New hardware platform-based deep learning co-design methodology for CPS prototyping : Objects recognition in autonomous vehicle case-study

Quentin Cabanes

Thèse de doctorat

NNT 2021UPASG042

université PARIS-SACLAY

New hardware platform-based deep learning co-design methodology for CPS prototyping: Objects recognition in autonomous vehicle case-study

Nouvelle méthodologie de co-conception pour de l'apprentissage en profondeur basée sur une plate-forme matérielle pour le prototypage de SCP: reconnaissance d'objets dans une étude de cas de véhicule autonome

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Unité de recherche : Université Paris-Saclay, UVSQ, LISV, 78124, Vélizy -Villacoublay, France.
Référent : Université de Versailles -Saint-Quentin-en-Yvelines

Thèse présentée et soutenue à Paris-Saclay, le 07/06/2021, par

Quentin CABANES

Composition du Jury

El-Bey BOURENNANE Professeur des Universités, Université de Bourgogne	Président
Dong Seog HAN Professeur, Kyungpook National University	Rapporteur & Examinateur
Nicole LEVY Professeure des Universités, Conservatoire National des Arts et Métiers (CEDRIC)	Rapporteuse & Examinatrice
Akash KUMAR Professeur, Technische Universität Dresden	Examinateur

Direction de la thèse

Amar RAMDANE-CHERIF Professeur, UVSQ (LISV)	Directeur de thèse
Benaoumeur SENOUCI Maître de conférences, ECE Paris	Co-Encadrant

To my late grandfather Robert, and my family.

Acknowledgements

I would like to sincerely thank my supervisors, Dr Benaoumeur Senouci and Pr Amar Ramdane-Cherif, for their help and contribution during this work.

I would also like to thank the members of the jury, who accepted to evaluate my work.

A big thanks to all the ECE Paris lab team for their help and support during this thesis.

I would like to thank Alain Houelle for his review and advices.

A special thanks to my partner, Anaïs, who was by my side during that tumultuous time, and to Nikki, who proofread my thesis.

Finally, a big thanks to my family for believing in me all this time and helped me achieve all these projects.

Contents

1 General Introduction **11**

1.1 Introduction 12

1.2 Context and Motivations 12

1.3 Problematics 15

1.4 Thesis Contributions 16

1.5 Thesis Outline 18

2 Cyber-physical systems and embedded artificial intelligence **21**

2.1 Introduction 22

2.2 Cyber-physical systems design 22

2.3 Hardware accelerators for smart cyber-physical systems 24

2.4 Artificial intelligence in cyber-physical systems 25

2.5 3D object detection and recognition for CPS 27

2.5.1 3D vision techniques using 2D/3D sensors 27

2.5.2 Software deep learning for 3D object detection and recognition . . . 28

2.5.3 Hardware acceleration of 3D object detection and recognition application 29

2.6 Design and prototyping time of a hardware accelerated object detection and recognition application for CPS 30

2.7 Conclusion 31

3 An embedded Deep Learning methodology for hybrid CPU/FPGA-

based Cyber-Physical Systems platform design using a hardware Neural Network Processor **33**

3.1 Introduction . . . 35

3.2 Hybrid CPU/FPGA platform . . . 37

3.3 A standard process for HW/SW co-design prototyping . . . 38

3.4 HW/SW co-design methodology in the deep learning AI era . . . 41

3.4.1 Understanding the standard workflow for deep learning algorithms . 41

3.4.2 Exploration of deep learning inside a HW/SW co-design application 43

3.4.3 Prototyping automation tools for hybrid CPU/FPGA platforms . . 46

3.5 Proposed embedded Deep Learning methodology around a FPGA-based Neural Network Processor . . . 49

3.5.1 The methodology design flow . . . 49

3.5.2 Toward the automation of the design flow . . . 53

3.6 Challenges of deep learning in hybrid CPU/FPGA-based CPS . . . 54

3.7 Conclusion . . . 56

4 A hardware Neural Network Processor: core of the methodology **57**

4.1 Introduction . . . 58

4.2 Neural Network Processor . . . 58

4.2.1 Definition . . . 58

4.2.2 Design and Architecture . . . 59

4.2.3 Validation of the NNP . . . 67

4.3 NNP integration into the full prototype . . . 69

4.3.1 Embedded processing to improve performance . . . 69

4.3.2 Control software for reconfiguration purposes . . . 70

4.3.3 Workflow of an application using the NNP . . . 74

4.4 Experimentation and results . . . 75

4.5 Conclusion . . . 79

5 Implementation and validation: a smart LIDAR for pedestrian detection 81

5.1 Introduction 82

5.2 Autonomous Vehicle Case Study: Description 82

5.3 Design of the application 83

5.4 Experimentations 86

5.4.1 Embedded processing algorithms 87

5.4.2 Deep learning algorithms 96

5.4.3 Implementation and results 97

5.5 Conclusion 102

6 General conclusion and future works 105

6.1 General conclusion 106

6.2 Future works 109

A SystemC source code of the Neural Network Processor 113

B Tensorflow source code for Dense Neural Network topologies 123

C Tensorflow source code for SUOD dataset 129

D Résumé de thèse 133

List of Figures

1.2 Past and future evolution toward automated and cooperative driving . . . 14

1.3 Thesis outline 17

2.1 Cyber-physical systems architecture 23

3.1 Methodology global view around embedded DL 36

3.2 Hybrid CPU/FPGA platform with DRAM 38

3.3 HW/SW co-design prototyping for hybrid CPU/FPGA-based platform . . 39

3.4 V-model: HW/SW co-design prototyping based on hybrid CPU/FPGA platform 40

3.5 Deep learning classification standard steps 43

3.6 HW/SW co-design prototyping for a hybrid CPU/FPGA-based Deep learning application. Compared to the previous methodology (Figure 3.3): the deep learning requirements are added inside the software requirements, the data processing and feature calculation are added inside the hardware requirements, the model training and testing is added between the host application and host code compiler, and the model parameters are added before the host executable. 44

3.7 Design flow diagram for a hybrid CPU/FPGA-based HW/SW co-design deep learning software application 45

3.8 Application Tasks Graph 46

3.9 Prototype automated deployment 47

3.10 Our methodology implementation. The differences with previous methodology (Figure 3.6) are: the addition of the NNP IP integration and the removal of a host software development and compilation 50

4.1 Neuron in a neural network 59
4.2 Dense Neural Network (DNN) 60
4.3 Data and tasks representations of a NNP 61
4.4 NNP data flow with 4 cores 64
4.5 Neuron processing unit architecture. The weights, inputs and output arrows represents 32-bit floating points. 66
4.6 Vivado diagram of our NNP using the PS/PL with three DMA and the scheduler connected to four neural processing unit 69
4.7 Splitting the weight matrix into sub-matrices with their size depending on the number of cores. N is the number of neurons in the current layer. M is the number of neurons in the next layer. K is the number of neuron processing unit. K' is the size of the last sub-matrix, with $K' \leq K$. 71
4.8 Hybrid CPU/FPGA-based smart CPS workflow 74
4.9 DNN Topologies for each Dataset 76
4.10 Tests results of the NNP, the execution time is for one feed forward sequence 78
4.11 Execution time per parameter with different number of cores and topologies 78

5.1 Smart LIDAR use-case task flow 83
5.2 Smart LIDAR for object classification case study 85
5.3 Design flow for the embedded DL methodology 86
5.4 Box simple overlapping hierarchical problem. The blue zone represents no box overlapping, the red zone represents two boxes overlapping. 92
5.5 Box multiple overlapping hierarchical problem. The blue zone represents no box overlapping, the red zone represents two boxes overlapping. The green zone represents three boxes overlapping 93

5.6 DNN topology for SUOD dataset 97
5.7 Comparison between software and hardware application execution times . . 98
5.8 Example of the occupancy grid task 99
5.9 Example of the point filtering task results 100
5.10 Example of three sliding boxes on a pedestrian 100
5.11 Pedestrian extracted from a box 101
5.12 Time performance and time per parameter for the SUOD neural network topology 103

List of Tables

4.1 Number of bits for each piece of data in instruction word 63

4.2 Hardware resource utilization of the scheduler module with a 10 ns clock target 65

4.3 Hardware resource utilization of one neuron processing unit module with a 10 ns clock target 67

4.4 Direct register mode DMA registers [95] 72

4.5 Description of the different steps to configure a DMA 73

4.6 Accuracy for each dataset 77

5.1 Comparison of SW and HW Applications Execution Time 98

5.2 Hardware “points to voxels” resource utilization 101

5.3 Results from hardware "points to voxels" module 101

5.4 Accuracy results per dataset 102

Acronyms

ADAS Advanced Driver-Assistance System.

AI Artificial Intelligence.

AMBA Advanced Microcontroller Bus Architecture.

ANN Artificial Neural Network.

ASCII American Standard Code for Information Interchange.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

CD Continuous Deployment.

CI Continuous Integration.

CNN Convolutional Neural Network.

COTS Commercial Off-The Shelf.

CPS Cyber-Physical System.

CPU Central Processing Unit.

DL Deep Learning.

DMA Direct Memory Access.

DNN Dense Neural Network.

DRAM Dynamic Random Access Memory.

DSP Digital Signal Processor.

EEPROM Electrically-Erasable Programmable Read-Only Memory.

EXT4 4th EXTended file system.

FIFO First In First Out.

FPGA Field-Programmable Gate Array.

FSBL First Stage Boot Loader.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLL High Level Language.

HLS High Level Synthesis.

HW HardWare.

IoT Internet of Things.

IP Intellectual Property.

JTAG Joint Test Action Group.

LIDAR LIght Detection And Ranging.

MM2S Memory-Mapped to Stream.

NNP Neural Network Processor.

NoC Network on Chip.

NPU Neural Processing Unit.

NTFS NT File System.

OS Operating System.

PE Processing Element.

PLD Programmable Logic Devices.

RADAR RAdio Detection And Ranging.

RAM Random Access Memory.

RGB Red Green Blue.

RGBD Red Green Blue Depth.

RNN Recurrent Neural Network.

RTL Register-Transfer Level.

S2MM Stream to Memory-Mapped.

SPP Single Purpose Processor.

SW SoftWare.

U-Boot Universal Boot Loader.

V2X Vehicle-to-Everything.

Chapter 1

General Introduction

Contents

1.1 Introduction **12**

1.2 Context and Motivations **12**

1.3 Problematics **15**

1.4 Thesis Contributions **16**

1.5 Thesis Outline **18**

1.1 Introduction

This chapter introduces the thesis context with four parts: context and motivations, problematics, thesis contributions, and finally, the thesis outline. The context and motivation explains the reason of this subject. The problematic is the expression of the problem isolated in the motivation. The contribution outlines how this work can be useful for the scientific community. Finally, the thesis outline details all chapters in the manuscript.

1.2 Context and Motivations

Embedded systems are a robust solution for many technological challenges in our society and are able to deliver precise, predictable and robust behavior. Their evolution gave birth to a new type of system called Cyber-Physical Systems (CPS) which combine computation and physical processes. A CPS can be defined as a networked embedded system that can analyze a physical environment and make decisions from its current state to affect it toward a desired outcome. Such systems possess a great potential, because the physical components of such systems introduce safety and reliability requirements, different from those in general-purpose computing [1]. A large spectrum of fields can benefit from the use of CPS, like smart city, smart mobility, smart health care, etc [2]. Fig 1.1 is an example of the evolution from early embedded systems adapted for the physical world (mechatronics) toward CPS and modern Internet of Things (IoT) systems [3]. Although CPS are a great solution for problems related to physical environment, they are constrained with processing resources, real-time, prototyping, etc. In this work, we are focusing on specific constraints: embedded AI implementation and prototyping time. Because of the limited calculation resources available on such systems, the integration of modern AI, particularly machine learning inference, is a tough problem. Mainly because those types of algorithms are computational heavy when analyzing data to extract the features related to the learnt pattern. For this purpose, lightweight and optimized algorithms made their appearance, but because of modern sensors and new techniques, such

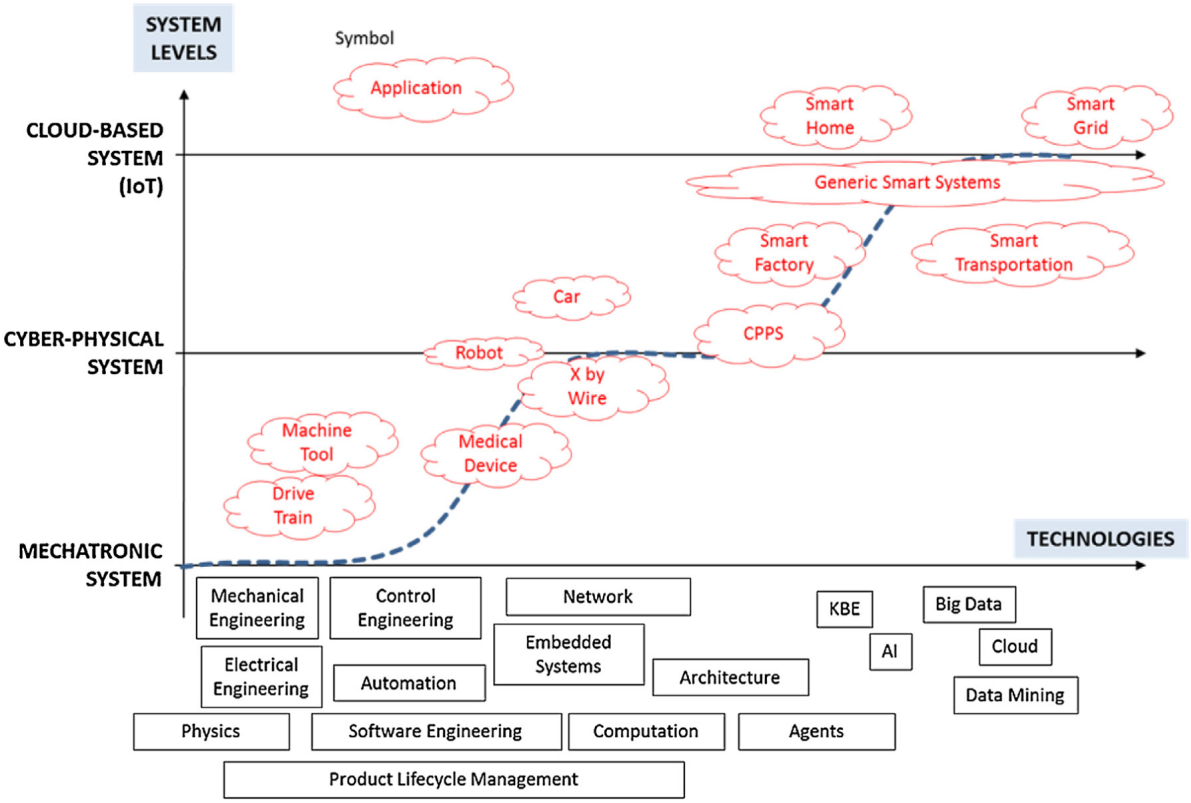


Figure 1.1 – Transition process from early Embedded Systems (Mechatronics) to CPS to Internet of Things [3]

as data fusion, data size keeps increasing faster than the processing power of general purpose systems. This is where Single Purpose Processors (SPP) come in handy. Designing a digital circuit for one specific task has many advantages, such as faster computation, lower power, or smaller memory footprint. However, the prototyping time of specific hardware (also called hardware accelerators or hardware threads) to process AI algorithms is longer than software solutions. This is where the use of reconfigurable digital circuits is advantageous. Programmable Logic Devices (PLD), such as Field-Programmable Gate Array (FPGA), decrease prototyping time compared to Application-Specific Integrated Circuits (ASIC) because there is no need to produce a circuit board for each version of the project. Moreover, using High Level Synthesis (HLS) software, which can translate high level programming language to Hardware Description Language (HDL), further decreases the prototyping time [4]. For all these reasons, this thesis is focused on a methodology for AI applications in CPS based on hybrid CPU/FPGA platforms. In addition, a use case

is chosen, according to the lab thematic: the autonomous vehicle. Since the apparition of vehicles as cheap transportation for individuals, a lot of progress has been made to improve the comfort of the driver and the safety of vehicles. But the interesting part of this evolution is the technologies used to implement such challenges. In most of today's approaches, the usage of AI in specific modules of the vehicle is one of the solutions to make it smart. Moreover, functionalities in cars are approached as independent parts that will analyze the environment and take decisions from the environmental state. In a way, an autonomous vehicle is a CPS. A key example of the evolution of those challenges is the development of vehicles and their evolution to smart vehicles thanks to Advanced Driver-Assistance Systems (ADAS) as seen in Figure 1.2. In this work, to analyze the

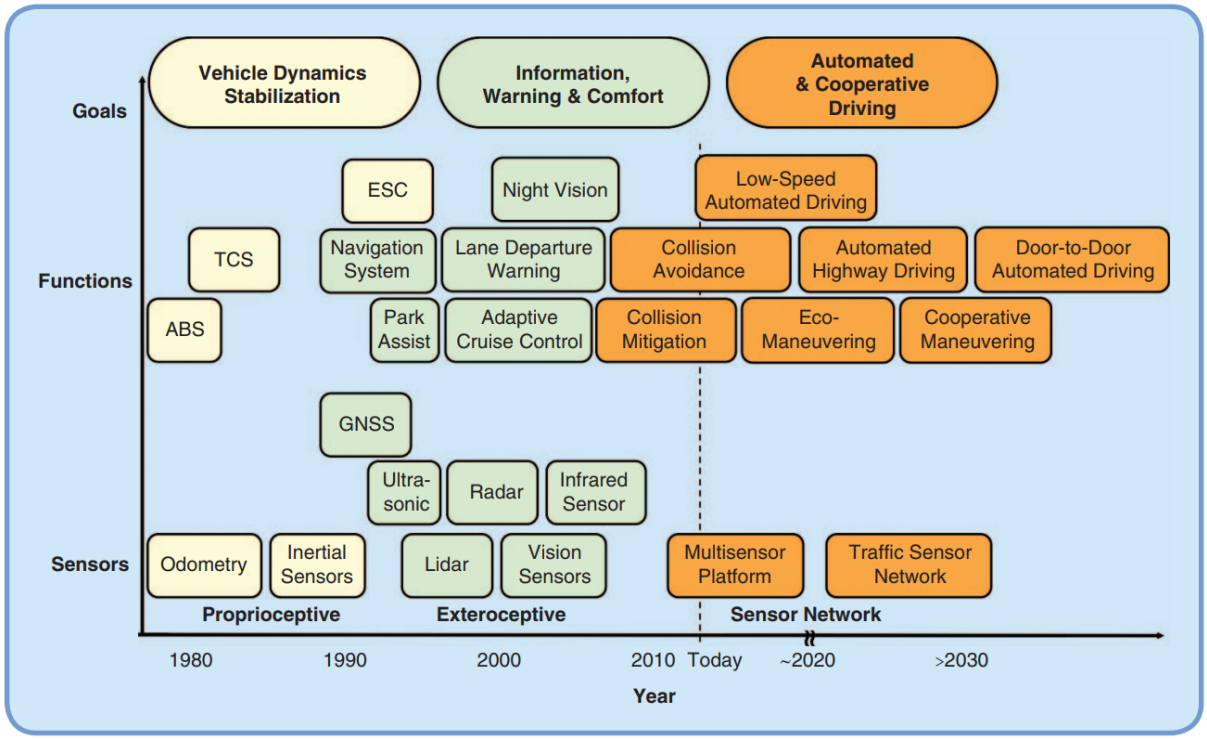


Figure 1.2 – Past and future evolution toward automated and cooperative driving [5].

environment, Artificial Intelligence (AI) technologies are used. More specifically, the use of Deep Learning (DL) with Neural Networks (NN). DL technologies became a hot topic in solving problems such as data analytics and object recognition [6]. Since the late 20th century, it has evolved in a substantial way and tends to be applied in many different fields and applications related to computer science and engineering, such as CPSs [7] [8].

For this purpose, we are using a specific field of the DL spectrum: object recognition. In the case of autonomous vehicles, object recognition is a way to make the vehicle recognize every object in its environment in order to smoothly pass across obstacles, obey traffic laws and predict the behavior of other agents (pedestrians, cars, bicycles, ...). However, with the increased accuracy requirements and complexity of NN architectures, DL technologies have been known to need a lot of computational power, mostly because of their huge number of parameters [9]. Unlike distributed cloud computing, where a lot of power processing is available, embedded systems impel some restrictions in the use of DL technologies. Even when optimizing/compressing NN or using Graphics Processing Units (GPU) for embedded systems, there is still some possible optimization through the usage of specialized processing systems [10] [11]. Also, if we want to build an application using specialized hardware processing for NN (ex: FPGA/ASIC based), we need a complete design methodology for embedded DL inference in order to decrease prototyping time.

1.3 Problematics

Nowadays, CPS are a great solution to analyze a physical environment and interact with it. Moreover, the use of DL algorithms increased the accuracy of such analysis. But integrating DL applications in a CPS comes with many constraints that either conflict with the calculation performances because of the needed calculation resources, or with the prototyping time because of the use of a specific hardware. In the case of a specific hardware, such as deep learning accelerators, we observed a lack of a defined methodology in the context of hybrid CPU/FPGA-based CPS. NN accelerators have become a hot topic since 2014+ [12] and many papers are proposed about it, such as Neuflow, DianNao, etc. But methodologies are missing to integrate those NN accelerators in real-world case studies using hybrid CPU/FPGA-based CPS. We feel this is a bummer because combining NN accelerators with a suitable design methodology means a lot of possibilities and solutions in the context of modern problematics. This is why we are trying to explore

a consistent way of mixing NN accelerators and CPS with the use of hybrid CPU/FPGA platforms. This type of platform is able to bring an easy way to prototype hardware threads specialized in NN calculation with the flexibility of software automation. Our belief is that a methodology using hardware principles such as design re-use [13] and design automation might decrease the prototyping time of DL-based CPS, while still being able to optimize calculation time. In this thesis, we explore the problematic of prototyping time for hardware platform-based deep learning in the context of CPS using a co-design methodology. We also consider the use of software tools for automation in our methodology. Nevertheless, we keep in mind the usage of specific purpose hardware to speed up deep learning calculation, in order to decrease prototyping time and increase CPS analysis performances.

1.4 Thesis Contributions

Our aim is to find a methodology to make CPS with DL algorithms hosted on hybrid CPU/FPGA platform while decreasing prototyping time of such systems. This issue is a real challenge, because of the constraints of CPS and the integration of AI in such systems. In this work, AI conception is mainly about DL architectures and inference. The contributions can be described as follows:

1. A methodology to develop DL applications for CPS using a hybrid CPU/FPGA platform

2. A hardware Neural Network Processor (NNP) architecture, design and prototype

3. A configuration and benchmarking software for the NNP

4. A validation of the NNP with different configurations and results

5. An automation tool to setup a hybrid CPU/FPGA prototype board for embedded DL applications

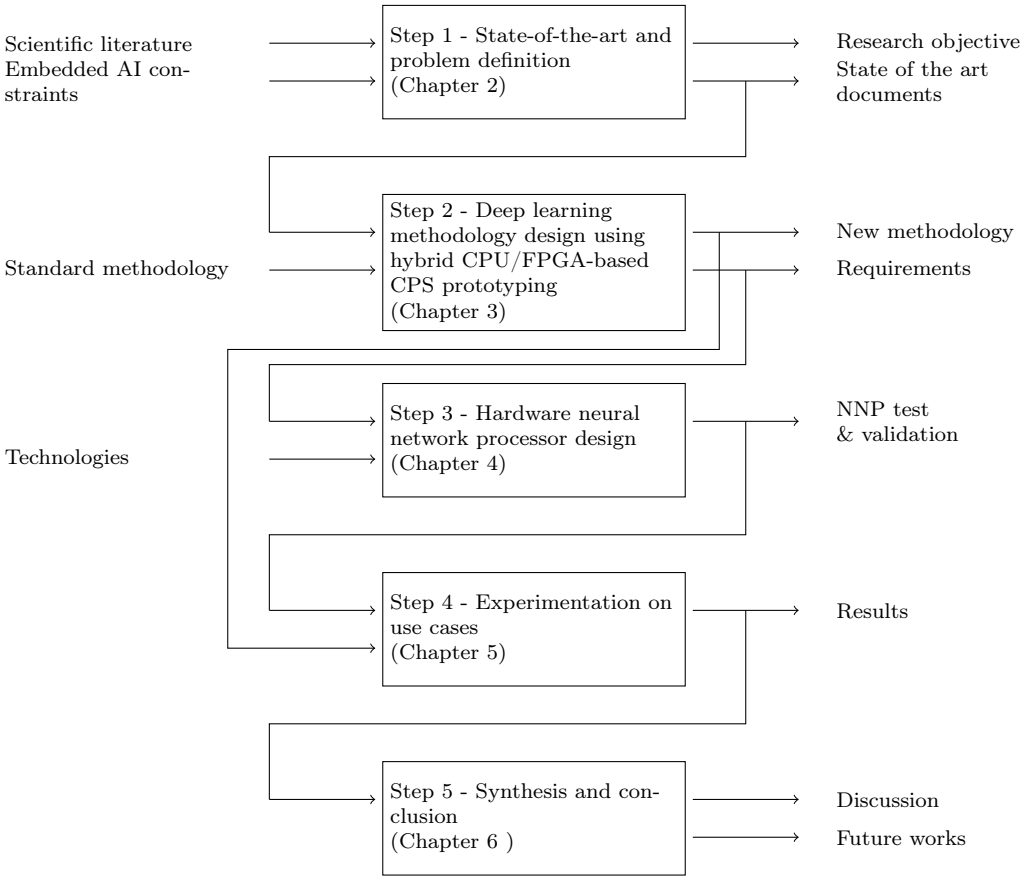


Figure 1.3 – Thesis outline

6. Case study validation: pedestrian detection for autonomous vehicles with several algorithms using a 3D PointCloud from a LIDAR

A thesis outline is described in Figure 1.3 and roughly describes the order of the chapters and the link between each chapter. The figure represents all chapters of this thesis except the context chapter (Chapter 1). This thesis outline presents a specific approach to explore a problem and find a logical pathway to a solution through components. The first step is the exploration of the scientific literature about embedded AI in order to delimit the research objectives and extract all documents necessary in order to study the problem. The second step will use the related works to extract from them the standard methodology for designing a hybrid CPU/FPGA-based prototype using DL. From this are inferred the requirements for a NNP and a new methodology is designed. The third step is about the design of this NNP, from the methodology requirements and the current

technologies, a hardware NNP is designed and validated. The fourth step will use the newly made NNP and the new methodology in order to experiment on a use case and get results. The fifth step is the synthesis of this work, which will highlight out discussions and future works. Embedded DL and hardware platform-based prototyping methodology are tackled, because they are great tools to improve prototyping time in CPS. Embedded DL is used as an analysis tool to infer an environmental state. The hardware platform-based prototyping methodology serves as a support for embedded DL application by easing the development and deployment. These domains are explored to find one solution among many that will fit the problematic and this exploration is described in the coming chapters.

1.5 Thesis Outline

Chapter 2 presents a background exploration of the different domains tackled in this work: hybrid CPU/FPGA-based design for CPS and their related prototyping methodology, DL techniques for object recognition, neural networks design for FPGA hardware and a review of the existing architectures. Those four domains are mixed together from a methodology to the hardware and software development.

Chapter 3 describes an embedded DL methodology for hybrid CPU/FPGA-based CPS platform design using a hardware NNP. Starting from the standard methodology for CPU/FPGA platform-based design, this chapter explains the required modification of the methodology to include DL applications and ease the development for implementing such application. A dedicated hardware for DL computing is made as part of the methodology, in order to simplify some development steps.

Chapter 4 provides the architecture and design of a hardware NNP. Those choices are explained in detail in this chapter, in order to understand how it can be used. Then is presented the NNP IP (Intellectual Property) architecture as hardware accelerators. Finally, some experimentation and results on the NNP are presented in order to display some performances. Experimentations are done to validate the NNP in standalone mode

in order to determine its limitation.

Chapter 5 explains how to use the methodology with a specific use case: a smart LIDAR for pedestrian detection. A first implementation of a hybrid CPU/FPGA-based DL application is presented using this work's methodology, and the already conceived NNP is validated in this real world case study. A design flow is presented, made from the methodology in chapter 3, in order to detail the different steps of prototyping.

Chapter 6 concludes this work with a synthesis of our contribution. The final solution is discussed and future works are highlighted.

Chapter 2

Cyber-physical systems and embedded artificial intelligence

"Prendre des p'tits bouts d'trucs et puis les assembler ensemble"

Stupeflip

Contents

2.1 Introduction 22

2.2 Cyber-physical systems design 22

2.3 Hardware accelerators for smart cyber-physical systems . . . 24

2.4 Artificial intelligence in cyber-physical systems 25

2.5 3D object detection and recognition for CPS 27

2.5.1 3D vision techniques using 2D/3D sensors 27

2.5.2 Software deep learning for 3D object detection and recognition 28

2.5.3 Hardware acceleration of 3D object detection and recognition application 29

2.6 Design and prototyping time of a hardware accelerated object detection and recognition application for CPS 30

2.7 Conclusion 31

2.1 Introduction

Nowadays, Cyber-Physical Systems (CPS) interact with the physical world by analyzing their environment using a variety of sensors. For this purpose, a powerful analysis tool is needed, such as Artificial Intelligence (AI), more precisely Deep Learning (DL) algorithms. Since the late 20th century, DL algorithms have evolved in a substantial way, and tend to be applied in many different fields and applications related to computer science and engineering, such as CPSs [7] [8]. However, with the increased accuracy requirements and complexity of Neural Networks (NN) architecture, DL technologies have been known to need a lot of computational power, mostly because of their huge number of parameters. Unlike distributed cloud computing where a lot of power processing is available, embedded systems impel some restrictions for the use of DL technologies. Optimizing/compressing NN or using Graphics Processing Units (GPU) for embedded systems is a great way to embed DL algorithms, nevertheless the usage of single-purpose processing systems boast great results [10] [11]. Moreover, if we want to build an application using single-purpose hardware processing for NN (eg. FPGA or ASIC-based), we need a complete design methodology for embedded DL in order to speed up the development. In this chapter, the state of the art is presented in five main points: 1) techniques for cyber-physical systems design in order to understand what a CPS is 2) the hardware used for CPSs and its constraints for smartness, 3) the integration of AI in CPSs from a hardware point of view, 4) the evolution of object recognition techniques in deep learning and the computation of such applications on specific hardware, 5) the prototyping time issues of a 3D object recognition application for CPS.

2.2 Cyber-physical systems design

Cyber-physical systems (CPS) are mainly characterized as systems with a physical input and output. Their goal is to analyze a physical environments from sensor data and determine the correct action to perform to guide the environment into a desired state [1].

