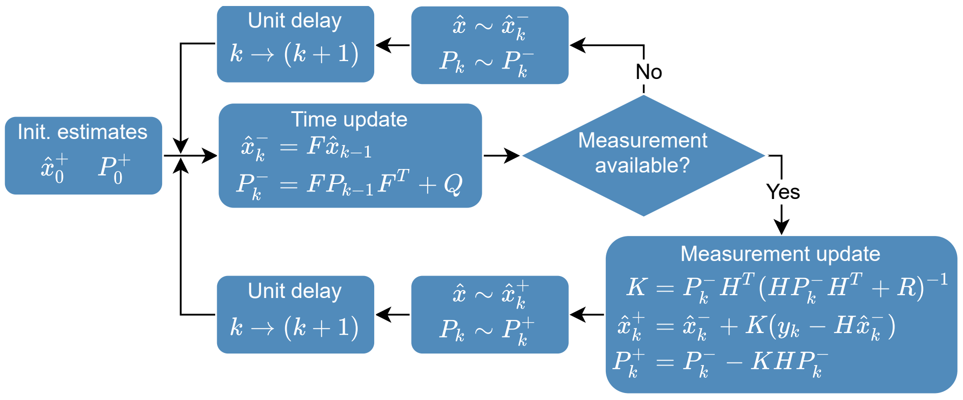

Unit delay
$k \rightarrow (k+1)$
$\hat{x} \sim \hat{x}_k^-$
$P_k \sim P_k^-$
No
Init. estimates
$\hat{x}_0^+$ P_0^+
Time update
$\hat{x}_k^- = F\hat{x}_{k-1}$
$P_k^- = FP_{k-1}F^T + Q$
Measurement available?
Yes
Measurement update
$K = P_k^- H^T (HP_k^- H^T + R)^{-1}$
$\hat{x}_k^+ = \hat{x}_k^- + K(y_k - H\hat{x}_k^-)$
$P_k^+ = P_k^- - KHP_k^-$
Unit delay
$k \rightarrow (k+1)$
$\hat{x} \sim \hat{x}_k^+$
$P_k \sim P_k^+$

Figure 10.10: Estimation algorithm

10.3.4 Application

A going vehicle has information only about the position of the ahead of it going lead vehicle. The adaptive cruise control needs LV's velocity for proper functionality. The model of LV si based on [52] and assumes that the unknown LV velocity is constant.

The LV model can be described as follows:

$$x_{k+1} = F \cdot x_k + w_k \tag{10.24}$$

$$y_k = H \cdot x_k + v_k \tag{10.25}$$

where $x = \begin{bmatrix} v \\ s \end{bmatrix}$, $v\left[\frac{m}{s}\right]$ is lead vehicle's velocity, $s[m]$ is its distance, $T_s[s]$ is the sampling time, v_k is the measurement noise with covariance R and v_k is the process white noise with covariance Q. The matrices are:

$$F = \begin{bmatrix} 1 & 0 \\ T_s & 1 \end{bmatrix} \quad H = \begin{bmatrix} 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 \cdot 10^{-3} & 0 \\ 0 & 1 \end{bmatrix} \quad R = 4$$

$$P_0^+ = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix} \quad x_0^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad T_s = 0.1s \quad T_{meas} = 5 \cdot T_s$$

In the first stage, the update function was modelled in Matlab Simulink according to the Figure 10.10. Similarly to [13], the basic blocks included within the Code replacement library were utilized (see Table 8.1), so that the generated code could be optimized for the PPU.

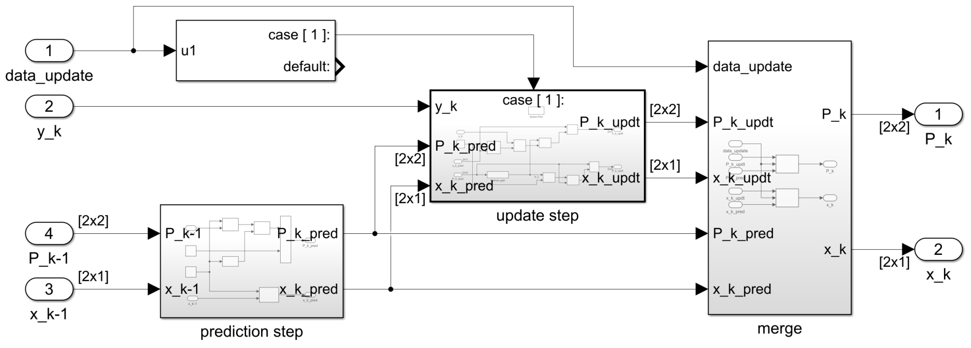


Figure 10.11: Simulink model of Kalman filter

Later, the functionality was tested in the simulation with a setup shown in Figure 10.12.

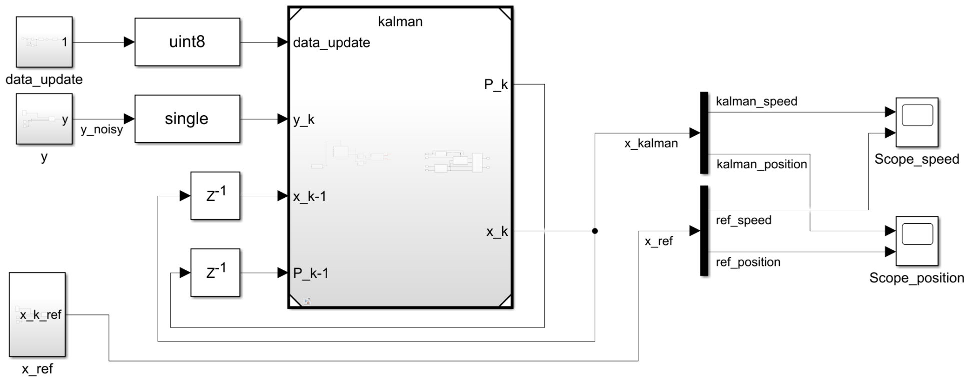


Figure 10.12: Model verification

The last part of the model-based design is source code generation optimized for a particular uC architecture. To achieve this, the Embedded coder toolbox accompanied by Metaware Toolbox [41] was exploited. The example of a generated function is shown in Appendix.

