



Deusto

Facultad de Ingeniería
Universidad de Deusto

Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Grado en Ingeniería en Electrónica Industrial y Automática

Industria Elektronikako eta Automatikako Ingeniaritzako Gradua

Proyecto fin de grado

Gradu amaierako proiektua

**Application Acceleration Using a
Heterogeneous MPSoC Architecture with
MPU and FPGA Processors**

Mikel Solabarrieta Román

Director: Ignacio Angulo Martínez

Bilbao, junio de 2020



Deusto

Facultad de Ingeniería
Universidad de Deusto

Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Grado en Ingeniería en Electrónica Industrial y Automática Industria Elektronikako eta Automatikako Ingeniaritzako Gradua

Proyecto fin de grado Gradu amaierako proiektua

Application Acceleration Using a
Heterogeneous MPSoC Architecture with
MPU and FPGA Processors

Mikel Solabarrieta Román

Director: Ignacio Angulo Martínez

Bilbao, junio de 2020

A handwritten signature in blue ink, appearing to read 'I. Angulo', enclosed within a blue oval scribble.

Abstract

This degree thesis tests the performance improvements of an MPU+FPGA multiprocessor system-on-chip (MPSoC) over standard microcontrollers in edge computing applications, namely object detection and facial recognition. Conclusions are deduced from execution profiles of similar facial recognition applications with and without FPGA acceleration.

The board used for this thesis is a Xilinx Zynq UltraScale+ MPSoC development board: the Ultra96-v1.

A brief description of Zynq is due before describing the application it will be used for. Zynq is a computer architecture designed by Xilinx that contains both an ARM processor and a Field-Programmable Gate Array. PYNQ is a Zynq framework that has been used to develop software for this thesis mainly because it allows the usage of the Python programming language to make use of Zynq. It runs on the Linux operating system and has Jupyter Notebooks installed by default. Programmable Logic is implemented with overlays: hardware libraries that enable a simple way of interacting with the FPGA.

Facial recognition and object detection are both relatively computationally heavy tasks that tasks a lot of time to perform on standard microcontrollers. The viability of performing the aforementioned tasks in real time will be tested by using existing overlays to accelerate Artificial Intelligence algorithms and computer vision utilities, both independently and integrating them.

Descriptors

Edge Computing, Embedded Devices, MPSoC, PYNQ, Zynq

Contents

1	Introduction	1
1.1	Context	1
1.2	Project Motivation	1
1.3	Problem and Need	2
1.4	Justification	3
2	Scope	5
2.1	Ethical Considerations	6
2.2	Objectives	6
2.3	Project Structure	7
3	Technical Requirements	9
3.1	Functional Objectives	9
3.2	Non-Functional Objectives	11
4	Methodology and Resources	15
4.1	Kanban	15
4.2	Resources	16
5	Budget and Planning	19
5.1	Budget	19
5.2	Planning	20

6	State of the Art	23
6.1	Key Areas	23
6.1.1	Edge Computing	24
6.1.2	System-on-Chip	24
6.2	Platform Overview	24
6.2.1	Zynq	25
6.2.2	PYNQ	28
6.3	Board Overview	33
6.4	High-Level Synthesis	35
6.5	Preceding Projects	36
6.5.1	FINN	36
6.5.2	Binarized Neural Networks	37
6.5.3	Quantized Neural Networks	39
7	Development	41
7.1	Familiarization with the PYNQ Platform	41
7.2	Facial Recognition	42
7.2.1	LFC Setup	42
7.2.2	LFC Usage	46
7.2.3	CNV Setup	48
7.2.4	CNV Usage	49
7.3	Object Detection	50
7.3.1	Setup	50
7.3.2	Usage	51
7.4	Integration of Facial Recognition and Object Detection	55
7.5	Forked Projects	57
8	Results	61

9	Conclusions	65
9.1	Future Work	66
9.2	Personal Assessment	67
	Bibliography	71
	A Vivado Block Designs	
	B Integration Utilization Report	
	C HLS Example	

List of Figures

1.1	Profile of object detection program	2
2.1	Characteristics of the BSD-3 license used for PYNQ	6
2.2	Project structure	8
4.1	Kanban board	15
4.2	Picture of the PYNQ-Z1 (left) and Ultra96 (right) boards	17
5.1	Project schedule	22
6.1	Zynq-7000 architectural overview	25
6.2	Zynq-UltraScale+ component overview	26
6.3	PL interface to PS memory	27
6.4	PYNQ application overview	28
6.5	State of the program with image in local and shared memory	30
6.6	State of the program when using the PL	31
6.7	Jupyter Notebook example	32
6.8	Picture of the Ultra96-v1 board	33
6.9	PS specifications	34
6.10	PL specifications	34
6.11	Ultra96 block diagram	35
6.12	HLS validation and verification	36
6.13	FINN data flow	38

6.14	LFC diagram	39
6.15	CNV diagram	39
6.16	TinierYOLO topology	40
7.1	Dataset directory structure	43
7.2	Best epoch when training LFCW1A2	46
7.3	GTSRB modified dataset directory structure	48
7.4	Best epoch when training CNVW1A1	49
7.5	Last step before creating the Vivado project	56
8.1	Execution times of each BNN topology	62
8.2	Alternative view of the execution times of each BNN topology	62
8.3	Comparison of HW+SW and SW QNN execution times	63
8.4	Comparison of border (SW) and middle (HW) layers	63
8.5	GitLab repository containing the Jupyter Notebooks	64

List of Tables

3.1	Requirements of objective O1	9
3.2	Requirements of objective O2	10
3.3	Requirements of objective O3	10
3.4	Requirements of objective O4	11
3.5	Requirements of objective O5	11
3.6	Requirements of objective O6	12
3.7	Requirements of objective O7	12
3.8	Requirements of objective O8	12
3.9	Requirements of objective O9	13
5.1	Cost of human resources	19
5.2	Cost of material resources	20

Listings

6.1	"IP extraction from the resizer overlay"	29
6.2	"Memory allocation"	29
6.3	"PL usage"	30
7.1	"Image modification script"	44
7.2	"Sample of mnist.py"	45
7.3	"Module versions"	45
7.4	"BNN LFC usage example"	46
7.5	"PynqBNN constructor"	47
7.6	"Training of CNV"	48
7.7	"BNN usage example"	49
7.8	"Ctypes example from darknet.py"	50
7.9	"Python dynamic library import in darknet.py"	51
7.10	"Function definition example from darknet.py"	51
7.11	"QNN global variables"	51
7.12	"QNN imports"	52
7.13	"Partial method of TinierYolo class"	52
7.14	"Network architecture"	52
7.15	Network creation and configuration"	53
7.16	"First QNN layer"	53
7.17	"Middle QNN layers"	54
7.18	"Middle QNN layers post-processing"	54
7.19	"Extraction of results from QNN"	54
8.1	"BNN and QNN fork installation command"	63

Chapter 1

Introduction

This thesis aims to provide relevant information about the end of degree project. This chapter intends to provide an introduction to the project, provide context for it, the motivation behind it and lay out the problem it intends to solve.

1.1 Context

The rise of Internet of Things (IoT) systems for any range of applications, from industrial to consumer ones, brings with it the benefit of bringing “intelligence” to applications that didn’t traditionally have it. IoT, in this context, is described as the convergence of technologies from the fields of data-capture devices, embedded systems and analytics, amongst others.

Inside IoT systems, edge computing consists on running some of the data-processing functions in the data-capture devices themselves rather than in centralized servers. Dual architectures such as the one provided by Xilinx enable edge devices, devices that do the processing, to be more flexible and more efficient regarding both computation and energy consumption than a standard, single-processor device. The advantages of this specific paradigm were a device with ARM+FPGA processors are used for edge computing include speed improvements, reduced bandwidth use and energy consumption, some security improvements and increased reliability. While there are some downsides, such as the increased programming difficulty, in most cases they are outweighed by the benefits.

1.2 Project Motivation

The adoption of an interesting, relatively new architecture with potential implications in fields such as Distributed Systems, Edge Computing and High Performance Computing. As discussed in the “Justification” section, the fact that the platform studied in this project seems to be the path many researchers and professionals will take is a great motivator. The interest in a research career

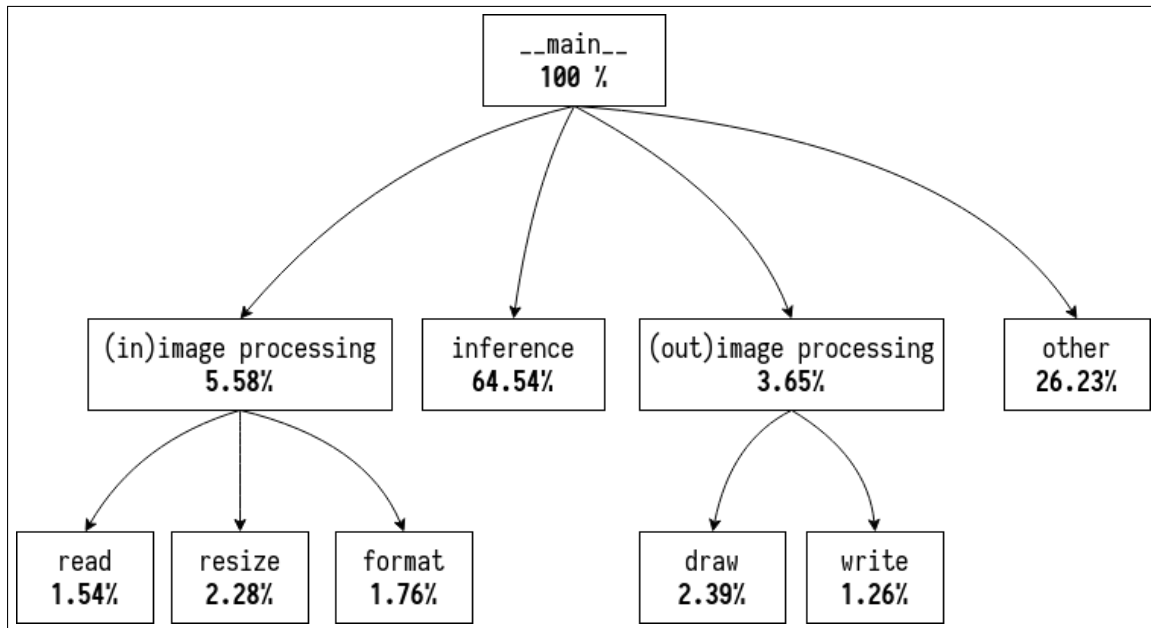


Fig. 1.1: Profile of object detection program

and both personal and professional interests in fields very related to this project, such as High Performance Computing (HPC) are also important factors. This will be the last project before pursuing a career in the HPC field, thus exploring what the personally completely new concept that is a heterogeneous, dual architecture that combines many of the interests adjoined during the studies is appropriate.

One of the first steps of the project was to run an object detection application in a computer with an ARM processor and profile the execution. Figure 1.1 shows a simplified version of the profile. The program run the facial recognition function in 43 different images and, as seen in the image, a highly parallelizable section of the code: the inference, was taking more than half of the execution times. The code section shown in the picture as “other” included displaying the images, which if handled correctly with concurrency could completely disappear from the profile. It is clear then the potential of a device such as the one studied in this project: the performance can be very significantly improved without the developer of the application needing to know the inner workings of any low-level device. Just take the input, pass it to the PL, and receive the output. This is significantly important in industrial environments where reducing the latency of error detection in any process can prevent downtime and lead to considerable savings.

1.3 Problem and Need

The way IoT technologies are currently implemented has a lot of room for improvement, such as the centralization of processing units and the need to send great amounts of data from the acquisition systems to the processing units potentially saturating the networking infrastructure. While solutions to some of the presented problems exist, most noticeable the development and deployment of 5G networking infrastructure, the Zynq architecture could prove to be an impactful

and further improve the situation when combined with other solutions.

Reiterating, the problem is that most edge devices have low computing power and can't run advanced processes on the data, requiring to be sent to central servers. Thus the need is an edge computing device that allows for such processing while being energy efficient and low cost. This project specifically looks at an edge computing device able of performing object detection and facial recognition without requiring the transmission of video feed to central servers or introducing latency and bandwidth issues to the network. Both of these applications are computationally heavy and can't be adequately performed in real-time by standard microcontrollers that have acceptable energy consumption. The dual ARM+FPGA enables the delegation of computationally heavy tasks that can be parallelized to the FPGA processing unit while implementing the rest of the functionality in the ARM processors. This project evaluates how the Zynq architecture and the PYNQ framework work as a solution to this problem.

1.4 Justification

The idea that specialized hardware is becoming increasingly relevant in multiple fields, albeit not a definitive solution to problems that require more flexibility, is a justifying argument in itself. The counter-argument: that some problems require a more flexible solution, can also be turned around in this specific case. The processor-centric approach means that designs downloaded to the FPGA can be changed on runtime, which brings flexibility single processor devices lack.

Despite being the final project of the Industrial Electronics and Automation Engineering degree, it involves concepts studied in both degrees of the double degree. The main approach is based on using an ARM processor, which combined with an FPGA in a single MPSoC device gives very significant performance improvements. The platform is intended for industrial uses and could have been applied in many of the degree subjects: signal processing, automatizing... Because it contains many different subjects studied throughout the degree, it seems appropriate as a final project for the electronics degree. At the same time, development is made more accessible by using a framework based on GNU/Linux and Python, both of which have been very relevant subjects in the Computer Engineering degree. The knowledge acquired about the GNU/Linux operating systems has proven useful not only for understanding the platform better but also for troubleshooting in several steps of the project development process. Knowledge of Python has also proven very useful in several sections where documentation was lacking and the provided code needed to be thoroughly inspected. It has also been useful to speed up development and modify things such as code of broken dependencies. At a certain point in the project, the C development done in the microcontrollers and automatizing subjects electronics degree also proved useful when modifying a library that while called from Python, was written in C. This will be discussed further in one of the final sections, section 9.2.

Chapter 2

Scope

The main goal of this project is to research and experiment with the state of the art technology that represents the PYNQ framework and its underlying technology, and its actual and potential impact on relevant applications. These applications are in this case object detection and facial recognition because, as explained in the introduction chapter, they are recurrent processes in countless scenarios and computationally demanding for most standard microcontrollers. They represent an example of how the range of data-processing functions that can be executed by devices near to where the data is captured instead of in central servers is not limited to basic operations and can have some level of complexity. Experimenting with these examples is, once again, the goal of this project and both the scope and the results are oriented to representing so.

Some concepts, such as “overlays” are mentioned in this chapter and in chapter 3 are not explained until chapter 6 but should be clear after reading it.

The expected final result of the project includes:

- Analysis of the Zynq architecture
- Analysis of existing PYNQ projects and their potential use in object detection and face recognition
- Adaptation of the existing PYNQ project to fit the use cases of object detection and face recognition
- Benchmarks of multiple hardware-accelerated applications and their not accelerated versions
- A program integrating various PYNQ overlays to meet the requirements of real-time object detection and face recognition in an edge computing device

2.1 Ethical Considerations

The first ethical aspect examined is software licensing. Software licenses determine what people can do with software, how they can use, modify and distribute it; and are often overlooked. Many of the components of this project have open licenses, namely the BSD-3 or “BSD new” license, its most relevant characteristic being the ability to modify and redistribute the source code as long as the original license is included. It is compatible with the widely used GNU General Public License and has some more specific details, such as forbidding the use of the names of the original authors for endorsements. The main characteristics of the license are listed in figure 2.1. Having a permissive software license is important because it gives more rights and opportunities to anyone interested in using the software, favoring accessibility and equality.