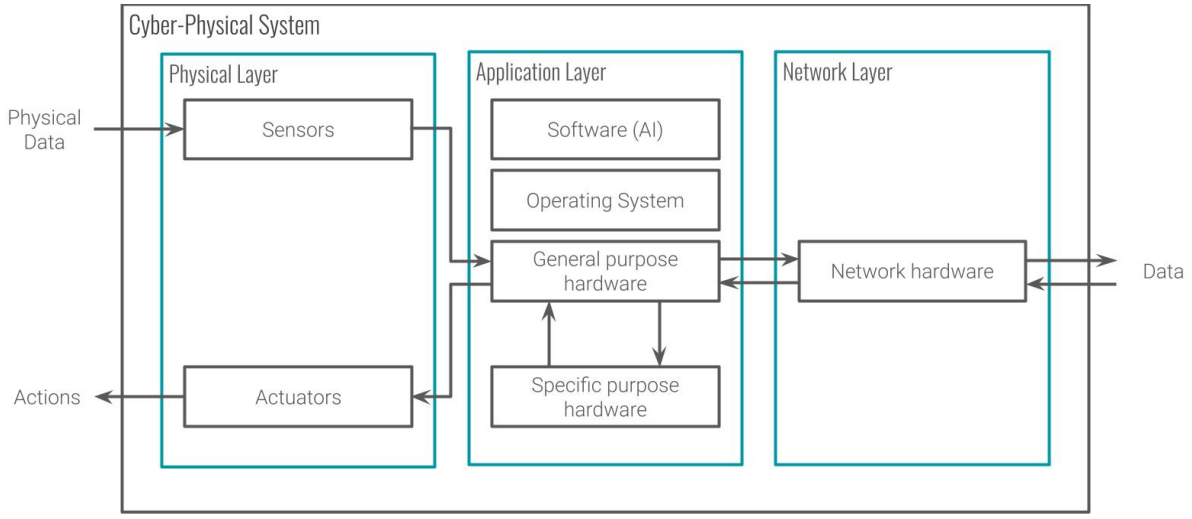


Figure 2.1 – Cyber-physical systems architecture

In this thesis, autonomous vehicles are represented as examples of networked CPS: the environment contains physical objects such as pedestrians, signs and other vehicles, with the sensors being cameras, RADARs, LIDARs, etc; and the actions to perform are about controlling the vehicle, with the final goal being reaching its intended destination. Figure 2.1 describes the architecture of a CPS as three layers communicating with each other. The physical layer is in charge of the interaction with the physical world: observing the surrounding environment to turn it into data for the system with sensors and influencing its environment to reach a desired state with actuators. The application layer is in charge of the analysis of the physical data and the communication with other CPSs on the network. This layer is composed of a general purpose hardware in charge of the software stack, which executes the main application of the CPS, and of an optional specific purpose hardware if some operations need to be accelerated (GPU, ASIC, FPGA, ...). The network layer is simply in charge of the communication between CPSs in the network to share data for a more resilient analysis of the physical world. In our context, the autonomous vehicle, the physical layer is composed of 3D sensors such as LIDARs, RADARs, and 2D sensors such as cameras. Actuators would be the vehicle movement such as braking, acceleration and controlling the wheel. The application layer would be composed of a software dedicated to decision-making running on a general purpose hardware, and a hardware

deep learning application represented by the specific purpose hardware. Meanwhile, the network layer would be represented as V2X (Vehicle-to-Everything) communications [14] [15] [16].

However the design of such systems is a real issue because of its multidisciplinary approach [17] [18]. The increasing complexity of such systems means the need for specific design methodology, such as contract-based or model-based, in order to decrease the prototyping time [19] [20] [3] [21]. In order to decrease the prototyping time in this thesis, we are oriented toward a platform-based design methodology for our CPS prototype. Because we are focusing the prototyping part of the design, full software solution would be the first approach; however to tackle performance issue we need the possibility of hardware accelerated solutions. What we have in mind in this work is to prototype on a hybrid CPU/FPGA platform to benefit the flexibility of the software and the performance of the hardware.

2.3 Hardware accelerators for smart cyber-physical systems

Hardware is an important constraint in CPS, because it will guide the development and deployment of the applications toward the prototype. Subsequently, the choice of hardware is an important decision to make. For all these reasons, FPGA-based chipsets are valuable, because they can emulate any type of hardware with performances near ASIC (Application Specific Integrated Circuit). Platform-based design and prototyping has seen an increase in its relevance, because of today's pressure on time-to-market and design costs. Techniques to design such systems have greatly progressed as explained by A. Sangiovanni-Vincentelli [13] and Pinto et al. [22]. The overall methodology also evolved, thanks to automation tools and design reuse. Platform-based design made evolve topics such as cyber-physical systems as shown by Nuzzo et al. [23]. But our interest is specifically focused on the integration of neural network in FPGA-based methodology. Guo et

al. [24] demonstrated the resource utilization of neural network calculation in terms of parameters and operations. Their survey shows that FPGA is a promising platform for neural network acceleration because of speed and energy efficiency. Li et al. [25] proposed a model-based design methodology involving deep neural network. They proposed an integrated set of tools and libraries alongside their methodology, in order to assist designers of signal processing systems. Shawahna et al. [26] made a survey about FPGA-based accelerators for deep learning networks, in particular convolutional neural network (CNN), and tried to isolate a methodology for their conception. Their survey revealed a specific pattern for FPGA-based accelerated neural network architecture, which is presented with techniques to optimize design and specific tools to automate the design process. Now we have a perspective for the design of a FPGA accelerator and specifically a NNP on FPGA platform. But there is a lack of prototyping methodology including a NNP as a tool in their design steps to prototype a fully functional embedded DL-based application.

2.4 Artificial intelligence in cyber-physical systems

In this work, we tackle the principle of smartness in embedded systems, especially embedded CPSs. Our vision of smartness in this thesis is mainly based on the concept of learning systems, because of their versatility toward analysing environments. Furthermore, as we are mainly working on hybrid CPU/FPGA-based systems, we are trying to find a way to compute neural networks with specialized hardware. Hardware accelerated neural network is not a new topic, either as an analog chip with Säckinger et al. [27] [28] or in FPGA: Botros et al. [29] who made a hardware implementation of an artificial neural network (ANN) in 1994. With the rise of cyber-physical systems and the progress in performance of deep learning, this topic soon became a hot one. Several FPGA implementations were done like Ferrer et al. [30], Ormondi and Rajapakse [31] or Sahin et al. [32] even using floating points arithmetic. Farabet et al [33] [34] [35] [36] [37] proposed several architectures for the hardware implementation of convolutional neural networks (CNN) and even

a runtime reconfigurable architecture. Pham et al [38] designed an ASIC to accelerate neural network with a DMA for configuration purposes from Farabet et al. FPGA-based NeuFlow system. Esmaeilzadeh et al. [39] proposed a reconfigurable neural accelerators architecture. Lozito et al. [40] proposed an FPGA implementation of a feed forward neural network with floating point arithmetic and presented their speed performance. Chen et al. [41] proposed a high-throughput ASIC processor for CNN and DNN computation. Architecture models for neural network have emerged and their optimization was tackled: Zhang et al. [42] [43] [43] showed a way to optimize CNN accelerators as well as sparse NN accelerators. Venkatesh et al. [44] showed a way to optimize sparse and low-precision neural network. Wang et al. [10] proposed a scalable deep learning accelerator in order to optimize performance and maintain low power cost for large neural network. Gokhale et al. [45] proposed a CNN accelerator which is agnostic to CNN workload. Kreinar [46] presented a C++ library for the Vivado HLS [47] software to deploy trained neural network on FPGA. Zhou et al. [48] proposed a binary neural network on FPGA. Mittal [49] did a survey for FPGA-based accelerators for CNNs to highlight the key ideas of several dozens of works. Venieris et al. [50] proposed a survey of CNN-to-FPGA toolflows and compared their characteristics and features. Wang et al. [51] presented a survey about FPGA-based DL accelerators in order to demonstrate their advantages and disadvantage for research purpose. Shawahna et al. [26] presented a survey of FPGA-based deep learning accelerators and introduced a table that compared 30 papers using, among others, the precision, operations per second and power. FPGA-based accelerators for DL is a rising topic, especially for CNN accelerators considering their accuracy. All those works inspired us when conceiving our own NN accelerator but the main difference is that our accelerator is made to be integrated in our prototyping methodology and focused on the ease to prototype and use. Moreover, those work are mainly focused on CNN, while we are working on DNN. Now we need to understand what type of DL architecture needs to be studied in order to find the equilibrium between accuracy, power and computing time.

2.5 3D object detection and recognition for CPS

Object detection and recognition applications are a crucial part of navigation inside a physical environment. The possibility to detect and understand a surrounding environment open doors to many opportunities in the scope of CPSs. In our case, 3D object detection and recognition for CPS is a way to give an autonomous vehicle an understanding of its surroundings to enforce traffic laws and reduce accidents. But in order to set up object detection and recognition, three parts are needed: sensors, object recognition algorithms and hardware capable of computing information in real time.

2.5.1 3D vision techniques using 2D/3D sensors

Nowadays, sensors boast improved accuracy and heterogeneous type of information perceived. With the current state of technologies, autonomous vehicles are equipped with a large variety of heterogeneous sensors. As such, it is needed to understand how those sensors work, related to 3D vision techniques, to enhance 3D object detection and recognition algorithms. Thus, in order to correctly design a 3D object detection and recognition application for CPS, we need to choose the correct algorithm, as well as the correct sensors and hardware, considering the whole system's requirements.

2.5.1.1 2D sensors

Most of the 2D sensors in autonomous vehicles are cameras. Their primary function is to convert light waves into a 2D images consisting of pixels (often a mix of the colors Red Green and Blue [RGB]). The resolution (number of pixels in an image) of modern cameras make it easier to detect and recognize objects in an image as well as increasing the detection distance. But the main problem about 2D sensors in autonomous vehicles is the lack of the depth dimension. This dimension is somewhat required for moving vehicles, even if it can be determined using specific algorithms [52] [53], but this means more calculations for the system. Hence, 2D sensors are often combined with 3D sensors

to improve the accuracy of the vehicle perception.

2.5.1.2 3D sensors

Autonomous vehicles are equipped with a large variety of 3D sensors like RADARs, LIDARs or 3D cameras. RADARs (RAdio Detection And Ranging) use radio waves in predetermined directions to find an object when the original signal is reflected or scattered back. LIDARs (LIght Detection And Ranging) use the same technique as a RADAR but with laser light. 3D cameras can either be RGBD (Red Green Blue Depth) cameras, a type of camera with a depth sensor embedded in it, or stereo camera, two or more lenses to simulate human binocular vision and thus able to capture 3D images. Moreover, it is possible to combine 2D and 3D sensor data (often called data fusion), to enhance the quality of the vehicle's vision [54]. Therefore, with such sensors, the capabilities of 3D object detection and recognition is improved, but the computation time of those algorithms is also increased because, with the addition of one more spacial dimension, data size increases exponentially.

2.5.2 Software deep learning for 3D object detection and recognition

3D object classification is a hot topic considering current sensors such as LIDAR or 3D camera. The usage of deep learning applications may help reach great accuracy in classification of 3D objects. Maturana and Scherer [55] proposed a 3D convolutional neural network (CNN) using voxels as input, and proposed a way to convert a point cloud to a voxel model. Brock et al. [56] proposed a voxel-based autoencoder and convolutional neural network to generate and classify 3D objects. Garcia-Garcia et al. [57], Hegde and Zadeh [58] and Jing Huang and Suya You [59] proposed different 3D convolutional neural networks (CNN) architectures using voxels as inputs to classify objects. Qi et al. [60] proposed a deep learning architecture to directly classify and segmentate point cloud instead of voxels. Zhi et al. [61] proposed a lightweight version of 3D convolutional network which

is interesting for embedded computation. Ioannidou et al. [62] proposed a residual neural network (ResNet) for 3D object classification using voxels. So 3D volumetric binary grid like voxels seems to be the way to process 3D data in order to make pattern prediction for object classification. 3D object classification using deep learning is a hot topic, because of today's 3D sensors and the accuracy they can yield. But software deep learning 3D object detection is heavy on computing power, especially for embedded systems.

2.5.3 Hardware acceleration of 3D object detection and recognition application

There are a lot of different hardware platforms available to accelerate 3D object detection and recognition applications. In this work, we will look into GPU, ASIC and FPGA hardware platforms and compare them. Birk et al [11] made a comparison between a GPU and a FPGA on a reflection image reconstruction application for 3D ultrasound computer tomography. Their results show that the GPU is 2.2 times faster, but with the estimated power consumption of the FPGA board (estimated to a maximum of 40W), the FPGA-based accelerator has a better performance per watts ratio (compared to the GPU using 250W). Nurvitadhi et al. [63] [64] [65] tried to answer the question about FPGAs beating GPUs in accelerating deep neural networks and showed that the current trend in deep learning algorithms may favor FPGAs, at least in term of performance/watts. FPGA-based hardware accelerated neural network seems to be a promising ways to compete with GPU in embedded approach such as CPS. The main conclusions about these comparisons is that ASIC-based platforms are the best choice for acceleration, but the prototyping time of such systems is the longest. GPU-based platforms are the easiest to prototype because it is fully reprogrammable, but they have the greatest power consumption, which means it is not always suitable in the context of CPS. FPGA-based platforms offer a middle ground between ASIC and GPU. Coupled with the current progress in FPGA design and the current hybrid CPU/FPGA platforms, it might become the best hardware platform choice when prototyping CPS-based applications.

2.6 Design and prototyping time of a hardware accelerated object detection and recognition application for CPS

Object detection and recognition applications are already difficult to develop and prototype because of the skill set needed, such as data processing and AI algorithms. If you add a layer with hardware acceleration, the complexity of such systems is increased exponentially because of the new heterogeneous skills needed and the cooperation between the software and hardware world. And if you add embedded system constraints such as the ones in CPS, it becomes a real challenge. That is why the prototyping time of those systems need to be tackled; in this thesis, we are specifically tackling the prototyping part and not the time to market. Andrews et al. [66] [67] is tackling the problem with Commercial Off-The Shelf (COTS) components in order to reduce design costs and time to market. Moreover, they are introducing hybrid CPU/FPGA chips with a thread-oriented programming model for a faster development. One of the interesting parts is about programming languages for reconfigurable architectures and the usage of high-level languages for system-level design. From this, High Level Synthesis (HLS) software comes into view. Inggs et al. [4] investigated into the maturity of HLS software for business. They concluded that HLS tools can be reliable for industrial business but an expertise into embedded systems and particularly FPGA systems is still needed to yield more performance. Nane et al. [68] made a survey about the different HLS tools available, in order to compare academic and commercial tools. They concluded that academic and commercial HLS tools are not drastically far apart in terms of quality, but they may yield different optimization depending the target application. If we now return to the CPSs world, Hehenberger et al. [3] presented the importance of design, modelling, simulation and integration of CPS, and particularly showed that it is still a multidisciplinary world. So prototyping time of hardware accelerated object detection and recognition application for CPS still needs to be improved with solutions such as the usage of COTS, design

re-use and specific methodologies. An example of COTS product for DL-application in CPS would be industrial neural network accelerators such as the Google TPU [69], Apple A12 Bionic [70], Intel Nervana NNP [71] or Intel Movidius [72].

2.7 Conclusion

In this chapter, we presented five main points: 1) techniques for CPSs design in order to understand what a CPS is 2) the hardware used for CPSs and its constraints for smartness, 3) the integration of AI in CPSs from a hardware point of view, 4) the evolution of object detection and recognition techniques in deep learning and the computation of such applications on specific hardware, 5) the prototyping time issues of an object recognition application on a hybrid CPU/FPGA hardware in the CPS context. The lack of methodologies for the integration of NN accelerators for hybrid CPU/FPGA-based CPS applications built up our interest into developing a methodology for DL-based 3D object detection and recognition applications accelerated on a hybrid CPU/FPGA hardware platform. The next chapter starts to introduce the standard methodology for HW/SW design and present how we achieved our methodology.

Chapter 3

An embedded Deep Learning methodology for hybrid CPU/FPGA-based Cyber-Physical Systems platform design using a hardware Neural Network Processor

Contents

3.1 Introduction 35

3.2 Hybrid CPU/FPGA platform 37

3.3 A standard process for HW/SW co-design prototyping 38

3.4 HW/SW co-design methodology in the deep learning AI era 41

3.4.1 Understanding the standard workflow for deep learning algorithms 41

3.4.2 Exploration of deep learning inside a HW/SW co-design application 43

3.4.3 Prototyping automation tools for hybrid CPU/FPGA platforms 46

3.5 Proposed embedded Deep Learning methodology around a FPGA-based Neural Network Processor 49

3.5.1 The methodology design flow 49

3.5.2 Toward the automation of the design flow 53

3.6 Challenges of deep learning in hybrid CPU/FPGA-based CPS 54

3.7 Conclusion 56

3.1 Introduction

In the previous chapter, we presented the state of the art and explained the need of a DL methodology for hybrid CPU/FPGA-based CPSs. In this chapter, we will present our work toward our embedded DL based methodology for CPU/FPGA-based CPS platform design using a hardware NNP (Neural Network Processor). We will introduce Hardware/-Software (HW/SW) co-design methods, their involvement in systems such as CPS and the usefulness of hybrid CPU/FPGA platforms. Because of modern algorithms, the hardware resources needed in order to reach a result are far more substantial than in the past. And because of the constraints of embedded systems, a full software calculation system may not have enough hardware resources to perform in a granted time. Using HW/SW co-design allows a system to expand software calculations to specific hardware accelerators in order to speed up computations. Because of this, HW/SW co-design methods have become more and more popular in recent years, when designing embedded systems, and thus CPS when designing modern systems. CPS comes from the evolution of embedded systems in the 20th century. Over recent years, significant effort has been put into understanding the relationship of the individual with the physical environment. CPS are a great base toward conceiving systems that interact with the physical environment but with an analysis of the environment. The better the analysis, the better the understanding of the environment, and thus, the better the interaction with the physical world. However, improving the analysis comes from two main factors: better sensors and better algorithms. One occurrence that greatly affected the CPS topic is the appearance of modern sensors that were much more accurate and could detect a large choice of physical information. In our case, we are mainly talking about 3D sensors such as RADAR, LIDAR and 3D cameras. Those sensors improved the quantity of data available and enhanced the performance of analysis algorithms. This is especially important when looking at modern methods using deep learning. The current understanding of deep learning shows that the more information available, the more accurate the result. Subsequently, the usage of deep learning algorithms in CPS can improve the analysis of the physical environment, but at

the cost of resource-heavy computation. This is where HW/SW co-design methods can be a great help toward the conception of smart CPS. That is why we are trying to embed DL algorithms inside the system using a hybrid CPU/FPGA platform. The use of software is an important part in CPS prototyping, and making a hardware deep neural network accelerator as part of this system is a way to reduce the toll of the computation. From this point, we want to describe our path toward a CPU/FPGA-based DL methodology using a NNP as its core. The exploration to propose a new methodology starts with the understanding of hybrid CPU/FPGA platform and the standard processes for HW/SW co-design prototyping. Then, we present a simple methodology to develop a HW/SW co-design application using deep learning software. Finally, we present a CPU/FPGA-based DL methodology using a hardware deep neural network accelerator as its core. Figure 3.1 is a global view of our final methodology. It consists of four parts: the deep learning software represents the deep learning architecture design, its training and testing to finally extract all trained weight matrices; the embedded processing is the design of the different data processing algorithms and describing them as embedded system code; the hardware acceleration transforms the embedded processing source code as HDL (Hard-

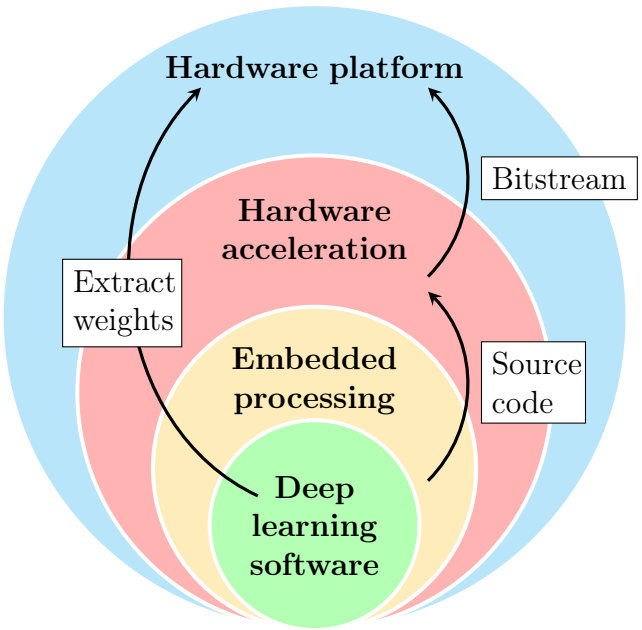


Figure 3.1 – Methodology global view around embedded DL

ware Description Language) either with HLS tools (High Level Synthesis) or manually; finally, the hardware platform can be prepared with the extracted weights loaded inside (using file systems, EEPROM, ...) and with the bitstream containing the FPGA architecture consisting of the hardware accelerated data processing and the NNP. In order to describe this methodology, we will explain from where we started and the different steps toward our goal.

3.2 Hybrid CPU/FPGA platform

Hybrid CPU/FPGA platforms are systems containing both CPU and FPGA components communicating between themselves. The CPU and FPGA are either two different chips or in the same chip such as the Xilinx Zynq, the Intel Altera Stratix and the MicroSemi SmartFusion. This union enables many types of design by combining the flexibility of software-driven development with the performance of a hardware-driven one. The main purpose of such platforms is to design a platform with hardware application-specific components to speed up performance-critical functions. Figure 3.2 presents a view of a simple hybrid CPU/FPGA plaform with a DRAM (Dynamic Random-Access Memory). An example of a simple application on this type of platform would be a software calling FPGA application specific components as hardware threads while computing something else. DRAM can be used here to exchange data between the software threads processed by the CPU and the hardware threads on the FPGA. Such structure is a great way to speed up performance-critical functions that require complex calculations. In this thesis, we consider performance-critical functions as data processing and deep neural network calculation. With application specific components computing inferences, the software threads can focus on other tasks such as network communication or UI display. We consider HW/SW co-design using hybrid CPU/FPGA platform a great way to speed up CPS prototyping, because it merges software development with hardware performances and enables prototype automation.

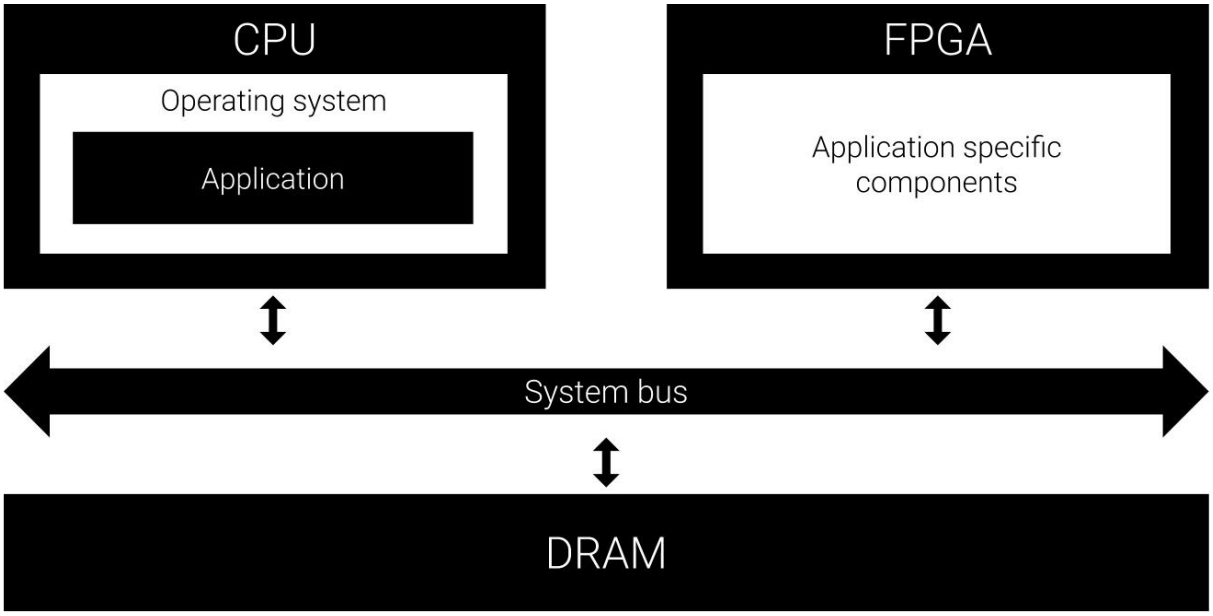


Figure 3.2 – Hybrid CPU/FPGA platform with DRAM

3.3 A standard process for HW/SW co-design prototyping

In order to make a new methodology using HW/SW co-design prototyping for hybrid CPU/FPGA-based platform, it is necessary to understand how it works. Figure 3.3 is a diagram representing a standard design flow for HW/SW co-design prototyping for CPU/FPGA-based platform. We can consider the standard design flow as a V-model [73] as seen in Figure 3.4. The system definition part is the first step to define and dissociate the software and the hardware processes. The goal is to isolate the different processes in the system in order to achieve the desired result. Once the processes defined, we need to determine if it should be executed as software or hardware threads. In the case of HW/SW co-design, hardware is mostly used for hardware acceleration, which means executing tasks the software will be slow at processing, whereas software processing is about controlling the hardware and manipulating data that hardware processing has trouble with. In this work, we have two ways of deciding: either the function is determined as performance-critical because we want it to be as fast as possible, thus a hardware thread is chosen, or we make the function as a software thread and measure its performance;

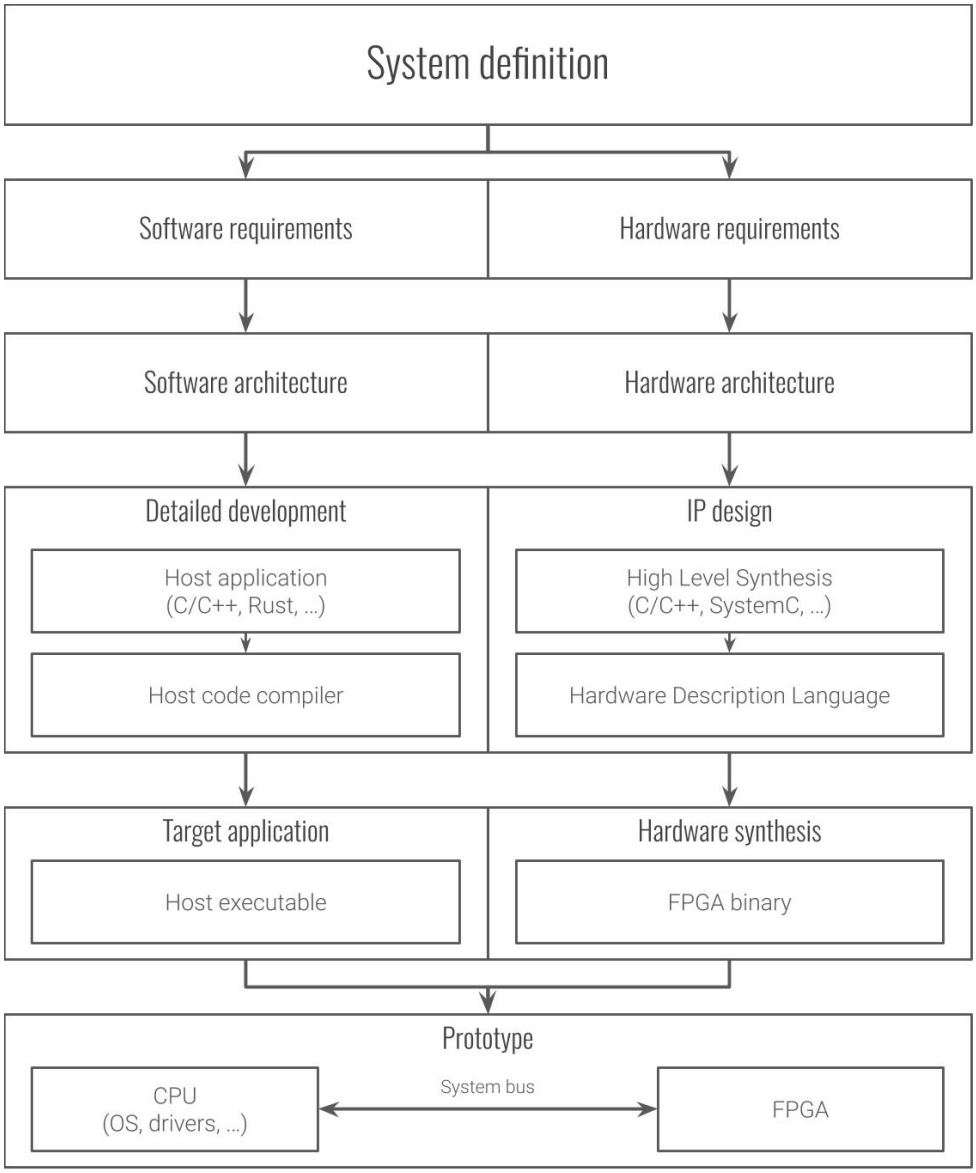


Figure 3.3 – HW/SW co-design prototyping for hybrid CPU/FPGA-based platform

depending on the results we choose between software and hardware threads. Once the requirements and architectures have been defined, the development part starts. This part is one of the most time-consuming tasks because of the iterations done to stick with the requirements set beforehand. Moreover, the need to separate software and hardware applications means that the skills required are heterogeneous, which mostly suggests two teams of individuals working on those tasks: one team for the software and the other for the hardware. Finally, the implementation and deployments are also key parts, and often subject to going back to the development stage when the integration between software

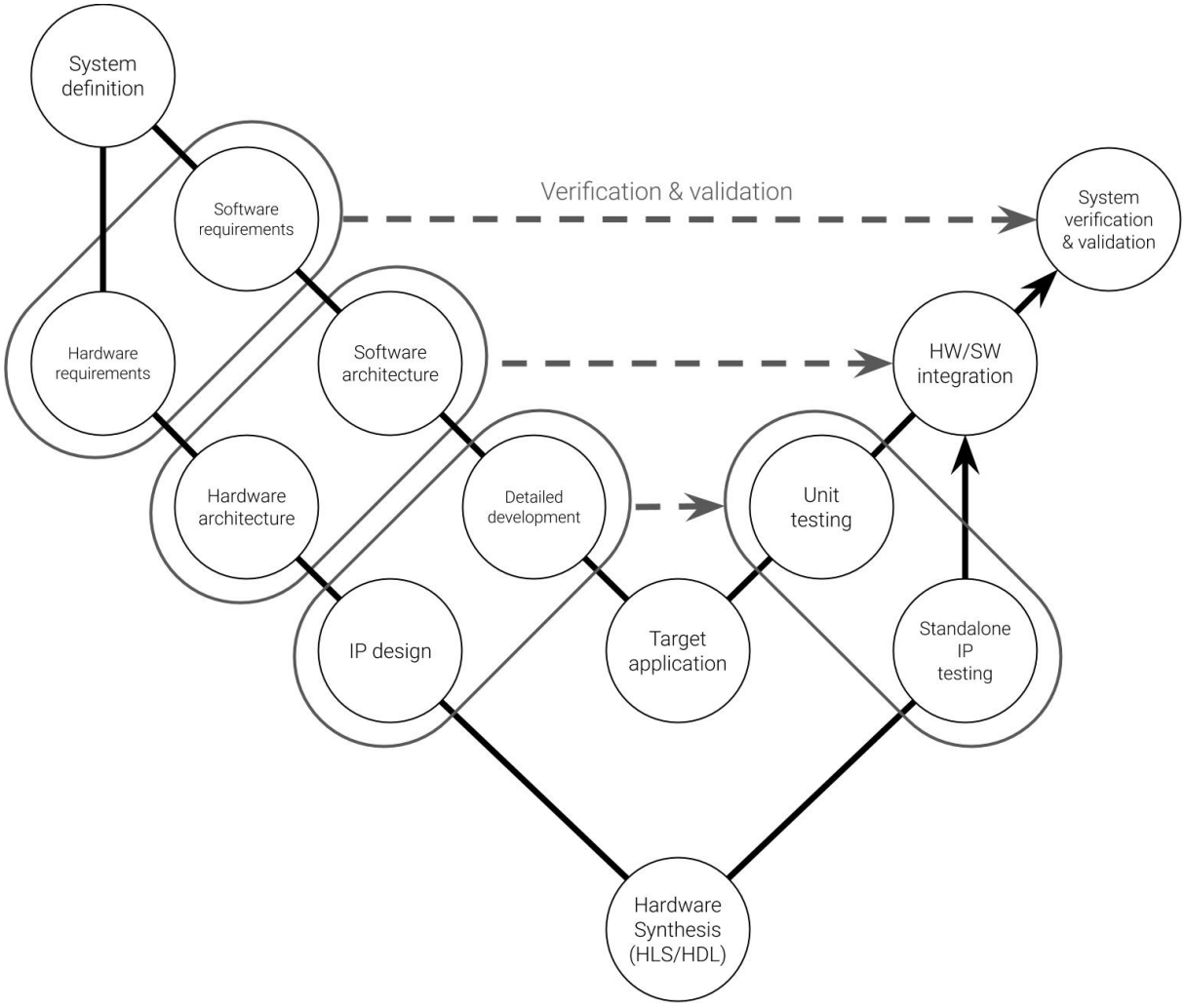


Figure 3.4 – V-model: HW/SW co-design prototyping based on hybrid CPU/FPGA platform

and hardware are not up to the system definition and requirements. Figure 3.4 represents the HW/SW co-design prototyping for hybrid CPU/FPGA-based platform but as a V-model. The system definition forks in two branches: one for software and the other for hardware. In the software branch, the software requirements refers to defining the behavior of the software and its constraints. The software architecture refers to designing the interactions between software threads. The detailed development refers to programming the functions. The target application is the compiled software adapted to the desired host. The unit testing refers to testing each software thread individually to verify and validate the detailed development. In the hardware branch, the hardware requirements refers to defining the behavior of the hardware and its constraints. The hardware architec-

ture refers to designing the interactions between hardware threads. The IP (Intellectual Property) design refers to describing the programmable logic of each hardware thread. The hardware synthesis is the use of HLS tools or/then HDL to describe the hardware architecture. The standalone IP testing is to verify the behavior of each IP individually. Finally, the HW/SW integration is the fusion of the hardware and software with their verification and validation toward the software and hardware architecture. The system verification and validation is the final test, checking that the current system is compliant with the requirements. Once this step has been cleared, the prototype can be started. The goal of this thesis is to simplify and optimize the standard design flow for embedded AI with the usage of specific constraints, dedicated hardware and automation tools.