Subsequently, the function was integrated into the example described in Section 6.2 as an independent kernel with two expected input parameters: the measurement availability and, eventually, the measurement. The output of the function is an estimate of the velocity. The function ran 1200 times, and the measurement was provided with a frequency of 5 ticks. The following Figure shows the estimated velocity together with the Matlab model's simulation.

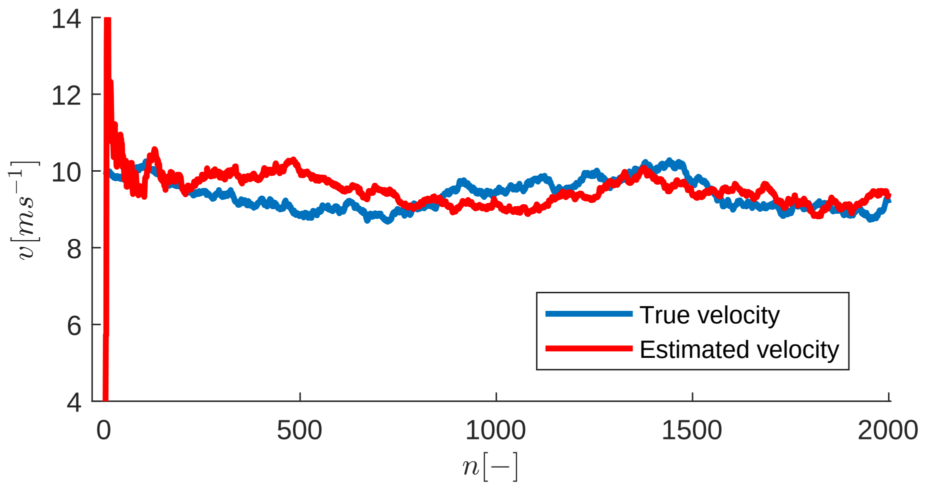


Figure 10.13: Kalman filter response

10.3.5 Results

The Matlab Simulink was used to implement an estimation part of the Adaptive cruise control application. Thanks to the model-based approach, it was possible to test the functionality before the source code generation and run it on the real hardware to correct possible bugs. The block implementation of the Kalman filter was verified experimentally on the PPU. Due to the high-level representation, porting the application to a scalar uC was possible too.

The speed-up compared to a scalar execution was negligible because of a very simple model with only two internal states. Matrix operations with such small dimensions bring additional overhead, as was shown in Section 7. On the other hand, the use of a model-based approach enables to quickly increase the complexity of the model according to real-world requirements, where the difference in run-time would be more significant.

Chapter 11

Results

11.1 Basic Software Support for TriCores

In the first part of the thesis, the requirements for basic software support (BSP) for TriCore processors were defined. Then a set of functions was created, logically grouped into these sub-modules:

- uC (Appendix C.1)
- board (Appendix C.3)
- isr (Appendix C.2)

Furthermore, a linker script was created. It correctly assigns code and symbols to the uC memory, defines stack size and creates the final executable binary. Using the functions and linker defined above, a multi-core application was created where each core flashes its corresponding LED at a different frequency based on the interrupts generated by a timer.

To extend the functionality of BSP, multi-core printing to UART using the following modules was developed:

- mutex - for handling exclusive access to the shared resources (Appendix C.4).
- asclin - for transmission and reception of data via the UART interface (Appendix C.6).
- printf - an implementation of C's formatted printing family of functions (Appendix C.5).

11.2 IPC and PPU Integration

As a follow-up to the previous project, support for booting PPU and generating inter-core interrupts has been added (Appendix C.7). In the course of implementation, the incompatibility of compilers for different platforms was discovered. Consequently, it was impossible to merge TriCore and PPU into one shared project, so the changes had to be applied to both projects.

To enable data exchange and potential task offloading on the PPU, inter-processor communication between computing units based on shared memory and software mailboxes has been defined and implemented (Appendix C.8). At this stage, differences in EABI leading to a different interpretation of certain data types were found. This caused the communication to be restricted only to specific data types. Subsequently, experiments on task offloading in different scenarios were performed with the following conclusion: It is advantageous to offload only highly parallel algorithms that are large enough to benefit from the PPU vector architecture. For better determinism and faster response, PPU services using less TriCores should be used. Otherwise, more sophisticated Iner-processor communication using task prioritization could be utilized [47].

11.3 Model-based Approach

The model-based approach was used to implement velocity estimation based on the Kalman filter in Matlab Simulink.

The *data-update* and *time-update* functions were modelled and tested within the simulation. Afterwards, the code was generated by the Embedded Coder and Synopsys toolbox

Code-replacement library from Synopsys toolbox allowed efficiently deploying the model on the Parallel unit by exploiting its enhanced matrix calculation capabilities. Thanks to the high-level approach, deploying a scalar version of the model was also possible. The use of parallel processing, in this case, did not significantly speed up the execution of the program. Because the application was too small, the overhead of moving data suppressed the benefits of parallelizing matrix operations.

11.4 Neural Network

A neural network detecting driver drowsiness was modelled and trained using the Keras library. Its design was adapted to the limitations of embedded devices, and the test accuracy reached 86%. Further, a CNN mapper called NN SDK was used. After applying quantization and other optimization methods, the code was generated, and the model was deployed in a simulation environment on the top of Synopsys runtime environment [45]. However, the computation time could not be determined because the simulator is not cycle-accurate.