Permissions	Limitations	Conditions
✓ Commercial use	✗ Liability	ⓘ License and copyright notice
✓ Modification	✗ Warranty	
✓ Distribution		
✓ Private use		

Fig. 2.1: Characteristics of the BSD-3 license used for PYNQ

Other ethical aspects of special relevance to this project are the potential uses of facial recognition and object detection. Facial recognition is a technology with many potential benefits, but also many risks. Due to this, many of the companies and institutions developing the technology are doing it with caution. While acknowledging the clear benefits it may offer (convenience, security...), this section will focus on analyzing the ethical issues it presents. The most commonly discussed risks are the privacy issues systems using facial recognition may imply, or the potential exploitation of the ability to instantly identify people. Assessing privacy issues, facial recognition is often used without the consent of the people being analyzed; and in some cases can be used to access a lot of public information about a person with just their face, even if that person did not agree to it. About exploitation, the ability to identify people instantly can be used for unethical purposes, for example by a government or company running a face recognition algorithm against a group of protesting people to identify and retaliate against them. When applied to critical applications such as automated criminal identification, false positives may result in innocent people being condemned which goes against the principle of justice.

It is important that the ethical issues of these applications are analyzed and regulated. Questions must be asked about why certain technologies are being developed and what potential risks they may have and how those affect people or the environment, before developing them.

2.2 Objectives

The main objective of the project is to evaluate how suitable the Zynq devices with the PYNQ framework are as a solution to the necessities edge computing presents in modern IoT systems.

Two computationally heavy tasks will be studied: object detection and facial recognition. As mentioned previously, these two tasks are too demanding for standard, mainstream homogeneous (mono-core or multi-core) embedded microcontrollers.

Functional objectives:

- **O1** An independent application must be able to identify a person's face given a frame.
- **O2** An independent application must be able to detect objects given a frame.
- **O3** An integrated application must be able to perform both previous tasks.

Non-functional objectives:

- **O4** Tasks must be able to be performed in real time, each one of them processing at least 5 frames per second.
- **O5** Code must be deployable in PYNQ-enabled Ultra96-v1 device.
- **O6** Accelerated processes must be noticeably faster than their pure software versions.
- **O7** The code must be clear and understandable.
- **O8** Documentation must be provided for the resulting code and underlying ideas.
- **O9** Software licenses must be compatible.

2.3 Project Structure

The structure of the project is shown in figure 2.2, and contains the following steps. Summarizing, it consists on researching state-of-the art technology and areas and trying them in different use cases, before trying to combine them into the intended final use.

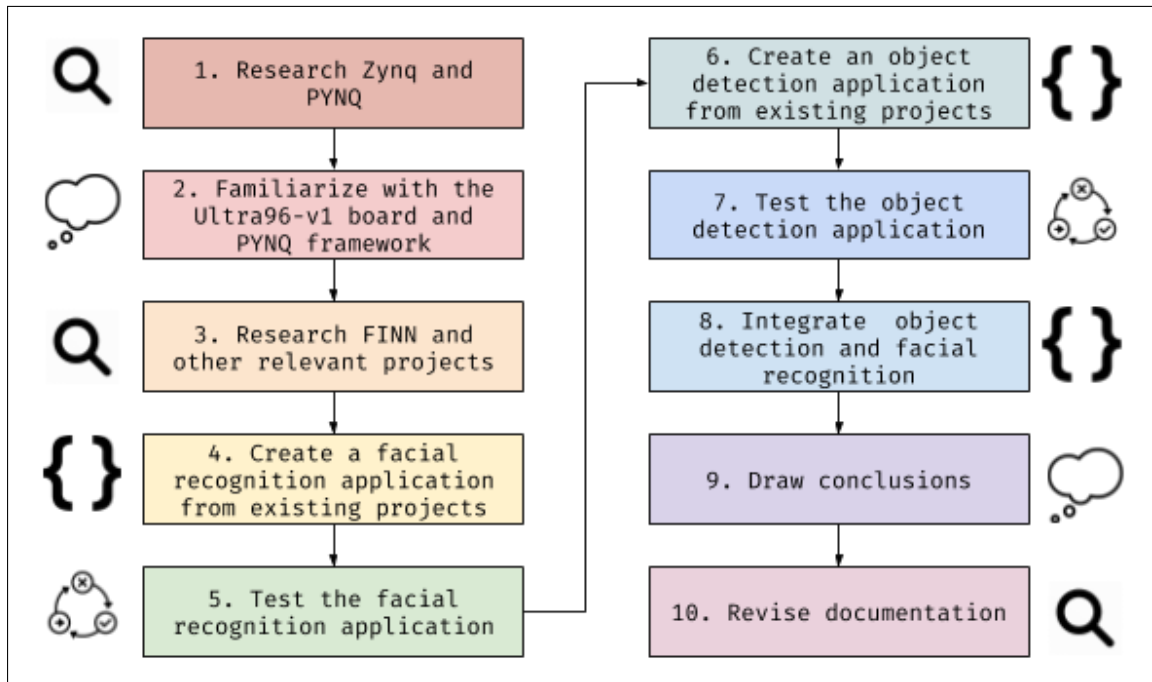


Fig. 2.2: Project structure

1. **Research Zynq and PYNQ.** Research Zynq architecture and the PYNQ framework. Understand their purpose and how they work.
2. **Familiarize with the Ultra96-v1 board and PYNQ framework.** The Ultra96-v1 is the physical device that runs PYNQ in this project.
3. **Research FINN and other relevant projects.** Research projects built over PYNQ that will be used in the next steps.
4. **Create a facial recognition application from existing projects.** Adapt an existing project to serve the facial recognition functionality.
5. **Test the facial recognition application.** Test the efficiency of the facial recognition solution.
6. **Create an object detection application from existing projects.** Adapt an existing project to serve the object detection functionality.
7. **Test the object detection application.** Test the efficiency of the object detection solution.
8. **Integrate object detection and facial recognition.** Integrate the previous solutions into a single project that meets all the requirements of the project.
9. **Draw conclusions.** Analyze the project and extract conclusions.
10. **Revise documentation.** Finish the project documentation.

The project structure will be further analyzed when detailing the objectives in chapter 3 and in the planning of chapter 5, which details how much time and resources will be dedicated to each one of the tasks.

Chapter 3

Technical Requirements

This chapter defines and explains each of the requirements of the project based on objectives from section 2.2. Each objective has certain requirements. Each requirement is fully identified by appending the requirement identifier to the objective identifier. Thus, the full identifier for requirement **R1** of objective **O1** is **O1R1**. Requirements that require additional specifications have a specification section. Specifications are fully identified by appending the objective, requirement and specification identifiers.

3.1 Functional Objectives

Objective 1

The first objective, **O1** refers to the facial recognition functionality and states that an application must be created to receive a frame as input and output what person the face belongs to.

Table 3.1: Requirements of objective O1

Requirement	
R1 Functional PYNQ overlay that runs a picture through an AI classifier and outputs the class the picture belongs to.	
Specification	Description
S1	The overlay must fit in the Ultra96-v1 board.
Requirement	
R2 Functional Python program that provides a provides an image as input to the overlay and displays the output in a readable manner.	

Objective 2

The second objective, **O2** details that an object detection algorithm must be accelerated with the Zynq architecture. All sections of code that are likely to offer a substantial improvement when accelerated by hardware must be implemented using programmable logic (PL) and those that cannot be accelerated must be efficiently implemented in Python.

Table 3.2: Requirements of objective O2

Requirement	
R1 Functional PYNQ overlay that runs a picture through an AI classifier and outputs a bounding box of each object detected in the picture.	
Specification	Description
S1	The overlay must fit in the Ultra96-v1 board.
Requirement	
R2 Functional Python program that provides a provides an image as input to the overlay and filters the bounding boxes outputted by the overlay that belong to people.	
Specification	Description
S1	The Python program must for compensate any functionalities the overlay lacks.

Objective 3

Objective 3 or **O3** refers to a program that combines the functionality of object detection and facial recognition, effectively detecting and recognizing people in an image. The ideal solution is an overlay with functions that do both as specified in requirement **O3R1**, however that is a complicated task that may fail. With that in mind, an alternative program design must be provided as stated in requirement **O3R2**.

Table 3.3: Requirements of objective O3

Requirement	
R2 Functional PYNQ overlay that combines O1R1 and O2R1	
Specification	Description
S1	The overlay must fit in the Ultra96-v1 board.
S2	The functions corresponding to each functionality must be provided separately.
Requirement	
R3 If O3R1 cannot be met an alternative program must be provided that effectively utilizes O1R1 and O2R1 to achieve the same functionality.	
Specification	Description
S1	The program must minimize the performance impact of not achieving O3R1 .
Requirement	

R3 Python program that wraps **O3R1**, feeds the overlays their corresponding inputs and manages the outputs. If **O3R1** is not met, the **O3R3** must wrap **O3R2**.

3.2 Non-Functional Objectives

Objective 4

Objective **O4** defines what is considered “real-time” in this project and does so as a minimum of 3 frames per second for application.

Table 3.4: Requirements of objective O4

Requirement	
R1 The image processing part of each application must be able to process 3 frames per second.	
Specification	Description
S1	Image processing is measured, but not image capture or posterior management.
S2	FPS can be extrapolated from single measures or averaged from multiple processing of consecutive frames to counter potential bottlenecks.

Objective 5

Objective **O5** defines that the developed programs must be deployable in any PYNQ-enabled Ultra96-v1 board.

Table 3.5: Requirements of objective O5

Requirement	
R1 Following the steps in chapter 7 any of the applications must be deployable in an Ultra96-v1 board with the PYNQ firmware.	
Specification	Description
S1	The firmware version used in development is specified.
S2	The code is publicly available.
S3	Either the training materials or trained models are provided.
S4	Installation scripts for some steps are provided for convenience.

Objective 6

Objective **O6** states that each accelerated component must have a non-accelerated counterpart to compare it to, and the HW version must perform better than the HW version.

3. Technical Requirements

Table 3.6: Requirements of objective O6

Requirement	
R1 A pure software implementation must be provided for each hardware-accelerated part. The functionality and internal logic of the SW versions must be comparable to the HW ones.	
Specification	Description
S1	Each of the applications must also have a pure software implementation.
S2	The SW implementation must work similarly to the HW one.
S3	Benchmarks will compare both versions.
Requirement	
R2 Accelerated versions must have an execution time that is at least 60% smaller than the non-accelerated version.	

Objective 7

Objective **O7** states that the code must be clear and understandable.

Table 3.7: Requirements of objective O7

Requirement	
R1 The code must be consistent and as clear as possible.	
Specification	Description
S1	Function, type and variable names must be descriptive.
S2	Code syntax must be consistent within each program.
S3	Comments must clarify relevant sections of the code.

Objective 8

Objective **O8** states that the programs and underlying ideas must be documented.

Table 3.8: Requirements of objective O8

Requirement	
R1 Jupyter Notebooks detailing the steps followed in the program must be provided for each one of the applications.	
Specification	Description
S1	A single Jupyter Notebook must be provided for each application.
S2	The notebook must detail what is being done and why.
Requirement	
R1 The ideas behind the programs and the development documentation must be present in the state of the art and development sections of this document. Sections are chapters 6 and 7 respectively.	

Objective 9

Objective **O9** defines that the licenses given to the results of this project must be compatible with the ones of the dependencies and each other.

Table 3.9: Requirements of objective O9

Requirement
R1 The software licenses of the results must be compatible with the ones of the sources and libraries. There must be no licensing conflicts.

Chapter 4

Methodology and Resources

4.1 Kanban

Kanban methodology was used during the development of the project. Kanban comes from a procedure that was used to track the route of a product throughout a factory. Similarly, the current Kanban methodology tracks the state of tasks in a project. The methodology and word have Japanese origin, “kan” meaning “visual” and “ban” meaning “card”.

The Kanban methodology is based on a board. The board gives all the necessary information about the state of each task. It provides a convenient, quick way of determining the state of the project or how it is doing regarding the pacification. It also provides “agile” flexibility to the project as tasks can be easily and quickly moved or adapted depending on the requirements. Tasks in Kanban are assigned to cards. Cards are placed in columns. In this specific project, five columns were used. Columns are ordered and contain any number of cards. Cards can be moved from one column to any adjacent column. Figure 4.1 shows a screenshot of the Kanban board at one point of the project.

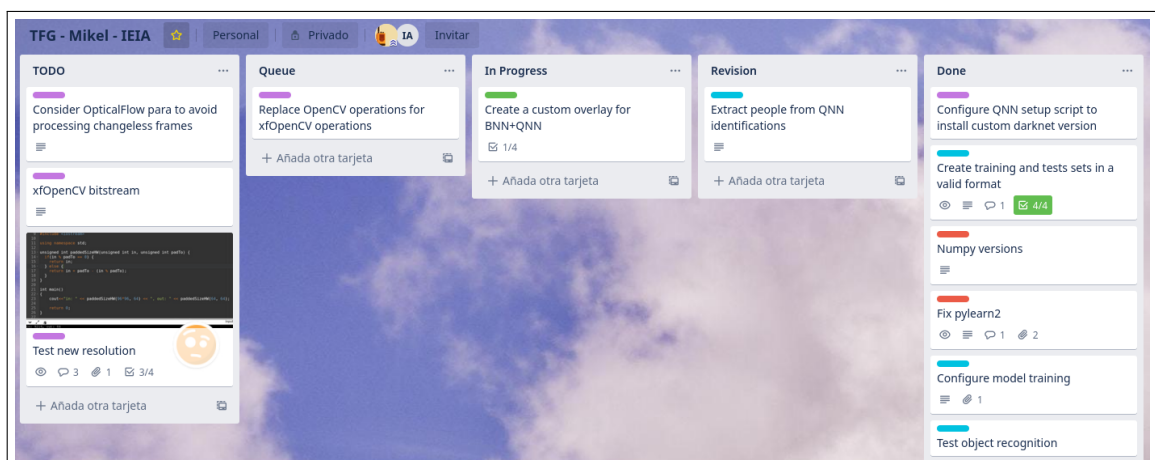


Fig. 4.1: Kanban board

4. Methodology and Resources

Instead of a physical board, a digital one provided by Trello was used, also seen in the screenshot of figure 4.1. The Trello version that was used is free, provides many functionalities, team and organizational capabilities. Each task can be commented on, contain checklists, media, descriptions and so on. It also has integration with many services including the ones of its parent company Atlassian or others such as GitHub. Although these integrations were not used in this project, they are still useful and worth mentioning. This is specially true when considering a lot of software developers use this methodology as means of development and communication with clients in the form of issues, which are user requests sometimes turned into tasks.

The five columns used in this project are listed below. The usual life cycle of a task starts in the first column and ends in the last, having gone through each other column in between at least one.

- **TODO.** This column contains tasks that have not been done yet and are not intended to be carried out in short term.
- **Queue.** The queue contains the tasks that have not been done yet but are intended to be carried out in the short term. Tasks are moved from Queue to the column to the right when started.
- **In Progress.** Contains all the tasks being currently under development.
- **Revision.** Tasks in this column have been developed but need to be revised to ensure everything is correct.
- **Done.** Contains all the completed tasks.

4.2 Resources

The human resources used for this project were the author, the director and an external consultant. The author and director will take many roles in the budget to more accurately convey the actual cost it would entail to have an engineer specialized in each field.

The digital resources will be analyzed in order of relevance. In the first place, the firmware, programming languages, frameworks, libraries and so on used during development. This includes PYNQ, interpreters and compilers... On second place, many Xilinx manuals were used. This, combined with multitude of articles about PYNQ applications and repositories such as BNN-PYNQ and QNN-MO-PYNQ was essential. All the code editing tools used in this project were free: Vim, Jupyter Notebooks and PyCharm. To edit this document, the Overleaf LaTeX editor was used with the template provided by the University of Deusto. Finally, the PYNQ forum was used to post a question regarding overlay compression, and even if no solution was found it was a helpful resource with great treatment and fast response times.