3.4 HW/SW co-design methodology in the deep learning AI era

This section is about the first step of this work toward the final methodology. We want to explore a simple methodology for HW/SW co-design using hybrid CPU/FPGA platform and include a deep learning application for classification as a software. It is divided in two parts: 1) the update of the standard methodologies to fit our goal, 2) a first methodology with software deep learning and hardware accelerated data processing.

3.4.1 Understanding the standard workflow for deep learning algorithms

In order to take a first step toward the wanted methodology, we want to modify the standard methodology to our needs. But first, it is necessary to understand the standard workflow of a deep learning application for classification, so it can be fused with the methodology. Figure 3.5 presents a standard workflow for a pre-trained deep learning-based classifier application. There are three steps before the inference of an object cate-

gory:

1. Data processing is the action of filtering a signal (our input) in order to remove noise or irrelevant pieces of information. This step is important to reveal meaningful information used for pattern recognition. What is filtered out of the input will depend on the nature of the input data and the pieces of information required to determine a pattern.

2. Calculating the features for the deep learning inference. A feature can be defined as a measurable property of an observed phenomenon. Ideally, a feature should be unique to each category of object in order to easily determine a pattern. As it is seldom possible to determine the category of an object with only one feature, multiple features are used to find the pattern. Although those features should be independent from each other. In the case of deep learning, those features are often learnt inside the neural network. But in some cases, it is better to pre-process those features to reduce the size of the neural network architecture.

3. Inferring the object class with the learnt model. It implies that training and testing of the DL application is already done. The inference will use the value of each feature to determine the category of the input corresponding to the pattern learned.

Of those three steps, the heaviest computations are the data processing and the feature calculation. As the data processing task is making calculations from the raw input, it means this is the part with the greatest amount of information. Moreover, the performance greatly depends on how noisy the input is. The feature calculation is also resource intensive depending on the number of features and the complexity of said features. This means that those tasks are the ones we are most likely to hardware accelerate.
With the deep learning workflow in mind, Figure 3.6 presents an example of how to prototype a deep learning application on top of a HW/SW co-design platform. The main changes are: 1) the hardware architecture which is only about hardware accelerating data processing and feature calculation algorithms, 2) the DL requirements, which refers to

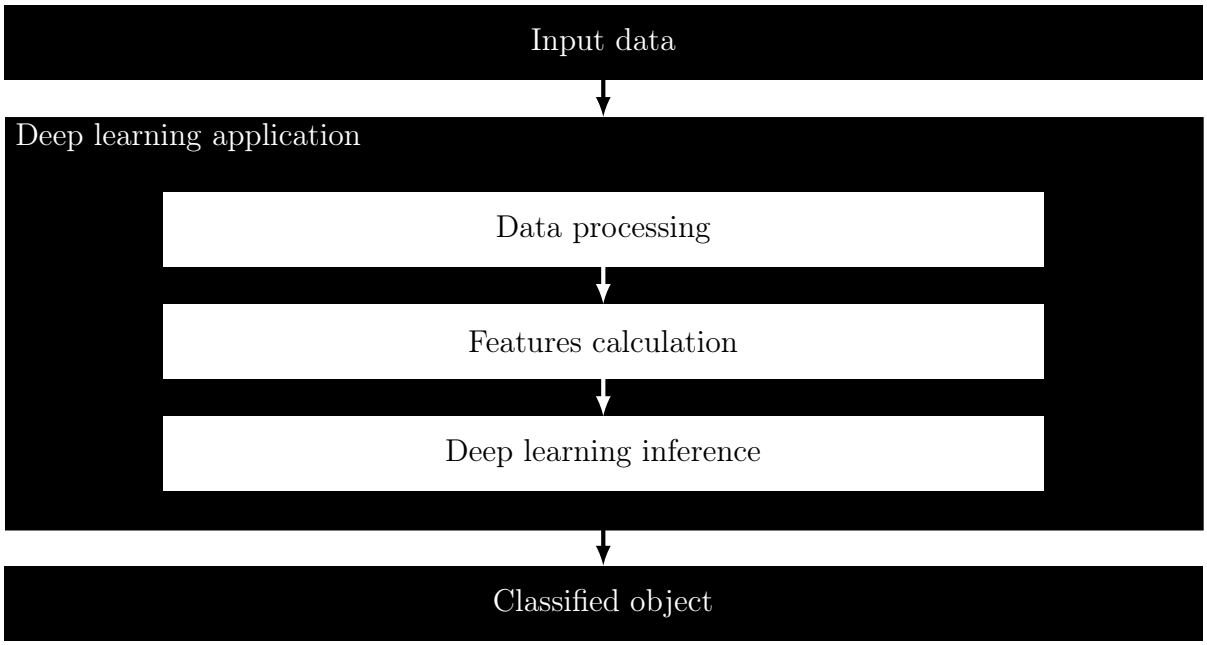


Figure 3.5 – Deep learning classification standard steps

determining the dataset to train and test the model, the algorithm to use for the learning, and the target accuracy and error, 3) the model parameters that need to be embedded either into the host executable or into the prototype hardware (File System, EPROM, DRAM, ...), so the host software can use it.

3.4.2 Exploration of deep learning inside a HW/SW co-design application

Now that we have a better understanding of the conception of a HW/SW co-design application using deep learning, lets first make a design flow in order to describe the steps before the final prototype. Our approach is oriented toward a platform-based design using a hybrid CPU/FPGA platform. Figure 3.7 shows the design flow of a co-design deep learning system where preliminary data processing tasks are hardware accelerated and the learning system is pure software. Some steps of this design flow have been automated to accelerate the development, prototyping and validation of the HW/SW co-design application, such as the configuration of the prototyped hardware co-design application [74]. There are four main steps toward the prototype phase: The **Specification & Native**

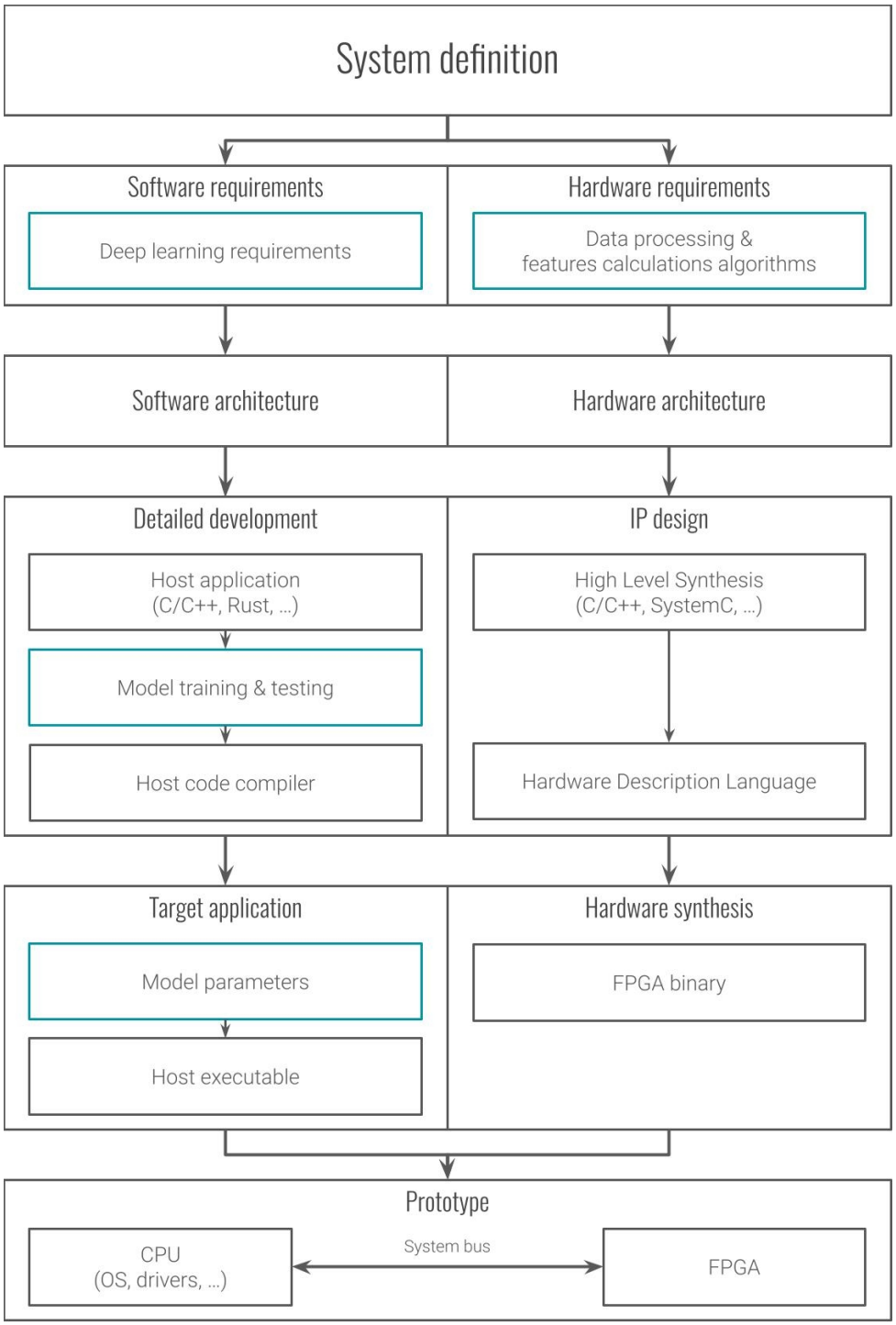


Figure 3.6 – HW/SW co-design prototyping for a hybrid CPU/FPGA-based Deep learning application. Compared to the previous methodology (Figure 3.3): the deep learning requirements are added inside the software requirements, the data processing and feature calculation are added inside the hardware requirements, the model training and testing is added between the host application and host code compiler, and the model parameters are added before the host executable.

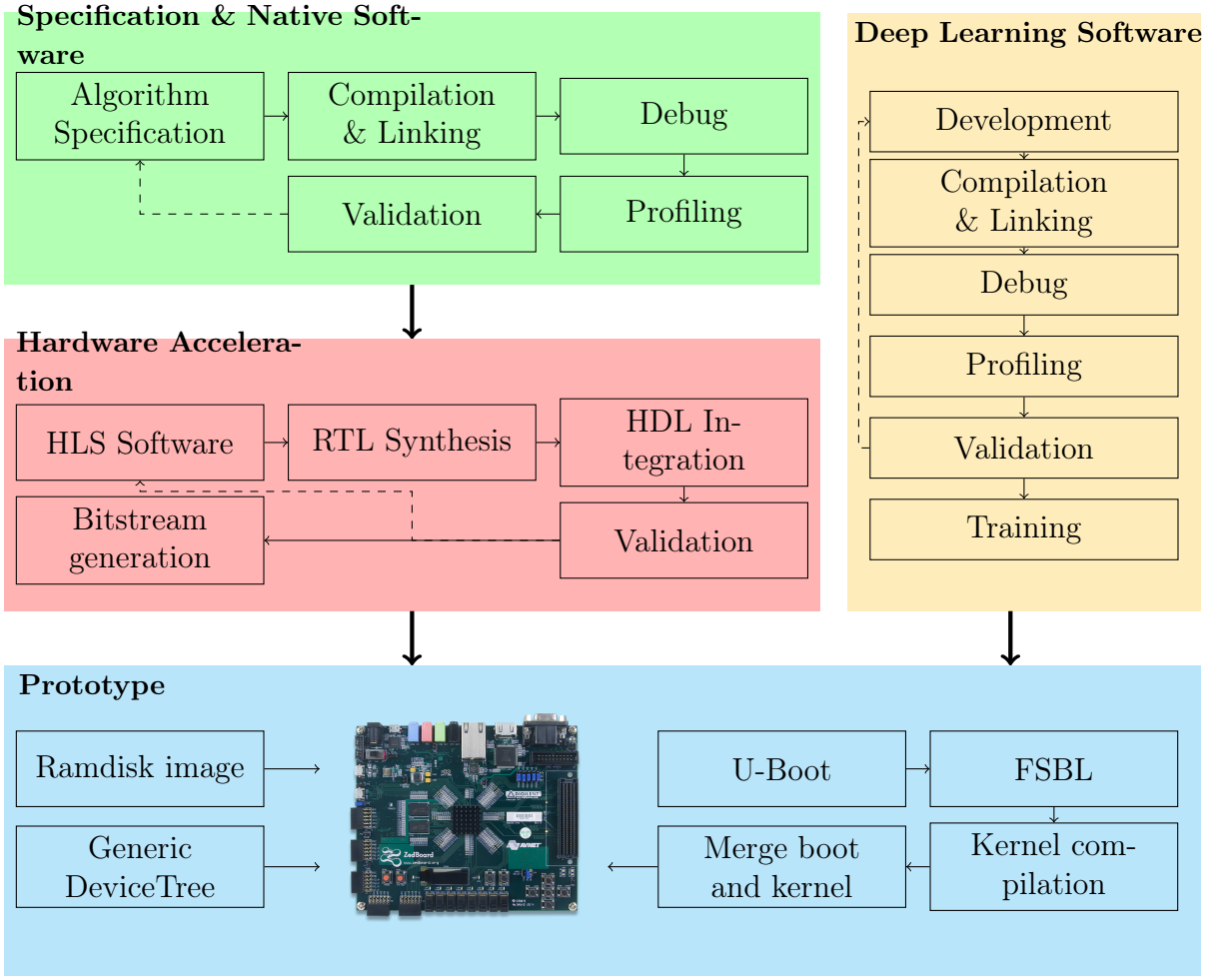


Figure 3.7 – Design flow diagram for a hybrid CPU/FPGA-based HW/SW co-design deep learning software application

Software part refers to the development of the data processing software on a native platform in order to validate it. Once the software has been profiled and its results meet requirements, the application source code is considered the input of the next step, the Hardware Acceleration. The **Hardware Acceleration** part refers to transforming the data processing software threads to hardware threads. The native software source code is to be converted to Register-Transfer Level abstraction (RTL) with a High-Level Synthesis (HLS) software, then to be implemented in any Hardware Description Language (HDL) project. If simulation results meet the requirements, the final bitstream file that came from the HDL project is deployed on the prototype. The **Deep Learning Software** part refers to the development of the learning system which represents the smartness of the

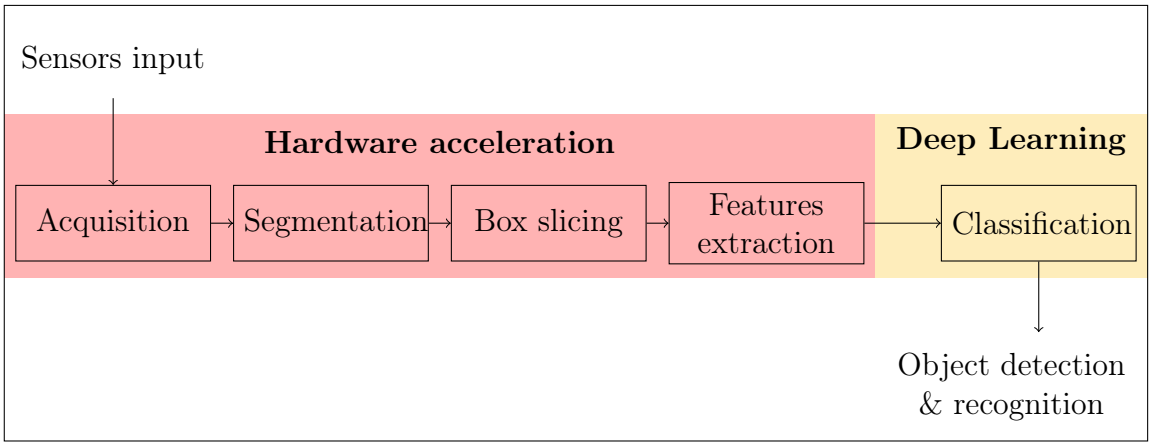


Figure 3.8 – Application Tasks Graph

system. This part does not differ from traditional embedded deep learning development. Once finished and compiled, the software is deployed on the prototype. The **Prototype** part refers to the platform on which the hardware threads and the deep learning software are deployed and tested. The prototype configuration and deployment are fully automated in our work (see Section 3.4.3). As an example of this design flow, let's imagine a simple application of pedestrian recognition using a LIDAR point cloud [75]. Figure 3.8 presents the task graph of this application, in red are the hardware threads and in yellow is the software thread. If applying the design flow, in order to design this system, we first need to define an algorithm that is validated through native software development. Once this software has been defined, we can hardware accelerate it thanks to HLS tools. Meanwhile we also need to develop the software deep learning and validate it correctly. Once everything is done, we can deploy the hardware threads as well as the deep learning software executable to the prototype.

3.4.3 Prototyping automation tools for hybrid CPU/FPGA platforms

As seen beforehand, we automated the deployment of our prototype platform in order to decrease prototyping time. We developed an automation software using *GNU make* [76] available in our git repository [74]. GNU make is an automation software using a

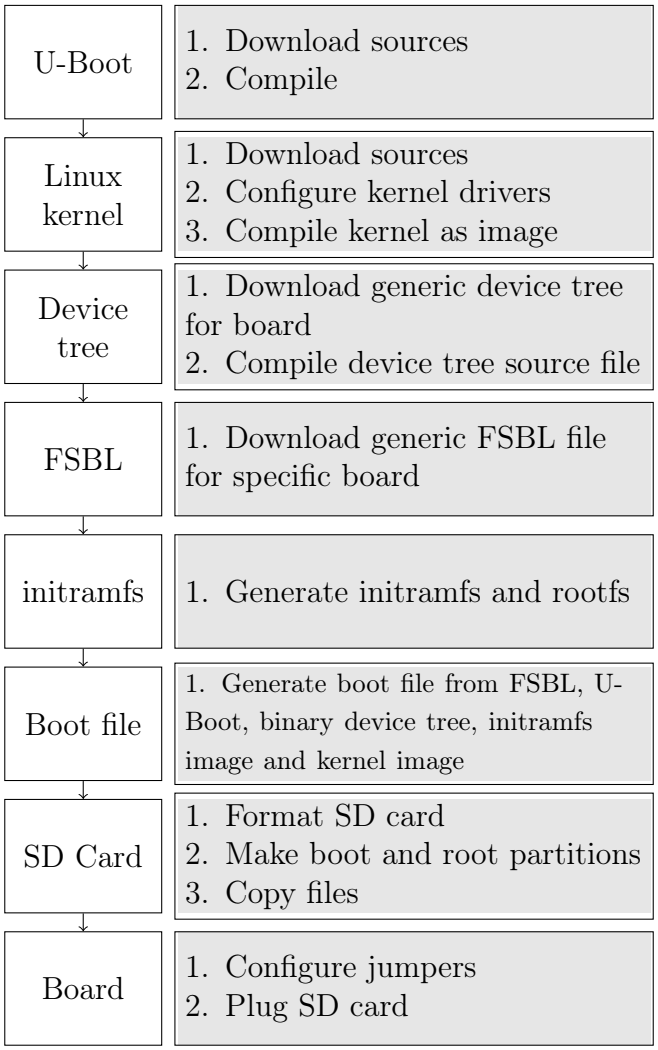


Figure 3.9 – Prototype automated deployment

Makefile to perform various actions. Figure 3.9 shows the different steps of our automation software to deploy the prototype. It automates the build of the U-Boot and Linux kernel by downloading the sources from their respective git repository and compiling them. U-boot (Universal Boot Loader) is an open-source boot loader for embedded device which contains instructions to boot the available operating system kernel. In our case, the kernel is a Linux one. The Linux kernel is a free and open-source operating system kernel. It is often deployed in embedded devices because of how light it is. The purpose of this kernel is to run an operating system which will execute our NNP control software. A generic device tree is available inside our repository in order to be compiled for the board beforehand. For now, only a device tree for a Zedboard [77] has been made, because

it was the development board used during this work. A device tree is a data structure describing the different hardware components available on our platform. Its purpose is to enable the operating system kernel to use and manage those components (e.g. FPGA and DRAM mostly in our case). A FSBL (First Stage Boot Loader) is also available, and already compiled, inside our repository, but only a ZedBoard one. A FSBL is a specific file responsible for loading the FPGA bitstream and configuring the Processing System (PS) at boot time. There also is an already made initramfs (initial RAM file system) image with necessary drivers and libraries for the OS and control software, in order to get access to a read-only file system inside the DRAM. The initramfs is an image of a temporary root file system loaded into memory (DRAM) that acts as a read-only file system for the operating system. Then a boot file is generated from the FSBL, U-Boot file, binary device tree, initramfs and kernel image. This boot file can be considered as a compressed and executable version of the necessary files needed for the booting of the kernel and the operating system. The FPGA bitstream can also be added to the boot file in order to be loaded at boot by the FSBL. Finally, all files are placed inside a folder to be copied into the SD card. The SD card needs to be correctly formated with a NTFS (NT File System) boot partition for files generated by our script, and an EXT4 (4th EXTended filesystem) partition for persistent storage. A script available inside the initramfs mounts the EXT4 partition to the */root* folder at OS startup. This means the embedded deep learning software is copied inside the EXT4 partition to be executed. It also allows a better control of continuous integration/continuous deployment (CI/CD), because it allows us to deploy our updated software through the network on which the prototype was connected. Moreover, we also succeeded in deploying the FPGA bitstream over a network thanks to the */dev/xdevcfg* channel, which allowed us to reprogram the FPGA at runtime by sending the bitstream to this FPGA device file.

3.5 Proposed embedded Deep Learning methodology around a FPGA-based Neural Network Processor

In this section, we propose an embedded DL based methodology for FPGA-based CPSs platform design using a hardware NNP. Furthermore, the automation of the methodology is tackled to determine how to decrease prototyping time.

As seen in Chapter 2, there are already several proposed neural network processor architectures and implementations. So in a way, developing another NNP would not look like a necessary part, at first glance, when considering it as a tool to be used. But in order to develop a new methodology, we need to have a better understanding of NNP design to find its advantages and constraints. The goals behind building our own NNP were to have a better understanding of NNP design and the constraints associated to it, to hone our skills toward hardware acceleration and be aware of all necessary hidden steps when hardware accelerating an algorithm. Thus, even if the benefits seem limited at first, the knowledge gained from this NNP design and implementation was a necessary part of this thesis.

3.5.1 The methodology design flow

The first step of our methodology is the definition of the system with its requirements and architecture. Then, different software algorithms are designed for data processing and DL. Those algorithms are hardware accelerated using a High Level Synthesis (HLS) software tool, or are described from scratch with a Hardware Description Language (HDL). Finally, the hardware accelerators (hardware threads) are synthesized and uploaded on a hardware platform to be tested. Considering those steps, several hidden tasks are present from data processing to data management, and the configuration of the hardware platform. The goal of our methodology is to mitigate those hidden tasks, either with simplification or automation. Our approach toward making smart application for CPSs is built around a FPGA-based DL methodology using a NNP. This methodology

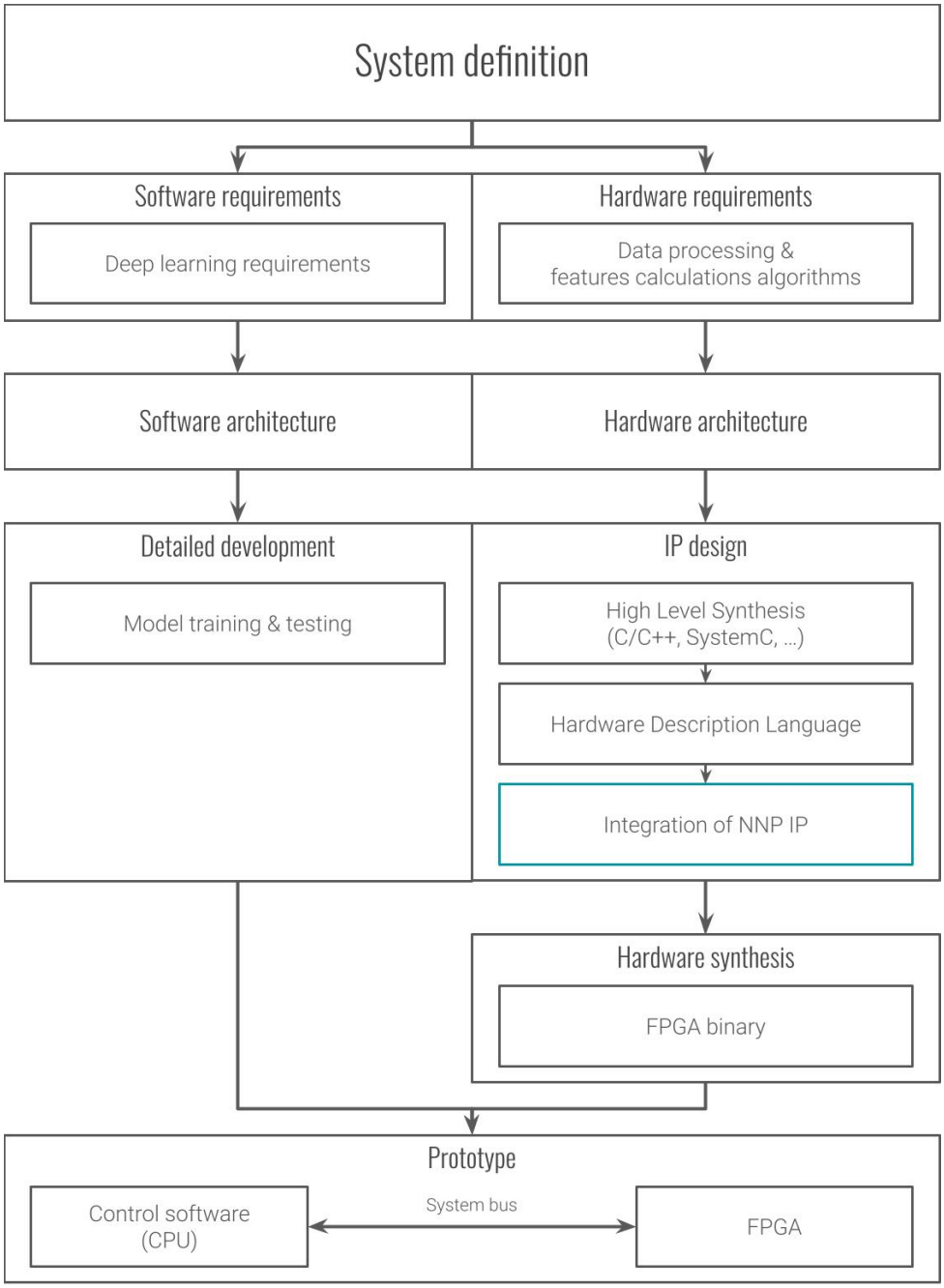


Figure 3.10 – Our methodology implementation. The differences with previous methodology (Figure 3.6) are: the addition of the NNP IP integration and the removal of a host software development and compilation

is divided in four parts (Figure 3.1): hardware platform, hardware acceleration, embedded processing and DL software. The transition between each part is as follows: the DL weight matrices are extracted and transferred to the hardware platform, the embedded processing is hardware accelerated. The development and use of the NNP as a part of the

methodology is an important step, in order to handle the DL processing. The description of the methodology is done with a top to bottom approach by disassembling the different tasks to make a prototype and explaining our design process to share our experiences. A design flow detailing the approach of our proposed methodology is presented in Figure 5.3. It shows the different steps of the four parts of the methodology, and indicates how the parts are connected to each other. Our proposed methodology for embedded DL differs from standard ones because of its constraints. The differences mostly reside in the integration of the NNP and its associated configuration software. Figure 3.10 illustrates the implementation of our methodology. Our DL is based on the NNP that is implemented firstly in software and then migrated to a hardware accelerator using HLS tools. The NNP is synthesized and integrated within the FPGA bitstream programming file. The HW/SW architecture programmed on the top of the hybrid CPU/FPGA platform will use the extracted weights, which come from the offline DL training. In order to present the methodology in detail, lets split it into the four designed parts: Hardware platform, hardware acceleration, embedded processing, deep learning software.

3.5.1.1 Hardware prototype

One contribution of this work is to develop a real prototype, by acquiring physical data from the real world and processing them, in order to obtain an accurate analysis. This analysis is done by transforming physical data into specific features that are used with a DL approach to be classified. The hardware platform needs to host an embedded processing application to transform physical data. Then, a DL application (NNP) will classify that data. Configuring and deploying a prototype was simplified using the developed automation tools [74]. Also, the usage of a bootloader (U-Boot) and a First Stage Bootloader (FSBL) helped to get the first stages of the platform. Moreover, platform operating system configuration is simplified using a Unix kernel with its initial ramdisk, preconfigured system files, and a generic devicetree to get access to all the components on the hardware platform. In the case of an FPGA-based platform, hardware threads

communicate with each other through data transmission over the system bus. Lets first consider the embedded processing thread: it needs to directly acquire data from sensors, and prepare them for the NNP. The NNP will classify the adjusted input received from the embedded processing application. With hindsight, the main goal for the embedded processing is to filter the signals coming from the sensors in order to reduce the processing needed by the NNP, because the NNP's main constraint is data management of the internal states, such as weight matrices and hidden unit communication. Thus, doing more data processing means we can use a smaller deep neural network model and speed up the NNP calculations. Finally, control software is needed to reconfigure the NNP. This control software will load weight matrices (which are considered binary files) into the external DRAM so the NNP can access it through Direct Memory Access (DMA) transfer. The control software will also initialize the DMA registers to read and write the correct DRAM addresses corresponding to the NNP data.

3.5.1.2 Hardware acceleration

Embedded processing is built as an FPGA hardware thread. Hardware accelerators development is simplified using HLS software tools or HDL, but data management is still a sensitive part of the development because of the FPGA's constraints. In this methodology, we consider that data is received and transmitted as a FIFO queue (First In, First Out) in order to simplify data management, even if it may mean extra calculation for processing tasks. This leads to the embedded processing application receiving data from sensors and directly transmitting the processed information to the NNP. It also means that the NNP is receiving its data (input vector and weight matrices) as a FIFO queue and needs to compute the classification as data transmission progress. The embedded processing algorithms need to be tweaked in order to compute FIFO transmitted data, and use as little internal cache as possible. We mainly consider the usage of a HLS software to synthesize NNP modules from a High Level Language (HLL) to Register-Transfer Level (RTL). The generated HDL files are imported to the whole project to generate a

FPGA bitstream.

3.5.1.3 Embedded processing

Data perceived by the CPS should be processed so the NNP can use it correctly. It is necessary for two main reasons: 1) the data needs to be transformed for the NN to handle it, 2) to decrease the size of the neural network by computing some features beforehand. The main constraint is data management. With data coming as a FIFO queue, most algorithms need to be redesigned in order to use as little memory cache (BRAM) as possible.

3.5.1.4 Deep learning software

The common method to make a DL application is by using specific tools to train and test NN architectures with a dataset. In this work, we consider that the NN architecture is already defined. We also consider the offline training as done. The weights are then extracted to be used directly by the NNP embedded in the hardware platform. In this methodology, we consider weights extracted as a binary file containing the weight matrices between all layers.