Based on the preceding, it is possible to conclude that using this tool requires a deep understanding of the used hardware to fully exploit its capabilities in order to deploy the model with satisfactory accuracy and sufficient speed. Another possibility to optimize the size of the model and thus its performance is to use a specific training approach [55] maximizing the sparseness of layers.

Chapter 12

Conclusion

This thesis aimed to contribute to the early development phase of the AURIX TC4xx heterogeneous architecture and to gradually build support for the execution of parallel applications on a computational unit with vector calculation capabilities.

After analyzing the TC4xx architecture, a set of functions and a linker script were created that allow the initialization and basic functionality of the TriCore units. On their basis, a multi-core application was created, which verified the correctness of the implementation. The software support was extended to include printing into UART with multi-core synchronization to share the cores' communication channels.

Further, the functionality of the parallel unit was investigated, and functions to boot it and generate inter-core interrupts were created. Consequently, the inter-processor communication for data exchange between TriCore processors and PPU necessary for offloading tasks to the vector processor was defined and implemented.

To demonstrate the advantages and disadvantages of task offloading from scalar to the vector processor on the heterogeneous TC4xx architecture, a set of experiments was performed. As a result, due to the shared communication channel and overhead, while preparing data for parallel processing, it is suitable only for more extensive well parallelizable algorithms.

Last but not least, driver assistance systems were addressed. Requirements necessary for their operation were specified, and commonly used architectures were compared.

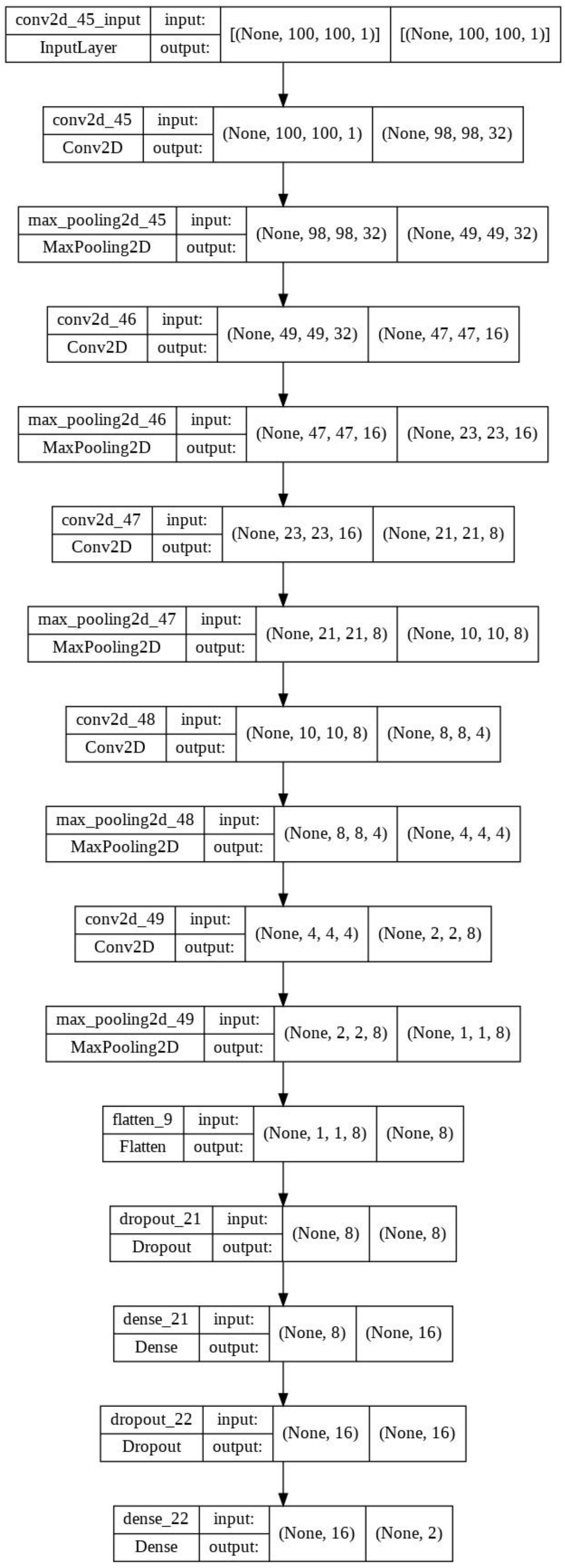
The possibilities of accelerating the development of TC4xx was explored. In particular, high-level tools, such as Matlab and CNN mapper, enable the generation of highly optimized code and replace the hand-written code.

Using a model-based approach, a Kalman filter for lead-vehicle estimation was created in Matlab Simulink. Its functionality was tested in simulation, and then, Embedded Coder, in conjunction with the Synopsys toolbox, was exploited for generating PPU-optimized code. Due to the higher level of representation, the model can be ported to different architectures. Using specialized code-replacement tools, the algorithm can be tailored to a heterogeneous HW architecture.

Finally, a convolutional neural network was implemented to detect the drowsy driver. The model was built and trained using the Keras library. The neural network SDK was then used to optimize it for resource-constrained devices. Generated code was partly running in a simulation environment. The parallel processing unit seems to be usable for advanced driving algorithms with a certain level of complexity. However, it was difficult to evaluate because it was impossible to deploy the created NN to the evaluation board due to the problems with HW.