About physical resources, a Raspberry Pi 3 model B was used to run the initial programs in an ARM device. PYNQ experimentation began on a PYNQ-Z1 board but it was changed for a Ultra96-v1 board very early in the project. Access to both PYNQ boards, research and a lot of the development was done in a personal computer running GNU/Linux. The personal computer has

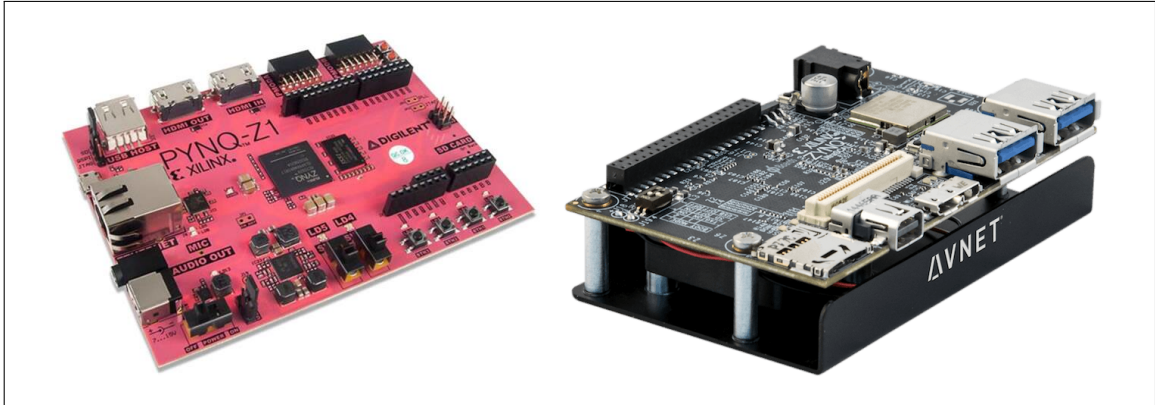


Fig. 4.2: Picture of the PYNQ-Z1 (left) and Ultra96 (right) boards

a GPU compatible with the machine learning model training libraries, which saved a considerable amount of ours. Vivado and other demanding software was all run in this computer. The boards were accessed either with Jupyter Notebooks or SSH.

Chapter 5

Budget and Planning

5.1 Budget

Table 5.1 shows the expected budget for human resources. There are four profiles: project director, researcher, developer and consultant. The project director coordinates the project and has an expected investment of 60 hours. The researcher profile handles research and documentation taking an expected amount 232 hours, while the developer writes, validates and tests the applications taking 120 expected hours. Consultant accounts for actual people who helped with the project either directly or through forums. The full schedule is detailed at section 5.2.

Table 5.1: Cost of human resources

Profile	€/hour	Hours	Total cost [€]
Project Director	45	60	2,700
Researcher (Student)	38	232	8,816
Developer (Student)	36	120	4,320
Consultant	72	10	720
Total cost:			16,556

The material cost of the project represents the cost of the material acquired for the development of the project, in this case the PYNQ-Z1 board and the Ultra96-board. It should be stated that neither supposed a real personal cost and both were provided by the university. The material costs disregarding electrical costs are shown in table 5.2.

Table 5.2: Cost of material resources

Material	Price (€)
PYNQ-V1	199
Ultra96-v1	259
Total cost:	458

The total cost is therefore 17,014€.

5.2 Planning

Figure 4.1 shows the expected schedule of the project. The schedule has been modified, although as little as possible. Sections about using certain libraries have been changed after choosing said library to fit them better. Decisions about linearity have also been made: although facial recognition and object detection could be developed in parallel, it was planned and done sequentially because it is more efficient for the researchers. The planning needed to take into account the documentation that was created during development.

Meetings with the project director took place weekly. These meetings consisted of discussing topics and results, reporting the state of the project and recent advancements and mapping out the immediate path to take afterwards. As a consequence of the *COVID-19* outbreak, remote meetings were had during great part of the project.

Figure 5.1 shows a Gantt diagram of the planned project schedule. The tasks correspond to the project structure explained in section 2.2. The first tasks of the project are centered around research and planning. The first development phase focuses on completing the facial recognition use case, while the second focuses on object detection. After that comes the integration of both use-cases, before writing the final documentation. Taking that into account, the following milestones can be defined:

- **Research on the new technology.** Expected start and finish dates: *from 2020-02-05 to 2020-02-24*. This phase involves learning about and familiarizing with the Zynq architecture and PYNQ framework. It is divided into three tasks: General research on the topics, the creation of simple programs for PYNQ and deciding what accelerated applications will be experimented next. *Result: planning of the next steps.*
- **Facial recognition.** Expected start and finish dates: *from 2020-02-25 to 2020-03-27*. This milestone will be reached when a functional facial recognition is successfully created and tested. With such a wide scope, it had to be split into three sub-stages while each of the sub-stages is divided into multiple tasks. Sub-stages are explained next. *Result: functional, accelerated facial recognition program.*
 - **Research and library verification.** Expected start and finish dates: *from 2020-02-24 to 2020-03-05*. This phase involves researching current developments on facial recognition applications on PYNQ or similar projects that could be useful for implementing it.

After the initial research, a library must be chosen, tested and benchmarked to ensure that results will be positive.

- **Development.** Expected start and finish dates: *from 2020-03-06 to 2020-03-24*. The library will likely need to be modified to serve the purpose of this project, which involves researching on the AI technology being used and how models can be trained. After that, a dataset needs to be created and the models trained, followed by validation and testing of the results.
- **Documentation.** Expected start and finish dates: *from 2020-03-25 to 2020-03-27*. Steps followed in the previous sub-phases must be documented to be reproducible.
- **Object detection.** Expected start and finish dates: *from 2020-03-30 to 2020-04-14*. Similarly to the facial recognition milestone, this one marks the successful development of an object detection application. Steps are similar to the ones of facial recognition, but knowledge gained during the previous phase should prove useful and reduce the development time of this phase. Each of the sub-phases is described below. *Result: functional, accelerated object detection program.*
 - **Research and library verification.** Expected start and finish dates: *from 2020-03-30 to 2020-04-02*. Research what the current state of the art is in object recognition on PYNQ, select a library and test it.
 - **Development.** Expected start and finish dates: *from 2020-04-03 to 2020-04-09*. This sub-phase involves modifying the library to fit the use-case before validating and testing the results.
 - **Documentation.** Expected start and finish dates: *from 2020-04-10 to 2020-04-14*. This sub-phase involves documenting the entire phase to ensure previous steps reproducible.
- **Integration of facial recognition and object detection.** Expected start and finish dates: *from 2020-04-15 to 2020-04-30*. The goal of this phase is to integrate both use cases into a single application. The proposed way of doing so is by integrating both overlays into one, therefore overlays and overlay design need to be more thoroughly studied. After that, a new design will be created, validated and tested. Finally the application will be benchmarked. Tasks from research to validation are expected to take *from 2020-04-15 to 2020-04-28*. The steps need to be documented, which is expected to happen *from 2020-04-29 to 2020-04-30*. *Result: integrated application of facial recognition and object detection.*
- **Documentation.** Expected start and finish dates: *from 2020-05-01 to 2020-06-04*. This milestone represents the completion of the project documentation that will be delivered. It needs to be written, verified and corrected iterative. *Result: project documentation.*

Figure 5.1 shows an additional task at the end of the project: “Project closure”. This involves preparing results by, for example, publishing private repositories where code was hosted during the development. The diagram shows a total of 88 days dedicated to the project, assuming four hours are dedicated each day. The amount of dedicated time will however depend on factors such as academic or professional workload. There is room for adjustment between the final day (June 3) and the thesis delivery deadline (June 24).

5. Budget and Planning

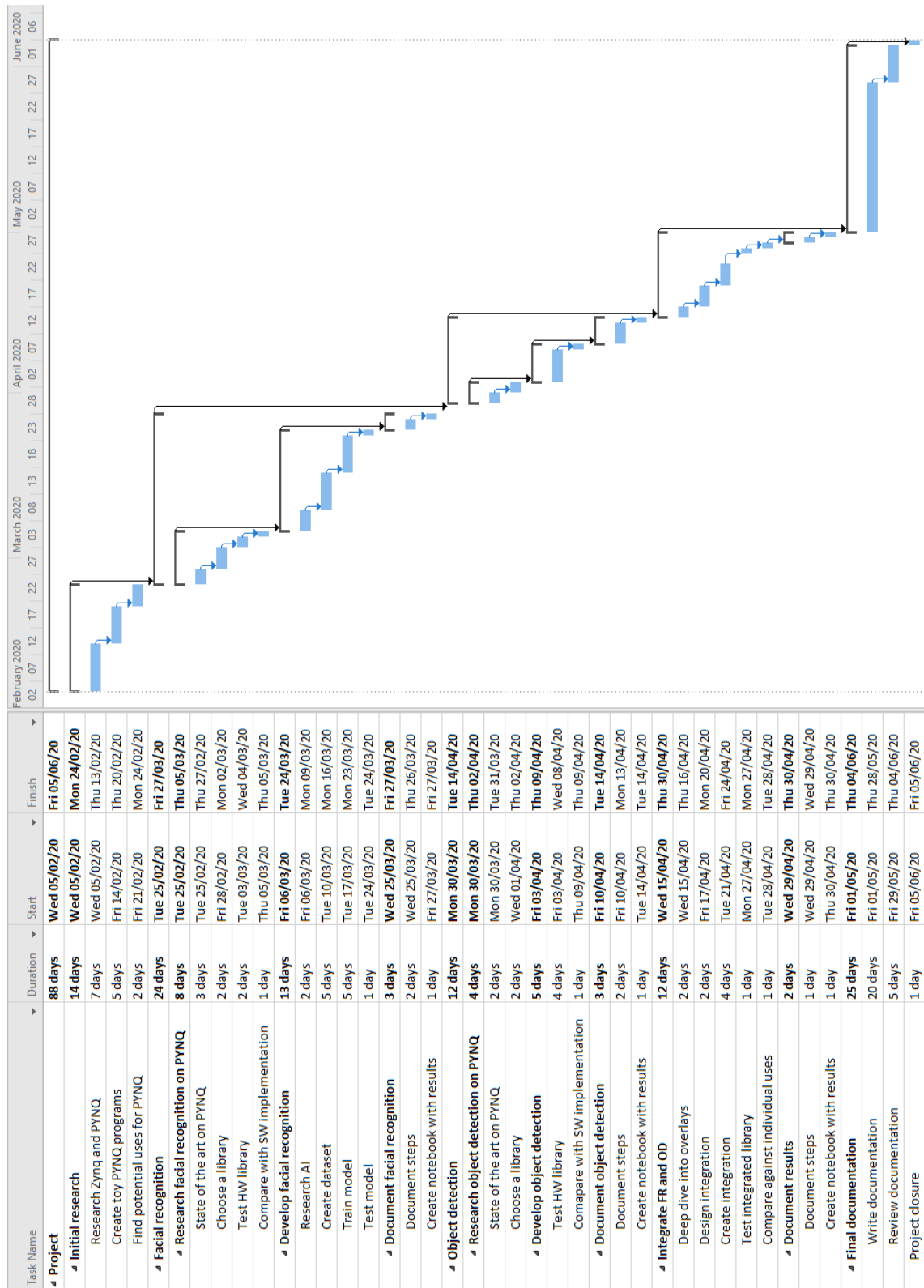


Fig. 5.1: Project schedule

Chapter 6

State of the Art

This section provides an overview to the relevant fields this project encompasses. Before continuing it is worth mentioning novelty of the adopted architecture and platform, and the studied projects such as FINN are the state of the art in the field of artificial vision in embedded devices. The other two analyzed topics in the “Preceding Projects” section are also pushing forward different scenarios where the platform provides performance advantages.

The platform is also innovative in another aspect: despite the advanced knowledge in programmable logic, as well as in the architecture of the buses that allow the intercommunication between the processors and the FPGA, the PYNQ architecture is not only intended for people with knowledge on that area. If a “library” (it is not officially called library, but in this point the concept helps shedding a light on the idea) with the intended functionality exists, the programmer has only to download it to the PL and call it from C, or in this case Python, code. While still in development and maturing, the PYNQ platform and underlying technologies discussed in this section may have a great impact on future embedded devices.

6.1 Key Areas

This section discusses edge computing and SoC. These are two of the areas that might be more impacted by the success of Zynq and PYNQ. Edge Computing methodology and applications are currently designed majorly for non-accelerated embedded devices or local servers, but Zynq could mean that embedded devices are capable of running tasks currently assigned to local servers in specific, accelerate cases. Regarding SoCs, the way Zynq contains and communicates two heterogeneous processors could find greater adoption and research if Zynq (or PYNQ and therefore Zynq) gain even more traction.

6.1.1 Edge Computing

Edge computing refers to a paradigm shift where data processing and storage of IoT applications are partially moved near where the data is acquired. It is not a new concept but is becoming increasingly popular with the rise of IoT related industries. In the context of this project, edge computing is done with low-power devices not comparable to the central servers of a service and thus lacking the required power to perform complex manipulations to the data. That however may change with the heterogeneous MPSoC architectures introduced in the SoC section. Edge computing can reduce the latency and broadband requirements of the system, as well as lighten the server load. The main advantage is clear: it benefits efficiency and flexibility. It may, however, involve increased complexity and risks. Each device adds more complexity to the overall system, which is specially true when each device is complex on itself. Adding more critical devices reduces the importance of each device, which translates to a reduced impact if a single device fails. From a security point of view, it also increases the attack surface of the system, aside from requiring new data policies (e.g. an edge computing with confidential data may be more vulnerable than central servers).

The benefits and shortcomings of edge computing should be analyzed for each application, but in this specific project, it is considered favorable as it would considerably reduce critical factors such as network saturation or latency. The device and platform examined from this point of view in this thesis: as potential edge computing devices that stand out from other products for their unique characteristics.

6.1.2 System-on-Chip

A System-on-Chip or SoC integrates the components of an electronic system in a single, integrated circuit. In the case of the Zynq architecture explained in section 6.2, the system contains an ARM processor, an FPGA, memory and IO components. Having multiple processing units means it is a Multiprocessor System-on-Chip or MPSoC. SoC is a common technology in modern markets such as mobile and edge computing, which is closely related to this subject.

The integration of SoC implies multiple benefits, the most notable ones being the reduced area usage, power consumption and latency, and increased performance than their multi-chip counterparts. As mentioned before, MPSoCs could gain popularity amongst engineers and researchers based on the success of Zynq.

6.2 Platform Overview

This section studies both the Zynq architecture and the PYNQ framework. Zynq UltraScale+ and PYNQ are very recent developments, in fact, PYNQ was only created a few years ago and while it has already proven itself really useful and grown a community, it still has a lot of growth potential.

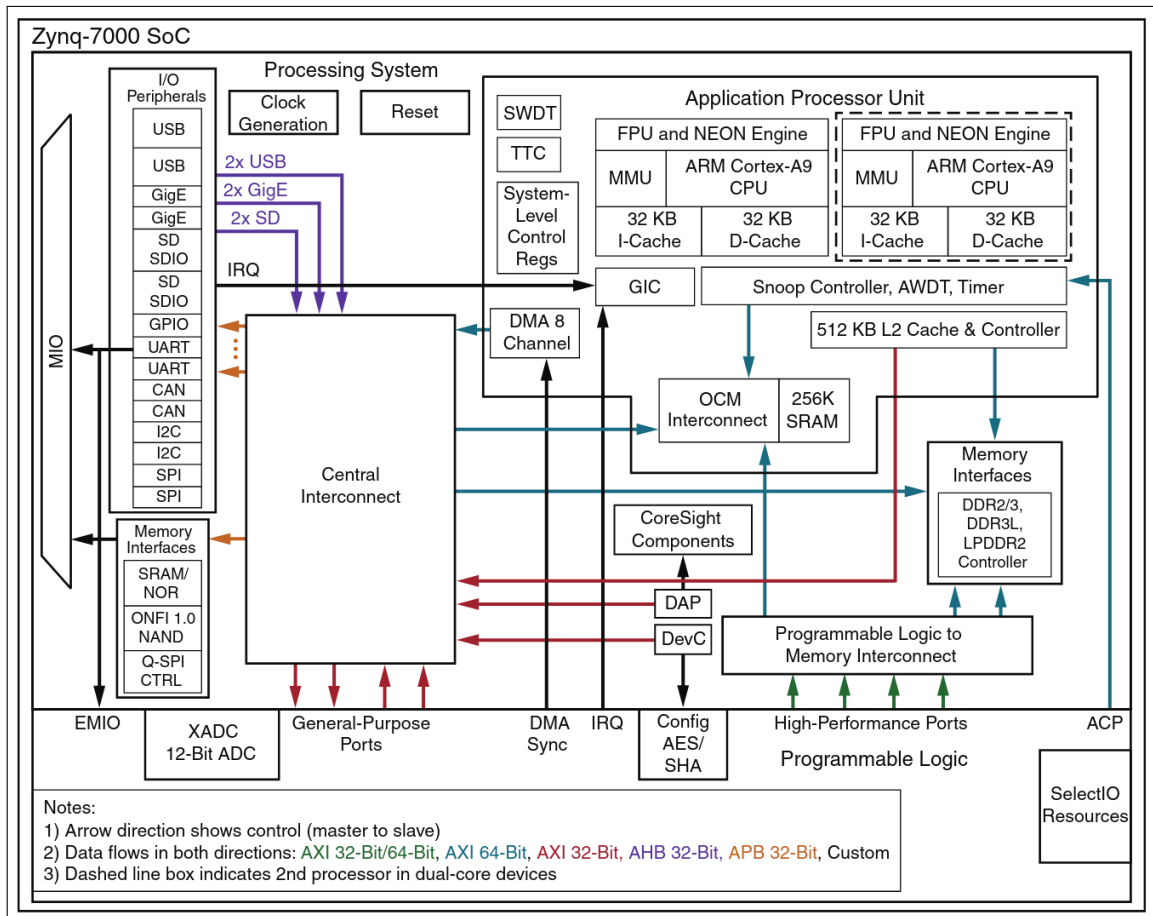


Fig. 6.1: Zynq-7000 architectural overview (source: [2])

6.2.1 Zynq

This project uses Zynq UltraScale+ architecture [1], the successor to Zynq-7000. This document refers to Zynq as an inclusion of both, knowing that UltraScale+ is the most modern architecture. Zynq-7000 is a family of heterogeneous SoCs designed by Xilinx for complex tasks such as the machine vision studied in this project. It is designed with a processor-centric approach instead of the more traditional FPGA-centric approach. The processor-centric approach means the PS boots first and standard processes such as running the operating system or interacting with the board are done as in other, single processor devices. This makes the development process more natural to the majority of developers. This section analyzes the Zynq-7000 architecture and explicitly remarks differences with UltraScale when relevant.