3.5.2 Toward the automation of the design flow

Automating the methodology is an important issue toward a decreasing the time-to-prototype. Moreover, automation can help reduce the required skills necessary for this kind of tasks and thus make it more accessible. We believe that each part of the design flow for the methodology (Figure 3.7) can be more or less automated. The deep learning software is already simplified with the usage of either programming tools or visual editors to develop a deep learning architecture. The embedded processing part is a complicated process to automate, because it depends on the data processing algorithm's complexity, required performances, and the HLS software's user friendliness. But the reusability of IP for embedded processing might be a good start toward automation, because it means

that there is no need for hardware development. Moreover, with the Vivado Design Suite [47], everything can be automated thanks to scripts that are interpreted by the Vivado software, which means that the integration of the embedded processing hardware threads with the NNP can be automated. The hardware platform part can be automated, either in a generic way or for specific configuration, thanks to modern tools (U-Boot, Linux kernel, Devicetree, ...).

3.6 Challenges of deep learning in hybrid CPU/FPGA-based CPS

When creating this methodology and experimenting with it, we encountered many challenges and difficulties because of the hybrid CPU/FPGA-based CPS context. The context we are talking about in this work mainly concerns the design of a deep learning application on a hybrid CPU/FPGA platform using a hardware neural network accelerator and hardware threads for data processing.

The first challenge pertains to the design and implementation of hardware threads, from HDL to RTL. Creating and integrating hardware processes is one of the hardest tasks in our design flow, especially when using an HDL. This is one of the reasons we used SystemC models to simulate the different hardware threads on a software platform and verify their behavior to ease the development. Then we used an HLS tool to convert SystemC models to HDL ones, which sped up the prototyping process as well. The only drawback of an HLS tool is whenever the optimization part comes. In the context of our methodology, editing an HLS-generated HDL code and optimizing the RTL synthesis is currently not so user friendly, and needs expert-level skills. By this, we do not mean it is an impossible task, nevertheless the needed skills to optimize HLS-generated hardware threads are separated from the skills needed for AI design. Which means, in the context of our methodology, that the optimization of hardware threads needs specialized tools to elevate low-level skills to high-level ones.

Another challenge is from the data transfer between the CPU and the FPGA. In our context, we need a way to transfer data back and forth between the CPU and FPGA while the CPU is not focused on data transfer instructions. This is mainly to enable the CPU to control different hardware threads while not waiting for data transfer. For this reason we chose a DMA-oriented approach. As known with DMA-based systems, once the CPU has transferred the initial data to the DRAM, it can simply watch the different DMA statuses and do something else while the data is being transferred between the FPGA and the DRAM. But using a DMA approach also adds more challenges, such as the system bus communication bottleneck due to the maximum data bandwidth and throughput. This is particularly true in the context of deep learning algorithms, where modern parameter matrices are in the MB scale because of the size of neural networks. In the case of our experimentation with a DMA, using an AMBA system bus of 32-bit running at 100 MHz, which meant that transferring one 32-bit floating point would take around 10 ns. Hence, in the case of a neural network with 1 million 32-bit floating point parameters, 10 ms are dedicated to the data transfer. And this time will increase if the number of parameters increases. There are currently three solutions for this challenge: 1) specific hardware with high-speed data transfer, 2) reducing the number of parameters in the neural network architecture, 3) compressing the parameters as 16-bit fixed point integers or even binary.

One more challenge comes from the HW/SW integration. The main problem in the integration phase is the debugging. While it is easy to debug software, it is far more complex to debug a hardware thread. The use of a JTAG (Joint Test Action Group) connection allowed us to understand what is going on inside the FPGA with our hardware threads and is a debugging tool, but it requires a JTAG port and the use of a JTAG library, either with HLS or HDL. But JTAG has some limitations when debugging complex signals and behaviors. Another solution we used for debugging was the creation of our own debugger IP which was connected to a DMA for us to be able to send ASCII characters, so we could print some kind of debugging logs from our hardware threads. While this

practice did not involve JTAG, it was longer to make because of the need to change our hardware designs.

The last challenge we encountered was automation. Automating the prototyping phase can be difficult because of the numerous constructors and standards. This is why we tried to use open source technologies and generic configuration. But this also comes with some disadvantages related to the optimization of our platform. While going bare-metal can be the best way for a performance-oriented implementation, it also comes with the setback of a longer prototyping time. Which is why we approach the automation with a generic method, but at the loss of performances. In our vision, automation is a really important part of this thesis because it brings hardware accelerated solutions to high-level skilled users. Our final goal would be a full automation of the methodology so that AI designers could simply deploy their prototype on a "ready-to-go" hardware accelerated platform.

3.7 Conclusion

We presented our work toward an embedded DL methodology for hybrid CPU/FPGA-based CPSs platform design using a hardware NNP. The standard process for HW/SW co-design prototyping is explained, then a first methodology to develop HW/SW co-design applications using software deep learning is introduced. Finally, the FPGA-based DL methodology using a NNP as its core is explained, in addition to a discussion toward its automation. This new methodology shows another way to develop and deploy DL applications on FPGA platforms. But to make this methodology completely functional, the NNP needs to be designed following the methodology requirements.

Chapter 4

A hardware Neural Network Processor: core of the methodology

Contents

4.1 Introduction 58

4.2 Neural Network Processor 58

4.2.1 Definition 58

4.2.2 Design and Architecture 59

4.2.3 Validation of the NNP 67

4.3 NNP integration into the full prototype 69

4.3.1 Embedded processing to improve performance 69

4.3.2 Control software for reconfiguration purposes 70

4.3.3 Workflow of an application using the NNP 74

4.4 Experimentation and results 75

4.5 Conclusion 79

4.1 Introduction

In the previous chapter, we introduced our DL-based methodology using a CPU/FPGA hybrid platform. However, to easily prototype an application using this methodology, a Neural Network Processor (NNP) is needed. The NNP is designed to simplify the integration of DL in embedded CPS applications. The purpose is to create some kind of single-purpose processor dedicated to the calculation of deep learning feed-forward algorithms. To keep this processor simple, some constraints were defined: process the simplest Neural Network (NN) architecture (fully connected NN without bias) with as few activation functions as possible, process any number of sequential layers independently of their depth and width, and be re-configurable at runtime. A no bias architecture is chosen here because bias calculation needs more computational power, time and FPGA resources.

4.2 Neural Network Processor

4.2.1 Definition

A NNP can be qualified as an Application Specific Instruction Processor (ASIP). Its purpose is to compute neural network-based deep learning through processor instructions. The most needed instructions are the weight matrices of the trained NN, its network topology and activation functions for each layer. With this information, it should be enough for computation once an input is loaded into the NNP. But it seldom is that a NNP can compute any type of NN. For performance purposes, it is sometimes better to limit it to a special case, like Dense Neural Network (DNN) in our case, or as often seen in the literature, Convolutional Neural Network (CNN). This can be seen as one of the limitation of NNP, but since it is mostly application specific, it is more of a constraints set beforehand.

Now in terms of architecture, a NNP has several parts that are necessary. It needs at

least three buffers for the input, output and weight matrices. Those buffers can either be directly in the FPGA cache, or in the DRAM and accessed through DMA (Direct Memory Access). One or more module dedicated to NN calculation is also needed, it is often called Processing Element (PE) or Neuron Processing Unit (NPU) in this work. The purpose of PE is to access the input and weights buffer to make the layer calculations or part of the layer calculations [78]. PE should either output its results in a specific buffer or directly in the output buffer which will be swapped with the input buffer to compute the next layer. The PE communication network can be either directly connected the system bus, or as an access point in a Network on Chip (NoC) [36]. Lastly, a controller component is needed in order to order the instructions between the different elements, like instructing the PE what to process, making the swap between buffers, scheduling computation, etc.

4.2.2 Design and Architecture

In order to understand how to design a NNP, we first need to understand how a feed-forward neural network without biases is computed [6], especially DNNs in our case. A DNN can be defined as layers of neurons where each neuron in a layer is connected to all other neurons in the next layer. A neuron, also called node, is defined in Figure 4.1 and with the calculation model being $y = f(\sum_i x_i \cdot w_i)$, where x_i are the inputs of the neuron, w_i are the weights for each path to the neuron (sometimes called "synapse"), $f()$ is the activation function of the neuron which is a specific function associated to the neuron, and y is the output of the neuron.

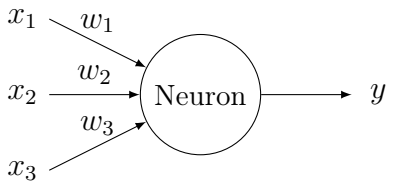


Figure 4.1 – Neuron in a neural network

Now that we defined a neuron, Figure 4.2 represents a multi-layer perceptron without biases, meaning a DNN, also called fully-connected neural network. There are three

types of layers: input layer, hidden layers and output layer. The input layer is the first layer which receives external data. The output layer is the last layer from which results the neural network calculation. And in-between those two layers are zero or more hidden layers which make calculations from the input layer to the output layers. Each neuron in a layer has the same activation function, but it is not always the same function in each layer. The input layer does not have an activation function and just transmits external data to the next layer. A layer can be mathematically modeled as $Y_n^{L+1} = f_{L+1}(W_{mn}^{L/L+1} \cdot Y_m^L)$ where Y_n^{L+1} is the matrix of size $n \times 1$ containing all neurons output from the $L+1$ layer, Y_m^L is the matrix of size $m \times 1$ containing neurons output from the L layer, $W_{mn}^{L/L+1}$ is the matrix of size $m \times n$ containing the weights between the layer L and $L+1$, finally $f()$ is the activation function of the layer $L+1$ applied to all elements of the dot product between $W_{mn}^{L/L+1}$ and Y_m^L e.g. if $C_n = W_{mn}^{L/L+1} \cdot Y_m^L$ then $f(C_n) = [f(c_0), \ldots, f(c_n)]$

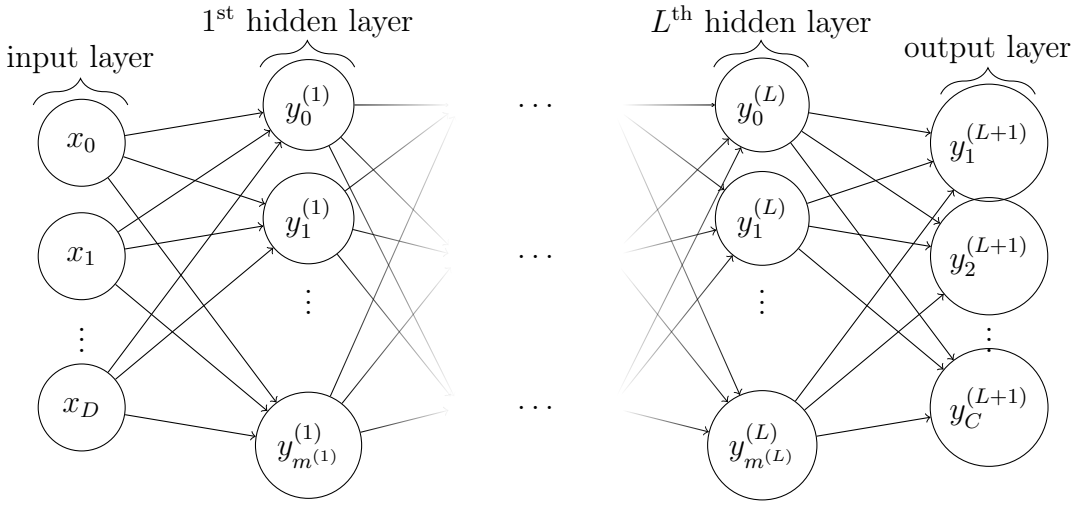
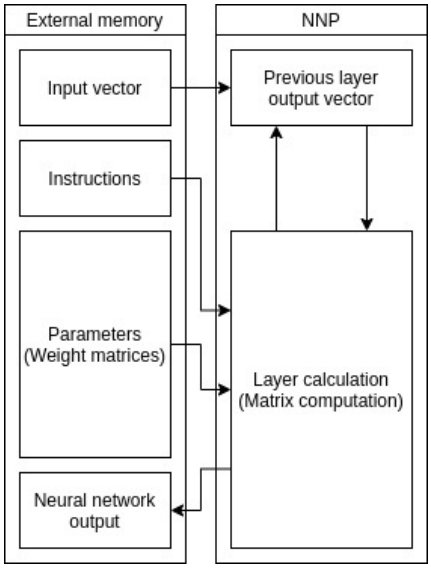


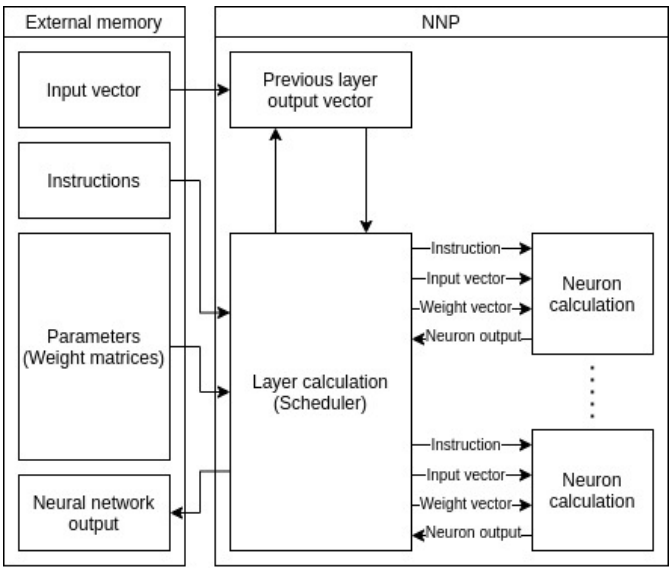
Figure 4.2 – Dense Neural Network (DNN)

Computing a DNN is mainly about matrix calculation which means two problems arise from the implementation: matrix calculation and data management. For the calculation part, two approaches are possible: calculating layer per layer, or each neuron per layer. Whole layer calculation is interesting in the case of optimized hardware for matrices calculation, but calculating each neuron individually can also be interesting when using parallel operations or compressed neural networks such as binary neural networks.

But the biggest issue is about data management, nowadays neural networks have millions of parameters which make weight matrices using hundreds of megabytes[9]. Even if optimizing neural networks by reducing the number of parameters, the size is still important enough to need to carefully design data management. First, we need to define where data is stored, meaning weight matrices, input vector and instructions (layer size and activation function). In most cases, parameters are loaded from an external memory component such as DRAM, EEPROM, etc. There is several reasons for this: 1) because of the limited FPGA memory cache compared to the size of today's weight matrices if not compressed, 2) to allow different configurations by updating the weight matrices and instructions, 3) to have an interface between the FPGA and other components wanting to use the NNP. Now that we know where data is stored and we know it is not stored inside the FPGA memory cache, we need to determine which data is transmitted to avoid congestion in the system bus. The main data is parameters inside the weight matrices, this data is mandatory for the feed-forward calculation, and the instructions, layer size and activation function. In the case of layer calculation, it is only needed to convey the weight matrix between the current layer and the next layer with the input vector of the



(a) NNP calculating layers per layers



(b) NNP calculating each neuron of a layer

Figure 4.3 – Data and tasks representations of a NNP

next layer (which is the output vector of the current layer). Because of the often large number of parameters, it is needed to take into account the data bus bandwidth to correctly estimate the performances. But in the case of individual neuron calculation, the weight matrix might also need to be sorted considering the scheduling of the processing engines in charge of the neuron's output calculation. Instructions are lightweight directives to the processing engines in order to read the correct weight matrix and input vector. Moreover we have volatile information such as the matrix resulting of the calculation of a layer which is needed to calculate the next layer. In the case of the last layer, this matrix is the NN output. We need this layer to be quickly read and written for the NPU performance, so in order to optimize data transmission, each hidden layer output needs to be kept in the FPGA cache to be used as inputs for the next layer. But in the case of the last layer, it might be useful to convey the output vector to the external memory so other components can have access to the results. Figure 4.3 represents the architecture of a NNP when calculating whole layers as matrix dot products, or when calculating a layer neuron per neuron. It represents all data needed for the calculations and the interactions between the different tasks and data. Now that we have in mind the data management and calculation inside the NNP, we need to determine the data precision.

4.2.2.1 Data precision and hardware performances

Data precision can really influence the performance of the NNP as well as the resources used, but it will also reduce the accuracy of the processed neural network [79]. Floating points are often used in NN because of their precision, but it requires a lot of hardware resources to process them [80][81][82]. Fixed points are less precise than floating points, but are better suited for FPGA since they use far less resources, which means there is more space for parallelism [83]. Finally, the most compressed data type for NN is binarized NN, which is really suited for FPGA using logic gates, but this method implies the most precision loss [84][85]. Another way to deduce the correct data precision would be to take a fully trained NN using floating-point or fixed-point arithmetic and reducing the

number of bits used until the behavior of the NN totally diverges from the original model [86]. But this method only works for a specific NN model, which means, in our context, that the NN accelerator would need a way to adapt the data precision at run time. Data precision is a tough choice depending on the NNP application. Precision is an important part in decision making in the case of CPSs, but performance and real-time calculation might also be critical in some applications.

4.2.2.2 Hardware architecture design and constraints

First, we will detail the hardware architecture of our NNP and how it calculates layers and neurons. Figure 4.4 represents the different part of the processor and the communication interfaces in-between. There are four communication channels with the NNP for different data: the input vector which comes directly from another FPGA component (the embedded processing), the instructions and weight matrices are loaded from the external DRAM, and the output vector is loaded into the external DRAM. The data type used for this prototype was 32-bit floating points, in order to easily use weights from an external deep learning training software. In the context of our prototype, data between the external DRAM and the FPGA and the is using the same system bus, which is a 32-bit AMBA AXI bus.

Scheduler module The scheduler module loads all instructions from the DRAM to know how many weights and inputs should be loaded and the activation function to be used for each layer. Each instruction represents information about one layer and is coded by a 64-bit word containing three pieces of information: the number of neurons in the previous layer (30 bits), the number of neurons in this layer (30 bits) and the activation function of this layer (4 bits). As seen in Table 4.1.

Data type	Previous layer size	Current layer size	Activation
Bits	30 bits	30 bits	4 bits

Table 4.1 – Number of bits for each piece of data in instruction word

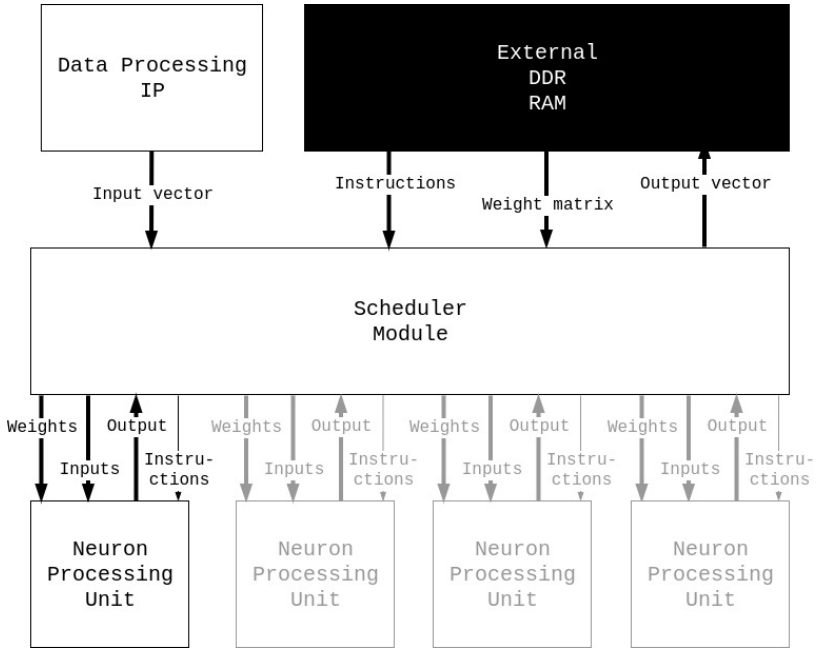


Figure 4.4 – NNP data flow with 4 cores

Algorithm 4.1: Scheduler module algorithm

Data: Input vector, Instructions, Weight matrix
Result: Output vector
instructionsCache ← read all from instructions;
inputVectorCache ← read all from input vector;
repeat
  processedNeurons ← 0;
  while *all neurons are not processed* **do**
    foreach *core* **do**
      write instructions to core;
    foreach *input* ***in*** *inputVectorCache* **do**
      foreach *core* **do**
        write *input* to core;
        weight ← read one from weight matrix;
        write *weight* to core;
    foreach *core* **do**
      outputVectorCache ← read output from core;
      processedNeurons increment by 1;
  inputVectorCache ← *outputVectorCache*;
until *all instructions are processed*;
write *outputVectorCache* to output vector;

Once all instructions are loaded, the input vector is read from the embedded processing IP (Intellectual Property) into a local cache and processing is started. For each layer,

each neuron is represented by a neuron processing unit, also called a core, and thus each neuron is computed individually. The scheduler starts a processing unit by sending specific instructions to it, which are not the same as the ones that the scheduler module receives. Then each input and weight connected to the specific computed neuron is sent. Once the processing unit has finished the calculation, the output is returned to the scheduler, which stores it in the local cache to be used for the next layer. Once all neurons of the layer are computed, the output vector is used as the input vector of the next layer and the process starts again. Once the last layer is reached, the output vector of the NNP is written to the external DRAM. Algorithm 4.1 describes in a shorter way the scheduler module process.

Name	BRAM_18K	DSP48E	FF	LUT
Utilization	260	6	4462	4867

Table 4.2 – Hardware resource utilization of the scheduler module with a 10 ns clock target

Because of the size of the number of neurons in the scheduler module instructions (30 bits), a layer should be able to have over 1 billion nodes. But there is a hard limit inside the scheduler module cache for resource utilization purposes, which means a layer cannot be larger than **65,536** nodes. The number of instructions that can be loaded in the scheduler is set to **512**, which makes the instructions buffer size **4,096** bytes. Thus, the scheduler module uses at least **528,384** bytes of FPGA memory cache. The resources used for the hardware scheduler component is showed in Table 4.2.

Neuron processing unit Each neuron processing unit calculates one neuron at a time. It takes instructions from the scheduler module, each instruction is a 34-bit word containing two pieces of information: the number of inputs and weights (30 bits) and the activation function to be used (4 bits). When an instruction is received, the processing engine module starts listening to weights and inputs. Each time a pair of inputs and weights are received, they are multiplied and summed to previous results. Once all inputs and weights for one neuron are received, the activation function is calculated and sent

to the output. Algorithm 4.2 describes in a shorter way the processing engine module process.

Algorithm 4.2: Neuron processing unit algorithm

Data: Weights, Inputs, Instructions
Result: Output
instructionCache ← read one from instruction;
sum ← 0;
repeat
  input ← read weight from scheduler;
  weight ← read weight from scheduler;
  sum ← *sum* + *input* ∗ *weight*;
until *all weights and input are received*;
`/* activation function is defined by instruction */`
output ← *activationFunction*(*sum*);
write *output* to scheduler;

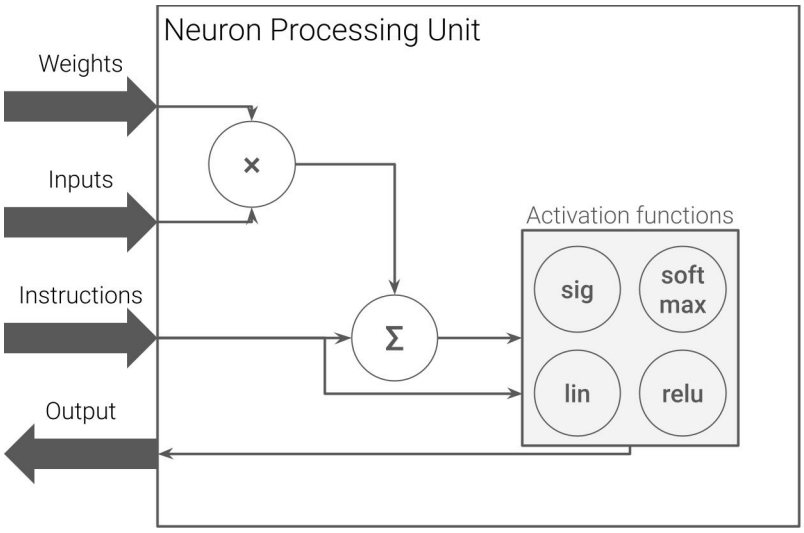


Figure 4.5 – Neuron processing unit architecture. The weights, inputs and output arrows represents 32-bit floating points.

As said before, the activation functions are limited to four activation functions: ***relu***, ***linear***, ***sigmoid*** and ***softmax***. ***Relu***, ***linear*** and ***sigmoid*** activation functions are computed directly inside the neuron processing unit. But in the case of the ***softmax*** activation function, the exponential part is done in the neuron processing unit, and the division by the sum of the output vector is done inside the scheduler module, because only the scheduler has access to the whole output vector. Figure 4.5 present the architecture

of a neuron processing unit. All the source code for the NNP is written in SystemC and is available online [87]. A testbench is available to load datasets and neural network models. The synthesis for each of those components is done with Vivado HLS [47]. The resources used for the hardware neuron processing component is showed in Table 4.3.

Name	BRAM_18K	DSP48E	FF	LUT
Utilization	0	48	3623	6115

Table 4.3 – Hardware resource utilization of one neuron processing unit module with a 10 ns clock target

4.2.2.3 Can DNN do the job?

DNN, specifically fully-connected NN in this work, is still used for a broad type of applications; either as the only type of layers or as a part of the network (e.g. CNN with fully-connected NN as last layers for classification purposes). But above all, DNN is one of the most hardware-friendly type of NN. Even though DNN are not as powerful as CNN for feature extractions, they can still be efficient to a certain degree [88]. This main difference comes from the way a DNN perceives input compared to CNN, but it can be improved with the use of data processing.

4.2.3 Validation of the NNP

4.2.3.1 Software simulation with SystemC

In order to develop the NNP, a SystemC model is first made with the code available in a git repository [87], as well as some parts of the source code in the Appendix A. The SystemC code is then transformed to HDL thanks to Vivado HLS software [47]. Two external libraries were used as cited in the repository: the CNPY library is a C++ library to read Numpy files and the TQDM library is C++ library, coming from the Python TQDM library, to display progress bar in a CLI (Command Line Interface) environment. The repository is seperated in 4 folders: *headers*, *sources*, *models* and *testbench*. The *headers* and *sources* folders contains the SystemC code for each component of the NNP

(scheduler and neuron processing unit), the top module is only necessary for the testbench, but is not used for the hardware creation with Vivado HLS. The *models* folder contains datasets and trained neural networks for the testbench to use with a compressed Numpy format [89]. The datasets available are Cifar10, Cifar100 [90], MNIST [91], SUOD [92] [93] and a simple XOR neural net. The *testbench* folder contains a SystemC main file whose purpose is to test the simulated hardware neural network and displays its final accuracy to compare it to training software such as Tensorflow. For now, lets focus on the hardware NNP SystemC components in the *sources* folder: the scheduler and the neuron processing unit. Those two files contain the source code used by the HLS software to create the HDL code for the hardware NNP. Their code is basically the SystemC equivalent to Algorithms 4.1 and 4.2. But it is encapsulated in an infinite loop to simulate their permanent processing. There are also some "pragma" macro used for the HLS software configuration. Furthermore, to emulate the system bus inside the FPGA, in our case AXI4Stream channels, the *sc_fifo* class is used to simulate a FIFO communication channel. Now, lets talk about the testbench. Its purpose is to load a model (dataset and trained neural network) to test its accuracy with the simulated NNP. First, the dataset and NN layer files are loaded to generate the instructions for the NNP. Then the NN weights are loaded inside the FIFO. Finally, we loop for each sample in the dataset in order to load the input, run the SystemC NNP and test the resulting classification. Once all samples processed, the accuracy is calculating. Each NN model was trained beforehand with Tensorflow [94] and was then stored in a compressed Numpy file along with the testing samples of the dataset. Those files are the ones in the *models* folder.

4.2.3.2 HDL Synthesis

Once the NNP simulation is working and the testbench results are OK, each component is converted to HDL code with Vivado HLS directly from their SystemC source code. Once the HDL has been generated, we then use the Vivado tool (Figure 4.6) to make the FPGA

architecture and generate the bitstream. The bitstream is then deployed to the FPGA, either with a hardware storage such as a SD Card or by sending it through a network access.

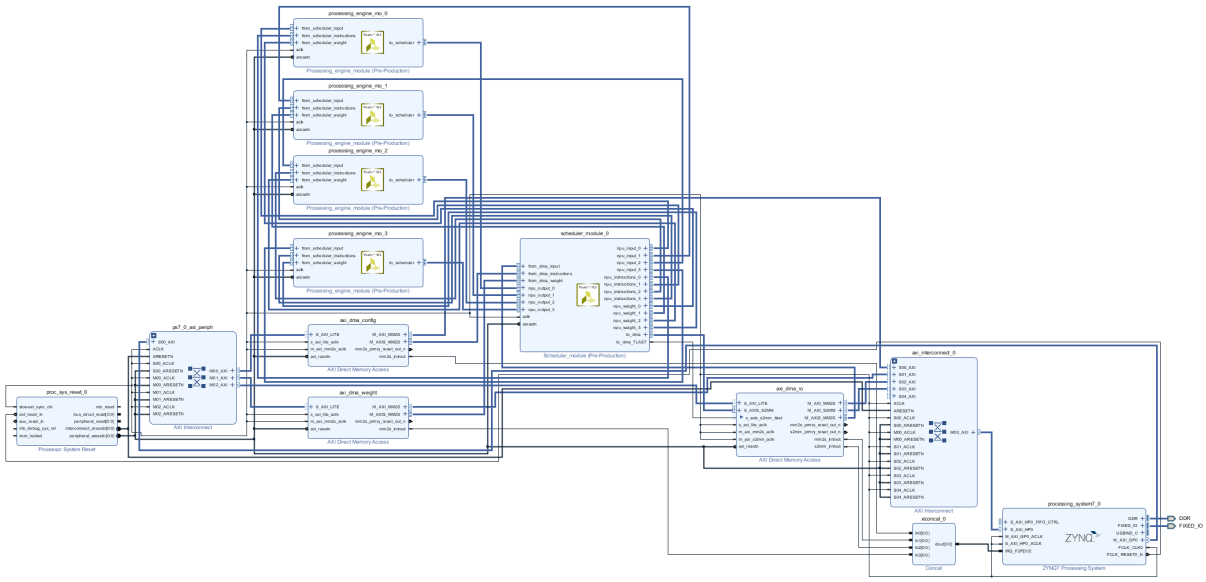


Figure 4.6 – Vivado diagram of our NNP using the PS/PL with three DMA and the scheduler connected to four neural processing unit

4.3 NNP integration into the full prototype

This section will focus on the integration of the NNP into a prototype. Three parts are tackled: the design of embedded data processing on FPGA, the NNP control software and the NNP as part of the application workflow.

4.3.1 Embedded processing to improve performance

Because of the usage of a DNN, embedded data processing design might be a necessary part for the application in order to improve the performance. Designing data processing hardware is not new, but in the case of DL there is one question that needs to be asked: How can we highlight the data features? The answer to this question is the main job of the embedded processing. It should be noted that some applications might not require

embedded processing, and data can be directly transferred to the NNP throught the usage of the control software.

4.3.2 Control software for reconfiguration purposes

With the NNP hardware designed, a software stack is needed to load data into the DRAM to control the NNP. The purpose of this software is twofold: to read a configuration file that regroups all weight matrix binary file paths to determines the instructions, and to sort all weights in matrices for scheduling purposes. Once the architecture is inferred, instructions are generated to process this specific architecture. The weight matrices are then written in the external DRAM, the NNP is started and waits for the input vector from the data processing IP. Every time the NNP finishes a calculation, the output vector is read from the external DRAM. The control of the NNP is also done with the configuration of DMA registers since the processor is waiting until it can read DRAM data. Algorithm 4.3 describes how the configuration software behaves.