Appendix A

Keras Model of Driver Drowsiness Classification


conv2d_45_input | InputLayer | input: [(None, 100, 100, 1)] | output: [(None, 100, 100, 1)]
conv2d_45 | Conv2D | input: (None, 100, 100, 1) | output: (None, 98, 98, 32)
max_pooling2d_45 | MaxPooling2D | input: (None, 98, 98, 32) | output: (None, 49, 49, 32)
conv2d_46 | Conv2D | input: (None, 49, 49, 32) | output: (None, 47, 47, 16)
max_pooling2d_46 | MaxPooling2D | input: (None, 47, 47, 16) | output: (None, 23, 23, 16)
conv2d_47 | Conv2D | input: (None, 23, 23, 16) | output: (None, 21, 21, 8)
max_pooling2d_47 | MaxPooling2D | input: (None, 21, 21, 8) | output: (None, 10, 10, 8)
conv2d_48 | Conv2D | input: (None, 10, 10, 8) | output: (None, 8, 8, 4)
max_pooling2d_48 | MaxPooling2D | input: (None, 8, 8, 4) | output: (None, 4, 4, 4)
conv2d_49 | Conv2D | input: (None, 4, 4, 4) | output: (None, 2, 2, 8)
max_pooling2d_49 | MaxPooling2D | input: (None, 2, 2, 8) | output: (None, 1, 1, 8)
flatten_9 | Flatten | input: (None, 1, 1, 8) | output: (None, 8)
dropout_21 | Dropout | input: (None, 8) | output: (None, 8)
dense_21 | Dense | input: (None, 8) | output: (None, 16)
dropout_22 | Dropout | input: (None, 16) | output: (None, 16)
dense_22 | Dense | input: (None, 16) | output: (None, 2)