Figure 6.1 shows the functional blocks and connections of functional blocks of the Zynq-7000 architecture. The processors are divided into the processing system or PS and the programmable logic or PL. The PS and PL communicate via AXI interfaces. AXI interfaces are designed for on-chip communications and offer benefits for high-performance, parallel, synchronous communication. The bit size of the ports varies, some being of 32 bits, others of 64 and 128 bits. The AXI interfaces offer lower latency and higher bandwidth than more traditional connections. The AXI connections are seen in figure 6.1 as arrows between functional blocks. The legend of the figure gives additional

6.State of the Art

information as to how to interpret the connections.

The PS can be seen in the upper half of figure 6.1 contains the application processor unit, memory interfaces, IO peripherals and the interconnect interface. Zynq 7000 devices have 32 bit ARM processors, while Zynq UltraScale+ devices have 64-bit processors. Zynq UltraScale+ devices also include a dual-core RPU and an optional GPU [3]. These paragraphs will examine each one of the main components in more detail. Each core of the processor in the APU has an L1 cache, there is an L2 cache that is shared between processors. It is worth noting that while the APU in figure 6.1 has two cores, the number of cores for Zynq UltraScale+ devices may be either two or four. It has low latency access to the on-chip RAM described later in this section. Although not explicitly discussed here, it also has standard features such as interruptions, timers, tracing mechanisms and a MMU to manage virtual memory.

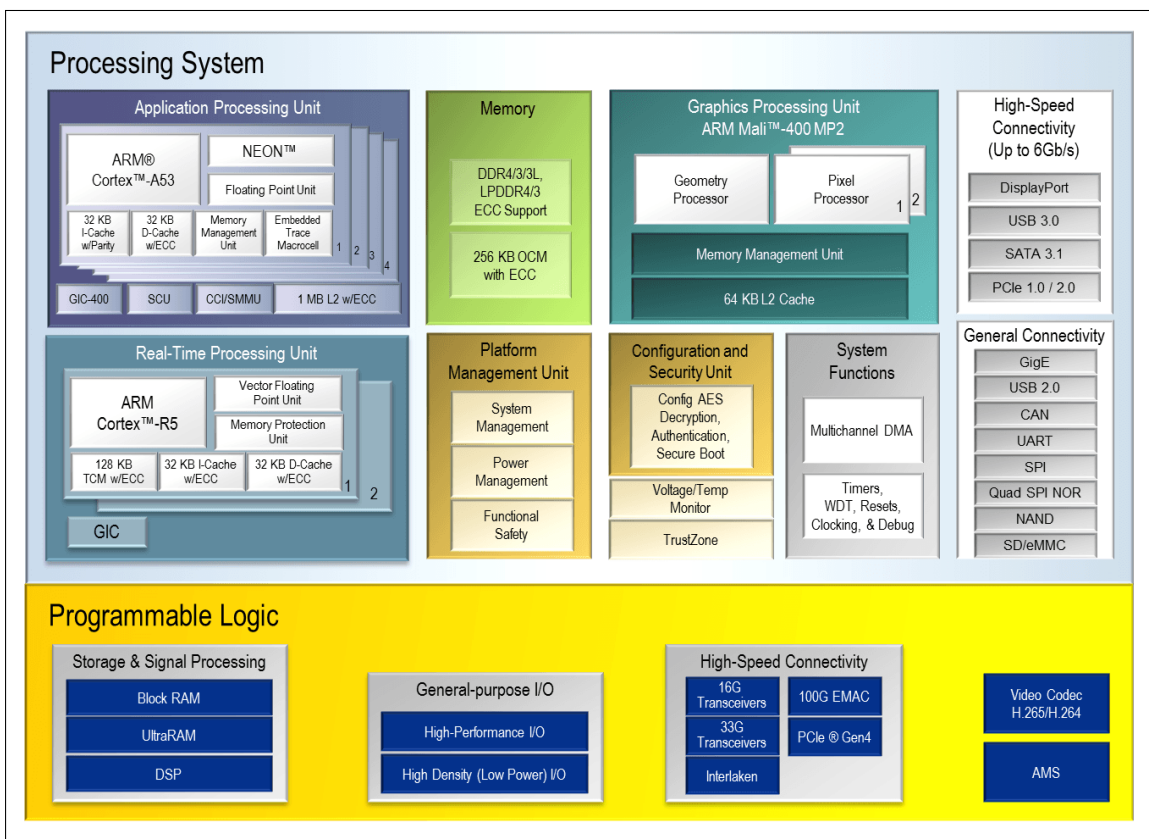


Fig. 6.2: Zynq-UltraScale+ component overview (source: [4])

The PS has two memory interface types: dynamic and static. The dynamic memory interfaces handle the CPU L2 caches, shared memory with the PL and interconnect components. They include a multi-protocol DDR memory controller that enables shared access to memory from both the PS and PL via AXI ports. The IO peripherals are explained in section 6.3, but it is worth noting that they can be exposed via multiplexed IO pins (MIO) or combining MIO and pins belonging to the PL. That combination is denominated extended MIO or EMIO. “Interconnect” refers to the connection between architectural components: APU, PL, IO peripherals and so on are connected to each other via multilayered ARM AMBA AXI connections, again as shown in

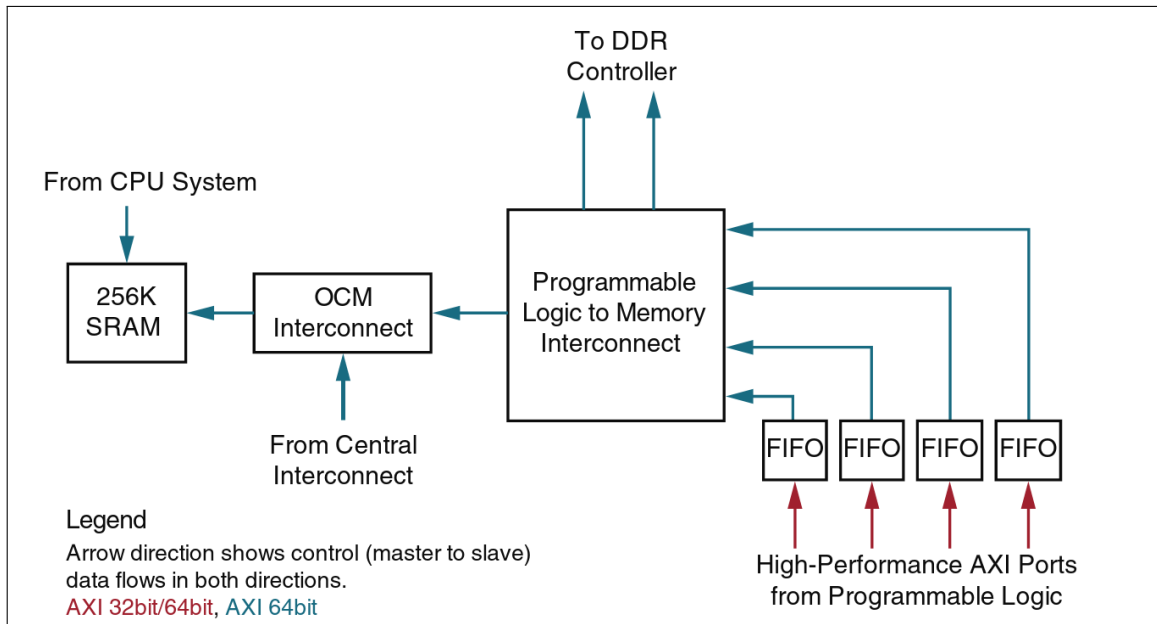


Fig. 6.3: PL interface to PS memory (source: [4])

figure 6.1. According to the datasheets [4] the interconnect is designed in such a way that the most time-sensitive modules that require minimal latency have the shortest paths to memory. A QoS block exists to regulate traffic between the components. Figure 6.2 shows the components of the Zynq UltraScale+ architecture, which as already mentioned contains more features than the Zynq 7000 architecture.

Zynq UltraScale+ devices also include memory protection and peripheral protection units developed by Xilinx, additional units such as a platform management unit or a configuration and security unit, and a more advanced DMA controller. Lastly, more standard and high-speed peripheral connections are available than in Zynq-7000.

Regarding the PL, LUTs can be used either as a single 6 input 1 output LUT or as two 5 input 1 output LUTs. The equivalence is a 64 bit ROM for the single, 6 input LUT and two 32 bit ROMs for the dual 5 input LUTs. CLBs are composed of two slices that are themselves composed of four LUTs and corresponding components (e.g. two flip-flops corresponding to each LUT).

The PS-PL interface is complex and includes many components, protocols and ports. As seen in figures 6.1 and 6.3, communication from PL to PS happens when the PL writes to the four corresponding AXI ports. The writes are buffered with FIFO controllers. The output ports then go to either the DDR or OCM. This provides the PL a way of accessing the DDR and OCM memories of the PS, but when coherence is required in the access, the ACP interface should be used, shown in figure 6.1, which goes through the corresponding controllers.

Summing up, Zynq is an architecture that offers a PS-centric approach and a low-consumption, low-latency, highly efficient connection between the PS and PL.

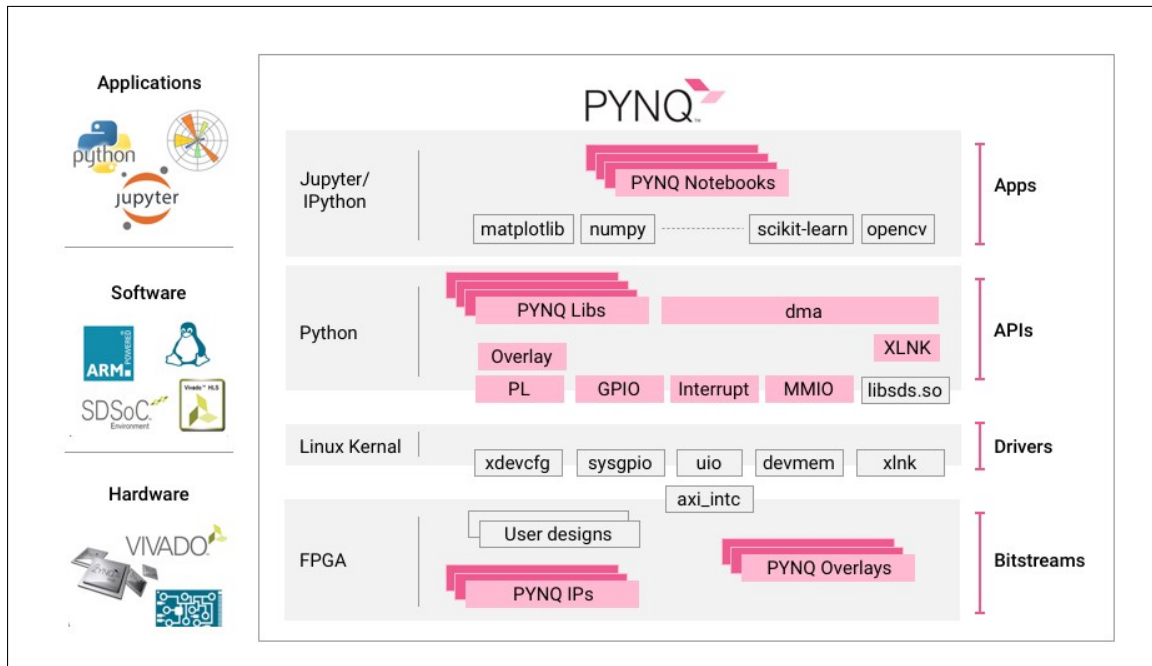


Fig. 6.4: PYNQ application overview (source [5])

6.2.2 PYNQ

PYNQ is an open-source framework that enables development for the Zynq architecture without requiring the design of PL circuits. Software for PYNQ is developed in Python. The framework has Jupyter Notebooks installed so academical or explanatory documents containing executable, live code can be created. This is useful not only for getting used to the framework while going through the notebooks but also to develop in an organized way and to publish the research results in an accessible manner. Multiple introductory notebooks are included in PYNQ to get familiarized with the framework. An overview of PYNQ is shown in figure 6.4. The framework includes a Python library, documentation, programs that serve as examples, amongst others.

PYNQ is also the name of the Python library. The library setup scripts contain functions to validate the device it is installed on and to install additional utilities such as the documentation or the introductory Jupyter Notebooks. PYNQ can be downloaded from prebuilt binaries or manually compiled from an appropriate version of the repository in a supported device.

The library exposes many functionalities of Zynq, such as IO operations, PS/PL management or memory management. It provides acceleration abstraction using overlays. A PYNQ overlay is a hardware library or PL design that allows the implementation of applications to the PL in addition of the PS. An API for controlling the overlays and thus the PL from Python is offered in the PYNQ library. The accelerated part of an application is located in an overlay. Designs can be downloaded to the PL without overlays by using bitstreams. Overlays include both a bitstream (.bit) and scripting (.tcl) files that usually enables, amongst other things, to identify Zynq system configurations or to rebuild the Vivado project from the overlay. Both overlays and bitstreams have corresponding Python classes in the library.

Additionally, PYNQ is reliant on NumPy [6] for data management and is compatible with Asyncio [7]. As explained in section 6.2.2.2, NumPy is a scientific computing library and Asyncio offers a high level concurrency model, both very useful for PYNQ enabled devices.

PYNQ is pushing boundaries, allowing developers without expertise on programmable logic to develop programs that run on FPGAs with the considerable performance benefits it implies.