Algorithm 4.3: NNP configuration software algorithm

```
Data: Weight matrix files
Read weight matrix files;
Infer layer dimensions from each matrix;
foreach layer do
    Generate instruction;
Sort weight matrices to correspond to neuron processing unit scheduling;
Load sorted weight matrices into DRAM;
Load instructions into DRAM;
repeat
    Wait for output from NNP;
    Save output from DRAM;
    Raise flag;
until system stop;
```

Finally, regarding the weight sorting, since data is coming as a FIFO queue and also to reduce data memory cache usage inside the FPGA, weights shall be transmitted in the same order as the transmissions to their associated neuron processing unit. The scheduling algorithm loads each pair of inputs and weights to each core until all neurons

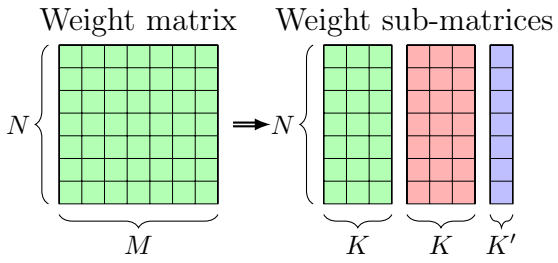


Figure 4.7 – Splitting the weight matrix into sub-matrices with their size depending on the number of cores. N is the number of neurons in the current layer. M is the number of neurons in the next layer. K is the number of neuron processing unit. K' is the size of the last sub-matrix, with $K' \leq K$.

are processed, which means that weights must be sorted depending on the layer size and the number of available cores. The basic process is about dividing the weight matrix into sub-matrices with a size depending on the number of available cores (Figure 4.7). Each sub-matrices represent weights for one set of K neuron processing units. Those weights need to be sorted by processing unit and this is done with a transposition. Then, each transposed matrix is vectorized for memory writing purposes. All vectors are then merged into one vector and written into DRAM.

The main issue with the control software is the DMA configuration. Accessing the DMA registers is done using an access to the system physical memory from the OS. From this point, it is necessary to follow the instructions of the documentation to correctly configure and control the DMA [95]. First, let's detail the basics of the DMA, and what we need to use for our application. A DMA (Direct Memory Access) is a hardware component that transfers data from a memory storage, such as the DRAM in our case, to another hardware component, our NNP here. It transfers slices of bytes from or to a configured address. The primary functioning mode of the DMA is called "direct register mode" which means that we need to setup the DRAM address and the length in bytes of the data to read from or to write to. There are two available channels for the data, the MM2S (Memory-Mapped to Stream) from which the DMA read from the DRAM and write the data to the connected hardware component, and the S2MM (Stream to Memory-Mapped) which transfers the data from the connected hardware component to the DRAM. All data

is streamed as FIFO (First In First Out). In order to configure the DMA with the DRAM addresses and data length, the DMA has an internal register with specific bits for each task. In the case of our application, we only need 8 specific registers, 4 for MM2S and 4 for S2MM, displayed in Table 4.4 (note that this table is for direct register mode only).

Address space offset	Register size (bytes)	Name	Description
00h	4	MM2S_DMACR	MM2S DMA Control register
04h	4	MM2S_DMASR	MM2S DMA Status register
18h	4	MM2S_SA	MM2S Source Address. Lower 32 bits of address.
28h	4	MM2S_LENGTH	MM2S Transfer Length (Bytes)
30h	4	S2MM_DMACR	S2MM DMA Control register
34h	4	S2MM_DMASR	S2MM DMA Status register
48h	4	S2MM_DA	S2MM Destination Address. Lower 32 bit address.
58h	4	S2MM_LENGTH	S2MM Buffer Length (Bytes)

Table 4.4 – Direct register mode DMA registers [95]

Those registers are all those needed for this application using the DMA in direct register mode to transfer data as FIFO from the DRAM to the NNP and vice versa. The registers named MM2S_DMACR and S2MM_DMACR are dedicated to the control of the DMA such as starting and stopping it, and configuring interruptions. The registers named MM2S_DMASR and S2MM_DMASR are dedicated to the status of the DMA which means those are the registers to watch in order to see what the DMA is doing. The register named MM2S_SA stores the source address, which is the DRAM address from which the DMA reads the data. The register named MM2S_LENGTH stores the number of bytes to read from the source address, and the real register length is 26 bytes. The register named S2MM_SA stores the destination address, which is the DRAM address to which the DMA writes the data. The register named S2MM_LENGTH stores the number of bytes to write to the destination address, and the real register length is 26 bytes. Moreover, there are some special behavior from the MM2S_LENGTH and S2MM_LENGTH registers. For the S2MM_LENGTH register, whenever a non-zero value is written, it enables the S2MM transfer, furthermore the number of actual bytes written transferred through the S2MM channel is updated to the register at the end of the transfer. For the MM2S_LENGTH register, whenever a non-zero value is written, it starts the MM2S transfer. Now, let's

Step	Description
1	Reserve memory block for the DMA to read (called source)
2	Reserve memory block for the DMA to write (called destination)
3	Write data inside the source memory block
4	Reset the destination memory block to value 0 so that no previous values can be read by accident
5	Reset the DMA: write 1 to bit 2 of MM2S_DMACR/S2MM_DMACR
6	Halt the DMA: write 0 to bit 0 of MM2S_DMACR/S2MM_DMACR
7	Enable the Interrupts on Complete (IOC): write 1 to bit 12 of MM2S_DMACR/S2MM_DMACR
8	Enable the Interrupt on Error: write 1 to bit 14 of MM2S_DMACR/S2MM_DMACR
9	Set DMA to ready: : write 1 to bit 0 of MM2S_DMACR/S2MM_DMACR
10	Set destination address: write memory address to S2MM_DA
11	Set destination transfer length: write length in bytes to S2MM_LENGTH. Writing a non-zero value enables S2MM channel to receive data packet!
12	Set source address: write memory address to MM2S_SA
13	Set source transfer length: write length in bytes to MM2S_LENGTH. Writing a non-zero value starts the MM2S transfer!
14	Wait for MM2S transfer to finish by looking to MM2S_DMASR: bit 0 to check if it is running correctly, bit 1 to check if it is not idle, bit 12 to check the IOC and bit 14 to check if there are any errors
15	Wait for S2MM transfer to finish by looking to S2MM_DMASR: bit 0 to check if it is running correctly, bit 1 to check if it is not idle, bit 12 to check the IOC and bit 14 to check if there are any errors
16	Read data inside the destination memory block

Table 4.5 – Description of the different steps to configure a DMA

details the steps to correctly configure the DMA and manage the data flow (Table 4.5).

But there is still some limitations when using a DMA. The first one is the transfer bandwidth which is limited by the system bus bandwidth. So depending on your application, you might need to choose a system or design one with internal bandwidth constraints according to your needs. Moreover, in the case of the Vivado DMA in direct register mode, the amount of data transmitted is limited by the MM2S_LENGTH and S2MM_LENGTH register real size, which is 26 bit. Which means that only 2^{26} (a bit more than 67 million) words can be transferred.

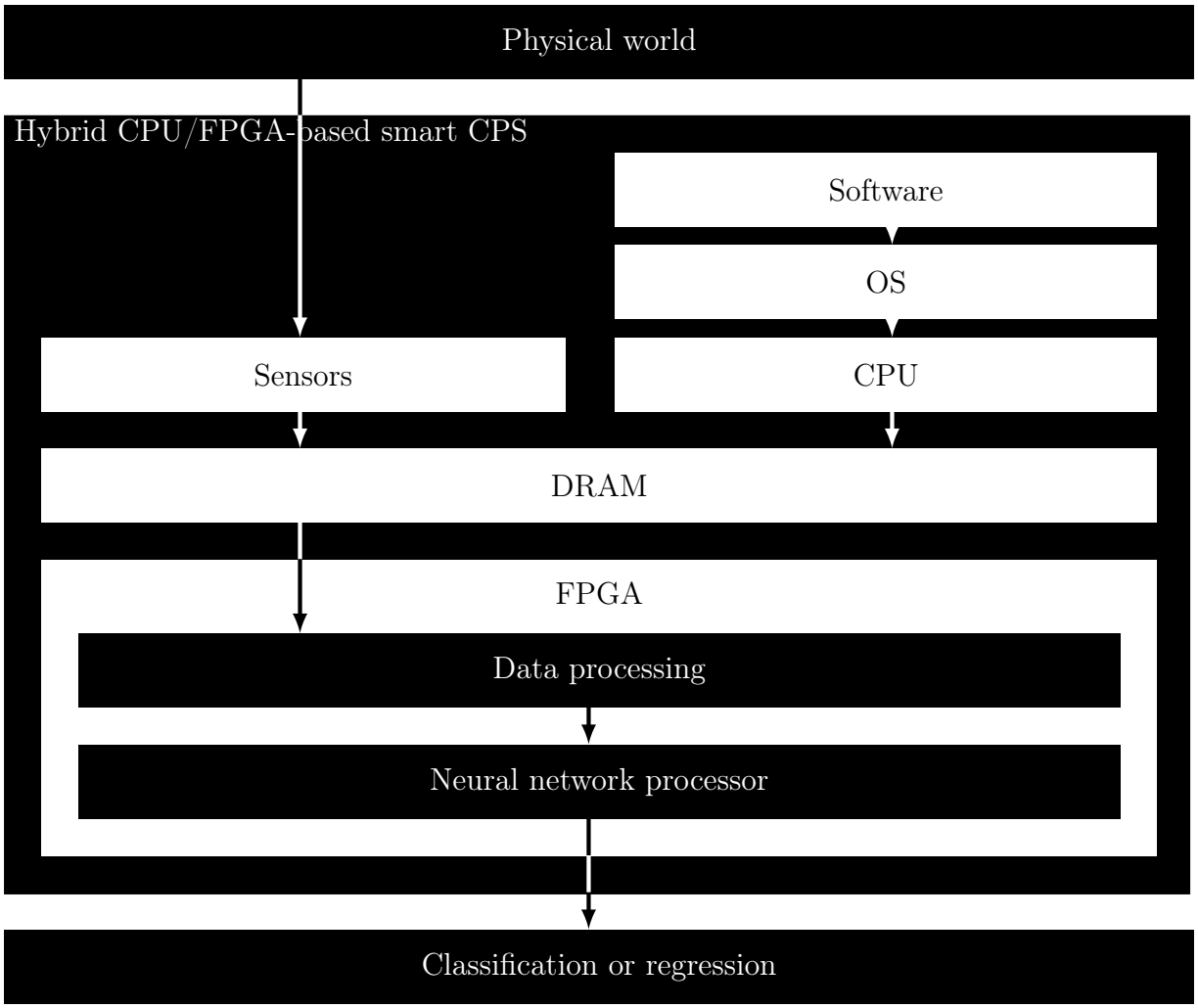


Figure 4.8 – Hybrid CPU/FPGA-based smart CPS workflow

4.3.3 Workflow of an application using the NNP

In the case of a CPS prototype using our methodology with the NNP, a specific workflow (Figure 4.8) is required. As a CPS, the first step is to observe the physical worlds and acquire data from it. This can be done with sensors that will queue their data either to the CPU, or directly into a buffer as part of the DRAM. Meanwhile, the CPU is running an OS with different services on it depending on the CPS needs. But one of the most important services is the control software to load the DNN parameters into the DRAM and configure the FPGA DMA to access the parameters and the sensor's output. This is mainly needed for the DMA configuration, so the FPGA knows where the data is in the DRAM. Inside the FPGA, the embedded processing corrects data from sensors in order

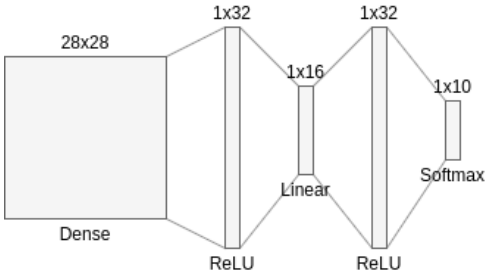
to highlight features for the DNN. A DMA module is dedicated to pick data from sensors from the specified address in the DRAM. Then the NNP receives the processed data as its input and loads the parameters from the DRAM to either classify or regress. There is a DMA module dedicated to the NNP to get the parameters from the DRAM and then to output the NNP results into the DRAM for the CPU to use it. The embedded processing can directly communicate with the NNP through the use of system buses, no need for a DMA. Finally, the system can make decisions based on the NNP output when the CPU is reading the results from the DRAM.

4.4 Experimentation and results

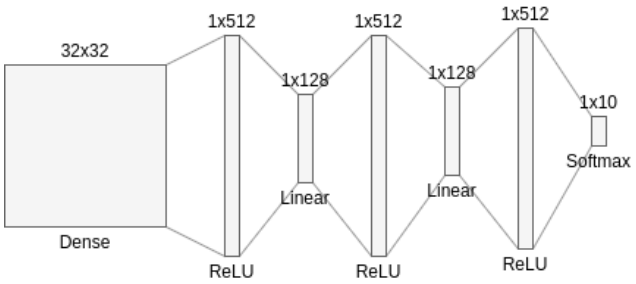
In this section, the NNP performances are tested in standalone mode, the goal is to understand its limitation. Tests have been done on a Zedboard development kit using the Xilinx Zynq-7000 SoC (XC7Z020). Each test corresponds to the usage of a specific known dataset with a specific NN architecture for each dataset. Those datasets are the MNIST [91], Fashion MNIST [96], Cifar10 and Cifar100 [90]. NN model training and testing is done with Tensorflow [94].

NN topologies are shown in Figure 4.9. These topologies are chosen in order to improve the performance of DNN thanks to linear bottleneck layers [88]. Fully connected layers generally cause loss of spatial information because all neurons are connected, which often means that DNN struggles for image classification while it may have better results for tasks such as segmentation. Using a linear activation function between two non-linear functions such as ReLU, and without any bias, means two things:

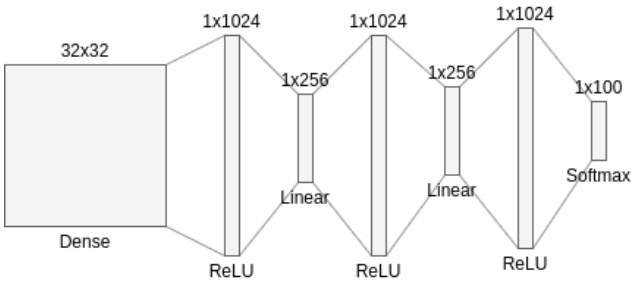
- Better calculation efficiency, either for feed-forward or back-propagation because, as demonstrated by Lin et al, two fully connected non-linear ReLU layers of size N means $N^2 + N$ parameters to calculate, whereas adding a linear layer of size L in-between means $2NL + L + N$ parameters to calculate because there is no need to calculate negative ReLU input since the output will be zero.



(a) **Topology for MNIST and Fashion MNIST dataset**



(b) **Topology for Cifar10 dataset**



(c) **Topology for Cifar100 dataset**

Figure 4.9 – DNN Topologies for each Dataset

- Better accuracy as seen empirically. Though there is no mathematical proof that explains why it is more accurate compared to other fully-connected models.

The Tensorflow source code to train those neural networks is available in Appendix B. The accuracy results achieved by our NNP are exactly the same as evaluated by Tensorflow whatever the number of cores, as shown in Table 4.6. The accuracy for Cifar 10 and Cifar 100 datasets are low compared to a CNN because of the usage of a DNN, though it is possible to achieve better results with unsupervised pre-training using Restricted Blotzmann Machines (RBM) [97] [98] or Zero-bias AutoEncoders (ZAE) [99].

It is to be noted that the maximum number of cores (neuron processing units) embedded in the NNP is 4 because of the FPGA DSP (Digital Signal Processor) limitation (220

Dataset	Tensorflow	1 core	2 cores	3 cores	4 cores
MNIST	99.71%	99.71%	99.71%	99.71%	99.71%
Fashion MNIST	98.53%	98.53%	98.53%	98.53%	98.53%
Cifar10	44.13%	44.13%	44.13%	44.13%	44.13%
Cifar100	14.71%	14.71%	14.71%	14.71%	14.71%

Table 4.6 – Accuracy for each dataset

DSP available on our FPGA chip and each core uses 48 DSP, see Table 4.3). In this work, the execution time is our main concern, and it is obviously related to the number of parameters in the NN and the number of cores in the processor. Figure 4.10 shows the execution time of three datasets depending on the number of cores. The execution time seems to be close to linear, with the same architecture and a different number of cores, except for when there is only one core, which shows a bottleneck. Moreover the execution time per parameter seems to be the same between the different datasets as shown in Figure 4.11. With Vivado HLS transforming the SystemC models to HDL, our hardware threads are running in parallel (the scheduler and neuron processing units are independent finite state machines using the same clock). The clock for the hardware threads is running at 100 MHz, which is 150 MHz lower than the maximal frequency on our hardware. But increasing more than this frequency means that time constraints are not met. The use of parallel hardware threads improved the processing time of our system. However, we want to point out the data transfer bottleneck in the AXI system bus which affects the whole processing time of the system. This bottleneck is mainly due to the number of parameters transmitted. Since we are using 32-bit floating points, the parameters matrices of the NN are in the MB scale and our AXI channels are running at a theoretical maximum of 300 MB/s. We would get better results if using compression such as 16-bit fixed point integer or binary weights. Another option to improve the time consumption would be to run the scheduler which controls the neural processing units with a faster clock than the neuron processing units so that data is read faster from DMA and distributed faster to the processing units, but we did not confirm this will bypass the data transfer bottleneck. In the context of the defined topologies, MNIST has **26,432** parameters,

Cifar 10 has **791,552** parameters (because since the input is grayscaled, the image size is 32x32x1), and Cifar 100 has **2,107,392** parameters. Figure 4.11 analysis shows that the execution time of the feed forward sequence of a specific NN model may be predicted, which means we can determine the needed NNP cores for a given application with its real-time constraint. Moreover, it might be possible to estimate the maximal number of cores before reaching the data transmission bandwidth bottleneck, which would be when the estimated parameter calculation time is less than the transmission time. But because those measures take into account the data transmission bandwidth, the estimation would be a rough one.

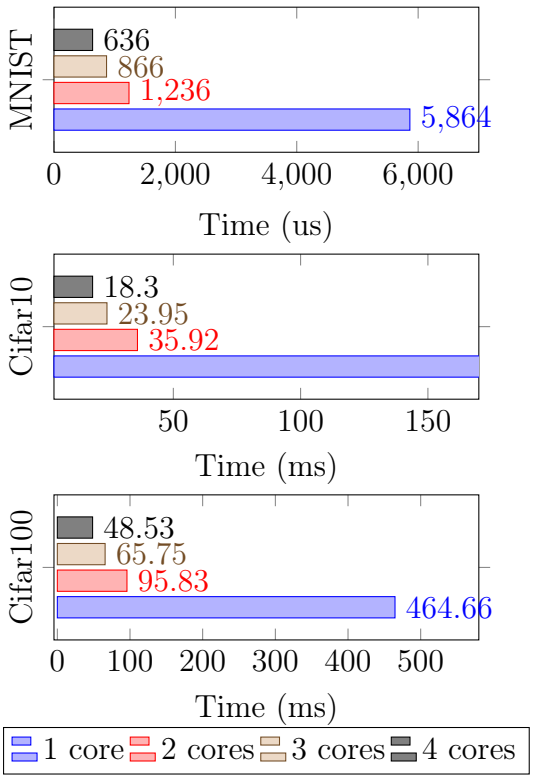


Figure 4.10 – Tests results of the NNP, the execution time is for one feed forward sequence

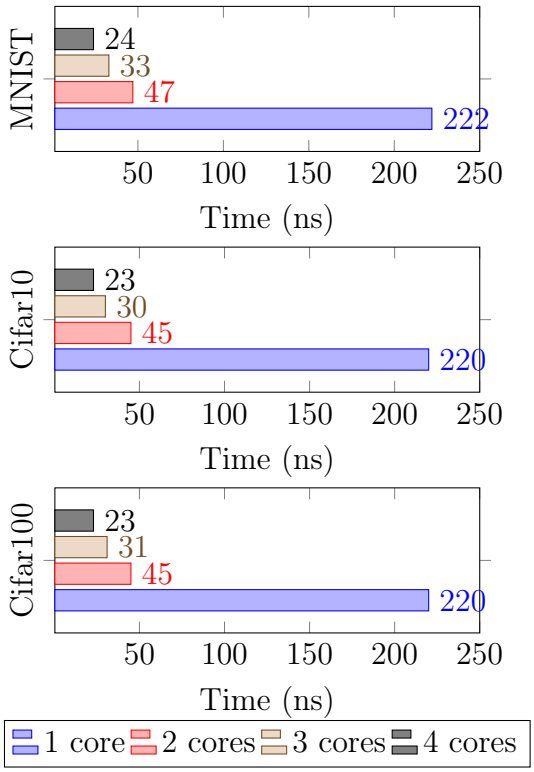


Figure 4.11 – Execution time per parameter with different number of cores and topologies

4.5 Conclusion

In this chapter, we introduced the concept of NNP (Neural Network Processor) and how they are designed. We explained the math behind the feed-forward of a fully-connected neural network and defined some guidelines to design our NNP such as data precision and main architecture. We displayed the methodology toward the integration of our NNP into a real prototype, the software as well as the hardware. We described the source code of our NNP so that other people can use it for their own purposes. Finally, we presented some unit testing on our NNP, to show the accuracy and execution time of our NNP. The different contributions in this chapter are as follows:

- We proposed an architecture, a design and a prototype for a hardware Neural Network Processor (NNP)
- We created a configuration and benchmarking software for the NNP
- We validated the NNP with different configurations and we got different results
- We shared the source code of the NNP in a public online Git repository
- We also shared the source code of the different neural network topologies training and testing with Tensorflow in the appendices

In the next chapter, we will present the full experiment and results from our case study implementation using our methodology and NNP.

Chapter 5

Implementation and validation: a smart LIDAR for pedestrian detection

Contents

5.1 Introduction 82

5.2 Autonomous Vehicle Case Study: Description 82

5.3 Design of the application 83

5.4 Experimentations 86

5.4.1 Embedded processing algorithms 87

5.4.2 Deep learning algorithms 96

5.4.3 Implementation and results 97

5.5 Conclusion 102

5.1 Introduction

In previous chapters, we presented a methodology for FPGA-based CPS using Deep Learning (DL) as well as a Neural Network Processor (NNP) used to hardware accelerate DL calculation. Now that we have a methodology to design an FPGA-based DL application and that the NNP is designed, we can now test all those together with a use-case. In this chapter, a smart LIDAR for pedestrian detection use-case will be presented, in order to test our methodology along with our NNP, and display its limitations. In this work, our definition of a "smart LIDAR" is a LIDAR sensor that, instead of outputting a 3D representation of its physical environment (a 3D point cloud in our case), will output the detected elements in this environment. Those elements could be any type of object present in a constrained environment. In our case, since we chose the autonomous car, because of our lab thematic, those elements could be any type of urban objects such as pedestrians, cars, bicycles, etc. But because of dataset constraints, we chose to only detect pedestrians, in order to have enough data to train our DL models.

5.2 Autonomous Vehicle Case Study: Description

The chosen case study is a smart LIDAR for pedestrian detection, which fits the autonomous vehicle context. The goal is to use a 3D sensor such as a LIDAR to directly detect pedestrians as the output of this sensor instead of outputting 3D mapping data. We want to create a smart sensor system [100] as a CPS using a LIDAR sensor and a hybrid CPU/FPGA platform using our NNP. The features required in this case study are the ability to detect pedestrians in the observed environment through the use of a 3D point cloud made by the LIDAR, and communicating the position of those pedestrians relative to the LIDAR position. The main constraint in this case study is a real-time one. We want to be able to detect all pedestrians mapped for each 3D frame sent by the LIDAR, which is running at 10/15 Hz in this case study, so between 66.67 and 100 ms per 3D frame. This means that we need to accelerate all tasks related to the processing of

each frame in order to reach the real-time constraint. However, the use of deep learning, particularly a modern model with high accuracy, is not meant to run in such a short time on an embedded system with low resources compared to other systems. This is why the acceleration of the NN processing is mandatory in such cases. Consequently, we are using our methodology described in a previous chapter to speed up the prototyping of this system.

5.3 Design of the application

The first step toward this application is to define the goal and the different tasks to reach this goal. The purpose of this smart LIDAR is to sense its environment, then detect pedestrians in it. A LIDAR works kind of like a RADAR (based on echolocation) but it uses a different electromagnetic wavelength: ultraviolet, visible or near infrared lights. With such wavelengths, a narrow laser beam is projected on a rotating mirror to probe the environment with a specified vertical and horizontal range angle. For each movement of the mirror, the reflected wave is recorded to map physical features as points in a 3D space. Once the mirror movement phase is completed, a map of all the points recorded in the 3D space (often called point cloud) is generated and sent to the system linked to the LIDAR sensor. From this point cloud, we need to detect and recognize pedestrians.

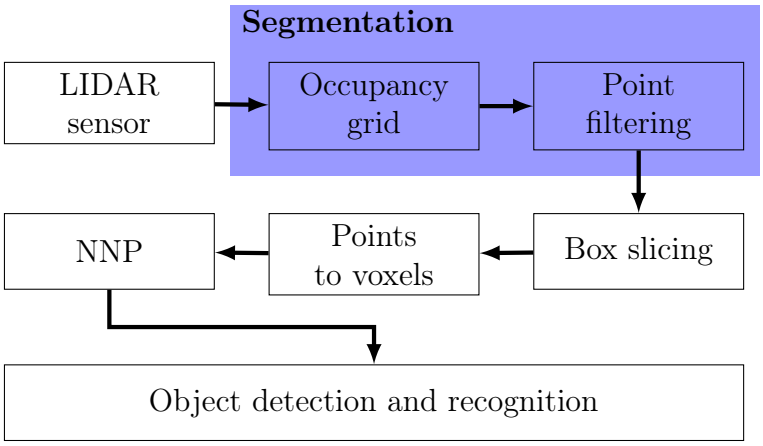


Figure 5.1 – Smart LIDAR use-case task flow

For this purpose, there are several algorithms available. We will be using the ones most used in robotics, especially for robotics and autonomous cars [101]. First, we will use segmentation to detect points categorized as an obstacle and remove non-obstacle points. This will reduce the number of points to remove the floor and other noise points. This operation can be split in two parts: occupancy grid and point filtering. The occupancy grid maps points in a cell in the XY dimension parallel to the floor and determine the mean height (defined on the Z axis) of those points. If the mean height is above a defined threshold, all points in the cell are categorized as an obstacle and the cell is defined as active. Otherwise, the cell is defined as inactive if there are no obstacles. The point filtering removes all points in inactive cells to keep only the active cell's points. Once the segmentation is done, we need to isolate points in boxes to analyze them afterwards. This process is similar to the window sliding [102] but is used in a 3D space instead of a 2D one. Once the points have been isolated in boxes, we need to convert them to a data model that can be used by deep learning algorithms. We can look into images, often used for object classification with cameras. They are made of pixels, a 2D datum that indicates information at a specified coordinate. The 3D equivalent of a pixel is called a voxel, a 3D datum that indicates information at a specified 3D coordinate. So the last task before the deep learning classification is to convert points to voxels. Figure 5.1 is a visual representation of all those tasks and the links between them.

The next step is to define the workflow of the application. Figure 5.2 represents all the steps of the application. It first starts with the physical world data that is acquired through the LIDAR sensor. The sensor transmits its raw data to the embedded processing hardware module in order to process and transform the information so that it can be used by the NNP, which analyzes the data and classifies the objects. Meanwhile the embedded processing part is split in four tasks: Occupancy grid, point filtering, sliding box and points to voxels. The goal of the occupancy grid is to create binary cells based on the Z-axis mean to detect if those points are relevant. The point filtering stage takes care of discarding all points which are in inactive cells. The sliding box splits the filtered point

cloud in predetermined boxes that can overlap themselves in order to extract objects. The points to voxels transforms each box from a continuous 3D space to a discrete one, with voxels.

Now we can proceed to the implementation of the workflow following our methodology. There are four main steps: deep learning software, embedded processing, hardware acceleration and hardware platform. The deep learning software implements a neural network made to analyze a voxelized 3D space and determine if its a pedestrian or not. The embedded processing is the first step toward hardware acceleration; the goal is to determine data processing algorithms to be hardware accelerated then implement it as a software to

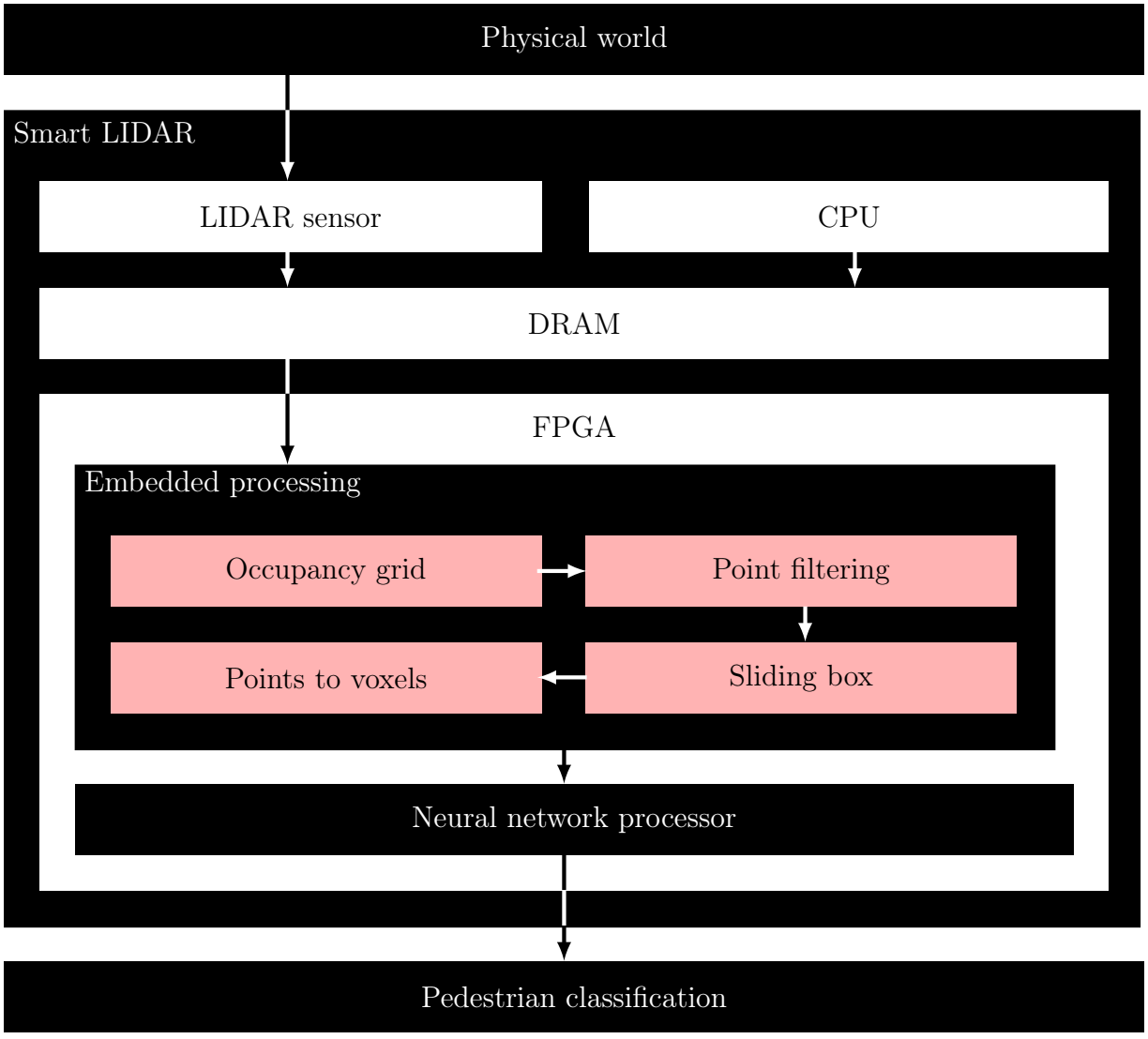


Figure 5.2 – Smart LIDAR for object classification case study

test the performances. The hardware acceleration transforms the embedded processing algorithms as HDL (Hardware Description Language) and implements it along the NNP. Finally, the hardware accelerated application along the deep learning weight matrices are loaded on the configured hardware platform in order to have a prototype. Figure 5.3 is the illustration of this implementation following our methodology.