Appendix B

Matlab-generated Kalman Filter

```
/*
 * Academic License - for use in teaching, academic research,
 ↪  and meeting
 * course requirements at degree granting institutions only.
 ↪  Not for
 * government, commercial, or other organizational use.
 *
 * File: kalman.c
 *
 * Code generated for Simulink model 'kalman'.
 *
 * Model version                  : 4.26
 * Simulink Coder version         : 9.7 (R2022a) 13-Nov-2021
 * C/C++ source code generated on : Wed Apr 27 11:04:31 2022
 *
 * Target selection: snps.tlc
 * Embedded hardware selection: Synopsys->ARC
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#include "vec_blas_sgemm.h"
#include "vdsplib.h"
#include "rtwtypes.h"
#include "kalman.h"

/* Storage class 'VCCM' */
__vccm real32_T rtA[4];                /* '<Root>/A' */
__vccm real32_T rtC[2];                /* '<Root>/C' */
__vccm real32_T rtQ[4];                /* '<Root>/Q' */
__vccm real32_T rtx_k_pred[2];         /* '<Root>/x_k_pred' */
__vccm real32_T rtP_k_pred[4];         /* '<Root>/P_k_pred' */
__vccm real32_T rtP_k1_pred[4];        /* '<Root>/P_k1_pred' */
__vccm real32_T rtx_k1_pred[2];        /* '<Root>/x_k1_pred' */
__vccm real32_T rtAdd1[4];             /* '<S1>/Add1' */
__vccm real32_T rtAdd8[2];             /* '<S1>/Add8' */
__vccm real32_T rtR;                   /* '<Root>/R' */
__vccm real32_T rty_k;                 /* '<Root>/y_k' */
__vccm real32_T rtDivide1_DWORK4;      /* '<S3>/Divide1' */
__vccm uint8_T rtdata_update;          /* '<Root>/data_update'
 ↪  */

/* Model step function */
void kalman_step(void)
{
  __vccm real32_T *tmp = (__vccm real32_T
   ↪  *)__vccm_alloca(sizeof(real32_T) * 4);
  __vccm real32_T *tmp_0 = (__vccm real32_T
   ↪  *)__vccm_alloca(sizeof(real32_T) * 4);
  __vccm real32_T *rtb_K_k = (__vccm real32_T
   ↪  *)__vccm_alloca(sizeof(real32_T) *
    2);
  __vccm real32_T *tmp_1 = (__vccm real32_T
   ↪  *)__vccm_alloca(sizeof(real32_T) * 2);

  /* SwitchCase: '<Root>/Switch Case' incorporates:
   *  Inport: '<Root>/data_update'
   *  Product: '<S3>/Divide1'
   *  Product: '<S3>/Matrix Multiply11'
   */
  if (rtdata_update == 1) {
    real32_T rtb_Add5;

    /* Outputs for IfAction SubSystem: '<Root>/Switch Case
     ↪  Action Subsystem' incorporates:
     *  ActionPort: '<S1>/Action Port'
     */
    /* Product: '<S3>/Matrix Multiply3' incorporates:
     *  Inport: '<Root>/C'
     *  Inport: '<Root>/P_k_pred'
     */
    vec_blas_sgemm(101, 111, 111, 1, 2, 2, 1.0F, &rtC[0], 2,
     ↪  &rtP_k_pred[0], 2,
                   0.0F, &tmp_1[0], 2);

    /* Sum: '<S3>/Add5' incorporates:
     *  Inport: '<Root>/C'
     *  Inport: '<Root>/R'
 *  Math: '<S3>/Transpose2'
 *  Product: '<S3>/Matrix Multiply10'
 *  Product: '<S3>/Matrix Multiply3'
 */
rtb_Add5 = (rtC[0] * tmp_1[0] + rtC[1] * tmp_1[1]) + rtR;

/* Product: '<S3>/Matrix Multiply11' incorporates:
 *  Inport: '<Root>/C'
 *  Inport: '<Root>/P_k_pred'
 *  Math: '<S3>/Transpose2'
 */
vec_blas_sgemm(101, 111, 112, 2, 1, 2, 1.0F,
↪  &rtP_k_pred[0], 2, &rtC[0], 2,
               0.0F, &rtb_K_k[0], 1);
rtb_K_k[0] /= rtb_Add5;
rtb_K_k[1] /= rtb_Add5;

/* Product: '<S1>/Matrix Multiply15' incorporates:
 *  Inport: '<Root>/C'
 *  Product: '<S3>/Divide1'
 *  Product: '<S3>/Matrix Multiply11'
 */
vec_blas_sgemm(101, 112, 111, 2, 2, 1, 1.0F, &rtb_K_k[0],
↪  2, &rtC[0], 2,
               0.0F, &tmp[0], 2);

/* Product: '<S1>/Matrix Multiply1' incorporates:
 *  Inport: '<Root>/P_k_pred'
 *  Product: '<S1>/Matrix Multiply15'
 */
vec_blas_sgemm(101, 111, 111, 2, 2, 2, 1.0F, &tmp[0], 2,
↪  &rtP_k_pred[0], 2,
               0.0F, &tmp_0[0], 2);

/* Sum: '<S1>/Add1' incorporates:
 *  Inport: '<Root>/P_k_pred'
 *  Product: '<S1>/Matrix Multiply1'
 */
vdsp_sub_f32(&rtP_k_pred[0], &tmp_0[0], &rtAdd1[0], 4U);

/* Product: '<S1>/Matrix Multiply14' incorporates:
 *  Inport: '<Root>/C'
 *  Inport: '<Root>/x_k_pred'
 *  Inport: '<Root>/y_k'
 *  Product: '<S1>/Matrix Multiply3'
 *  Product: '<S3>/Divide1'
```

```
     *  Sum: '<S1>/Add5'
     */
    vdsp_scale_f32(&rtb_K_k[0], rty_k - (rtx_k_pred[0] * rtC[0]
     ↪  + rtx_k_pred[1] * rtC[1]), &tmp_1[0], 2U);

    /* Sum: '<S1>/Add8' incorporates:
     *  Inport: '<Root>/x_k_pred'
     *  Product: '<S1>/Matrix Multiply14'
     */
    vdsp_add_f32(&rtx_k_pred[0], &tmp_1[0], &rtAdd8[0], 2U);

    /* End of Outputs for SubSystem: '<Root>/Switch Case Action
     ↪  Subsystem' */
  }

  /* End of SwitchCase: '<Root>/Switch Case' */

  /* Switch: '<Root>/Switch1' incorporates:
   *  Inport: '<Root>/data_update'
   */
  if (rtdata_update > 0) {
    /* Product: '<S2>/Matrix Multiply2' incorporates:
     *  Sum: '<S1>/Add1'
     */
    tmp[0] = rtAdd1[0];
    tmp[1] = rtAdd1[1];
    tmp[2] = rtAdd1[2];
    tmp[3] = rtAdd1[3];
  } else {
    /* Product: '<S2>/Matrix Multiply2' incorporates:
     *  Inport: '<Root>/P_k_pred'
     *  Sum: '<S1>/Add1'
     */
    tmp[0] = rtP_k_pred[0];
    tmp[1] = rtP_k_pred[1];
    tmp[2] = rtP_k_pred[2];
    tmp[3] = rtP_k_pred[3];
  }

  /* Product: '<S2>/Matrix Multiply2' incorporates:
   *  Inport: '<Root>/A'
   *  Switch: '<Root>/Switch1'
   */
  vec_blas_sgemm(101, 111, 111, 2, 2, 2, 1.0F, &rtA[0], 2,
   ↪  &tmp[0], 2, 0.0F, &tmp_0[0], 2);
  /* Math: '<S2>/Transpose' incorporates:
   *  Inport: '<Root>/A'
   *  Product: '<S2>/Matrix Multiply2'
   *  Product: '<S2>/Matrix Multiply3'
   */
  vec_blas_sgemm(101, 111, 112, 2, 2, 2, 1.0F, &tmp_0[0], 2,
   ↪  &rtA[0], 2, 0.0F, &tmp[0], 2);

  /* Outport: '<Root>/P_k1_pred' incorporates:
   *  Inport: '<Root>/Q'
   *  Math: '<S2>/Transpose'
   *  Product: '<S2>/Matrix Multiply3'
   *  Sum: '<S2>/Add1'
   */
  vdsp_add_f32(&tmp[0], &rtQ[0], &rtP_k1_pred[0], 4U);

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/data_update'
   */
  if (rtdata_update > 0) {
    /* Product: '<S2>/Matrix Multiply1' incorporates:
     *  Sum: '<S1>/Add8'
     */
    tmp_1[0] = rtAdd8[0];
    tmp_1[1] = rtAdd8[1];
  } else {
    /* Product: '<S2>/Matrix Multiply1' incorporates:
     *  Inport: '<Root>/x_k_pred'
     *  Sum: '<S1>/Add8'
     */
    tmp_1[0] = rtx_k_pred[0];
    tmp_1[1] = rtx_k_pred[1];
  }

  /* Outport: '<Root>/x_k1_pred' incorporates:
   *  Inport: '<Root>/A'
   *  Product: '<S2>/Matrix Multiply1'
   *  Switch: '<Root>/Switch'
   */
  vec_blas_sgemm(101, 111, 111, 2, 1, 2, 1.0F, &rtA[0], 2,
   ↪  &tmp_1[0], 1, 0.0F, &rtx_k1_pred[0], 1);
}
```

Appendix C

Implemented Functions

C.1 BSP-uC sub-module API

Function	**Description**
wtu_wdt_DisableSysWatchdog	Disable System watchdog module (one in TC4x uC).
wtu_wdt_DisableCpuWatchdog	Disable Core watchdog module.
wtu_wdt_DisableSecWatchdog	Disable Cyber Security watchdog module.
InitClock	Configure the uC clock system to the optimum setting for the max clock and peripherals
clock_EnableXOSC	Enable an external oscillator.
clock_SwitchSystemClock	Select requested clock input for system clock.
clock_SwitchPeripheralClock	Selects requested clock input for peripheral clock.
clock_EnableSystemPLL	Enable system PLL with required parameters.
clock_EnablePeripheralPLL	Enable peripheral PLL with required parameters.
clock_SetCon	Set a clock control register using a posting mechanism.
clock_SetClockFreq	Configure the clock distribution (dividers) of the system and peripheral clocks.
core_EnableICache	Enable Instruction cache on the current core.
core_EnableDCache	Enable Data cache on the current core.
core_StartCore	Start requested core from given *reset_vector*.