6.2.2.1 PYNQ application example

The PYNQ Hello World repository [8] created by Xilinx provides a good example to visualize how PYNQ allows exploiting the benefits offered by Zynq without diving too deep into architecture specifics. The example shows only the most relevant segments, but the full example can be accessed at the source. In this specific example, the PL will be used to scale an image from dimensions $A \times B$ to dimensions $C \times D$. This is done with an IP of the `resizer.bit` overlay. Listing 6.1 shows the overlay download and the assignation of the DMA, resizer IP and memory allocator to Python variables.

```

1 resize_design = Overlay("resizer.bit")
2 dma = resize_design.axi_dma_0
3 resizer = resize_design.resize_accel_0
4 xlnk = Xlnk()

```

Listing 6.1: "IP extraction from the resizer overlay"

After opening an image and saving it in a variable called `image`, the values forming the image exist only in the local memory of Python. By allocating shared memory with `xlnk`, the developer is able to save and read data from addresses that will also be accessible to the PL thanks to the DMA. This is done with the code shown in listing 6.2. The image will be resized from $A \times B$ to $C \times D$, each with three 8-bit color channels, thus the shape of the array.

```

1 in_buffer = xlnk.cma_array(shape=(A, B, 3), dtype=np.uint8, cacheable=1)
2 out_buffer = xlnk.cma_array(shape=(C, D, 3), dtype=np.uint8, cacheable=1)
3 in_buffer[:] = np.array(image)

```

Listing 6.2: "Memory allocation"

The last statement makes a deep copy of the image that was on the local Python memory to the shared memory. It is necessary to make a deep copy, as by default Python uses references when copying lists and other objects. Figure 6.5 shows an scheme of Zynq at this point of execution.

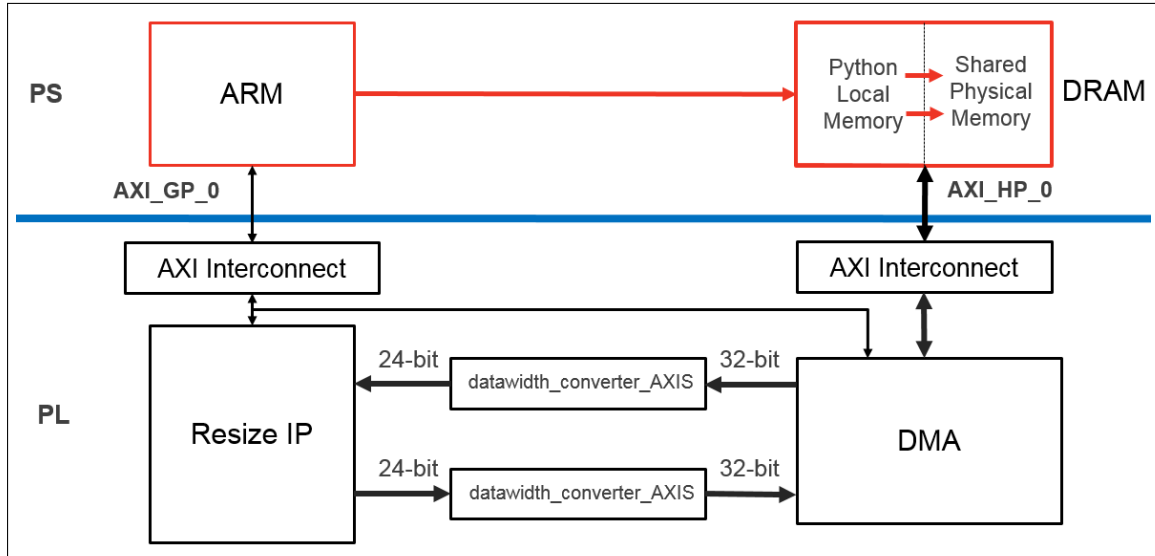


Fig. 6.5: State of the program with image in local and shared memory (source [8])

The next step is to make the data accessible to the PL, wait for it to process it, and read the results. Listing 6.3 writes the old and new dimensions to the corresponding registers (0x10 for A, 0x18 for B and so on), associates the shared memory previously allocated with the PL, starts the process by writing to the corresponding segment (0x81) and waits for the result to be written to the memory address associated to the output.

```

1 resizer.write(0x10, A)
2 resizer.write(0x18, B)
3 resizer.write(0x20, C)
4 resizer.write(0x28, D)
5
6 dma.sendchannel.transfer(in_buffer)
7 dma.recvchannel.transfer(out_buffer)
8 resizer.write(0x00, 0x81)
9 dma.sendchannel.wait()
10 dma.recvchannel.wait()

```

Listing 6.3: "PL usage"

It should be mentioned that these steps can easily be abstracted from the end user by providing higher-level functions that handle specific addresses, thus in some PYNQ applications it will be managed by the library itself instead of the library user. Figure 6.6 shows the components involved in the data being processed by the PL.

This example shows how, despite requiring some knowledge of the system, developing applications with PYNQ can be far more productive than doing so for other architectures that require different programming languages, architectural knowledge and communication management. Setting up the PL is trivial, as well as communicating it with the PS. Having the overlays loaded from files means they can be easily loaded for specific applications as long as the bitstreams are available on the board. The full example in the form of a Jupyter Notebook can be accessed at the original repository [8].

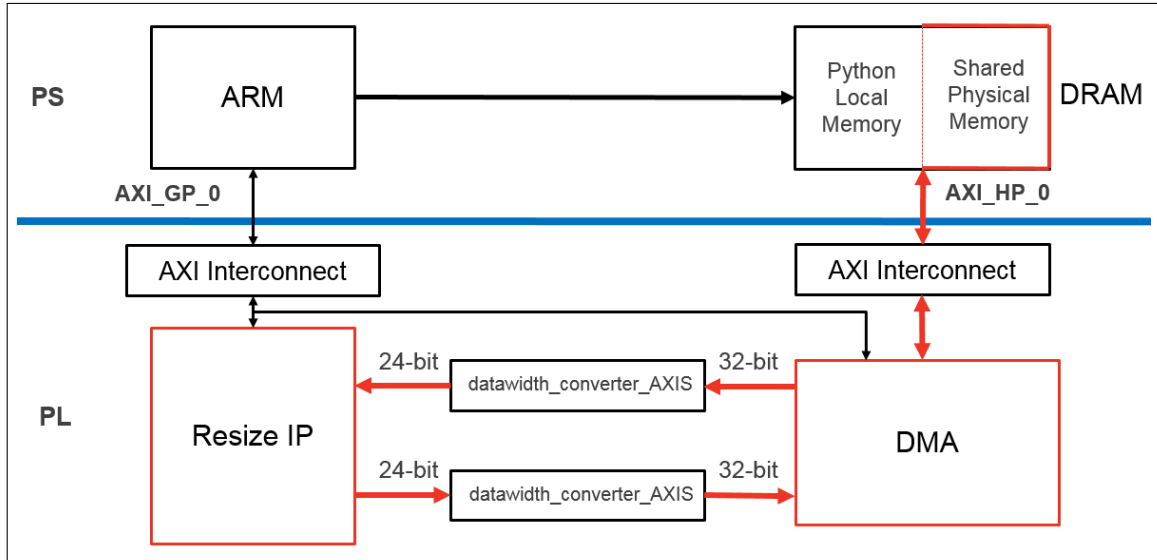


Fig. 6.6: State of the program when using the PL (source [8])

6.2.2.2 Python

Python is a multi-paradigm programming language and is one of the most popular, growing programming languages. Its high level design but low-level capabilities make it the language of choice for a very wide range of applications. Being an interpreted language and having interactive tools like Jupyter Notebook also makes it a good choice for PYNQ. The standard implementation of the Python interpreter, CPython, is written in the C programming language. The language has compatibility with C via bindings, which are handy when performance is required and allow the usage of C libraries from Python. This is specially useful in microcontrollers and is used in this project, as explained in section 7.3.1. Its wide adoption, multi-paradigm design, compatibility with other languages and range of options when choosing interpreters or compilers have resulted in Python being available for many different types of devices: from high-performing servers to personal computers and microcontrollers.

The aforementioned NumPy is a python package that intended for vector and matrix manipulation. The performance demanding parts are written in C instead of Python, which is advantageous not only regarding performance but also data type compatibility. Asyncio, as mentioned before, is used to write concurrent python code by running functions as co-routines that are executed concurrently. Actions such as waiting for resources can be executed without blocking the flow of the program, useful for devices that use low power or have a low core count such as the one used in this project.

Despite its many advantages, Python is still slower and consumes more resources than C. C is called multiple times from Python in the libraries used in this project, and how this is done will be explained in sections 7.2.2 and 7.3 in the “Development” chapter.

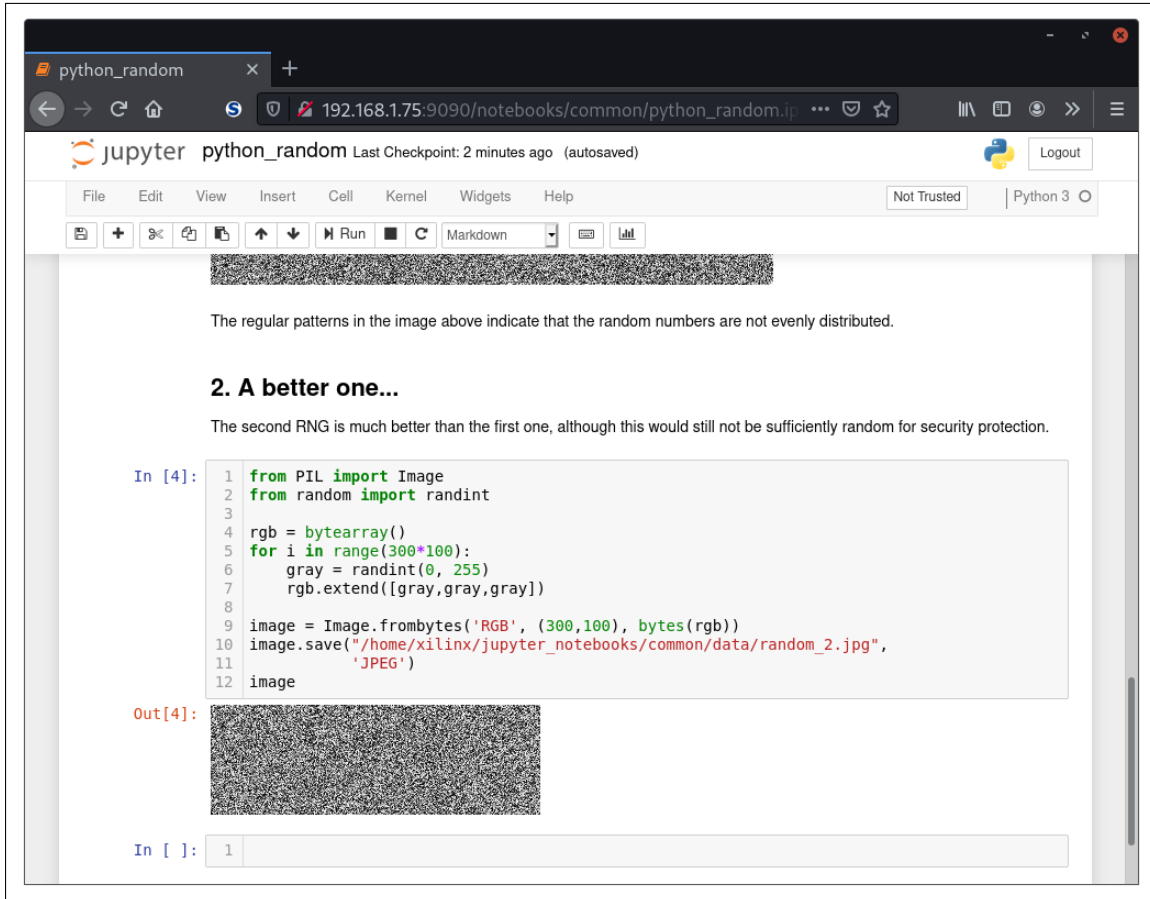


Fig. 6.7: Jupyter Notebook example

6.2.2.3 Jupyter Notebooks

Jupyter Notebook is a web application used to create documents that contain executable code as well as multimedia resources, equations, text-formatting and many of the features expected from document editors. The code can be either Julia, Python or R forming the name *Ju-Pyt-R* or *Jupyter*. Like many other tools and resources used in this project, Jupyter Notebooks is open-sourced and free software. The code inside a document is organized in blocks that are managed by “kernels”, programs in charge of event management (including code execution) that can be managed separately. The kernels, as well as the rest of the components, run in the host machine and thus the execution of the code is remote.

Figure 6.7 shows a screenshot of a Jupyter Notebook running on a PYNQ device. The notebook shown in the figure is included in PYNQ to show how Jupyter Notebook works with a random noise generation example. The notebook is accessed with a web browser and has two main parts: the menu and the notebook. The menu contains options for the Jupyter Notebook environment, the specific notebook and information about the status of the notebook or the programming language being used. The notebook contains text and blocks of code that can be executed at any moment, with the output of the execution being displayed right beneath the block.

6.3 Board Overview

The board used in this project is the Ultra96-V1. It reached the end of its short life when it stopped being produced in April 2019 to be replaced by the Ultra96-V2. The main difference between the V1 and V2 is the radio module, which does not affect this project. From this point the Ultra96-V1 will also be referred to as Ultra96 for brevity, despite actually Ultra96 being the family of boards the Ultra96-V1 and the Ultra96-V2 are part of. Figure 6.8 shows a picture of the Ultra96-v1 board.

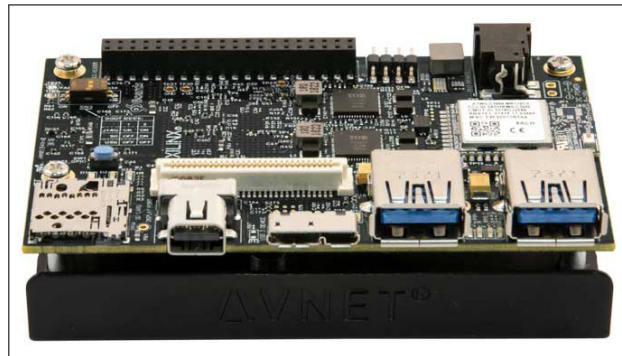


Fig. 6.8: Picture of the Ultra96-v1 board (source: [9])

It's a Zynq UltraScale+ device and the basic specifications of the board components are listed below.

- Xilinx Zynq UltraScale+ MPSoC ZU3EG A484 SoC
- 2GB LPDDR4 memory in a 512M x 32 configuration by Micron
- 802.11b/g/n Wi-Fi and Bluetooth 4.2 modules. The Bluetooth module supports for BC and BLE connections.
- micro-SD card slot
- Mini DisplayPort
- 8V-18V, 3A power source with connector with 1.7mm inner and 4.8mm outer diameters

According to the product tables [2] the ZU3EG MPSoC, the processing system has the specifications listed in figure 6.9. The specifications of the programmable logic are listed in figure 6.10. The PS and PS components are connected by 12 AXI ports of 32/64/128b. According to the MPSoC ordering nomenclature described in [2], the complete name of the device, ZU3EG SBVA484, can be divided into the following descriptors:

- **XC**: "Xilinx Commercial"
- **ZU**: Zynq UltraScale+ product.
- **3**: Value index of the product. Some specifications of the product vary according to the value index, and all variations are detailed in the product table [2]. The most notable ones include the number of memory blocks and the signal data transmission rate.

Processing System (PS)	Application	Processor Core	Quad-core ARM® Cortex™-A53 MPCore™ up to 1.5GHz	
	Processor Unit	Memory w/ECC	L1 Cache 32KB I / D per core, L2 Cache 1MB, on-chip Memory 256KB	
	Real-Time	Processor Core	Dual-core ARM Cortex-R5 MPCore™ up to 600MHz	
	Processor Unit	Memory w/ECC	L1 Cache 32KB I / D per core, Tightly Coupled Memory 128KB per core	
	Graphic & Video Acceleration	Graphics Processing Unit		Mali™-400 MP2 up to 667MHz
		Memory		L2 Cache 64KB
	External Memory	Dynamic Memory Interface		x32/x64: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3 with ECC
		Static Memory Interfaces		NAND, 2x Quad-SPI
	Connectivity	High-Speed Connectivity		PCIe® Gen2 x4, 2x USB3.0, SATA 3.1, DisplayPort, 4x Tri-mode Gigabit Ethernet
		General Connectivity		2xUSB 2.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
	Integrated Block Functionality	Power Management		Full / Low / PL / Battery Power Domains
		Security		RSA, AES, and SHA
AMS - System Monitor			10-bit, 1MSPS – Temperature and Voltage Monitor	

Fig. 6.9: PS specifications (source: [2])

- **E:** Quad APU, Dual RPU, Single GPU architecture.
- **G:** Specifies that the engine is designed for general purposes.
- **S:** Flip-chip with 0.8 mm ball pitch.
- **B:** Lidless.
- **V:** Refers to a hazardous substance use in electronic devices, in this case it is RoHS 6/6.
- **A:** Package designator.
- **484:** Pin count.