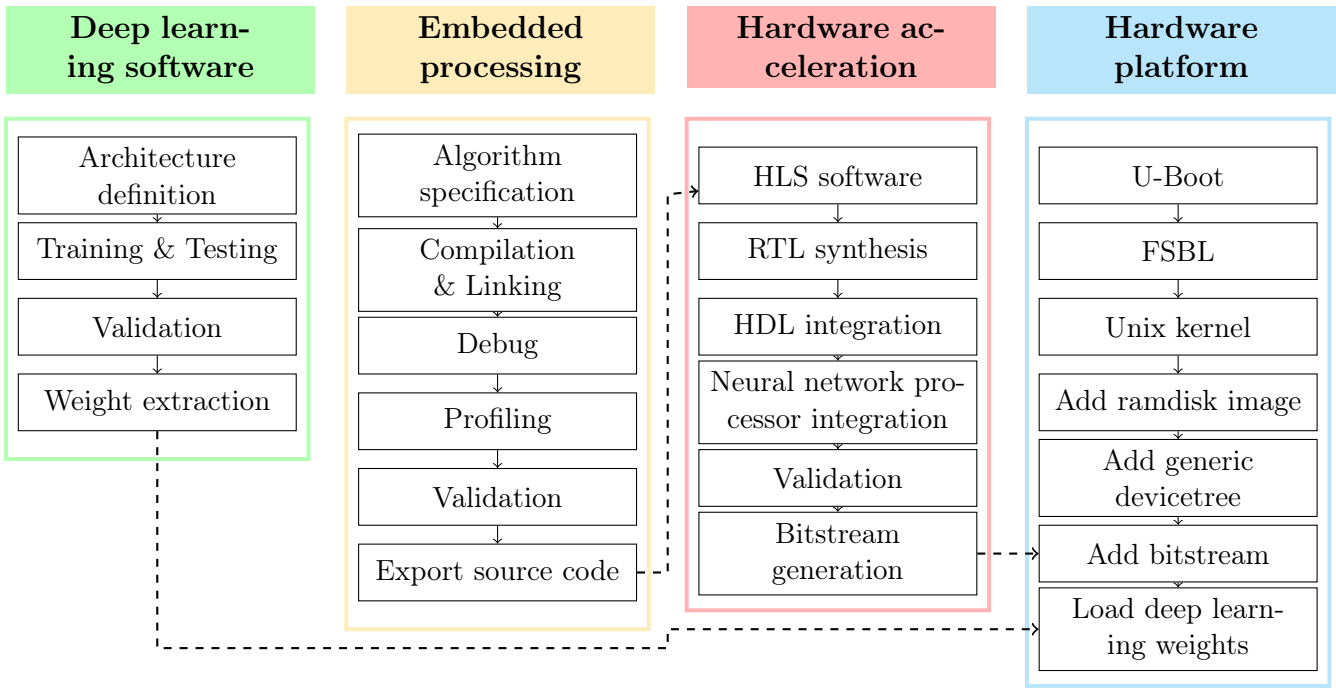


Figure 5.3 – Design flow for the embedded DL methodology

5.4 Experimentations

Following the workflow of the smart LIDAR use case, we now need to design the different algorithms for the embedded processing, as well as determining the DL architecture to do its training/testing. We also need to determine the experimental protocol to correctly measure the performances of our experimentation. And finally we implement those algorithms on our final prototype.

5.4.1 Embedded processing algorithms

We first start to detail the different embedded processing algorithms. The goal is to transform LIDAR sensor data to 3D space voxel objects which are then classified by the NNP. There are four parts in the embedded processing. The first one is the **segmentation**, but because of the hardware constraints, **segmentation** needs to be separated in two sub-tasks: *occupancy grid* and *point filtering*. Then comes the **box slicing** and finally the **points to voxels**. Each tasks were first made as different pieces of software in order to verify the algorithms, then they were tweaked for the hardware requirements.

5.4.1.1 Segmentation

For the segmentation part as a piece of software, the process (see Algorithm 5.1) is about making an occupancy grid on the XY plan of the point cloud i.e. calculating the mean Z axis distance on each cell and comparing it to a threshold so a binary grid of the 3D space is created. The purpose is to calculate the Z mean axis density of an object in order to estimate if it is an obstacle, and thus determining if it is an object to be classified. If the occupancy sub-area (cell) is inactive, then the points covered by the cell will be discarded from the output filtered point cloud. The filtered point cloud can be considered as the point cloud containing all the "to be classified" objects. This step improves the performance of the box slicing module, removes the ground from the original point cloud, and simplifies object classification. The algorithm runs through the point cloud with a predetermined step size on the XY plan. For each step, we calculate the mean Z axis for all the point cloud box, if it is greater than the threshold, then it means that an obstacle has been detected, and will be stored in the final point cloud. It is distinguished that the processing is done on one plan of the point cloud, which corresponds to the ground. If the first **Segmentation** (Algorithm 5.1) is refined as an hardware component, its tasks need to be run as two steps in a row and not as concurrent tasks, because of the FIFO data access. Those two tasks are defined as: the occupancy grid and the point filtering. The occupancy grid creates binary cells based on the Z-axis mean to detect if those points are

relevant. The point filtering discards all points which are in inactive cells.

Algorithm 5.1: The software segmentation algorithm

```
Data: Point Cloud, Sub-area/Cell (width, height), Threshold
Result: Filtered Point Cloud
point_MIN ← find the Minimum in the point cloud;
point_MAX ← find the Maximum in the point cloud;
step_X ← (point_MAX.X − point_MIN.X) / cell_WIDTH;
step_Y ← (point_MAX.Y − point_MIN.Y) / cell_HEIGHT;
for Y ← 0 to step_Y do
    for X ← 0 to step_X do
        box_MIN ←
          {point_MIN.X + cell_WIDTH * X, point_MIN.Y + cell_HEIGHT * Y, point_MIN.Z};
        box_MAX ← {box_MIN.X + cell_WIDTH, box_MIN.Y + cell_HEIGHT, point_MAX.Z};
        box ← get points from point cloud between box_MIN and box_MAX;
        if the box is not empty then
            Z_mean ← make the Z axis mean for points in box;
            if Z_mean > threshold then
                add the box in the filtered_cloud;
            end
        end
    end
end
return filtered_cloud;
```

For the **Occupancy Grid** (Algorithm 5.2), since data is streamed, when a point is received, the cell it belongs to is identified and the Z-axis mean of the cell is updated. Once all points are received, the mean of each cell is compared to the threshold to compute the occupancy grid. In this work, the *threshold*, *width* and *height* data, as well as the implicit number of points, are all sent from the CPU. The Point Cloud is then streamed point per point to the occupancy grid IP (Intellectual Property). For each points received, the X and Y axis coordinates are normalized between 0 and the grid width/height. The 0.5 offset in the algorithm is due to the X and Y axis range of original points:

$$X \in \mathbb{R}| -\frac{width}{2} \leq X \leq \frac{width}{2} \tag{5.1}$$

$$Y \in \mathbb{R}| -\frac{height}{2} \leq Y \leq \frac{height}{2} \tag{5.2}$$

Algorithm 5.2: The occupancy grid algorithm from the hardware segmentation task

```
threshold ← threshold configuration;
width ← point cloud map width;
height ← point cloud map height;
repeat
    point ← receive point from CPU;
    x ← trunc[(point.x / width + 0.5) * grid width];
    y ← trunc[(point.y / height + 0.5) * grid height];
    if point.z < cell(x, y).z_min then
        cell(x, y).z_min ← point.z;
    end
    cell(x, y).z_sum ← cell(x, y).z_sum + point.z;
    cell(x, y).z_count ← cell(x, y).z_count + 1;
until no more points received;
for y ← 0 to grid height do
    for x ← 0 to grid width do
        mean ← cell(x,y).z_sum / cell(x,y).z_count − cell(x, y).z_min;
        if mean > threshold then
            Send cell(x, y) active to next IP;
        else
            Send cell(x, y) inactive to next IP;
        end
    end
end
```

Once normalized, each X and Y is truncated and associated to a cell. For the corresponding cell, Z-axis points are summed, the point counter is incremented and the minimum Z value of the cell is stored for later. When all points have been received from the CPU, each cell is computed. The mean with an offset is calculated in order to find the mean size of the object from the zero point origin. The mean is then compared to the threshold, and the result of each cell, is called the occupancy grid, which is send to the next IP: the point filtering.

The second part of the segmentation task is the **Point Filtering** (Algorithm 5.3). Each point from the Point Cloud is compared to the occupancy grid. If the cell is active, the point is kept, otherwise it is discarded. As in the previous algorithm, the *width* and

height data, as well as the implicit number of points, are all sent from CPU. For each point received, it is scaled to the same range as in the occupancy grid part and mapped to the correct cell. If the cell is active, the point is sent to the CPU so it can be used for the next task, the box slicing. With those two tasks as part of the segmentation, points from the Point Cloud are sent two times within the system bus.

Algorithm 5.3: The points filtering algorithm from the hardware segmentation task

$width \leftarrow$ point cloud map width;
$height \leftarrow$ point cloud map height;
$occupancy\ grid \leftarrow$ receive occupancy grid from previous IP;
repeat
  $point \leftarrow$ receive point from CPU;
  $x \leftarrow trunc[(\frac{point.x}{width} + 0.5) * grid\ width]$;
  $y \leftarrow trunc[(\frac{point.y}{height} + 0.5) * grid\ height]$;
  if $cell(x, y)$ *is active* **then**
    Send point to CPU
  end
until *no more points received*;

5.4.1.2 Box slicing

The box slicing module split the filtered point cloud in predetermined boxes that can overlap themselves. The box size is chosen to contain the object to be detected (see algorithm [5.4]). In order to filter noise, boxes with few points are discarded. The box slicing methodology is inspired from the concept of sliding window [103] in image processing, and object detection. The box slicing presents a generic way to detect any object and find its position in the vehicle's environment. In our experience, we conclude that the sliding consumes more processing time than the slicing technique, even if it consumes more storage. The algorithm runs through the point cloud with a predetermined step size on an XY plan. For each step, get all points in the box (box size can be greater than step size, which means that boxes can overlap). If the number of points in the box is greater than a threshold, then add the box to the returning array. It is notable that the run is

made on one plan of the point cloud, which should correspond to the ground.

Algorithm 5.4: The box slicing software algorithm

Data: Point Cloud, Box (width, height, depth), Step (width, height), Threshold
Result: Point Cloud Boxes

$point_{MIN} \leftarrow$ find minimum point in input point cloud;
$point_{MAX} \leftarrow$ find maximum point in input point cloud;
$step_X \leftarrow \frac{point_{MAX}.X - point_{MIN}.X}{step_{WIDTH}}$;
$step_Y \leftarrow \frac{point_{MAX}.Y - point_{MIN}.Y}{step_{HEIGHT}}$;
for $Y \leftarrow 0$ **to** $step_Y$ **do**
  for $X \leftarrow 0$ **to** $step_X$ **do**
    $box_{MIN} \leftarrow$
     $\{point_{MIN}.X + step_{WIDTH} * X, point_{MIN}.Y + step_{HEIGHT} * Y, point_{MIN}.Z\}$;
    $box_{MAX} \leftarrow$
     $\{box_{MIN}.X + box_{WIDTH}, box_{MIN}.Y + box_{HEIGHT}, box_{MIN}.Z + box_{DEPTH}\}$;
    $box \leftarrow$ get points from point cloud between box_{MIN} and box_{MAX};
    if *number of points in box* $>$ *threshold* **then**
      add *box* in array;
    end
  end
end
return array of *box*;

For the **Box Slicing** hardware algorithm, data streaming revealed a hierarchical problem. In the software algorithm, boxes were sliding from step to step with a double for-loop. With data coming as FIFO, to exactly reproduce this behavior, the whole point cloud needs to be sent for each box, which will lead to the congestion of the AMBA. So the initial task needs to be reversed as assigning a box to a point whenever the point is received by the IP in order to only transfer the point cloud once. For the explanation of this part, only the X dimension will be observed and the box will be reduced as 2D, but it is notable that everything written here applies to every other space dimensions. The constraint for this task is the overlapping of boxes depending on their width and step size. For this work, the step width was defined as:

$$step_{width} \in \mathbb{R} | 0 < step_{width} \leq box_{width} \tag{5.3}$$

Thus the hierarchical problem: the overlapping depends on the step width. Two kinds of overlapping are defined in this work. We defined the "simple overlapping" (see Figure 5.4) as the overlap case when:

$$step_{width} > \frac{box_{width}}{2} \tag{5.4}$$

In the case of “simple overlapping,” points are first mapped to the corresponding box identifier (ID) (Equation 5.5). Once the box is mapped, the position of the point in this box is processed in order to find out if the point is in the overlapping zone (Equation 5.6). There is an exception in the overlapping zone for the first and last boxes of the row, since there is no overlapping zone as defined in equation 5.6, so if box_{ID} is 0 or $box_{max\ ID}$, there is no box overlapping possible. The points are always matched to the most advanced box_{ID} and if the position of the point in the $step_{width}$ grid is considered in the overlapping zone, the point is also matched to the previous box ID which is $box_{ID} - 1$.

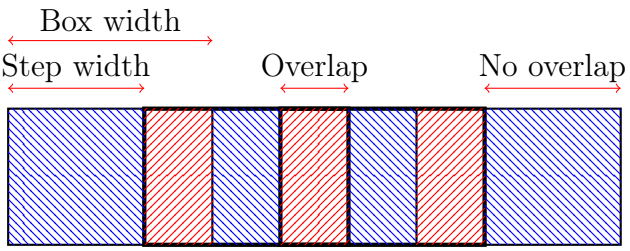


Figure 5.4 – Box simple overlapping hierarchical problem. The blue zone represents no box overlapping, the red zone represents two boxes overlapping.

$$box_{ID} = trunc(\frac{X}{step_{width}}) \tag{5.5}$$

$$Overlap(X) = \begin{cases} 1, & \text{if } X - box_{ID} * step_{width} < box_{width} - step_{width} \\ 0, & \text{otherwise} \end{cases} \tag{5.6}$$

Then there is the "multiple overlapping" (Figure 5.5), which we defined as the overlap

case when:

$$step_{width} \leq \frac{box_{width}}{2} \tag{5.7}$$

This problem is a bit more complex, because boxes always overlap, there is no "no-overlapping" zone between overlapping as in the "simple overlapping" problem.

In the case of "multiple overlapping," points are also matched to a specific box_{ID} with the same equation 5.5. The main difference is the handling of the overlapping zone. The box_{ID} is always the most advanced box, so it is now important to calculate the maximum overlap of the zone in order to match the correct previous boxes (see Equation 5.8). Once the point is matched with equation 5.5, the point is determined to be inside all boxes between box_{ID} and $box_{ID} - Overlap(X)$.

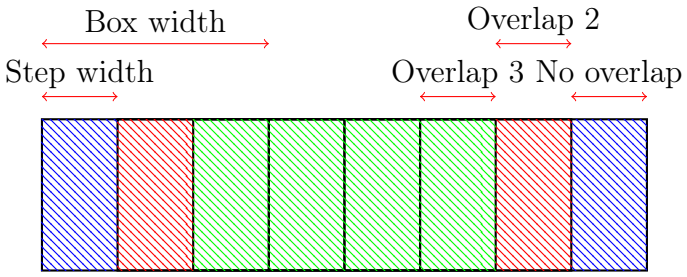


Figure 5.5 – Box multiple overlapping hierarchical problem. The blue zone represents no box overlapping, the red zone represents two boxes overlapping. The green zone represents three boxes overlapping

$$Overlap(X) = \begin{cases} box_{ID}, & \text{if } X_{ID} < \frac{box_{width}}{step_{width}} \\ box_{max\ ID} - box_{ID}, & \text{if } X_{ID} \geq box_{max\ ID} - \frac{box_{width}}{step_{width}} \\ \lfloor \frac{box_{width}}{step_{width}} \rfloor, & \text{otherwise} \end{cases} \tag{5.8}$$

This problem applies to each dimension needed, so dimensions X and Y in this work. Algorithm 5.5 is the implementation of this problem for one dimension. But processing each dimension individually lacks the dimension intersection. So once each dimension is processed for the point, the box_{ID} need to be finally adjusted to the two dimensions X and Y (see Algorithm 5.6) to correctly be mapped as boxes in the Point Cloud. Finally, when all points and boxes are sent to the CPU, it is necessary to regroup all points to their

respective box to completely terminate the main HW/SW co-design application. After this, all processing can be done by any supervised machine learning classification system to identify the objects in the boxes. Object position in space is also known since any box position can be decrypted from the box_{ID}.

Algorithm 5.5: The part of the box slicing algorithm for one dimension from the hardware box slicing task

$width \leftarrow$ point cloud map width;
$box_{width} \leftarrow$ box width;
$step_{width} \leftarrow$ step width;
repeat
  $point \leftarrow$ receive point from CPU;
  $x \leftarrow point.x + \frac{width}{2}$;
  $x_{overlap} \leftarrow 0$;
  $x_{ID} \leftarrow \frac{x}{step_{width}}$;
  if $step_{width} > \frac{box_{width}}{2}$ **then**
    if $x - x_{ID} * step_{width} < box_{width} - step_{width}$ **then**
      $x_{overlap} \leftarrow 1$;
    end
  else
    if $x_{ID} < \frac{box_{width}}{step_{width}}$ **then**
      $x_{overlap} \leftarrow x_{ID}$;
    else if $x_{ID} \geq \frac{width - box_{width}}{step_{width}}$ **then**
      $x_{overlap} \leftarrow \frac{width}{step_{width}} - X_{ID}$;
    else
      $x_{overlap} \leftarrow \frac{box_{width}}{step_{width}}$;
    end
  end
  for $id \leftarrow x_{ID} - x_{overlap}$ **to** x_{ID} **do**
    Transmit *point* and *id*;
  end
until *no more points received*;

5.4.1.3 Points to Voxels

3D Point Cloud data is the output of the LIDAR, but it is not the input of the NNP. Deep learning algorithms are not particularly made to learn on continuous space, thus we need to transform the 3D Point Cloud to a discrete space. A 3D discrete space is made

Algorithm 5.6: The dimension merging algorithm from the hardware box slicing task

$width \leftarrow$ point cloud map width;
$step_{width} \leftarrow$ step width;
repeat
  $point \leftarrow$ receive point;
  $x_{ID} \leftarrow$ receive X dimension ID of $point$;
  $y_{ID} \leftarrow$ receive Y dimension ID of $point$;
  $box_{ID} \leftarrow x_{ID} + y_{ID} * \frac{width}{step_{width}}$;
  Send $point$ and box_{ID} to CPU;
until *no more points and ids received*;

Algorithm 5.7: Point cloud to voxel grid hardware algorithm

$voxel\ size \leftarrow (24, 24, 24)$;
$padding\ size \leftarrow (32, 32, 32)$;
$resolution \leftarrow 0.1$m;
$minimum\ coordinates \leftarrow (+\text{inf}, +\text{inf}, +\text{inf})$;
$voxel\ grid \leftarrow$ 32x32x32 cells of 1 bit;
repeat
  if $point\ coordinates < minimum\ coordinates$ **then**
    $minimum\ coordinates \leftarrow point\ coordinates$;
until *all points received*;
repeat
  $point\ coordinates \leftarrow point\ coordinates - minimum\ coordinates$;
  if $0 \leq point\ coordinates\ AND\ point\ coordinates < voxel\ size * resolution$ **then**
    $center\ point\ coordinates \leftarrow point\ coordinates + (padding\ size - voxel\ size) * \frac{resolution}{2}$;
    $voxel\ coordinates \leftarrow trunc(\frac{center\ point\ coordinates}{resolution})$;
    $voxel\ grid[voxel\ coordinates] \leftarrow 1$;
until *all points received*;
return *voxel grid*;

of entities called **voxels**. Each voxel represents the occupation of the space in a certain continuous volume. This occupation can either be proportional or binary. In the case of a proportional occupation, we describe the voxel as a percentage of occupation. In the case of a binary occupation, we describe the voxel as an occupied space or an empty space.

In our case, we are using binary occupation. Hence, each object in the 3D Point Cloud needs to be extracted and transformed to voxels as an input for the NNP. The final step is to convert extracted objects from the sliding box into a volumetric binary occupancy grid. The algorithm for the "points to voxels" module is presented in Algorithm 5.7. The module receives all the points from a box two times: the first time to calculate the bounding box, the second time to calculate the volumetric binary occupancy grid. Once the object has been transformed into a 32x32x32 voxels grid, it is sent to the NNP to be classified.

5.4.2 Deep learning algorithms

Once the data has been processed, we need DL algorithms in order to detect and recognize pedestrians. The presented DL architecture is the one taking 3D voxel objects and classifying them. One of the steps when working on this prototype is to define how to classify objects from 3D data such as the point cloud received from the LIDAR. One way is to convert the point cloud to voxels, then use deep learning to determine the category [55]. The dataset used in our case study is the Sydney Urban Objects Dataset (SUOD) [93][92] but we converted point clouds into grids of 32x32x32 voxels using a volumetric binary occupancy grid approach. The training is done with the architecture represented in Figure 5.6 thanks to the Tensorflow software [94]. Once the architecture has been defined and the training/testing has been done, the weights were extracted in a NumPy binary format [89]. The SUOD dataset is used for training here because it contains several types of urban objects (pedestrians, cars, cyclists, ...). We wanted to use this dataset in order to have a complete training with different objects in order to differentiate pedestrians from them. Moreover, it also means that multi-object classification is possible, but the current processing algorithms are made with pedestrians in mind, and as such will misclassify other objects. Now, in order to export weights, we used Tensorflow weight matrices after the training and converted them into a Numpy format file. This file is then exported into our SDCard prototype to be used by the control software in order to feed those param-

eters to the NNP. The source code for the training and the weight export is available in Appendice C. The SUOD dataset we used, which is a convertion from 3D Point Cloud to voxels, is available in our NNP repository [87] inside the *models/suod/dataset.npz* file.

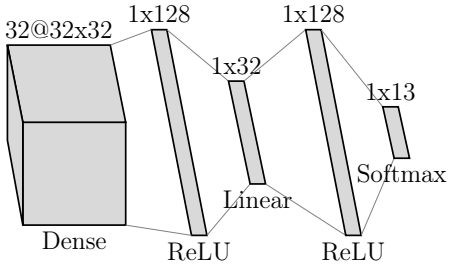


Figure 5.6 – DNN topology for SUOD dataset

5.4.3 Implementation and results

Now the workflow and all algorithms have been defined, it is time to implement everything on top of a real application and a hardware prototype. The hardware platform used is a ZedBoard Zynq-7000 [47]. The SD card generation is automated using our automation software [74]. This software deploys a UNIX Operating System (OS) with a Linux kernel and all other needed resources to boot this OS. All hardware modules are written in SystemC and synthesized to RTL with an HLS software (Xilinx Vivado HLS [47]). The LIDAR data comes from the 3D Point Cloud People Dataset [104] [105] which uses a Velodyne HDL 64E S2 sensor with a frequency of 5Hz at a maximum range limited to 20m. The different steps presented here will be the four embedded processing tasks presented in Figure 4.8, then NNP results will be presented.

5.4.3.1 Embedded processing

Table 5.1 shows the execution time for the embedded processing tasks when running as software and when hardware accelerated. As shown here, hardware acceleration has a huge impact on the execution time. The software time execution on the Zedboard was **103,460 ms** and it comes to **344 ms** when it is hardware accelerated. This represents a **300 times acceleration**.

SW	Occupancy Grid + Point Filtering		Box Slicing	**Total**
Time	90,230 ms		13,230 ms	**103,460 ms**
HW	Occupancy Grid	Point Filtering	Box Slicing	**Total**
Time	98 ms	132 ms	114 ms	**344 ms**

Table 5.1 – Comparison of SW and HW Applications Execution Time

In Figure 5.7, we compare the execution time between the software profiling and the hardware profiling for the Segmentation (Occupancy Grid and Point Filtering) and the Box Slicing tasks. As shown in this figure, the tasks that benefited the hardware acceleration the most were the Occupancy Grid and Point Filtering (OG+PF) tasks, with a 392 times acceleration. This difference is mainly due to the hardware task parallelism, because of the use of a FPGA compared to a single software thread when benchmarking our software. But the task is still considered as slow when compared to the real time constraint of the use case. This is mainly due to a bottleneck in the system bus because of the large amount of data transmitted, as well as the use of 32-bit floating points which use a lot of resources and are not as fast as integers calculation.

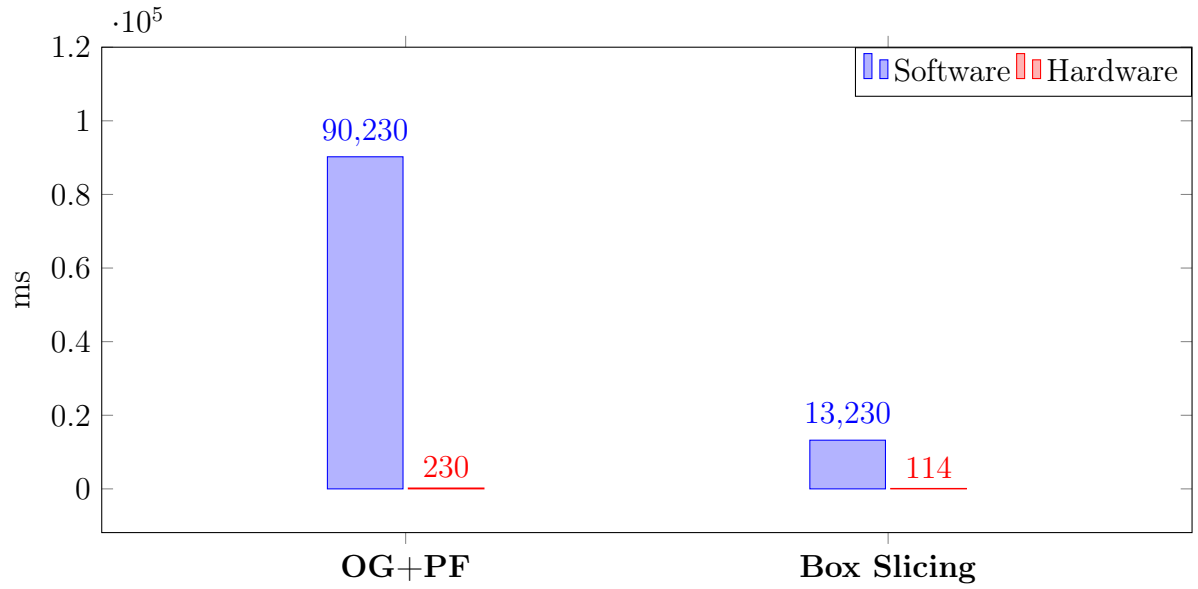
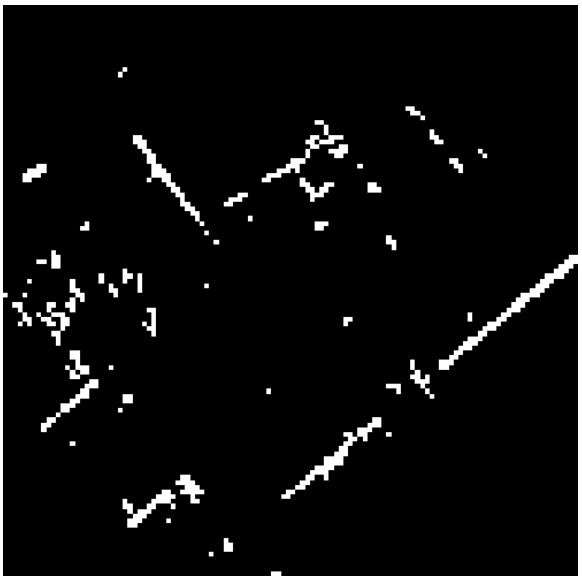


Figure 5.7 – Comparison between software and hardware application execution times

Occupancy grid

The occupancy grid task uses the initial 3D point cloud and determines with a 2D binary grid, parallel to the ground, in which cells are occupied by objects. Figure 5.8 shows the result of the occupancy grid. The occupancy grid generated here considers all mean heights greater than 0.5 meters as an obstacle. The grid is composed of 120 × 120 cells on a 3D point cloud which is around 40m × 40m. This means that one cell represents a 33cm × 33cm square.

(a) The initial point cloud [104][105]



(b) The generated 2D binary occupancy grid

Figure 5.8 – Example of the occupancy grid task

The occupancy grid processes **205,300 points** and takes **98 ms** to compute [106]. This is way faster than the software occupancy grid and point filtering which took 90,230 ms to complete. Once the occupancy grid is done, the point filtering will remove all unnecessary points.

Point filtering

The point filtering task uses the binary occupancy grid and removes the points not considered as obstacles. Figure 5.9 represents the 3D point cloud after the point filtering task. **205,300** points are processed within **132 ms**. Once the point filtering is done, the sliding box task follows.

Figure 5.9 – Example of the point filtering task results

Sliding box

Once the 3D point cloud has been cleared of unnecessary points, we need to apply a sliding box to isolate objects. Figure 5.10 is an example of a sliding box that measures 1×1 meters (width $\times$ height) with a step width of 1m. The mean box size is around **163 points** and the processing time is **144 ms**, which is way less compared to the software box slicing, with a processing time of **13,230 ms**.

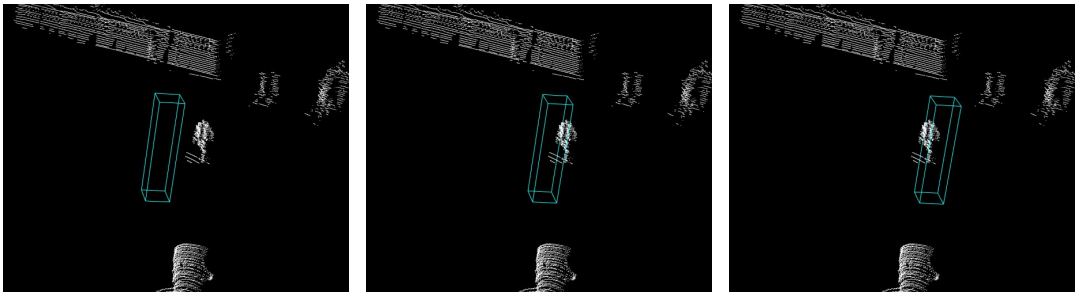
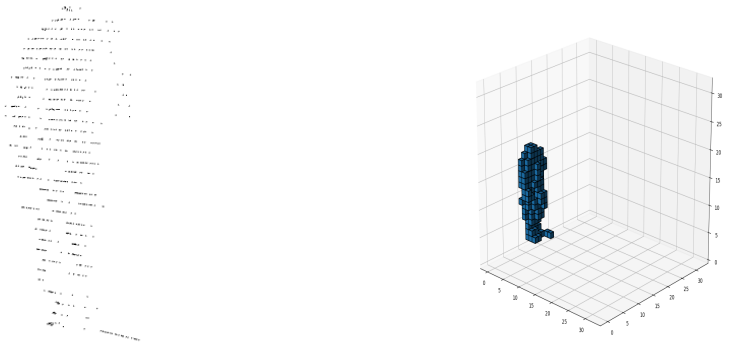


Figure 5.10 – Example of three sliding boxes on a pedestrian

Points to voxels

The input and ouput of the **Points to voxels** module is shown in Figure 5.11. The pedestrians have been extracted in boxes of $2 \times 2 \times 2$ meters then converted to a $32 \times$

32×32 voxel grid. FPGA synthesized results are shown in Table 5.2. After the hardware modules have been implemented, we tested all extracted pedestrian boxes to find the mean execution time per point (Table 5.3).