core_StartAllCores	Start all inactive cores on multi-core uC derivative.
core_DisableCallDepthCounter	Disable call depth counter for HRA, HRB and HRHV resources.
core_DisableVirtualization	Disable virtualization feature.
core_GetVirtualMachine	Return currently executing virtual machine number.
core_GetCurrentCore	Read value of the current execution Core.
core_GetCurrentInterruptPriority	Read value of the current core interrupt priority.
core_stm_ReloadChannel	Initialize requested channel with reload value.
core_stm_EnableChannelIsr	Enable STM Compare interrupt.
core_stm_ClearChannelIsrFlag	Clear active interrupt flag on requested channel.
core_stm_Wait	Wait for <time> in micro seconds using core local STM channel.
core_stm_IsChannelIsrFlag	Read the interrupt flag status.
core_stm_GetChannelCurrentValue	Read current value of the timer from requested channel.
port_SetGPIO	Set output value to the GPIO pin.
port_GetGPIO	Reads input value from the GPIO pin.
port_EnableInput	Configure requested pin to Input mode.
port_EnableOutput	Configure requested pin to output mode with requested characteristics.
pflash_SetWs	Wait States Configuration of PLASH controller Port0.
dflash_SetWs	Wait States Configuration of DLASH controller Port0.
intc_InitBIV	Initialize BSP provided ISR Vector Table to the current Core's BIV register.
intc_SetBIV	Register Interrupt vector table in the current Core.
intc_stm_SetSRC	Configure STM Interrupt in SRC Interrupt module.
intc_EnableExternalInterrupts	Enable external interrupt source in the current Core's ICR register.
intc_InitBTV	Initialize BSP provided Trap Vector Table to the current Core's BTV register.

intc_SetBTV	Register Trap vector table in the current Core.

Table C.1: uC derivative specific API

C.2 BSP ISR Sub-module API

Function	Description
bsp_isr_Init	Initialize RAM ISR Table with all priority vectors pointing to an Undefined Handler routine.
bsp_isr_RegisterHandler	Store the ISR Handler coordinates in ISR RAM table.

Table C.2: Interrupt sub-module API

C.3 BSP BOARD Sub-module API

Function	Description
wdg_Disable	Disable external WATCHDOG present on evaluation board.
led_InitAll	Initialize all LED pins to output mode with a required value.
led_Init	Initialize a particular LED pin to output mode with a required value.
led_Set	Set output value of a particular LED

Table C.3: Board sub-module API

C.4 Mutex Module API

Function	Description
lock	Try to lock the mutex object. If the mutex is already locked, the function immediately returns failure.
trylock	Lock the mutex object. If the mutex is already locked, the function waits until the mutex is freed.
unlock	Try to unlock the mutex object. If the function is called by a non-owner, mutex is not unlocked.

Table C.4: Mutex module API

C.5 Printf Module API

Function	Description
printf_ vprintf_ sprintf_ vsprintf_ snprintf_ vsnprintf_	Upon successful return, all functions return the number of characters written, excluding the string-terminating character. If any error is encountered, negative number is returned. Functions *snprintf* and *vsnprintf* have a limited maximum amount of written characters. A value equal or larger than the limit indicates a truncation. In the case of truncation, the return value is the number of characters that could have been written. Only when the returned value is non-negative and less than the limit, the string has been completely written with a terminating character. It is recommended to use *snprintf*/*vsnprintf* with the limited buffer size instead of *sprintf*/*vsprintf*, unaware of the amount of allocated memory.

Table C.5: Printf module API

C.6 Asclin Module API

Function	Description
initProt	Enable the read and write access to the peripheral registers to all TriCores
init	Initialize the Asclin0 in asynchronous mode according to the defined parameters.
putchar	Send a character to the Communication Port.
puts	Sends string to the Communication Port. The function returns the number of characters written, excluding the terminating NUL character. In case of any error, -1 is returned.
getchar	Read a character from the Communication Port. This function returns the character read as an unsigned char cast to an integer. The function waits in the infinite loop until a character is received.
getcharNoWait	Read a character from the Communication Port. This function returns the character read as an unsigned char cast to an int. If any error is encountered, -1 is returned. The function checks the receive FIFO queue whether there is a character to read. If there is not any character, the function returns -1.

gets	Reads one line (until CR/LF character occurs) from the COM port and saves it into the buffer. Upon successful return, it returns the number of characters read, excluding the line-terminating CR/LF character. If any error is encountered, -1 is returned.
flushFIFO	Clear transmit and receive FIFO queues.

Table C.6: Asclin0 module API

C.7 PPU Boot API

Function	Description
initProt	Enable the read and write access to LMU memory blocks
Start	Release the PPU from reset
waitForInit	Wait until the PPU has booted.
clearTable	Clear target memory for uninitialised data.
copyTable	Copy initialised data from LMU to VMU

Table C.7: PPU boot API

C.8 PPU IPC API

Function	Description
Rpc	Request execution of a kernel on PPU. Return 0 for success, 1 and above otherwise.
setSRC	Configure IPC Interrupt in SRC Interrupt module.
clearSRC	Clear active IPC interrupt flag.