Figure 6.10 is missing the speed grades, which in EG devices can take the following three values depending on the temperature grade. If the temperature grade is Extended (E), the values are -1, -2 or -2L (slowest, mid and low-power respectively) and support temperatures between 0° and 100°C. If the temperature grade is Industrial (I) the temperature ranges from -40°C to 100°C and the -2L value is replaced by -1L, a slower low-power mode. The following values are missing decimals in the figure: System Logic Cells: 154,350; CLB-Flip-Flops: 141,120; CLB LUTs: 71,560. These values are still in (K).

Programmable Logic (PL)	Programmable Functionality	System Logic Cells (K)	154
		CLB Flip-Flops (K)	141
		CLB LUTs (K)	71
	Memory	Max. Distributed RAM (Mb)	1.8
		Total Block RAM (Mb)	7.6
		UltraRAM (Mb)	-
	Clocking	Clock Management Tiles (CMTs)	3
		DSP Slices	360
	Integrated IP	PCI Express® Gen 3x16	-
		150G Interlaken	-
		100G Ethernet MAC/PCS w/RS-FEC	-
		AMS - System Monitor	1
	Transceivers	GTH 16.3Gb/s Transceivers	-
		GTY 32.75Gb/s Transceivers	-

Fig. 6.10: PL specifications (source: [2])

Artificial vision, the target application of this project, is comparable to some of the activities listed in the aforementioned product table as an intended use for “EG” devices. The full list includes cloud computing security, data center, flight navigation, machine vision, medical endoscopy, missile and munitions, military construction, networking, and secure solutions.

Figure 6.11 shows a block diagram of the main components of the board. The center of the image shows the memory, PS and PL blocks while the laterals show IO components.

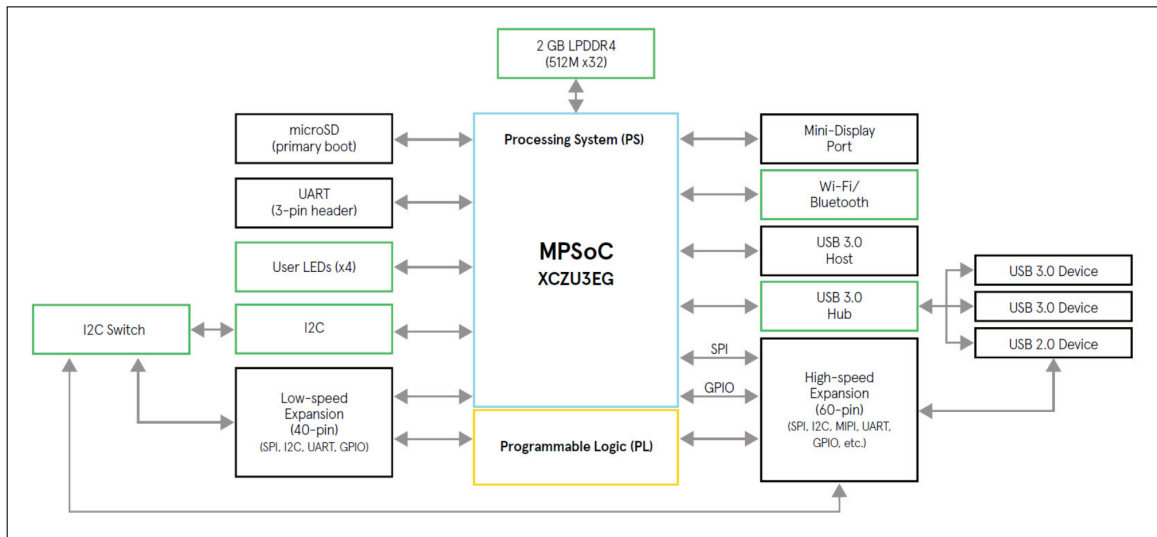


Fig. 6.11: Ultra96 block diagram (source: [10])

A micro-SD card is used as permanent storage and contains the operating system, as explained in section 6.2. The firmware version used in this project is the v2.4, which contains the “pynq 4.14.0-xilinx-v2018.3 aarch64” Linux version.

6.4 High-Level Synthesis

Historically, developing complex applications for FPGAs has in many cases been noticeably more difficult than doing so for other processor types, and has usually required more resource and time investment as well as a deep understanding of the architecture the programs are being design for. Vivado High-Level Synthesis or HLS [11] aims to solve that problem by providing a way of develop software to be run on FPGAs using programming languages such as C or C++. The HLS compiler receives source code written in C, C++ or SystemC and converts the functionality of the program to FPGA designs on Verilog and VHDL. It does so by extracting the data-flow and control structures from the source code and implementing its hardware equivalent. Some structures can be implemented in several ways, HLS deals with this ambiguity by following specific design directives that may, for example, consume less resources but offer less throughput or the other way around. HLS provides some degree of flexibility as to how functions and control structures are implemented in order to best fit specific designs by optimizing latency or area. Code segments are converted into state-machines that can be implemented in hardware, talking into account types, variables, functions, I/O and operations. An RTL is created from each function [12]. Figure 6.12 shows an

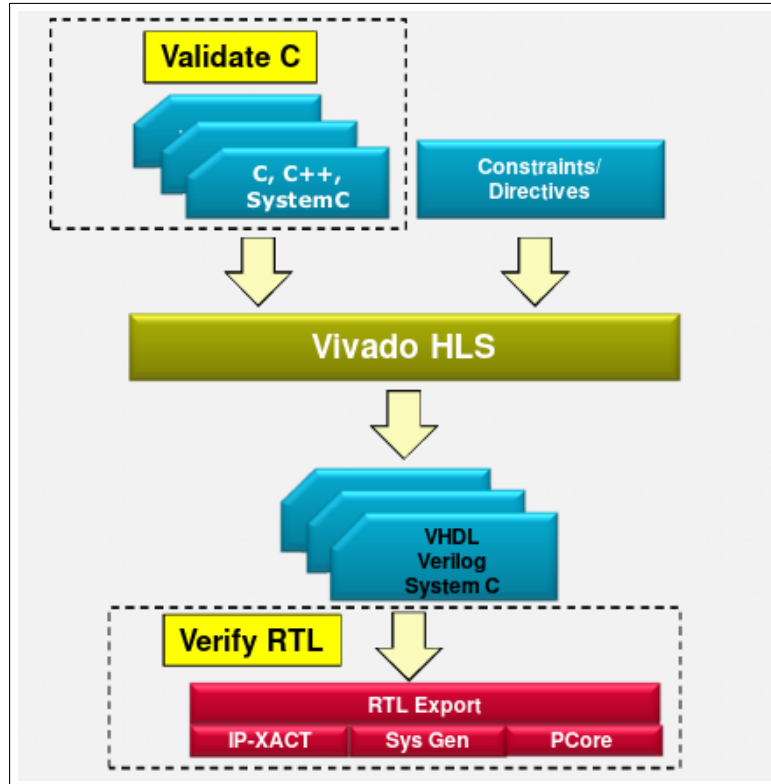


Fig. 6.12: HLS validation and verification (source: [12])

overview of the input and output of the HLS compiler.

The compiler has two main phases: scheduling and binding. Scheduling is the action of determining when operations will occur while binding assigns hardware units to the operations. The HLS compiler offers source validation and design verification as shown in figure 6.12. The “Vivado High-Level Synthesis” chapter of [11] gives deeper insight into how the HLS compiler performs transformations and optimizations. The capability of automatically creating FPGA designs from C source code is invaluable regarding project complexity and efficiency, which is why the applications described in section use HLS to implement certain types of neural networks on FPGAs.

6.5 Preceding Projects

This section discusses two relevant, existing projects for this thesis, and FINN which is a dependency both preceding projects have.

6.5.1 FINN

FINN [13] is a framework developed by Xilinx Research Labs that enables deep neural network inference on FPGAs. It is based on the idea that convolutional neural networks usually have redundancies that mean the network can be reduced with little cost to their accuracy, making

them a valid target for FPGAs. Figure 6.13 shows every step a trained neural network goes through to being implemented in a FPGA and be callable from PYNQ.

The creation of the trained neural network is detailed in sections 7.2.1 and 7.3.1 where the steps are followed to create different networks for the project applications. For now, the network is trained using tools described in the aforementioned sections and formatted for FINN. After that, FINN converts the model to the ONNX open standard. The next stage is the network preparation: sequential steps were the model is transformed, converted to Vivado HLS layers and optimized. HLS enables programs written, in this case, in C++ to be targeted into Xilinx devices, in this case, the PL of the Zynq UltraScale+ device. The HLS compiler optimizes the design latency, power and throughput, checking program operations and control structures for potential improvements. The C++ source files of the FINN library are available on a repository hosted on GitHub [15]. The next steps consist on creating the IPs and Vivado designs of the program, which will become vital in section 7.4. The last phase, which in 6.13 is called “PYNQ HW Gen & Deployment”, considers both the bitstream generation and the PYNQ code generation, which can be imported from Python when using the generated bitstream. In some cases, like the ones mentioned at the beginning of the paragraph, said code is wrapped and abstracts developers from manually downloading the overlay.

6.5.2 Binarized Neural Networks

Xilinx developed a BNN repository, BNN-PYNQ [16], for PYNQ which uses FINN and includes explanations on how it works and how to train models of the repository. BNNs use binary (± 1) weight and activation instead of floating point values [17], significantly reducing the computational complexity of connections in the neural network. The BNNs of this project can take different values for weights and activation functions. There are three implementations in the project:

- 1b weights and 2b activation functions, W1A1
- 1b weights and 2b activation functions, W1A2
- 1b weights and 2b activation functions, W2A2

W1A1 and W1A1 are used for both LFC and CNV network topologies, while W2A2 are only used for CNV topologies [13]. LFC networks have three fully connected layers, each layer with 1024 neurons. They take a 28x28 (grayscale image of 28x28 pixels) array as input and output a 10 bit value in One-Hot format, explained in section 7.2.1. CNV networks on the other hand have different types of concatenated layers. They take a 3x8x32x32 (8 bit RGB image of 32x32 pixels) array and output a 10x16 vector. LFC and CNV topology diagrams can be seen in figures 6.14 and 6.15 respectively.

BNNs are used both in [17] and the BNN-PYNQ repository to classify the same types of images. The first type of images they are used to classify are handwritten digits, for which the models are trained with the MNIST data-set. The second type classifies single objects in an image and is trained with the CIFRAR-10 data-set. BNNs are usually used for machine vision, as can be seen in the aforementioned examples.

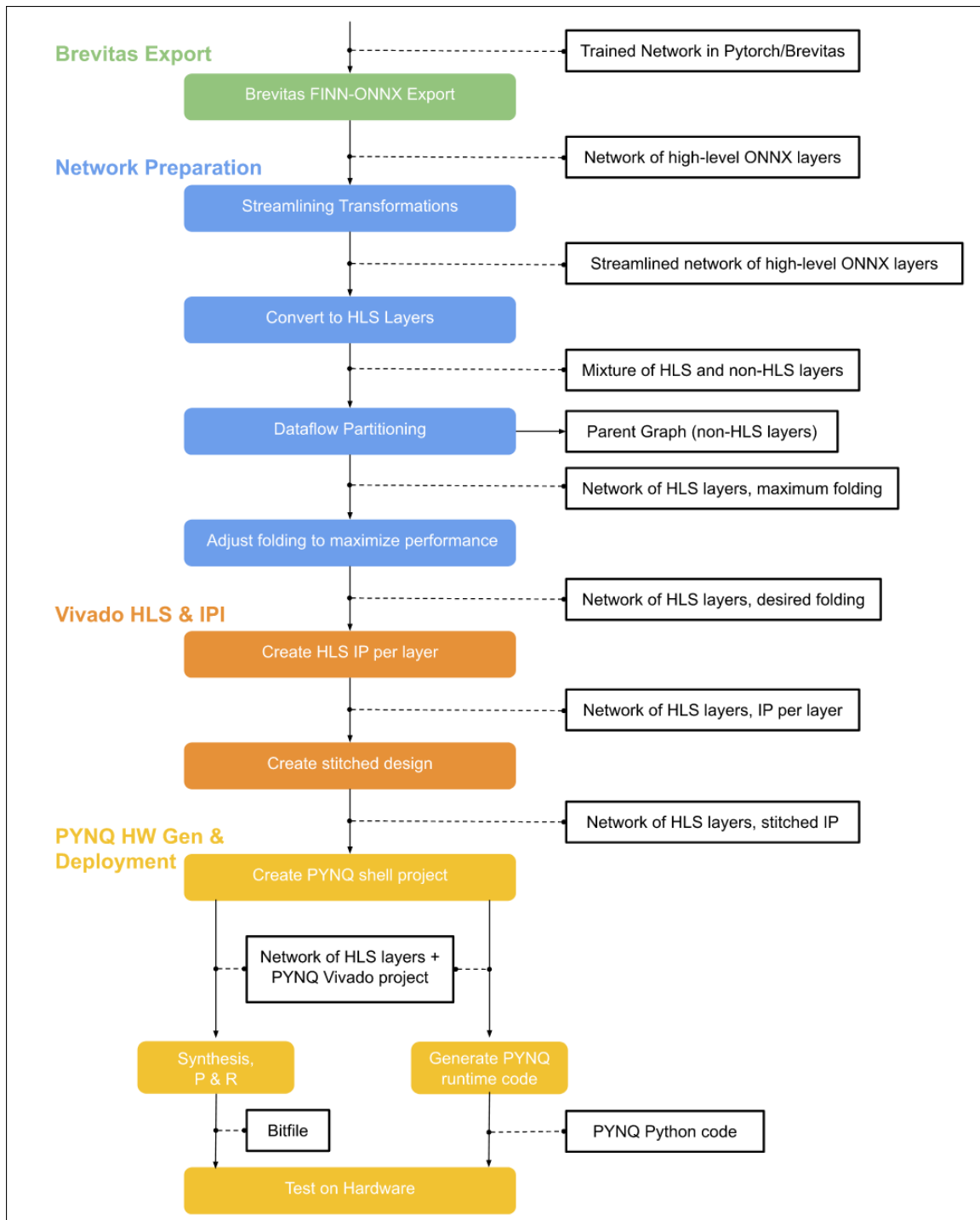


Fig. 6.13: FINN flow (source: [14])

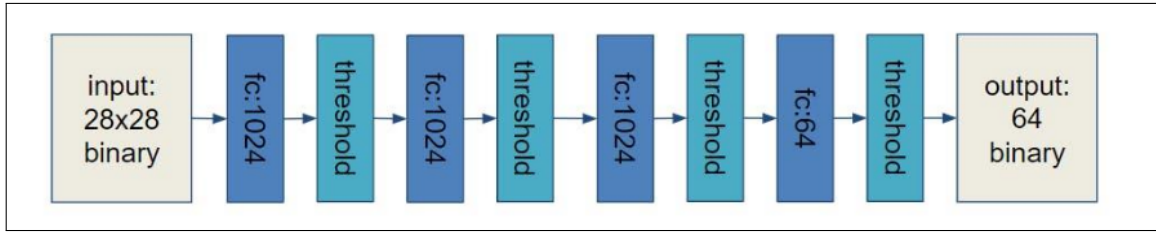


Fig. 6.14: LFC diagram (source: [18])

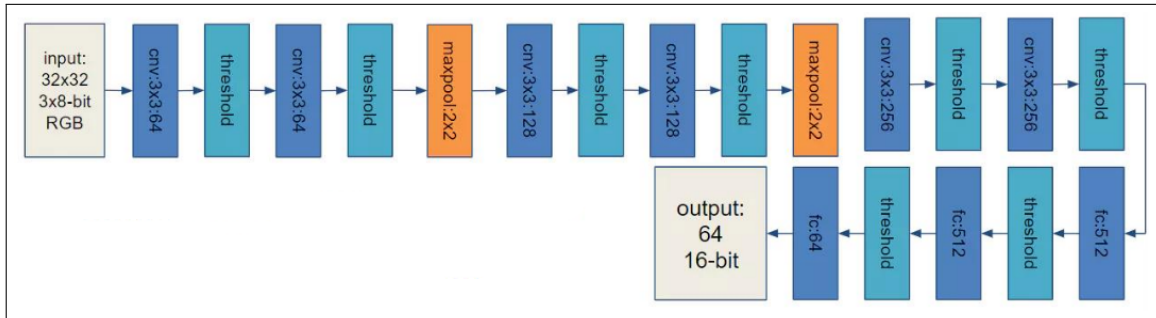


Fig. 6.15: CNV diagram (source [18])

As previously mentioned, the BNN-PYNQ repository contains documented sections about how to train models for different topologies and contains overlays (.bit + .tc1) for each of the pre-trained networks. The Vivado projects can be reconstructed with the provided script, which is a requirement for 7.4.