(a) LIDAR Point Cloud (b) Voxel Grid

Figure 5.11 – Pedestrian extracted from a box

Name	BRAM_18K	DSP48E	FF	LUT
Utilization	2	6	4043	7778

Table 5.2 – Hardware “points to voxels” resource utilization

Mean box size	Mean execution time	Mean time per point
163 points	880,240 ns	5,431 ns/point

Table 5.3 – Results from hardware "points to voxels" module

5.4.3.2 Neural Network Processor

The NN models for the NNP were trained using Tensorflow, the source code is available in Appendix C. The hardware used for the training was 16 GB of RAM with an Intel Core i7-8550U CPU with 4 cores, 8 threads, a base frequency of 1.8GHz up to a turbo frequency of 4GHz, as well as an 8 MB cache. Once the model has been trained, the parameters are extracted in a Numpy format. Then the embedded processing is synthesized with the NNP. The bitstream is then ported on top of the platform. The weight matrices are integrated within the system SD card along with the configuration software. In order to

evaluate the system, two tests are done: The first test is related to the SUOD dataset. The accuracy is evaluated with all classes contained in the dataset. The second test is related to the 3D Point Cloud People Dataset. We extracted all pedestrian boxes from the *Polyterrasse* set to test if they were correctly classified, which means **599** fully visible pedestrians. Thus the accuracy is related to the number of box that are correctly classified as pedestrians. Results are shown in Table 5.4. The results of the SUOD accuracy for multiple object detection are really low compared to state of the art neural networks. This is mainly due to two things: the use of Dense NN instead of CNN, and the limited number of parameters in the NN compared to the number of classified objects. But when trying to apply the same topology to only detecting pedestrians, the results are far better, which means the use of Dense NN and the number of parameters are enough to classify one type of object. The processing time of this network topology, using **4,204,160** parameters, is shown in Figure 5.12. The time per parameter is an interesting metric here, because it shows two things. First, that the use of 32-bit floating points is slowing down the processing compared to the state-of-the-art neural network accelerators. Secondly, there is still a data transfer bottleneck as pointed out in Chapter 4.

Dataset	SUOD	3D Point Cloud People Dataset
Accuracy	37.22%	93.99%

Table 5.4 – Accuracy results per dataset

5.5 Conclusion

In this chapter, we presented a smart LIDAR use-case using our methodology for FPGA DL application using a NNP. We showed the workflow of the application and described its different components. We detailed the algorithms of the embedded processing tasks and the DL architecture computed by the NNP. We finally presented detailed results for each part of the application. The embedded processing went from 103,460 ms for the software segmentation and box slicing to 344 ms using hardware threads, which represents

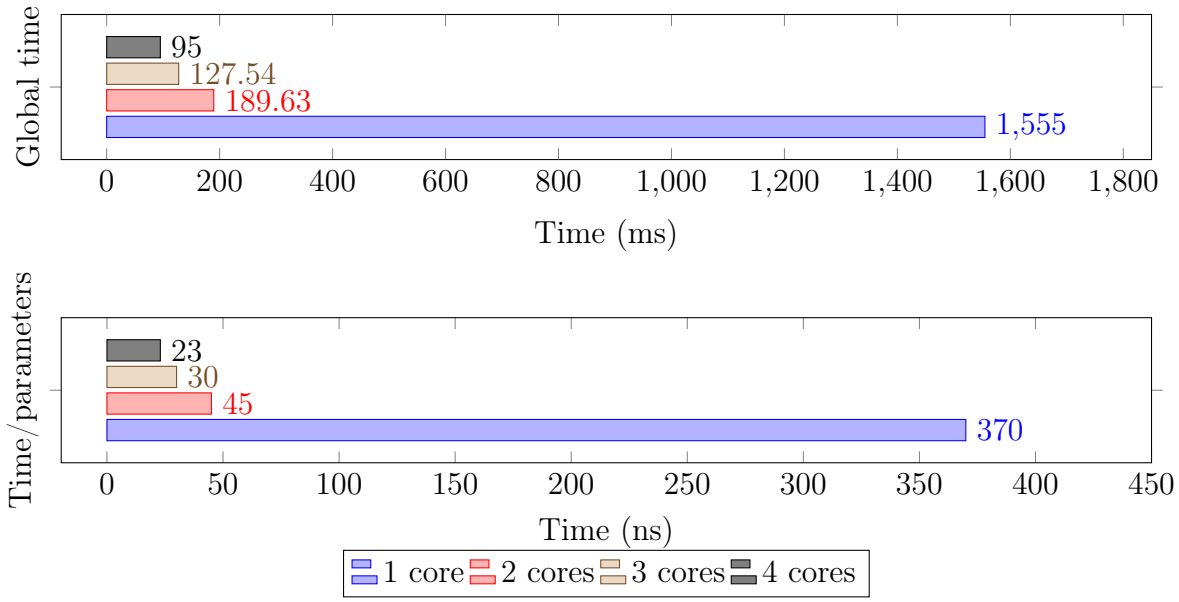


Figure 5.12 – Time performance and time per parameter for the SUOD neural network topology

a 300 times acceleration. Using our NNP, we have a low accuracy of 37.22% on the SUOD dataset, but when using the 3D Point Cloud People Dataset we reach an accuracy of 93.99%, which is enough to correctly detect pedestrians. And we can estimate the execution time of our NNP to be 23 ns per parameter of the neural network, when using four cores.

Chapter 6

General conclusion and future works

"Still sane, exile?"

Zana, Master Cartographer in Path of Exile

Contents

6.1 General conclusion 106

6.2 Future works 109

6.1 General conclusion

Cyber-Physical Systems (CPS) are a robust solution for monitoring a physical environment and responding to it with the usage of analysis algorithms such as Deep Learning (DL). But because of DL algorithms being resource greedy, hardware acceleration is a great solution to embed a DL application inside a CPS. Moreover, a prototyping methodology for DL application in CPS is needed in order to speed up the prototyping. In the context of this thesis, the CPS is represented by an autonomous vehicle and a case study is done with a smart LIDAR for pedestrian detections.

In Chapter 1, we explained the context of CPS, the use of DL applications for CPS, and the needs for a prototyping methodology for DL applications on hybrid CPU/FPGA platform-based CPS. In Chapter 2, we present: what a CPS is, what the DL hardware accelerators for CPS are, which AI algorithms to use in the context of CPS, what 3D vision techniques to use in order to detect 3D objects, and the challenge of prototyping a hardware accelerated DL application for CPS. In Chapter 3, we present our journey toward a DL methodology around a hybrid CPU/FPGA-based neural network accelerator in the context of CPS and the challenges we experienced. In Chapter 4, we present standard design for neural network accelerators and how we design our own neural network processor with its experimentation results. Finally, in Chapter 5, we present the case study of this thesis and all our experimentation results.

In this thesis, we focused on the creation of a new hardware platform-based DL co-design methodology for CPS prototyping and the design of a neural network accelerator, that we named a Neural Network Processor (NNP). We tackled the challenges of making a DL application in the context of CPS using a hybrid CPU/FPGA platform. During our exploration, we encountered several challenges around several topics such as hardware thread design, data transfer, HW/SW integration, the automation of our methodology, and so on. From those challenges, we found some solutions and we achieved the following contributions:

- A methodology to develop DL applications for CPS using a hybrid CPU/FPGA platform which is based on the usage of a neural network accelerator and a piece of automation software to speed up the prototyping time.

- A hardware Neural Network Processor (NNP) architecture, design and prototype which is used as the core of our methodology in order to simplify the deployment of DL application on a hybrid CPU/FPGA platform.

- A configuration and benchmarking software for the NNP, in order to test it and yield detailed results for analysis.

- A validation of the NNP with different configurations and results which are presented and analysed to understand the strengh and weakness of our NNP.

- An automation tool to setup a hybrid CPU/FPGA prototype board, which we used to automate some parts of our prototyping phase.

- A case study about pedestrian detection for autonomous vehicles based on a LIDAR sensor and the analysis of the 3D PointCloud from the LIDAR with a DL algorithm executed by our NNP on a hybrid CPU/FPGA platform.

- Several algorithms for pedestrian detection using a 3D PointCloud from a LIDAR that we first tried as software and hardware accelerated afterwards in order to speed up the execution time. All the results regarding those algorithms are presented: execution time as well as hardware resources.

The main difference with our work, compared to the related work presented in Chapter 2, is the methodology steps used in the design and prototyping of these new CPS systems. We shared a way to develop deep learning based CPS architectures, using Multi-CPU/FPGA-based HW/SW co-design with a neural network accelerator. In our methodology steps, we explore a way to speed up CPS systems prototyping with predetermined steps and automation. That is why we propose a methodology based on specific constraints, an

already conceived neural network accelerator and an automation software to develop a prototype. Automating Multi-CPU/FPGA based prototyping steps is one of the key parts of our design thinking. This automation is a key point for fast design and prototyping of new CPS architecture, and it allows a large test spectrum in terms of DSE (Design Space Exploration). The software source code of some automation steps are already shared on a Git repository for any design and reuse. Automating the methodology is an important issue toward decreasing the time-to-prototype. Moreover, automation can help reduce the required skills necessary for this kind of task, and thus make it more accessible. We believe that each part of the design steps for the methodology can be automated. The deep learning software is already simplified with the use of either programming tools or visual editors to develop a deep learning architecture. Deep learning hardware IP reuse in the context of CPS design and prototyping automation is a motivation for faster hardware development. In our methodology, the neural network accelerator IP (Intellectual Property) is considered as a pre-designed component from the IP library that can be used for the automation process (we also shared our neural network accelerator source code as a reusable IP component). Moreover, the specific use case "a smart LIDAR for pedestrian detection" is in here just as a specific example in order to demonstrate our methodology work. It is a better way for readers to have a deep understanding of how to use our methodology and our different developed tools. So, they can use our methodology to develop their own CPS applications using FPGA-based HW/SW co-design and neural network accelerator. Again, our work is oriented toward new automated methodology steps and new approaches to designing and prototyping an embedded DL application on a hybrid FPGA/CPU platform for CPS systems. Our approach is based on an existing platform (Xilinx Zynq Platform). Even if, and as we mention, we were inspired by some very interesting related work at the neural network accelerator architecture level, the automation of the design and prototyping steps of DL-based CPS systems using hybrid CPU/FPGA architecture is the main aim and goal of our work. We believe that this automation is the key step of better design and fast prototyping.

6.2 Future works

In our future works, we plan to completely automate our methodology with specific tools to easily synthesize the hardware threads and deploy the software system onto the hardware prototype, as well as a software tool to predict the performance of the NNP and implement the correct number of cores depending of the required performances.

Furthermore, automation tools would be a must have combined with the use of design reuse toward simplifying the prototyping of a hyrbid CPU/FPGA-based DL application in a CPS context. By this, we mean a way to define different IPs and state their work flow to automate the generation of an RTL model to use on the FPGA. The ultimate automation goal would be an interface for a DL application developer to specify the different hardware threads, in order to process data between the CPS input and the NNP. Then, depending on some time constraints, the number of NNP cores would be chosen to reach this constraint. The whole system would finally be synthesized and ready to be implemented on a hardware prototype.

Finally, we plan to improve the NNP performances, specifically by refactoring the data transfer and the calculation bottlenecks, as well as trying to use fixed point integers or binary data types. We also plan to improve the NNP capability with more activation functions and neural network topology available, such as Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN).

Publications

Cabanes, Q., & Senouci, B. (2017, July). Objects detection and recognition in smart vehicle applications: Point cloud based approach. In 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN) (pp. 287-289). IEEE. https://doi.org/10.1109/ICUFN.2017.7993795

Abid, O., Cabanes, Q., & Senouci, B. (2018, March). Supervisor and control investigation in smart/autonomous vehicles: Environment recognition and objects detection ADAS application case study. In 2018 11th International Symposium on Mechatronics and its Applications (ISMA) (pp. 1-7). IEEE. https://doi.org/10.1109/ISMA.2018.8330135

Cabanes, Q., Senouci, B., & Ramdane-Cherif, A. (2019, February). A Complete Multi-CPU/FPGA-based Design and Prototyping Methodology for Autonomous Vehicles: Multiple Object Detection and Recognition Case Study. In 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC) (pp. 158-163). IEEE. https://doi.org/10.1109/ICAIIC.2019.8669047

Senouci, B., Rouis, H., Cabanes, Q., Ramdan, A. C., & Han, D. S. (2019). HW/SW Co-design and Prototyping Approach for Embedded Smart Camera: ADAS Case Study. Journal of Telecommunication, Electronic and Computer Engineering (JTEC), 11(4), 31-40. http://journal.utem.edu.my/index.php/jtec/issue/view/242

Cabanes Q, Senouci B, Ramdane-Cherif A. Embedded Deep Learning Prototyping Approach for Cyber-Physical Systems: Smart LIDAR Case Study. Journal of Sensor and Actuator Networks. 2021; 10(1):18. https://doi.org/10.3390/jsan10010018

Appendix A

SystemC source code of the Neural Network Processor

Listing A.1 - Scheduler header code

```
#ifndef SCHEDULER_HPP
#define SCHEDULER_HPP

#include <systemc.h>
#include "verbose.hpp"

#define CORE 4
#define INSTRUCTION_BUFFER 512
#define INPUT_BUFFER 65536

SC_MODULE(scheduler_module)
{
    // PORTS
    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_fifo_in<float> from_dma_weight;
    sc_fifo_in<float> from_dma_input;
    sc_fifo_in< sc_uint<64> > from_dma_instructions;
```

```
19      sc_fifo_out<float> to_dma;
20
21      // PROCESSING ENGINES
22      sc_fifo_out< sc_uint<34> > npu_instructions[CORE];
23      sc_fifo_out<float> npu_weight[CORE];
24      sc_fifo_out<float> npu_input[CORE];
25      sc_fifo_in<float> npu_output[CORE];
26
27      // STATES
28      sc_uint<64> state_instruction_counter;
29      sc_uint<64> state_instruction_buffer[INSTRUCTION_BUFFER];
30      float state_input_buffer[INPUT_BUFFER];
31      float state_output_buffer[INPUT_BUFFER];
32
33      // PROCESS
34      void process(void);
35
36      SC_CTOR(scheduler_module)
37      {
38          // Init STATES
39          state_instruction_counter = 0;
40
41          SC_CTHREAD(process, clk.pos());
42          reset_signal_is(reset, true);
43      }
44  };
45
46  #endif
```

Listing A.2 - Scheduler source code

```
1   #include "../headers/scheduler.hpp"
2
3   void scheduler_module::process(void)
4   {
```

```
#pragma HLS resource core = AXI4Stream variable = from_dma_weight
#pragma HLS resource core = AXI4Stream variable = from_dma_input
#pragma HLS resource core = AXI4Stream variable = from_dma_instructions
#pragma HLS resource core = AXI4Stream variable = to_dma

#pragma HLS array_partition variable = npu_instructions complete dim = 1
#pragma HLS array_partition variable = npu_weight complete dim = 1
#pragma HLS array_partition variable = npu_output complete dim = 1
#pragma HLS array_partition variable = npu_input complete dim = 1

#pragma HLS resource core = AXI4Stream variable = npu_instructions
#pragma HLS resource core = AXI4Stream variable = npu_weight
#pragma HLS resource core = AXI4Stream variable = npu_input
#pragma HLS resource core = AXI4Stream variable = npu_output

  sc_uint<30> instruction_input_layer, instruction_output_layer;

  while (true)
  {
    // Load instructions
    from_dma_instructions.read(state_instruction_counter);
    for (unsigned int i = 0; i < state_instruction_counter; i++)
    {
      state_instruction_buffer[i] = from_dma_instructions.read();
    }
    instruction_input_layer = (state_instruction_buffer[0]) >> 34;
    instruction_output_layer = state_instruction_buffer[
   state_instruction_counter - 1] >> 4;

    // Load inputs
    for (unsigned int i = 0; i < instruction_input_layer; i++)
    {
      state_input_buffer[i] = from_dma_input.read();
    }
```

```
38
39      // Process neural network
40      for (unsigned int instruction_index = 0; instruction_index <
    state_instruction_counter; instruction_index++)
41      {
42        sc_uint<30> state_current_length = state_instruction_buffer[
    instruction_index] >> 34;
43        sc_uint<30> state_next_length = state_instruction_buffer[
    instruction_index] >> 4;
44        sc_uint<4> state_activation_function = state_instruction_buffer[
    instruction_index] & 0b1111;
45
46  #ifndef __SYNTHESIS__
47  #if VERBOSITY_LEVEL >= 2
48        cout << "[scheduler_module] @" << sc_time_stamp() << " inputs loaded
    (" << state_current_length << ")" << endl;
49        cout << "[scheduler_module] @" << sc_time_stamp() << " next loaded ("
     << state_next_length << ")" << endl;
50  #endif
51  #endif
52
53        // Schedule
54        for (unsigned int core_done = 0; core_done < state_next_length;
    core_done += CORE)
55        {
56          // Load cores
57          // If there is less nodes than cores, do not start unused cores
58          for (unsigned int i = 0; i + core_done < state_next_length && i <
    CORE; i++)
59          {
60            npu_instructions[i].write((state_current_length << 4) +
    state_activation_function);
61          }
62
```

```
63          // Send data to cores
64          for (unsigned int current_counter = 0; current_counter <
        state_current_length; current_counter++)
65          {
66  #pragma HLS pipeline II = 1 enable_flush
67            for (unsigned int i = 0; i + core_done < state_next_length && i <
         CORE; i++)
68            {
69              npu_weight[i % CORE].write(from_dma_weight.read());
70              npu_input[i % CORE].write(state_input_buffer[current_counter %
        state_current_length]);
71            }
72          }
73
74  // Return output
75  #pragma HLS pipeline II = 1 enable_flush
76          for (unsigned int i = 0; i + core_done < state_next_length && i <
        CORE; i++)
77          {
78            state_output_buffer[i + core_done] = npu_output[i % CORE].read();
79          }
80        }
81
82        // Process activation function that need full output vector
83        if (state_activation_function == 3) // Softmax
84        {
85          float sum = 0;
86          for (unsigned int i = 0; i < state_next_length; i++)
87          {
88            sum += state_output_buffer[i];
89          }
90
91          for (unsigned int i = 0; i < state_next_length; i++)
92          {
```

```
93              state_output_buffer[i] /= sum;
94            }
95          }
96
97          for (unsigned int i = 0; i < state_next_length; i++)
98          {
99            state_input_buffer[i] = state_output_buffer[i];
100         }
101       }
102
103       for (unsigned int i = 0; i < instruction_output_layer; i++)
104       {
105         to_dma.write(state_input_buffer[i]);
106
107         // This is for Vivado for the DMA IP
108         // if (i < (instruction_output_layer - 1))
109         //  to_dma_TLAST.write(0);
110         // else {
111         //  to_dma_TLAST.write(1);
112         //  wait();
113         // }
114       }
115
116       // This is for Vivado for the DMA IP
117       // wait();
118       // to_dma_TLAST.write(0);
119     }
120 }
```

Listing A.3 - Neuron processing engine header code

```
1 #ifndef PROCESSING_ENGINE_HPP
2 #define PROCESSING_ENGINE_HPP
3
4 #include <systemc.h>
```

```
5  #include "verbose.hpp"
6
7  SC_MODULE(processing_engine_module)
8  {
9      // PORTS
10     sc_in<bool> clk;
11     sc_in<bool> reset;
12     sc_fifo_in<float> from_scheduler_weight;
13     sc_fifo_in<float> from_scheduler_input;
14     sc_fifo_in< sc_uint<34> > from_scheduler_instructions;
15     sc_fifo_out<float> to_scheduler;
16
17     // STATES
18     sc_uint<30> state_length;
19     sc_uint<4> state_activation_function;
20
21     // PROCESS
22     void process(void);
23
24     // UTIL
25     float sigmoid(float input);
26     float relu(float input);
27     float softmax(float input);
28
29     SC_CTOR(processing_engine_module)
30     {
31         state_length = 0;
32         state_activation_function = 0;
33
34         SC_CTHREAD(process, clk.pos());
35         reset_signal_is(reset,true);
36     }
37 };
38
```

```
#endif
```

Listing A.4 - Neuron processing engine source code

```
#include "../headers/processing_engine.hpp"

void processing_engine_module::process(void)
{
#pragma HLS resource core = AXI4Stream variable = from_scheduler_weight
#pragma HLS resource core = AXI4Stream variable = from_scheduler_input
#pragma HLS resource core = AXI4Stream variable =
    from_scheduler_instructions
#pragma HLS resource core = AXI4Stream variable = to_scheduler

    // Init
    sc_uint<34> instructions;
    float input, weight, output;

    while (true)
    {
        output = 0.f;
        from_scheduler_instructions.read(instructions);
        state_length = instructions >> 4;
        state_activation_function = instructions & 0b1111;

#ifndef __SYNTHESIS__
#if VERBOSITY_LEVEL >= 2
        cout << "[processing_engine_module] @" << sc_time_stamp() << "
    loading length (" << state_length << ") and activation (#" <<
    state_activation_function << ")" << endl;
#endif
#endif

// Process
#pragma HLS pipeline II = 1 enable_flush
```

```
29          for (unsigned int i = 0; i < state_length; i++)
30          {
31              from_scheduler_input.read(input);
32              from_scheduler_weight.read(weight);
33              output += weight * input;
34          }
35
36          if (state_activation_function == 1) // Sigmoid
37          {
38              output = sigmoid(output);
39          }
40          else if (state_activation_function == 2) // Relu
41          {
42              output = relu(output);
43          }
44          else if (state_activation_function == 3) // Softmax
45          {
46              output = softmax(output);
47          }
48
49          to_scheduler.write(output);
50
51  #ifndef __SYNTHESIS__
52  #if VERBOSITY_LEVEL >= 2
53          cout << "[processing_engine_module] @" << sc_time_stamp() << "
        returning result (" << output << ")" << endl;
54  #endif
55  #endif
56      }
57  }
58
59  float processing_engine_module::sigmoid(float input)
60  {
61      return 1.f / (1.f + exp(-input));
```

```
62 }
63
64 float processing_engine_module::relu(float input)
65 {
66     return input < 0.f ? 0.f : input;
67 }
68
69 float processing_engine_module::softmax(float input)
70 {
71     return exp(input);
72 }
```

Appendix B

Tensorflow source code for Dense Neural Network topologies

Listing B.1 - Tensorflow source code for MNIST

```
from __future__ import absolute_import, division, print_function

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image;
from tensorflow.keras.models import load_model

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

batch_size = 128
epochs = 20

# sess = tf.compat.v1.InteractiveSession()
dataset = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = dataset.load_data()
```

```
19
20 # # Normalize pixel values to be between 0 and 1
21 x_train = x_train / 255.0;
22 x_test = x_test / 255.0;
23
24 # # Flatten images
25 x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.
       shape[2])
26 x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])
27
28 # # Create and train model
29 model = keras.Sequential([
30     keras.layers.Dense(32, activation=keras.activations.relu, use_bias=
       False),
31     keras.layers.Dense(16, activation=keras.activations.linear, use_bias=
       False),
32     keras.layers.Dense(32, activation=keras.activations.relu, use_bias=
       False),
33     keras.layers.Dense(10, activation=keras.activations.softmax, use_bias=
       False)
34 ])
35 model.compile(optimizer='adam',
36               loss='mean_squared_error',
37               # loss='sparse_categorical_crossentropy',
38               metrics=['accuracy'])
39 model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs)
40
41 # Saving weights and model
42 model.save("model.h5");
43
44 # Evaluation
45 print('\n# Evaluate on test data')
46 results = model.evaluate(x_test, y_test, batch_size=batch_size)
47 print('test loss, test acc:', results)
```

Listing B.2 - Tensorflow source code for CIFAR10

```
from __future__ import absolute_import, division, print_function
import tensorflow as tf
from tensorflow import keras
import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

batch_size = 500
epochs = 100

dataset = keras.datasets.cifar10
(x_train, y_train), (x_test, y_test) = dataset.load_data()

# Grayscale
x_train = rgb2gray(x_train);
x_test = rgb2gray(x_test);

# Normalize pixel values to be between 0 and 1
x_train = x_train / 255.0;
x_test = x_test / 255.0;

# Flatten images
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.
    shape[2])
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])

# Create and train model
model = keras.Sequential([
    keras.layers.Dense(512, activation=keras.activations.relu, use_bias=
    False),
```

```
30      keras.layers.Dense(128, activation=keras.activations.linear, use_bias=
        False),
31      keras.layers.Dense(512, activation=keras.activations.relu, use_bias=
        False),
32      keras.layers.Dense(128, activation=keras.activations.linear, use_bias=
        False),
33      keras.layers.Dense(512, activation=keras.activations.relu, use_bias=
        False),
34      keras.layers.Dense(10, activation=keras.activations.softmax, use_bias=
        False)
35  ])
36  model.compile(optimizer='adam',
37                loss='sparse_categorical_crossentropy',
38                metrics=['accuracy'])
39  model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs)
40
41  # Saving model
42  model.save("model.h5");
43
44  # Evaluation
45  print('\n# Evaluate on test data')
46  results = model.evaluate(x_test, y_test, batch_size=batch_size)
47  print('test loss, test acc:', results)
```

Listing B.3 - Tensorflow source code for CIFAR100

```
1  from __future__ import absolute_import, division, print_function
2  import tensorflow as tf
3  from tensorflow import keras
4  import numpy as np
5
6  def assert_parameters(model):
7      max_params = 8388608
8      params = 0
9
```

```
    model.build(x_train.shape)
    for layer in model.layers:
        params += layer.count_params()

    if params > max_params:
        print("Too many parameters ({})".format(params))
        exit(0)

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

batch_size = 500
epochs = 100

dataset = keras.datasets.cifar100
(x_train, y_train), (x_test, y_test) = dataset.load_data()

# Grayscale
x_train = rgb2gray(x_train);
x_test = rgb2gray(x_test);

# Normalize pixel values to be between 0 and 1
x_train = x_train / 255.0;
x_test = x_test / 255.0;

# Flatten images
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] * x_train.
    shape[2])
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] * x_test.shape[2])

# Create and train model
model = keras.Sequential([
    keras.layers.Dense(1024, activation=keras.activations.relu, use_bias=
    False),
```

```
42      keras.layers.Dense(256, activation=keras.activations.linear, use_bias=
        False),
43      keras.layers.Dense(1024, activation=keras.activations.relu, use_bias=
        False),
44      keras.layers.Dense(256, activation=keras.activations.linear, use_bias=
        False),
45      keras.layers.Dense(1024, activation=keras.activations.relu, use_bias=
        False),
46      keras.layers.Dense(100, activation=keras.activations.sigmoid, use_bias=
        False)
47  ])
48  model.compile(optimizer='adam',
49                loss='sparse_categorical_crossentropy',
50                metrics=['accuracy'])
51  assert_parameters(model)
52  model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs)
53
54  # Saving model
55  model.save("model.h5");
56
57  # Evaluation
58  print('\n# Evaluate on test data')
59  results = model.evaluate(x_test, y_test, batch_size=batch_size)
60  print('test loss, test acc:', results)
```

Appendix C

Tensorflow source code for SUOD dataset

Listing C.1 - Tensorflow source code for SUOD training

```
from __future__ import absolute_import, division, print_function,
    unicode_literals

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import load_model

BATCH = 64
EPOCHS = 20

sess = tf.compat.v1.InteractiveSession()
with np.load('suod_dataset.npz') as data:
    x_train = data['x_train']
    y_train = data['y_train']
    x_test = data['x_test']
    y_test = data['y_test']

```

```
model = keras.Sequential([
    keras.layers.Dropout(0.5),
    keras.layers.Dense(128, activation='relu', use_bias=False),
    keras.layers.Dense(32, activation='linear', use_bias=False),
    keras.layers.Dense(128, activation='relu', use_bias=False),
    keras.layers.Dense(13, activation='softmax', use_bias=False),
]);
model.compile(optimizer="adam",
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=BATCH, epochs=EPOCHS)

## Saving weights and model
model.save("model.h5");

## Evaluation
print('\n# Evaluate on test data')
results = model.evaluate(x_test, y_test, batch_size=BATCH)
print('test loss, test acc:', results)
```

Listing C.2 - Tensorflow source code for SUOD weight export

```
from tensorflow import keras
from tensorflow.keras.models import load_model
import numpy as np
import re

with np.load('suod_dataset.npz') as data:
    np.savez("dataset.npz", x=data['x_test'], y=data['y_test'])

# Export layers
model = load_model('model.h5');
pattern = r'<function ([a-z]+) at .*>'
layers_data = {}
function_counter = {}
```

```
w_size = 0
for i, layer in enumerate(model.layers):
    try:
        activation = str(layer.activation)
        m = re.match(pattern, activation)
        function_name = m.group(1)
        W = layer.get_weights()
        w_size += W[0].shape[0] * W[0].shape[1];
        function_counter[function_name] = function_counter.get(function_name, -1) + 1;
        key = "a{}_{}_{}".format(i, function_name, function_counter[function_name])
        layers_data[key] = np.array(W[0], dtype="f");
        print("{} ({})".format(W[0].shape[1], function_name))
    except:
        pass
np.savez("layers.npz", **layers_data)

print("Number of parameters: {}".format(w_size))
print("Number of bytes: {}".format(w_size * 4))
print("Estimated time: {} us".format(w_size * 0.023))
```

Appendix D

Résumé de thèse

Les Systèmes Cyber-Physiques (SCP) sont un sujet de recherche mature qui interagissent avec l'intelligence artificielle (IA) et les systèmes embarqués (SE). Un SCP peut être défini comme un SE en réseau qui peut analyser un environnement physique et prendre des décisions à partir de son état actuel pour affecter son environnement physique vers un résultat souhaité, via des actionneurs. Les SCP interagissent avec le monde physique via des capteurs/actionneurs pour résoudre des problèmes dans plusieurs applications (robotique, transport, santé, etc.). Ils nécessitent des algorithmes d'analyse de données puissants associés à des architectures matérielles robustes. D'une part, l'Ap-prentissage en Profondeur (AP) est proposé comme algorithme principal. D'autre part, les méthodologies de conception et de prototypage standard pour SE ne sont pas adaptées au SCP moderne basé sur de l'AP, et plus particulièrement, des accélérateurs matériels d'AP. En effet, l'intégration des applications d'AP dans un SCP s'accompagne de nombreuses contraintes qui sont soit en conflit avec les performances de calcul, en raison des contraintes de ressources de calcul nécessaires, soit avec le temps de prototypage, en raison de l'utilisation d'une plate-forme logicielle/matérielle spéci-fique. Dans le cas d'une plate-forme matérielle/logicielle utilisant des accélérateurs de réseau de neu-rones, nous avons observé un manque de méthodologie définie dans le contexte du SCP basé sur les plate-formes hybride CPU/FPGA (Field-Programmable Gate Array). Ce type de plate-forme

est en mesure d'apporter un moyen simple de prototyper des algorithmes matériels spécialisés dans le calcul de réseau de neurones avec la flexibilité de l'automatisation logicielle et une plus grande rapidité de prototypage comparé à une plateforme purement matérielle. Notre conviction est qu'une méthodolo-gie mélangeant les principes matériels tels que la réutilisation de conception, et les principes logi-cielles tels que l'automatisation de la conception, pourrait réduire le temps de prototypage des SCP basés sur des applications d'AP, tout en étant capable d'optimiser le temps de calcul.

Dans cette thèse, nous étudions la conception d'IA pour SCP autour d'application d'AP embarquée avec une plate-forme hybride CPU/FPGA. Nous proposons une méthodologie de co-conception matérielle/logicielle pour développer des applications d'AP pour SCP qui est basée sur l'utilisation d'un accélérateur de réseau de neurones et d'un logiciel d'automatisation des étapes de la méthodo-logie pour accélérer le temps de prototypage. L'accélérateur de réseau de neurones fonctionne comme un outil faisant partie intégrante de la méthodologie afin de simplifier le transfert du logiciel vers le matériel pour les applications d'AP. Le logiciel d'automatisation fonctionne comme un outil de support à cette méthodologie dont l'objectif est de faciliter les différentes tâches de prototypages qui nécessitent souvent des compétences de très bas niveau. Nous présentons ensuite la conception et le prototypage de notre accélérateur matériel de réseau de neurones ainsi que le résultat de ses performances. De plus, le code source de cet accélérateur de réseau de neurone est accessible en source libre. Cet accélérateur est capable de traiter des réseaux de neurones entièrement connecté sans biais avec un choix de fonctions d'activation limité (linéaire, relu, sigmoid, softmax). Des tests sont effectués sur notre accélérateur de réseau de neurone avec des base de données connu de la communauté scientifiques, tels que MNIST ou CIFAR10. Nous estimons le temps de calcul de notre accélérateur à 23 ns par paramètres du modèle d'AP, en prenant en compte le temps de transfert des données dans le bus système de la plateforme. Enfin, nous validons notre travail à l'aide d'un cas d'usage : un LIDAR (LIght Detection And Ranging) intelligent pour de la détection d'objet autour du véhicule autonome. L'objectif est d'utiliser un capteur 3D tel qu'un

LIDAR pour détecter direc-tement des objets, plus spécifiquement des piétons, et leurs positions dans l'espace 3D en sortie de ce capteur, au lieu de retourner des données cartographiques 3D. Ce cas d'usage est accompagné de plusieurs algorithmes de détection de piétons à l'aide du nuage de points 3D provenant d'un LI-DAR. Les données 3D du LIDAR sont traitées afin de les transformés en une grille de voxels 3D puis sont envoyé à notre accélérateur de réseau de neurone pour effectuer la classification de chaque objet détecté. Les données du LIDAR proviennent d'une base de données utilisant un vrai LIDAR sur une place piétonne et l'entrainement de notre modèle d'AP est effectué sur une base de données classifiant de multiples objets urbains afin de se rapprocher le plus possible d'un cas d'étude réel.