Table C.8: PPU IPC API

Appendix D

Printf Module Configuration

A format specifier follows this prototype:

```
%[flags][width][.precision][length]type
```

The following format specifiers are supported:

Type	Output
d or i	Signed decimal integer
u	Unsigned decimal integer
b	Unsigned binary
o	Unsigned octal
x	Unsigned hexadecimal integer (lowercase)
X	Unsigned hexadecimal integer (uppercase)
f or F	Decimal floating point
e or E	Scientific-notation (exponential) floating point
g or G	Scientific or decimal floating point
c	Single character
s	String of characters
p	Pointer address

Table D.1: Supported Format Specifiers [33]

Flag	Description
-	Left-justify within the given field width; Right justification is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.

#	Used with o, b, x or X specifiers the value is preceded with 0, 0b, 0x or 0X respectively for values different than zero. Used with f, F it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeros (0) instead of spaces when padding is specified (see width sub-specifier).

Table D.2: Supported flags [33]

Width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Table D.3: Supported width specifiers

Precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For f and F specifiers: this is the number of digits to be printed after the decimal point. By default, this is 6, and a maximum is defined when building the library. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Table D.4: Supported precision specifiers

Appendix E

List of Abbreviations

Abbreviation	Meaning
ADAS	Advanced Driver Assistance Systems
API	Application program interface
ASCLIN	Asynchronous/Synchronous Interface controller with LIN support
ASIL	Automotive Safety Integrity Level
AUTOSAR CP	AUTOmotive Open System Architecture Classic Platform
BFM	Bit-accurate Functional Model
BLAS	Basic Linear Algebra Subprograms
BSP	Board Startup Package
CNN	Convolutional Neural Network
CPU	Central processing unit
crt0	C' run-time environment initialization code
CSM	Cluster shared memory
CSRM	Cyber Security Real-time Module
CTOR	Constructor
DMA	Direct Memory Access
DMI	Direct Memory Interface
DSP	Digital Signal Processor
EABI	Embedded-application binary interface
EVB	Evaluation Board
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HW	Hardware
HLIR	High-Level Intermediate Representation
IR	Interrupt Router
ISO	International Organization for Standardization
LAPACK	Linear Algebra Package
LMU	Local memory unit

MCU	Microcontroller unit
NN	Neural network
NNAC	Neural network ARC compiler
PHY	Physical layer
PLL	Phase-locked loop
PPU	Parallel processing unit
RAM	Random Access Memory
SDK	Software development kit
SFR	Special Function Register
SoC	System on a chip
STM	System Timer
STU	Streamin Transfer Unit
T32	Lauterbach Trace32 debugger
TC	TriCore
TCF	Tool Configuration File
UART	Universal asynchronous receiver-transmitter
VCCM	Vector Closely Coupled Memory
VDK	Virtualizer Development Kit
VLIW	Very long instruction word
VM	Virtual Machine
VP	Virtual Prototype
uC	Microcontroller
XTAL OSC	Crystal oscillator

Appendix F

Bibliography

[1] Yadav Abhishek. *Driver drowsiness detection using Deep Learning.* https://github.com/. 2021.

[3] Infineon Technologies AG. *AURIX™ 32-bit microcontrollers for automotive and industrial applications.* Online available at: https://www.infineon.com/. (cited 16.05.2022).

[5] John E. Ball and Bo Tang. “Machine Learning and Embedded Computing in Advanced Driver Assistance Systems (ADAS)”. In: *Electronics* (2019). Online available at: https://www.mdpi.com/.

[6] Alex Becker. *Introduction to kalman filter.* Online available at: https://www.kalmanfilter.net/. (cited 08.05.2022).

[7] Lukas Bielesch. *TC49x BSP example Quick Guide. Version 1.0.* HighTec EDV-Systeme GmbH. 2022.

[8] Lukas Bielesch. *TC49x PPU base example. Version 1.0.* HighTec EDV-Systeme GmbH. 2022.

[9] Lukas Bielesch. *TC49x UART example Quick Guide. Version 1.0.* HighTec EDV-Systeme GmbH. 2022.

[10] Lukas Bielesch, Jiri Hanus, and Milan Zongor. *Real time gesture detector - Project Report.* Tech. rep. Computer Science Department NTUT Taiwan, 2020.

[12] Gordon Cooper. *Flexible embedded vision processing architectures for machine-learning applications.* Online available at: https://www.techdesignforums.com/. (cited 11.05.2022).

[13] Tiago Davi Curi Busarello and Marcelo Godoy Simões. “A Tutorial on Implementing Kalman Filters with Commonly Used Blocks”. In: *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society.* Vol. 1. https://ieeexplore.ieee.org/. 2019, pp. 60–67. DOI: `10.1109/IECON.2019.8927549`.

[16] Djamila Dekkiche. “Programming methodologies for ADAS applications in parallel heterogeneous architectures”. Online available at: https://tel.archives-ouvertes.fr/. PhD thesis. Computer Vision and Pattern Recognition [cs.CV]. Université Paris Saclay (COmUE), 2017.

[18] Embedded Vision Alliance. *Implementing Vision with Deep Learning in Resource-constrained Designs.* Online available at: https://www.edge-ai-vision.com/. (cited 11.05.2022).

[19] Marc Greeen. “"How long does it take to stop?" Methodological analysis of driver perception-brake times”. In: *Transportation human factors* (2000).

[20] Rajeshwari Hegde and Gurumurthy Kargal. “Model Based Approach for the Integration of ECUs”. In: *Lecture Notes in Engineering and Computer Science* 2170 (July 2008).

[21] HighTec EDV-Systeme GmbH. *TriCore Development Platform User Guide. Version 6.0.1.* 2022.

[22] Infineon Technologies AG. *Asclin: Aurix™ TC2xx Microcontroller Training. v1.0.* 2019.

[23] Infineon Technologies AG. *Aurix™ TC49x User’s Manual. v0.73.* 2021.

[24] Infineon Technologies AG. *Aurix™ TC4xx architecture reference. v2.0.* 2021.