6.5.3 Quantized Neural Networks

There is a QNN-MO-PYNQ repository [19] developed by Xilinx that implements QNNs for PYNQ. Quantized Neural Networks have low precision values for weights and activation functions [20], and could be considered a super-set of BNNs. When QNNs take values with such low precision as single bits they are equivalent to BNNs.

The repository implements an object detection algorithm called YOLO [21]. It has evolved over time and there are multiple versions of it, but the one used in this project is a modification of YOLOv2 that gives up accuracy for the sake of performance. YOLO is explained in depth in [22], but a brief explanation is due in this section. It is a computer vision algorithm that implements both a classifier and object locator in a single neural network. It divides the input image into regions, checks and compares the probability of each class to be on each region, and outputs the score of the classes for the regions. Threshold parameters determine required certainty values for predictions to be accepted, as explained in section 7.3.1. Humans are amongst the classes contemplated by the classifier, which is convenient for this thesis. In theory, it means the overlay generated from the pre-trained model can be used directly, although as detailed in section 7.3.1, that was not the case.

As seen in figure 6.16 the first and last layers have weight and activation sizes that are not compatible with the QNN accelerations. The 8 bits for each weight and activations means the first and

6.State of the Art

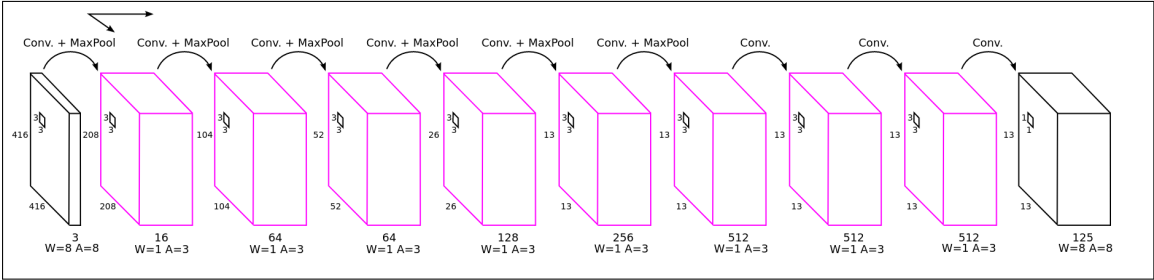


Fig. 6.16: TinierYOLO topology (source: [19])

last layers must be run in the ARM processor instead of the PL. For this the Darknet [23] library is used, an open source neural network framework developed in C.

Chapter 7

Development

This chapter describes the more relevant aspects of the development phase of the project more or less in chronological order starting with getting familiarized with the board, later with the facial recognition and object detection use cases and finally with the integration of both. There is also a small section at the end listing changes made to libraries that were forked from their original repositories for this project.

7.1 Familiarization with the PYNQ Platform

After finding possible uses cases by profiling programs in an ARM device, as told in the motivation section (section 1.2), familiarization with the board began. Although the project was started with a PYNQ-Z1 board, this section describes the process on an Ultra96-v1 board. The first step is to download the firmware and dump in on a micro-SD card. The firmware version 2.4 was used for development, however the 2.5 is currently available.

After writing the firmware, the board boots as a normal Debian-based GNU/Linux system. It has SSH enabled by default, but more accessible configuration options are available as Jupyter Notebooks in a folder with the name “common”. Connections to and from the board were initially made with the USB interface but were switched to WiFi for convenience. The Jupyter Notebook available to configure the WiFi was not making changes persistent, the `wpa_supplicant` configuration files were edited to make the configuration persistent. After being assigned a static local IP address to avoid depending on services such as Avahi, and with both SSH and Jupyter Notebooks available, accessing the board is trivial.

With the board running, the next step is to create some toy programs for PYNQ. The first programs that were created used components of OpenCV accelerated by Xilinx and offered in the PYNQ CV repository [24]. The package exposes some OpenCV functions as accelerated versions from the `xfOpenCV` [25] and contains example Jupyter Notebooks. The Cython (C and Python compiler) version (0.26.x) installed with the board had compatibility issues with the latest version of PYNQ CV and needed to be updated to at least version 0.29.14. After doing so and installing PYNQ CV,

it was found that the version of OpenCV natively installed in PYNQ was missing features and it too needed to be updated. The simplest way of doing so was by cloning the OpenCV repository and manually compiling/installing it, which took several ours. Once OpenCV was updated, everything was ready to create programs.

The first programs consisted of taking image files, running them through OpenCV filters with and without acceleration and compare the performance in both cases. After that images were taken from a webcam stream and run through several filters to test the real-time capabilities of PYNQ, which turned out to be impressive: filters run a few orders of magnitude faster. After that, the facial recognition use case was studied.

7.2 Facial Recognition

After examining PYNQ projects, BNN-PYNQ [16] was chosen. This section details how the PYNQ applications used to classify handwritten digits and traffic signs were repurposed to identify faces. Two of the topologies described in section 6.5.2 were used: LFC and CNV. LFC was originally used for classifying digits and produces smaller networks which makes it the preferred choice to be integrated with the object recognition into a single program, despite its reduced accuracy. CNV models are more accurate, but also more demanding. They were used to identify traffic signs as described in the state-of-the-art chapter. What follows is a setup and a usage section for both LFC and CNV, despite some parts of the processes being very similar.

When designing applications for FPGA, the most relevant constraint may usually be resource usage, such as the limited amount of logic blocks available. This makes the design area usage very relevant, in contrast to processors with architectures like ARM in which the most usual constraint is execution time. That is why two topologies were tested with the idea of using the ones with less resource consumption for the integration described in section 7.4.

Facial recognition models were trained using the Yale Faces [26] dataset, which provides eleven different pictures of fifteen people. A dataset this size is manageable for the data conversion steps while still providing good results.

7.2.1 LFC Setup

Based on [13] and the topology descriptions, LFCs are less demanding than CNV and were the first choice for facial recognition before trying the CNV topology. There is a model included in the BNN-PYNQ repository used to classify handwritten digits. The steps described below are required to train it for facial recognition.

Data classification

This step is straightforward: classify pictures of faces of different people into folders with each person's name. If using the Yale Faces dataset, this is already done. Despite the initial assumption

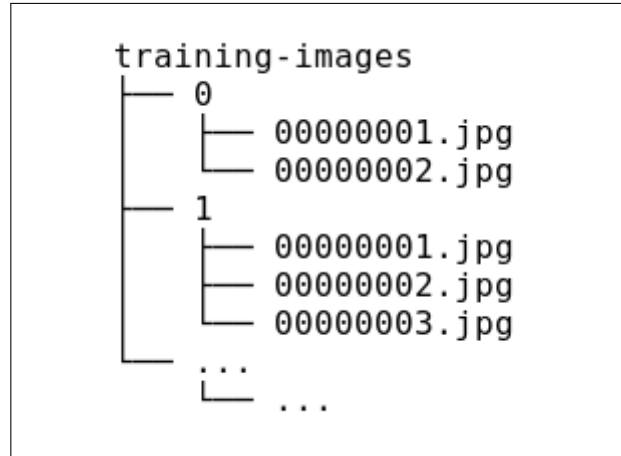


Fig. 7.1: Dataset directory structure

that only 10-bit output values were possible, after examining the training scripts it was found that the LFC network creation function accepts up to 64 bits.

Data splitting

The classified images must be split into these sets: the training set, validation set and test set. The arbitrarily chosen proportion is 70%, 15% and 15%, respectively. The training set will be used to test the model. Initially, the training and validation images will be in the same set and will be split into 70% and 15% during the training step described in section 7.2.1.

Data formatting

The images need to be grayscale images, have a size of 28x28 pixels and the labels, which identify the images, need to be in one-hot encoded format [27]. The one-hot format uses a set of bits with a single bit active at any time, to represent a state, and increases performance when used in the FPGA design [28]. The LFC topology used in this project do use the one-hot encoding internally [13]. Fortunately, labels are converted by the training script of section 7.2.1 and don't need to be changed manually.

A modification repository created by Gregory S. Kielian [29] for this specific data image conversion was used. The repository requires the dataset to have the structure shown in figure 7.1. The folder 0 represents the first class of the model, 1 the second and so on. The classes are defined in the file `batches.meta.txt`. Each line represents a class, in order and there must be one class for each person to be recognized. Therefore, if the first person were "Mikel", the first line of `batches.meta.txt` would be `mikel` and their pictures would be in folder 0. For the sake of simplicity, labels range from "Person 0" to "Person 14".

The resizing shell script is derived from `resize-script.sh` the file in the original repository, as shown in listing 7.1. In addition to resizing, all the original images in the RGB color model are converted to grayscale. In this case, the original files were in PNG format.

```

1 #!/bin/bash
2
3 height=28
4 width=28
5
6 for file in training-images/*/*.png; do
7     convert "$file" -colorspace Gray \
8         -resize "${width}x${height}"\! "${file%.*}.jpg"
9     rm "$file"
10 done
11
12 # ... repeat loop for each set

```

Listing 7.1: "Image modification script"

The final conversion script, which encoded the images in one-hot format, had issues that Phuong Le's fork [30] solved. The only modification the fork required was setting the color channels from three to one. Executing the file `converter.py` generates the `custom` folder with the output in four compressed files. The files with names matching `test*` must be renamed to `t10k*` for compatibility with the BNN-PYNQ repository [16]. The four files will therefore be: `t10k-images-idx3-ubyte.gz`, `t10k-labels-idx1-ubyte.gz`, `train-images-idx3-ubyte.gz`, and `train-labels-idx1-ubyte.gz`. These four files need to be extracted and moved to the directory pointed to by the environmental variable `PYLEARN_DATA_PATH` for `pylearn` to locate them. The directory is, in this case, `~/pylearn2/pylearn2/scripts/datasets/mnist`. It will be mentioned again before the description of the model training.

Model creation and training

Training the model took half an hour with an Nvidia GTX 1070 GPU. The necessary drivers and libraries were installed with steps similar to the ones described in the training section of the BNN-PYNQ repository but for an Arch Linux system instead of Debian-based distribution. No virtual environments were used because it is a system dedicated to this project.

There are some modifications that need to be made to the training scripts before proceeding. All of the files and folders mentioned during the next steps are in the training folder `bnn/src/training/`. For LFC, the file that must be changed is `mnist.py`. There are three main changes that need to be made: change the set sizes, change the output size number and the batch size. Set sizes must be modified to correspond to the training, validation and test sets. Output size must be changed from 10 to 15 bits to fit the fifteen faces of the dataset. The batch size does not have to change necessarily, but given the small size of the set it is recommended, as it dictates how often the model is updated during training.

The training, validation and test size is changed by modifying lines 122 to 124 have the correct number of images in each dataset. The validation images are a subset of the training dataset created in the previous step that is not used for training. In the example shown in listing 7.2, the total number of images is 200 and the proportion is still 70%, 15% and 15% for training, validation and testing respectively. Batch size is modified by changing the `batch_size` variable, and output size by changing "10" by "15" in the one-hot encoding and LFC generation sections.

```

121 ...
122 train_set = MNIST(which_set='train', start=0, stop=140, center=False)
123 valid_set = MNIST(which_set='train', start=140, stop=170, center=False)
124 test_set = MNIST(which_set='test', start=170, stop=200, center=False)
125 ...

```

Listing 7.2: "Sample of mnist.py"

Before executing the file, the following dependencies must be installed.

- **Python 2.7.** The training libraries require the 2.7 version of python, despite the rest of the project having been developed in the 3.6+ version.
- **Theano.** A mathematical expression evaluation compilation optimization library that compiles expressions to accelerate them in CPUs or GPUs like in this case.
- **Pylearn 2.** A machine learning framework that uses Theano under the hood. The MNIST library used by the `mnist.py` file in the BNN-PYNQ project is imported from Pylearn 2. Some difficulties were encountered with this library, described in section 7.5.
- **Lasagne.** Lasagne is a library used to create and train neural networks using Theano.
- **Additional packages.** General use packages that were not already installed in the system or environment, in this case `Pillow`, `scipy` and `six`.

The version of each module used in this project is displayed in listing 7.3. Pylearn2 is different than the rest, as during development the package being imported was under development to make the changes described in section 7.5. This means the `pip develop` command was used for Pylearn2 instead of the standard `pip install`.

```

1 Lasagne==0.2.dev1
2 numpy==1.16.6
3 Pillow==6.2.1
4 -e git+git@github.com:mikelsr/pylearn2.git@a2f970b8482b6aace80e41692090352c54011113
   #egg=pylearn2
5 scipy==1.2.3
6 six==1.14.0
7 Theano==0.8.0

```

Listing 7.3: "Module versions"

Additionally, the `*-ubyte.gz` files generated in a previous step must be extracted into a folder named "mnist" in the folder pointed at the `PYLEARN2_DATA_PATH` environment variable. Once this is done, the `mnist.py` will create a model and train the (in this case) facial recognition model. It can have either 1-bit or 2-bit activation functions by passing `-ab 1` or `-ab 2` as parameters. The next paragraph assumes two activation bits, but both 1 and 2 bit networks are included in the project results. The only difference with single bit activation is changing "2" for "1" in file and directory names. The best epoch out of 1000 from the LFCW1A2 training is shown in figure 7.2.

```

Epoch 992 of 1000 took 1.93348097801 seconds
LR:                3.25927687085e-07
training loss:     0.0916081467252
validation loss:   0.0945482701374
validation error rate: 12.5%
best epoch:       992
best validation error rate: 12.5%
test loss:        0.0483024631657
test error rate:  8.333333333333%

```

Fig. 7.2: Best epoch when training LFCW1A2

The resulting weights will be stored at `mnist-1w-2a.npz` in the training folder. Running `mnist-gen-weights-W1A2.py` will convert the weights to a format usable by both the hardware and software implementations of the BNN-PYNQ project by converting the floating values into binary values and packaging them into `.bin` files. It will also create the `classes.txt` file and the hardware configuration header file `config.h`. The results are located at the newly created `1fcw1A2` folder.

PYNQ Configuration

This is the last step: installing the BNN project in the PYNQ device and copying the results of the previous steps. As described in the README file of the BNN-PYNQ repository, installing the BNN library is as simple as running `sudo pip3 install git+https://github.com/Xilinx/BNN-PYNQ.git`.

The contents of the generated `1fcw1A2` directory must be copied to a specific folder in the PYNQ device, in this case the Ultra96. The directory is the parameter folder in the BNN installation path, likely to be `/usr/local/lib/python3.6/dist-packages/bnn/`. The arbitrary parameter name is “face_recognition”, thus the folder where the weights and configuration files are copied to in the Ultra96 is `/usr/local/lib/python3.6/dist-packages/bnn/params/face_recognition/`. This folder will contain another two folders when the two models are trained: `1fcw1A1` and `1fcw1A2`.