References

1. Lee, E. A. Cyber Physical Systems: Design Challenges. en, 10 (Jan. 2008).

2. Klotzer, C., WeiBenborn, J. & Pflaum, A. *The Evolution of Cyber-Physical Systems as a Driving Force Behind Digital Transformation* en. in *2017 IEEE 19th Conference on Business Informatics (CBI)* (IEEE, Thessaloniki, July 2017), 5–14. ISBN: 978-1-5386-3035-8. `http://ieeexplore.ieee.org/document/8012392/` (2020).

3. Hehenberger, P. *et al.* Design, modelling, simulation and integration of cyber physical systems: Methods and applications. en. *Computers in Industry* **82,** 273–289. ISSN: 01663615. `https://linkinghub.elsevier.com/retrieve/pii/S0166361516300902` (2020) (Oct. 2016).

4. Inggs, G., Fleming, S., Thomas, D. & Luk, W. Is High Level Synthesis ready for business? A computational finance case study. en, 8 (2014).

5. Bengler, K. *et al.* Three Decades of Driver Assistance Systems: Review and Future Perspectives. en. *IEEE Intelligent Transportation Systems Magazine* **6,** 6–22. ISSN: 1939-1390. `http://ieeexplore.ieee.org/document/6936444/` (2019) (2014).

6. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. en. *Nature* **521,** 436–444. ISSN: 0028-0836, 1476-4687. `http://www.nature.com/articles/nature14539` (2019) (May 2015).

7. Wickramasinghe, C. S., Marino, D. L., Amarasinghe, K. & Manic, M. *Generalization of Deep Learning for Cyber-Physical System Security: A Survey* en. in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society* (IEEE, Washington, DC, Oct. 2018), 745–751. ISBN: 978-1-5090-6684-1. `https://ieeexplore.ieee.org/document/8591773/` (2019).

8. Arif, A. *et al.* Performance and energy-efficient implementation of a smart city application on FPGAs. en. *Journal of Real-Time Image Processing.* ISSN: 1861-

8200, 1861-8219. http://link.springer.com/10.1007/s11554-018-0792-x (2020) (June 2018).

9. Iandola, F. & Keutzer, K. Small neural nets are beautiful: enabling embedded systems with small deep-neural-network architectures. en, 10. https://ieeexplore.ieee.org/abstract/document/8101283 (2017).

10. Wang, C. *et al.* DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. en. *arXiv:1605.06894 [cs].* arXiv: 1605.06894. http://arxiv.org/abs/1605.06894 (2019) (May 2016).

11. Birk, M., Zapf, M., Balzer, M., Ruiter, N. & Becker, J. A comprehensive comparison of GPU- and FPGA-based acceleration of reflection image reconstruction for 3D ultrasound computer tomography. en. *Journal of Real-Time Image Processing* **9,** 159–170. ISSN: 1861-8200, 1861-8219. http://link.springer.com/10.1007/s11554-012-0267-4 (2020) (Mar. 2014).

12. Sze, V., Chen, Y.-H., Yang, T.-J. & Emer, J. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. en. *arXiv:1703.09039 [cs].* arXiv: 1703.09039. http://arxiv.org/abs/1703.09039 (2020) (Aug. 2017).

13. Sangiovanni-Vincentelli, A. & Martin, G. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers* **18** (Nov. 2001).

14. ETSI, E. E. 302 665 V1. 1.1: Intelligent transport systems (ITS), communications architecture. *European Standard (Telecommunications Series)(September 2010)* (2010).

15. Committee, I. C. S. L. S. IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11^<TM>.* https://ci.nii.ac.jp/naid/10030068811/en/ (2007).

16. Bouchemal, N. *Quality of Service Provisioning and Performance Analysis in Vehicular Network* en. PhD thesis (June 2015).

17. Jensen, J. C., Chang, D. H. & Lee, E. A. *A model-based design methodology for cyber-physical systems* en. in *2011 7th International Wireless Communications and Mobile Computing Conference* (IEEE, Istanbul, Turkey, July 2011), 1666–1671. ISBN: 978-1-4244-9539-9. http://ieeexplore.ieee.org/document/5982785/ (2020).

18. Shi, J., Wan, J., Yan, H. & Suo, H. *A survey of Cyber-Physical Systems* en. in *2011 International Conference on Wireless Communications and Signal Processing (WCSP)* (IEEE, Nanjing, China, Nov. 2011), 1–6. ISBN: 978-1-4577-1010-0 978-1-4577-1009-4 978-1-4577-1007-0 978-1-4577-1008-7. http://ieeexplore.ieee.org/document/6096958/ (2020).

19. Sangiovanni-Vincentelli, A., Damm, W. & Passerone, R. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*. en. *European Journal of Control* **18,** 217–238. ISSN: 09473580. https://linkinghub.elsevier.com/retrieve/pii/S0947358012709433 (2020) (Jan. 2012).

20. Sztipanovits, J. *et al.* Toward a Science of Cyber–Physical System Integration. en. *Proceedings of the IEEE* **100,** 29–44. ISSN: 0018-9219, 1558-2256. http://ieeexplore.ieee.org/document/6008519/ (2020) (Jan. 2012).

21. Sztipanovits, J. *et al.* Model and Tool Integration Platforms for Cyber–Physical System Design. en. *Proceedings of the IEEE* **106,** 1501–1526. ISSN: 0018-9219, 1558-2256. https://ieeexplore.ieee.org/document/8396214/ (2020) (Sept. 2018).

22. Pinto, A., Bonivento, A., Sangiovanni-Vincentelli, A. L., Passerone, R. & Sgroi, M. System level design paradigms: Platform-based design and communication synthesis. en. *ACM Transactions on Design Automation of Electronic Systems* **11,** 537–563. ISSN: 10844309. http://portal.acm.org/citation.cfm?doid=1142980.1142982 (2019) (July 2006).

23. Nuzzo, P., Sangiovanni-Vincentelli, A. L., Bresolin, D., Geretti, L. & Villa, T. A Platform-Based Design Methodology With Contracts and Related Tools for the Design of Cyber-Physical Systems. en. *Proceedings of the IEEE* **103,** 2104–2132. ISSN: 0018-9219, 1558-2256. http://ieeexplore.ieee.org/document/7268792/ (2020) (Nov. 2015).

24. Guo, K., Zeng, S., Yu, J., Wang, Y. & Yang, H. [DL] A Survey of FPGA-based Neural Network Inference Accelerators. en. *ACM Transactions on Reconfigurable Technology and Systems* **12,** 1–26. ISSN: 19367406. http://dl.acm.org/citation.cfm?doid=3310278.3289185 (2019) (Mar. 2019).

25. Li, L. *et al.* An integrated hardware/software design methodology for signal processing systems. en. *Journal of Systems Architecture* **93,** 1–19. ISSN: 13837621. https://linkinghub.elsevier.com/retrieve/pii/S1383762118301735 (2020) (Feb. 2019).

26. Shawahna, A., Sait, S. M. & El-Maleh, A. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. en. *IEEE Access* **7,** 7823–7859. ISSN: 2169-3536. https://ieeexplore.ieee.org/document/8594633/ (2020) (2019).

27. Sackinger, E., Boser, B., Bromley, J., LeCun, Y. & Jackel, L. Application of the ANNA neural network chip to high-speed character recognition. *IEEE Transactions on Neural Networks* **3,** 498–505. ISSN: 10459227. http://ieeexplore.ieee.org/document/129422/ (2019) (May 1992).

28. Säckinger, E., Boser, B. E. & Jackel, L. D. A Neurocomputer Board Based on the ANNA Neural Network Chip. en, 8 (1992).

29. Botros, N. & Abdul-Aziz, M. Hardware implementation of an artificial neural network using field programmable gate arrays (FPGA's). en. *IEEE Transactions on Industrial Electronics* **41,** 665–667. ISSN: 02780046. http://ieeexplore.ieee.org/document/334585/ (2020) (Dec. 1994).

30. Ferrer, D., Gonzalez, R., Fleitas, R., Acle, J. & Canetti, R. *NeuroFPGA-implementing artificial neural networks on programmable logic devices* en. in *Proceedings Design, Automation and Test in Europe Conference and Exhibition* (IEEE Comput. Soc, Paris, France, 2004), 218–223. ISBN: 978-0-7695-2085-8. http://ieeexplore.ieee.org/document/1269233/ (2019).

31. *FPGA implementations of neural networks* en (eds Ormondi, A. R. & Rajapakse, J. C.) OCLC: ocm61477766. ISBN: 978-0-387-28485-9 978-0-387-28487-3 (Springer, Dordrecht, The Netherlands, 2006).

32. Sahin, S., Becerikli, Y. & Yazici, S. en. in *Neural Information Processing* (eds Hutchison, D. *et al.*) 1105–1112 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). ISBN: 978-3-540-46484-6 978-3-540-46485-3. http://link.springer.com/10.1007/11893295_122 (2020).

33. Farabet, C., Poulet, C. & LeCun, Y. *An FPGA-based stream processor for embedded real-time vision with Convolutional Networks* en. in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops* (IEEE, Kyoto, Sept. 2009), 878–885. ISBN: 978-1-4244-4442-7. http://ieeexplore.ieee.org/document/5457611/ (2019).

34. Farabet, C., Poulet, C., Han, J. Y. & LeCun, Y. *CNP: An FPGA-based processor for Convolutional Networks* en. in *2009 International Conference on Field Programmable Logic and Applications* (IEEE, Prague, Czech Republic, Aug. 2009), 32–37. http://ieeexplore.ieee.org/document/5272559/ (2019).

35. Farabet, C. *et al. Hardware accelerated convolutional neural networks for synthetic vision systems* en. in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (IEEE, Paris, France, May 2010), 257–260. ISBN: 978-1-4244-5308-5. http://ieeexplore.ieee.org/document/5537908/ (2019).

36. Farabet, C. *et al. NeuFlow: A runtime reconfigurable dataflow processor for vision* en. in *CVPR 2011 WORKSHOPS* (IEEE, Colorado Springs, CO, USA, June 2011),

109–116. ISBN: 978-1-4577-0529-8. http://ieeexplore.ieee.org/document/5981829/ (2019).

37. Farabet, C. *et al.* en. in *Scaling Up Machine Learning* (eds Bekkerman, R., Bilenko, M. & Langford, J.) 399–419 (Cambridge University Press, Cambridge, 2011). ISBN: 978-1-139-04291-8. https://www.cambridge.org/core/product/identifier/CBO9781139042918A158/type/book_part (2019).

38. Pham, P.-H. *et al. NeuFlow: Dataflow vision processing system-on-a-chip* en. in *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)* (IEEE, Boise, ID, USA, Aug. 2012), 1044–1047. ISBN: 978-1-4673-2527-1 978-1-4673-2526-4 978-1-4673-2525-7. http://ieeexplore.ieee.org/document/6292202/ (2019).

39. Esmaeilzadeh, H., Sampson, A., Ceze, L. & Burger, D. Neural acceleration for general-purpose approximate programs. en. *Communications of the ACM* **58,** 105–115. ISSN: 00010782. http://dl.acm.org/citation.cfm?doid=2688498.2589750 (2019) (Dec. 2014).

40. Lozito, G.-M., Laudani, A., Riganti Fulginei, F. & Salvini, A. FPGA Implementations of Feed Forward Neural Network by using Floating Point Hardware Accelerators. en. *Advances in Electrical and Electronic Engineering* **12,** 30–39. ISSN: 1804-3119, 1336-1376. http://advances.utc.sk/index.php/AEEE/article/view/831 (2020) (Mar. 2014).

41. Chen, T. *et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning* en. in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14* (ACM Press, Salt Lake City, Utah, USA, 2014), 269–284. ISBN: 978-1-4503-2305-5. http://dl.acm.org/citation.cfm?doid=2541940.2541967 (2019).

42. Zhang, C. *et al. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks* en. in *Proceedings of the 2015 ACM/SIGDA International Sym-*

posium on Field-Programmable Gate Arrays - FPGA '15 (ACM Press, Monterey, California, USA, 2015), 161–170. ISBN: 978-1-4503-3315-3. http://dl.acm.org/citation.cfm?doid=2684746.2689060 (2019).

43. Zhang, C., Fang, Z., Zhou, P., Pan, P. & Cong, J. *Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks* en. in *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16* (ACM Press, Austin, Texas, 2016), 1–8. ISBN: 978-1-4503-4466-1. http://dl.acm.org/citation.cfm?doid=2966986.2967011 (2019).

44. Venkatesh, G., Nurvitadhi, E. & Marr, D. Accelerating Deep Convolutional Networks using low-precision and sparsity. en. *arXiv:1610.00324 [cs].* arXiv: 1610.00324. http://arxiv.org/abs/1610.00324 (2019) (Oct. 2016).

45. Gokhale, V., Zaidy, A., Chang, A. X. M. & Culurciello, E. Snowflake: A Model Agnostic Accelerator for Deep Convolutional Neural Networks. en. *arXiv:1708.02579 [cs].* arXiv: 1708.02579. http://arxiv.org/abs/1708.02579 (2019) (Aug. 2017).

46. Kreinar, E. J. RFNoC Neural Network Library using Vivado HLS. en. *Proceedings of the GNU Radio Conference,* 7 (2017).

47. Xilinx. *Vivado Design Suite HLx Editions* 2018. https://www.xilinx.com/products/design-tools/vivado.html.

48. Zhou, Y., Redkar, S. & Huang, X. *Deep learning binary neural network on an FPGA* en. in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)* (IEEE, Boston, MA, USA, Aug. 2017), 281–284. ISBN: 978-1-5090-6389-5. http://ieeexplore.ieee.org/document/8052915/ (2019).

49. Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. en. *Neural Computing and Applications.* ISSN: 0941-0643, 1433-3058. http://link.springer.com/10.1007/s00521-018-3761-1 (2019) (Oct. 2018).

50. Venieris, S. I., Kouris, A. & Bouganis, C.-S. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. en. *arXiv:1803.05900 [cs].* arXiv: 1803.05900. http://arxiv.org/abs/1803.05900 (2019) (Mar. 2018).

51. Wang, T., Wang, C., Zhou, X. & Chen, H. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. en. *arXiv:1901.04988 [cs].* arXiv: 1901.04988. http://arxiv.org/abs/1901.04988 (2019) (Dec. 2018).

52. Liu, B., Gould, S. & Koller, D. *Single image depth estimation from predicted semantic labels* en. in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (IEEE, San Francisco, CA, USA, June 2010), 1253–1260. ISBN: 978-1-4244-6984-0. http://ieeexplore.ieee.org/document/5539823/ (2021).

53. Liu, F., Chunhua Shen & Guosheng Lin. *Deep convolutional neural fields for depth estimation from a single image* en. in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (IEEE, Boston, MA, USA, June 2015), 5162–5170. ISBN: 978-1-4673-6964-0. http://ieeexplore.ieee.org/document/7299152/ (2021).

54. Chen, X., Ma, H., Wan, J., Li, B. & Xia, T. *Multi-view 3D Object Detection Network for Autonomous Driving* en. in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (IEEE, Honolulu, HI, July 2017), 6526–6534. ISBN: 978-1-5386-0457-1. http://ieeexplore.ieee.org/document/8100174/ (2019).

55. Maturana, D. & Scherer, S. *VoxNet: A 3D Convolutional Neural Network for real-time object recognition* en. in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (IEEE, Hamburg, Germany, Sept. 2015), 922–928. ISBN: 978-1-4799-9994-1. http://ieeexplore.ieee.org/document/7353481/ (2019).

56. Brock, A., Lim, T., Ritchie, J. M. & Weston, N. Generative and Discriminative Voxel Modeling with Convolutional Neural Networks. en. *arXiv:1608.04236 [cs, stat].* arXiv: 1608.04236. http://arxiv.org/abs/1608.04236 (2020) (Aug. 2016).

57. Garcia-Garcia, A. *et al. PointNet: A 3D Convolutional Neural Network for real-time object class recognition* en. in *2016 International Joint Conference on Neural Networks (IJCNN)* (IEEE, Vancouver, BC, Canada, July 2016), 1578–1584. ISBN: 978-1-5090-0620-5. http://ieeexplore.ieee.org/document/7727386/ (2020).

58. Hegde, V. & Zadeh, R. FusionNet: 3D Object Classification Using Multiple Data Representations. en. *arXiv:1607.05695 [cs].* arXiv: 1607.05695. http://arxiv.org/abs/1607.05695 (2020) (Nov. 2016).

59. Jing Huang & Suya You. *Point cloud labeling using 3D Convolutional Neural Network* en. in *2016 23rd International Conference on Pattern Recognition (ICPR)* (IEEE, Cancun, Dec. 2016), 2670–2675. ISBN: 978-1-5090-4847-2. http://ieeexplore.ieee.org/document/7900038/ (2020).

60. Charles, R. Q., Su, H., Kaichun, M. & Guibas, L. J. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation* en. in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (IEEE, Honolulu, HI, July 2017), 77–85. ISBN: 978-1-5386-0457-1. http://ieeexplore.ieee.org/document/8099499/ (2020).

61. Zhi, S., Liu, Y., Li, X. & Guo, Y. LightNet: A Lightweight 3D Convolutional Neural Network for Real-Time 3D Object Recognition. en. *Eurographics Workshop on 3D Object Retrieval,* 8 pages. ISSN: 1997-0471. https://diglib.eg.org/handle/10.2312/3dor20171046 (2020) (2017).

62. Ioannidou, A., Chatzilari, E., Nikolopoulos, S. & Kompatsiaris, I. en. in *MultiMedia Modeling* (eds Kompatsiaris, I. *et al.*) 495–506 (Springer International Publishing, Cham, 2019). ISBN: 978-3-030-05709-1 978-3-030-05710-7. http://link.springer.com/10.1007/978-3-030-05710-7_41 (2020).

63. Nurvitadhi, E. *et al. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC* en. in *2016 International Conference on Field-Programmable*

Technology (FPT) (IEEE, Xi'an, China, Dec. 2016), 77–84. ISBN: 978-1-5090-5602-6. http://ieeexplore.ieee.org/document/7929192/ (2019).

64. Nurvitadhi, E. *et al. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC* en. in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)* (IEEE, Lausanne, Switzerland, Aug. 2016), 1–4. ISBN: 978-2-8399-1844-2. http://ieeexplore.ieee.org/document/7577314/ (2019).

65. Nurvitadhi, E. *et al. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?* en. in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17* (ACM Press, Monterey, California, USA, 2017), 5–14. ISBN: 978-1-4503-4354-1. http://dl.acm.org/citation.cfm?doid=3020078.3021740 (2019).

66. Andrews, D., Niehaus, D. & Ashenden, P. Programming models for hybrid CPU/FPGA chips. en. *Computer* **37,** 118–120. ISSN: 0018-9162. http://ieeexplore.ieee.org/document/1319290/ (2020) (Jan. 2004).

67. Andrews, D., Niehaus, D. & Jidin, R. Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components. en, 6 (2004).

68. Nane, R. *et al.* A Survey and Evaluation of FPGA High-Level Synthesis Tools. en. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35,** 1591–1604. ISSN: 0278-0070, 1937-4151. http://ieeexplore.ieee.org/document/7368920/ (2019) (Oct. 2016).

69. Jouppi, N. P. *et al.* In-Datacenter Performance Analysis of a Tensor Processing UnitTM. en, 17 (June 2017).

70. Apple Inc. *iPhone XS - Technical Specification* Sept. 2018. https://www.apple.com/iphone-xs/specs/.

71. Jeremy Hsu. *Nervana Systems Puts Deep Learning AI in the Cloud* Mar. 2016. https://spectrum.ieee.org/tech-talk/computing/software/nervana-systems-puts-deep-learning-ai-in-the-cloud.

72. Marantos, C. *et al. Efficient support vector machines implementation on Intel/-Movidius Myriad 2* en. in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)* (IEEE, Thessaloniki, May 2018), 1–4. ISBN: 978-1-5386-4788-2. https://ieeexplore.ieee.org/document/8376630/ (2021).

73. Eigner, M., Dickopf, T. & Apostolov, H. en. in *Product Lifecycle Management and the Industry of the Future* (eds Ríos, J., Bernard, A., Bouras, A. & Foufou, S.) Series Title: IFIP Advances in Information and Communication Technology, 382–393 (Springer International Publishing, Cham, 2017). ISBN: 978-3-319-72904-6 978-3-319-72905-3. http://link.springer.com/10.1007/978-3-319-72905-3_34 (2020).

74. Cabanes, Q. *Automation script of bootable SDcard for Zynq board* https://github.com/Tigralt/zynboot. 2018. https://github.com/Tigralt/zynq-boot.

75. Cabanes, Q. & Senouci, B. *Objects detection and recognition in smart vehicle applications: Point cloud based approach* en. in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)* (IEEE, Milan, July 2017), 287–289. ISBN: 978-1-5090-4749-9. http://ieeexplore.ieee.org/document/7993795/ (2019).

76. *GNU make* http://www.gnu.org/software/make/manual/make.html.

77. Xilinx. *ZedBoard* https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html.

78. Ovtcharov, K. *et al.* Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. en, 4 (2015).

79. Courbariaux, M., Bengio, Y. & David, J.-P. Training deep neural networks with low precision multiplications. en. *arXiv:1412.7024 [cs].* arXiv: 1412.7024. http://arxiv.org/abs/1412.7024 (2020) (Sept. 2015).

80. Govindu, G., Ling Zhuo, Seonil Choi & Prasanna, V. *Analysis of high-performance floating-point arithmetic on FPGAs* en. in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (IEEE, Santa Fe, NM, USA, 2004), 149–156. ISBN: 978-0-7695-2132-9. http://ieeexplore.ieee.org/document/1303135/ (2020).

81. Underwood, K. *FPGAs vs. CPUs: trends in peak floating-point performance* en. in *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays - FPGA '04* (ACM Press, Monterey, California, USA, 2004), 171. ISBN: 978-1-58113-829-0. http://portal.acm.org/citation.cfm?doid=968280.968305 (2020).

82. Dou, Y., Vassiliadis, S., Kuzmanov, G. K. & Gaydadjiev, G. N. *64-bit floating-point FPGA matrix multiplication* en. in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays - FPGA '05* (ACM Press, Monterey, California, USA, 2005), 86. ISBN: 978-1-59593-029-3. http://portal.acm.org/citation.cfm?doid=1046192.1046204 (2020).

83. Roldao Lopes, A. & Constantinides, G. A. en. in *Reconfigurable Computing: Architectures, Tools and Applications* (eds Hutchison, D. *et al.*) Series Title: Lecture Notes in Computer Science, 157–168 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010). ISBN: 978-3-642-12132-6 978-3-642-12133-3. http://link.springer.com/10.1007/978-3-642-12133-3_16 (2020).

84. Courbariaux, M., Bengio, Y. & David, J.-P. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. en, 9 (2015).

85. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R. & Bengio, Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations

Constrained to +1 or -1. en. *arXiv:1602.02830 [cs].* arXiv: 1602.02830. http://arxiv.org/abs/1602.02830 (2019) (Feb. 2016).

86. Ioualalen, A. & Martel, M. en. in *Quantitative Evaluation of Systems* (eds Parker, D. & Wolf, V.) Series Title: Lecture Notes in Computer Science, 129–143 (Springer International Publishing, Cham, 2019). ISBN: 978-3-030-30280-1 978-3-030-30281-8. http://link.springer.com/10.1007/978-3-030-30281-8_8 (2021).

87. Cabanes, Q. *Hardware Neural Network Processor* https://github.com/Tigralt/hardware-neural-network-processor. 2020. https://github.com/Tigralt/hardware-neural-network-processor.

88. Lin, Z., Memisevic, R. & Konda, K. How far can we go without convolution: Improving fully-connected networks. en. *arXiv:1511.02580 [cs].* arXiv: 1511.02580. http://arxiv.org/abs/1511.02580 (2020) (Nov. 2015).

89. Oliphant, T. *NumPy: A guide to NumPy* Published: USA: Trelgol Publishing. http://www.numpy.org/ (2006).

90. Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images* en. Tech Report. Apr. 2009.

91. LeCun, Y. & Cortes, C. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/ (2016) (2010).

92. Quadros, A. J. Representing 3D Shape in Sparse Range Images for Urban Object Classification. en, 204.

93. Deuge, M. D., Quadros, A., Hung, C. & Douillard, B. Unsupervised Feature Learning for Classification of Outdoor 3D Scans. en, 9 (2013).

94. Abadi, M. *et al.* TensorFlow: A system for large-scale machine learning. en, 21 (2016).

95. Vivado Design Suite. AXI DMA v7.1 LogiCORE IP Product Guide. en, 97. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (2019).

96. Xiao, H., Rasul, K. & Vollgraf, R. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms* arXiv: cs.LG/1708.07747. Aug. 2017.

97. Hinton, G. E. en. in *Neural Networks: Tricks of the Trade* (eds Montavon, G., Orr, G. B. & Müller, K.-R.) Series Title: Lecture Notes in Computer Science, 599–619 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012). ISBN: 978-3-642-35288-1 978-3-642-35289-8. http://link.springer.com/10.1007/978-3-642-35289-8_32 (2020).

98. Hinton, G. E. Training products of experts by minimizing contrastive divergence. *Neural computation* **14.** Publisher: MIT Press, 1771–1800 (2002).

99. Konda, K., Memisevic, R. & Krueger, D. Zero-bias autoencoders and the benefits of co-adapting features. en. *arXiv:1402.3337 [cs, stat].* arXiv: 1402.3337. http://arxiv.org/abs/1402.3337 (2020) (Apr. 2015).

100. Senouci, B., Charfi, I., Heyrman, B., Dubois, J. & Miteran, J. Fast prototyping of a SoC-based smart-camera: a real-time fall detection case study. en. *Journal of Real-Time Image Processing* **12,** 649–662. ISSN: 1861-8200, 1861-8219. http://link.springer.com/10.1007/s11554-014-0456-4 (2021) (Dec. 2016).

101. Thrun, S. *et al.* Stanley: The robot that won the DARPA Grand Challenge. en. *Journal of Field Robotics* **23,** 661–692. ISSN: 15564959, 15564967. http://doi.wiley.com/10.1002/rob.20147 (2020) (Sept. 2006).

102. Wojek, C., Dorkó, G., Schulz, A. & Schiele, B. en. in *Pattern Recognition* (eds Hutchison, D. *et al.*) 71–81 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008). ISBN: 978-3-540-69320-8 978-3-540-69321-5. http://link.springer.com/10.1007/978-3-540-69321-5_8 (2019).

103. Rosebrock, Adrian. *Sliding Windows for Object Detection with Python and OpenCV* Mar. 2015. http://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/.

104. Spinello, L., Arras, K. O., Triebel, R. & Siegwart, R. A Layered Approach to People Detection in 3D Range Data. en. *Proc. of The AAAI Conference on Artificial Intelligence (AAAI),* 6 (2010).

105. Spinello, L., Luber, M. & Arras, K. O. *Tracking people in 3D using a bottom-up top-down detector* en. in *2011 IEEE International Conference on Robotics and Automation* (IEEE, Shanghai, China, May 2011), 1304–1310. ISBN: 978-1-61284-386-5. http://ieeexplore.ieee.org/document/5980085/ (2019).

106. Cabanes, Q., Senouci, B. & Ramdane-Cherif, A. *A Complete Multi-CPU/FPGA-based Design and Prototyping Methodology for Autonomous Vehicles: Multiple Object Detection and Recognition Case Study* en. in *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)* (IEEE, Okinawa, Japan, Feb. 2019), 158–163. ISBN: 978-1-5386-7822-0. https://ieeexplore.ieee.org/document/8669047/ (2019).

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Nouvelle méthodologie de co-conception pour de l'apprentissage en profondeur basée sur une plate-forme matérielle pour le prototypage de SCP: reconnaissance d'objets dans une étude de cas de véhicule autonome

Mots clés : systèmes cyber-physiques; apprentissage en profondeur embarqué; FPGA; accélérateur de réseau de neurones; automatisation de prototype

Résumé : Les Systèmes Cyber-Physiques (SCP) sont un sujet de recherche mature qui interagissent avec l'intelligence artificielle (IA) et les systèmes embarqués (SE). Un SCP peut être défini comme un SE en réseau qui peut analyser un environnement physique, via des capteurs, et prendre des décisions à partir de son état actuel pour affecter son environnement physique vers un résultat souhaité, via des actionneurs. Ces SCP nécessitent des algorithmes d'analyse de données puissants associés à des architectures matérielles robustes. D'une part, l'Apprentissage en Profondeur (AP) est proposé comme algorithme principal. D'autre part, les méthodologies de conception et de prototypage standard pour SE ne sont pas adaptées au prototypage de SCP moderne basé sur de l'AP, et plus particulièrement, des accélérateurs matériels d'AP. Finalement, cette méthodologie est validée avec un cas d'usage autour du véhicule autonome.

Dans cette thèse, nous étudions la conception d'IA pour SCP autour de l'AP embarquée avec une plate-forme hybride CPU/FPGA (Field-Programmable Gate Array). Nous proposons une méthodologie de co-conception matérielle/logicielle pour développer des applications d'AP pour SCP qui est basée sur la conception et l'utilisation d'un accélérateur de réseau de neurones et d'un logiciel d'automatisation des étapes de la méthodologie pour accélérer le temps de prototypage. Nous présentons la conception et le prototypage de notre accélérateur matériel de réseau de neurones ainsi que le résultat de ses performances. Enfin, nous validons notre travail à l'aide d'un cas d'usage : un LIDAR (LIght Detection And Ranging) intelligent pour la détection d'objet autour d'un véhicule autonome. Ce cas d'usage est accompagné de plusieurs algorithmes de détection de piétons à l'aide du nuage de points 3D d'un LIDAR réalisé et testé sur plateforme logicielle et materielle.

Title: New hardware platform-based deep learning co-design methodology for CPS prototyping: Objects recognition in autonomous vehicle case-study

Keywords: cyber-physical systems; embedded deep learning; FPGA; neural network accelerator; hardware prototype automation

Abstract: Cyber-Physical Systems (CPS) are a mature research technology topic that deals with Artificial Intelligence (AI) and Embedded Systems (ES). A CPS can be defined as a networked ES that can analyze a physical environment, via sensors, and make decisions from its current state to affect this physical environment toward a desired outcome with actuators. These CPS deal with data analysis, which need powerful algorithms combined with robust hardware architectures. On one hand, Deep Learning (DL) is proposed as the main solution algorithm. On the other hand, the standard design and prototyping methodologies for ES are not adapted to modern DL-based CPS, especially when using hardware accelerated DL processor. Finally, this methodology is validated with a use-case around an autonomous vehicle.

In this thesis, we investigate AI design for CPS around embedded DL using a hybrid CPU/FPGA (Field-Programmable Gate Array) platform. We proposed a hardware/software co-design methodology to develop DL applications for CPS which is based on the usage of a neural network accelerator and automation software to speed up the prototyping time. We present our hardware neural network accelerator design and prototyping, as well as its performance results. Finally, we validate our work using a smart LIDAR (LIght Detection And Ranging) application use-case for object detection and recognition around an autonomous vehicle. This use-case is presented with several algorithms for pedestrians detection using a 3D point cloud from a LIDAR, designed and tested on software and hardware platform.

Maison du doctorat de l'Université Paris-Saclay
2ème étage aile ouest, Ecole normale supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France