[25] Infineon Technologies AG. *Aurix™ VDK User Manual. v1.0.* 2021.

[26] Infineon Technologies AG. *TC49x AA-step COM Target Datasheet. v0.61.* 2021.

[27] Infineon Technologies AG. *TriBoard TC499A COM V1.0 User’s Manual. v1.0.1.* 2022.

[28] *ISO 26262-1: Road vehicles — Functional safety.* Standard. Geneva, CH: International Organization for Standardization, Dec. 2018.

[29] Elena Magán et al. “Driver Drowsiness Detection by Applying Deep Learning Techniques to Sequences of Images”. In: *Applied Sciences* 12.3 (2022). ISSN: 2076-3417. DOI: 10.3390/app12031145. URL: https://www.mdpi.com/2076-3417/12/3/1145.

[31] Yair Moshe Pavel Lifshits Alon Eilam and Nimrod Peleg. *DSP in heterogeneous multicore embedded systems – a laboratory experiment.* Online available at: https://www.eurasip.org/. (cited 08.05.2022).

[32] Adrian Rosebrock. *Face detection (HOG and CNN).* https://pyimagesearch.com/. 2021.

[33] Eyal Rozenberg. *Standalone formatted printing function library.* Online available at: https://eyalroz.github.io/. (cited 17.03.2022). 2022.

[34] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches.* 1st ed. ISBN 978-0471708582. Wiley-Interscience, 2006.

[35] Grant Maloy Smith. *What is ADAS (Advanced Driver Assistance Systems)?* Online available at: https://dewesoft.com/. (cited 09.05.2022).

[36] Synopsys. *What is Virtual Prototyping?* Online available at: https://www.synopsys.com/. (cited 11.05.2022).

[37] Synopsys, Inc. *DesignWare ARCv2 ISA: Programmer's Reference Manual for DW EV7xFS Processors. Version 6367-005*. 2020.

[38] Synopsys, Inc. *Effective Vector Programming Guide MetaWare for AURIX™ TC4x. Version 4222-013*. 2021.

[39] Synopsys, Inc. *MetaWare C/C++ Programmer's Guide for the ccac Compiler. Version 4018-088*. 2021.

[40] Synopsys, Inc. *MetaWare ELF Linker and Utilities User's Guide. Version 0139-110*. 2021.

[41] Synopsys, Inc. *MetaWare MATLAB/SIMULINK Toolbox User's Guide. Version 4167-009*. 2021.

[42] Synopsys, Inc. *MetaWare NN SDK User Guide. Version 4210-002*. 2021.

[43] Synopsys, Inc. *MetaWare Vector DSP Library Databook. Version 6406-007*. 2021.

[44] Synopsys, Inc. *MetaWare Vector Linear Algebra Library User Guide and API Reference. Version 4214-006*. 2021.

[45] Synopsys, Inc. *SPEED Reference for AURIX™ TC4x. Version 4231-001*. 2021.

[46] Synopsys, Inc. *TC4x PPU Bare Metal Low-Level Driver (LLD) User Guide. Version 4227-001*. 2021.

[47] Synopsys, Inc. *TC4x PPU Dispatcher User Guide. Version 4229-001*. 2021.

[48] Synopsys, Inc. *Vector C Quick Reference. Version 4019-009*. 2021.

[49] Synopsys, Inc. *What is ADAS*. Online available at: https://www.synopsys.com/automotive/. (cited 17.03.2022).

[50] U.S. Department of Transportation. "Traffic safety facts: Research Note". In: *NHTSA's National Center for Statistics and Analysis* (2016). Online available at: https://crashstats.nhtsa.dot.gov/.

[52] David Vošahlík. *Estimation, filtering and detection (RM35OFD): Kalman filter*. Online available at: https://moodle.fel.cvut.cz/course/. (cited 17.03.2022).

[53] Lukas Westhofen. "Verifying Automotive C Code using Modern Software Model Checkers". PhD thesis. Mar. 2019. DOI: `10.13140/RG.2.2.24958.54081`.

[54] Oliver Willers et al. "Safety Concerns and Mitigation Approaches Regarding the Use of Deep Learning in Safety-Critical Perception Tasks". In: Sept. 2020, pp. 336–350. ISBN: 978-3-030-55582-5. DOI: `10.1007/978-3-030-55583-2_25`.

[55] Xiao Zhou et al. "Effective Sparsification of Neural Networks with Global Sparsity Constraint". In: *CoRR* abs/2105.01571 (2021). arXiv: `2105.01571`. URL: `https://arxiv.org/abs/2105.01571`.

Appendix G

Sources - Images

[2] *Adaptive cruise control.* https://www.audi-mediacenter.com/en/. [Online; accessed 01-05-2022].

[4] *Amdahl's law.* https://en.wikipedia.org/. [Online; accessed 12-05-2022].

[11] *Common architectures in convolutional neural networks.* https://www.jeremyjordan.me/. [Online; accessed 13-05-2022].

[14] *Deep Learning for Object Recognition: DSP and Specialized Processor Optimizations.* https://www.edge-ai-vision.com/. [Online; accessed 13-05-2022].

[15] *Deep Learning for Object Recognition: DSP and Specialized Processor Optimizations.* https://www.edge-ai-vision.com/. [Online; accessed 12-05-2022]. June 2016.

[17] Messaoud Samir Doudou, Abdelmadjid Bouabdallah, and Veronique Cherfaoui. "A Light on Physiological Sensors for Efficient Driver Drowsiness Detection System". In: *Sensors & Transducers Journal* 224.8 (Aug. 2018), pp. 39–50. URL: `https://hal.archives-ouvertes.fr/hal-02162758`.

[51] *USART in AVR ATmega16/ATmega32.* https://sanskrithitechnology.wordpress.com/. [Online; accessed 17-05-2022].