7.2.2 LFC Usage

Once a face is captured in an image and stored in a 28x28 numpy array, the only pre-processing left is giving it the appropriate format to be used by the MNIST classifier. An example of how to do this is provided in the `LFC-BNN_MNIST_Webcam` Jupyter Notebook included in the BNN repository. An example of how to use the trained model with a frame from a webcam is provided in listing 7.4. The example uses `take_picture()` and `pre_process()` as functions that capture a frame and convert it to MNIST format, respectively. The output variable is `class_out`, which contains the index of the identified person, e.g. if the identified person is the one whose pictures were in the folder “0” in the training step, `class_out` will have a value of 0. A full, working example is provided in the notebooks repository [31].

```

1 import bnn
2

```

```

3 # the parameters where named "faces" in the PYNQ Configuration section
4 hw_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A2, "faces", bnn.RUNTIME_HW)
5
6 face = take_picture() # capture a frame from the camera
7 face = pre_process(face) # image -> numpy array -> MNIST format
8
9 class_out = hw_classifier.classify_mnist(face)

```

Listing 7.4: "BNN LFC usage example"

It is worth taking a look at the source code [16] of the `bnn` Python package to better understand the process, located in the file `bnn.py` at the top-level package. When `bnn.LfcClassifier()` is called, a Python object of class `LfcClassifier` is created. The constructor of this class configures many of the classifier parameters and calls another constructor for one of its attributes of type `PynqBNN`. The constructor for this class, shown in listing 7.5, locates the bitstream to be downloaded into the PL and downloads it in line 91. In line 97, the `interface` attribute is loaded. This attribute is used to communicate with the PL, including launching the inference function used to classify an image. The communication functions are, however, in a shared library (`.so`, in the directory `bnn/libraries/ultra96`) and can't be directly accessed from Python. This is why in line 96, the C Foreign Function Interface (CFFI) library is used to open the shared library with the `_ffi` variable. This variable is an object from the CFFI package and knows what functions to expect from the shared library, as earlier in `bnn.py` the headers of the functions to be used have been passed to it, including the `inference` and `load_parameters` functions that will be named again later.

```

84 def __init__(self, runtime, network, load_overlay=True):
85     self.bitstream_name = None
86     if runtime == RUNTIME_HW:
87         self.bitstream_name="{0}-{1}.bit".format(network, PLATFORM)
88         self.bitstream_path=os.path.join(BNN_BIT_DIR, self.bitstream_name)
89         if PL.bitfile_name != self.bitstream_path:
90             if load_overlay:
91                 Overlay(self.bitstream_path).download()
92             else:
93                 raise RuntimeError("Incorrect Overlay loaded")
94     dllname = "{0}-{1}-{2}.so".format(runtime, network, PLATFORM)
95     if dllname not in _libraries:
96         _libraries[dllname] = _ffi.dlopen(os.path.join(BNN_LIB_DIR, dllname))
97     self.interface = _libraries[dllname]
98     self.num_classes = 0

```

Listing 7.5: "PynqBNN constructor"

After the `PynqBNN` constructor is called and the overlay downloaded, the `LfcClassifier` will configure the network with the aforementioned `load_parameter` function to create the network that will be used for facial recognition. The next function to analyze is `classify_mnist`, which is a method of the `LfcClassifier` class. This function will create the variables required to call the `inference` function located at the shared library, call it, print how much time inference has taken and return the results: the image classification.

7.2.3 CNV Setup

The CNV training process is a modification of the GTSRB training process used in the original repository. GTSRB is a dataset of German traffic signs, and the documentation provides steps on how to train the model given the dataset. Thus, rather than creating a new script to train the model, the dataset will be replaced by a modified version of the Yale Faces dataset and the training script will be modified to fit the new dataset.

Data formatting

The GTSRB dataset has a structure like the one shown in figure 7.3, with numerated folders containing a CSV file and a set of PPM files. The CSV files contain information about the PPM images in the folder. The CSV columns are separated by semicolons and contain a line per PPM file with the name of the file, the coordinates of the traffic sign in the image and the class the image belongs to. Scripts for creating the correct dataset are all provided in a repository created for this project and hosted on GitHub [32]. The scripts convert the image from `.gif` to `.ppm`, use a classifier included in OpenCV to extract the coordinates of the faces on the image, create the CSV files and put it all together.

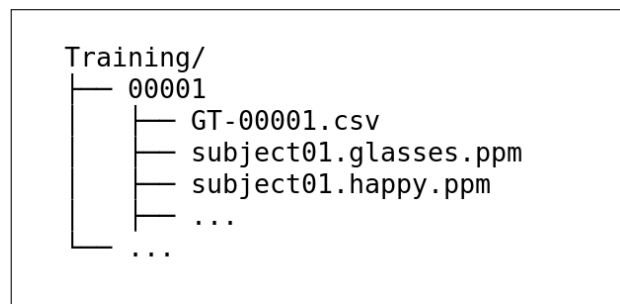


Fig. 7.3: GTSRB modified dataset directory structure

Model creation and training

Only two things need to be changed in the training file `gtsrb.py`. The first one is the variable `num_dup`, which improves the original GTSRB training process by grouping duplicates. Yale Faces contains no duplicates and therefore the value must be changed from 30 to 1. The second modification is disabling the GTSRB junk files that would worsen the model of this project. Setting the variable `junk_class` to `False` does the trick. The training script is now ready to be run as shown in listing 7.6.

```
1 python2.7 gtsrb.py -ip $PATH_TO_DATASET
```

Listing 7.6: "Training of CNV"

The process took five hours, considerably longer than the LFC topologies. Running the `gtsrb-generate-weights.py` converts and packs before storing them in the newly created folder `cnvW1A1-gtsrb` the weights so they can be used on PYNQ. The best epoch out of 1000 is shown in figure 7.4.


```

Epoch 997 of 1000 took 19.433754921s
LR: 3.09893715972e-07
training loss: 0.00693385727983
validation loss: 0.0849743406848
validation error rate: 13.3333333333%
best epoch: 997
best validation error rate: 13.3333333333%
test loss: 0.0115805853067
test error rate: 6.66666666667%

```

Fig. 7.4: Best epoch when training CNVW1A1

PYNQ Configuration

Similar to the LFC section, a folder was generated and needs to be installed on the board. For that, the generated `cnvW1A1-gtrsb` folder is copied from the personal computer to the board, specifically to the folder `/usr/local/lib/python3.6/dist-packages/bnn/params/facial_recognition/cnvW1A1`. As the path indicates, the folder is also renamed from `cnvW1A1-gtrsb` to `cnvW1A1`. The `facial_recognition` folder should now contain all three models.

7.2.4 CNV Usage

The usage of the CNV classifier is very similar to the one of the LFC classifier. The classifier is loaded by specifying the parameters and topology, a pre-processing function is run on the input image to prepare it for the classifier, and the image is run through the classifier resulting in the index of the detected class. The PYNQ notebooks [31] published with the results of the projects provide more insight into the whole process, but doing so in this section would fill it with code which is not appropriate. Once again, the parameter name is “`facial_recognition`”. There is a notable difference when compared to LFC: the classifier takes an array of images as input and returns an array of classes as output. The example of 7.7 feeds the classifier a single image and takes the first value of the output.

```

1 import bnn
2
3 hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNvw1A1, "facial_recognition", bnn.
  RUNTIME_HW)
4
5 face = take_picture() # capture a frame from the camera
6 face = pre_process(face) # image -> GTSRB format
7
8 class_out = hw_classifier.classify_image(face)

```

Listing 7.7: “BNN usage example”

The classifier construction and method are very similar to the ones of the LFC topology, excluding a few differences. The classifier class is `CnvClassifier` instead of `LfcClassifier`, the class has more classification methods for different purposes and the method used in listing 7.7 contains an image

formatting function, therefore the image does not need to have the exact format used by the classifier like in LFC.

7.3 Object Detection

The QNN-MO-PYNQ [19] provides an accelerated implementation for YOLO and is simpler than facial recognition in some ways due to the lack of necessity to train a new model. It is, however, more complicated to use. The following sections go through the set-up and usage processes in some detail.

7.3.1 Setup

The only step required to install the QNN library in the Ultra96 board is to run `sudo pip3 install git+https://github.com/mikelsr/QNN-MO-PYNQ.git`. It is a fork of the QNN-MO-PYNQ [19], the only modification being replacing the Darknet dependency of the original project with a custom one, as explained in the following section.

Darknet

The QNN-MO-PYNQ repository has a version of Darknet [33] modified by Giulio Gambardella from the Xilinx team [34] as a dependency. This modified version adds the C function named “forward_region_layer_pointer_nolayer” and the corresponding Python binding. It is however based on an older, less convenient version of Darknet, so another version was made to integrate the changes made by the Xilinx team to the current version.

Darknet Python Bindings

As previously explained, Python bindings allow calling code generated in other programming languages to be called directly from Python. There are multiple ways of using Python bindings, the two methods used in this library are ctypes and dynamic libraries. The examples that follow are from the `python/darknet.py` file in the custom fork of the Darknet repository. It is the file that is imported when importing darknet from Python.

Ctypes is a Python module that provides data structures compatible with those of the C programming language. The example provided in listing 7.8 shows the Python equivalent of a C structure with four variables of type `float`.

```

20 class BOX(ctypes.Structure):
21     _fields_ = [("x", ctypes.c_float),
22                ("y", ctypes.c_float),
23                ("w", ctypes.c_float),
24                ("h", ctypes.c_float)]

```

Listing 7.8: “Ctypes example from darknet.py”

Dynamic libraries, in this case in the form of shared object (.so) files due to PYNQ being a Linux system, reference C structures and functions and allow them to be referenced directly from Python without explicit bindings.

When building the custom darknet project in the Ultra96 with the Makefile the `libdarknet.so` shared objects file is generated. When importing Darknet from Python, the code in `python/darknet.py` is executed as mentioned before. The line shown in listing 7.9 shows how the dynamic library is loaded into the `lib` Python variable.

```
48 lib = CDLL("/opt/darknet/libdarknet.so", RTLD_GLOBAL)
```

Listing 7.9: "Python dynamic library import in darknet.py"

Explicit bindings are, however, the preferred way of creating bindings. The code sample in listing 7.10 shows how a C function imported when loading the dynamic library is assigned to a Python variable and the input/output parameter types are defined in the `argtypes` and `restype` attributes, respectively. It is important to note that Python objects are callable, and functions are implemented as objects. This results in callable objects having attributes, thus the attributes of the callable object `get_network_boxes` from listing 7.10.

```
65 get_network_boxes = lib.get_network_boxes
66 get_network_boxes.argtypes = [c_void_p, c_int, c_int, c_float, c_float, POINTER(
    c_int), c_int, POINTER(c_int)]
67 get_network_boxes.restype = POINTER(DETECTION)
```

Listing 7.10: "Function definition example from darknet.py"

7.3.2 Usage

The algorithm requires two configuration parameters: the detection and hierarchy thresholds. The first threshold, `QNN_THRESHOLD`, is used to decide whether a box contains an object. If the certainty is above the threshold, an object is considered to be in the box. The second threshold, `QNN_THRESHOLD_HIER`, determines whether an object is considered to belong to a class. Again, if the certainty surpasses the threshold, the class is accepted. Additional parameters, such as the path to the QNN and Darknet libraries and the size of the inputted to the network, are shown in listing 7.11.

The path to the Darknet module must be explicitly declared as the module is not in the Python path, as shown in line 17 of listing 7.12.

```
1 # Python module configuration
2 DARKNET_PATH = "/opt/darknet"
3 QNN_PATH = f"/usr/local/lib/python3.6}/dist-packages/qnn"
4
5 # Classifier configurations
6 QNN_SIZE = 416 # width and height of images used by YoloV2
7 QNN_THRESHOLD = 0.3 # certainty that there is an object in a box
8 # lower -> more results
9 QNN_THRESHOLD_HIER = 0.5 # threshold to consider a class in a box
```

Listing 7.11: "QNN global variables"

```

12 import qnn
13 from qnn import TinierYolo, utils
14
15 # ... other imports
16
17 sys.path.append(f"{DARKNET_PATH}/python")
18 import darknet

```

Listing 7.12: "QNN imports"

To create the classifier, the imported `TinierYolo` class must be instantiated. The instantiating method loads the libraries that will be used when using the hardware accelerator, although the SW implementation can also be loaded by changing the `runtime` parameter of the constructor. To download the corresponding bitstream into the FPGA, the `TinierYolo.init_accelerator` method must be called. As seen in listing 7.13, which is taken from the source code of the QNN library, the path to the bitstream has a default value and is not required. The functions `initParameters` and `initAccelerator` have also been imported from a shared library using CFFI as in the LFC example but this time under the `lib` attribute instead of `interface`. The source files that contain listings 7.13, 7.14, 7.15, 7.16, 7.17, 7.18 can be found in the QNN-MO-PYNQ repository [19].

```

97     def init_accelerator(self, bit_file=TINIER_YOLO_BIT_FILE, json_network=
98         W1A3_JSON, json_layer=JSON_TINIER_YOLO):
99         """ Initialize accelerator memory and configuration. """
100
101         if bit_file :
102             ol = Overlay(bit_file)
103             ol.download()
104             ...
105             self.lib.initParameters(1,0);
106             self.lib.initAccelerator(json_network.encode(), json_layer.encode())

```

Listing 7.13: "Partial method of TinierYolo class"

The network is still to be loaded after downloading the overlay. This is done by calling the `TinierYolo.load_network` passing the path to the network architecture description file as a parameter. The file is shown in listing 7.14, is in JSON format and contains both network parameters and descriptors for each layer of the network. The relevant descriptors for this project are the input size, output size and layer names.

```

1 {
2     "network": "tinier-yolo",
3     "input_image": "../../../tests/Test_image/tinier-yolo/input.bin",
4     ...,
5     "layers": [{
6         "name": "conv1",
7         "func": "conv_layer",
8         "input_bits": 3,
9         "output_bits": 3,
10        "weight_bits": 1,
11        "threshold_bits": 16,
12        "kernel_shape": 3,
13        "kernel_stride": 1,
14        "input_channels": 16,

```

```

15         "input": [16, 208, 208],
16         "output_channels": 64,
17         "output": [64, 208, 208],
18         "padding": 1
19     },
20     ...
21 }
22 }

```

Listing 7.14: "Network architecture"

Additionally, computations made with Darknet require the library to have the correct network configuration. The `parse_network_cfg` function takes the path to a configuration file as a parameter and sets the network values based on the configuration file. An edited example of the network creation, configuration and overlay download is shown in listing 7.15.

```

1 classifier = TinierYolo()
2 classifier.init_accelerator()
3 net = classifier.load_network(json_layer=f"{QNN_PATH}/params/tinier-yolo-layers.
   json")
4
5 conv0_weights = np.load(f"{QNN_PATH}/params/tinier-yolo-conv0-W.npy", ...)
6 conv0_weights_correct = np.transpose(conv0_weights, axes=(3, 2, 1, 0))
7 conv8_weights = np.load(f"{QNN_PATH}/params/tinier-yolo-conv8-W.npy", ...)
8 conv8_weights_correct = np.transpose(conv8_weights, axes=(3, 2, 1, 0))
9
10 # ... further configuration and data type conversions
11
12 net_darknet = darknet.lib.parse_network_cfg(conf_file)

```

Listing 7.15: Network creation and configuration"

Each frame to be processed requires to be in the format used by Darknet, as shown in the example Jupyter Notebook. After the image is correctly formatted, it must be run through the first layer. As explained in section 6.5.3, the first and last layers are not HW accelerated. The first layer execution is shown in listing 7.16.

```

1 conv0_ouput = utils.conv_layer(img, conv0_weights_correct,
2                               b=conv0_bias_broadcast,
3                               stride=2, padding=1)
4 conv0_output_quant = conv0_ouput.clip(0.0, 4.0)
5 conv0_output_quant = utils.quantize(conv0_output_quant/4, 3)

```

Listing 7.16: "First QNN layer"

Middle layers are run as shown in listing 7.17, which is explained in this paragraph. The function `get_accel_buffer` prepares a buffer for the inference to dump the results at. Next, `prepare_buffer` runs a series of operations over the input data to prepare it. Finally, `inference` loads the input data into the accelerator so that `postprocess_buffer` can read the output and perform a series of operations to convert it back to the original format. Inside the inference function shown in the listing, the function `singleInference` function is called. This function, as well as the aforementioned `initParameters`, `initAccelerator` and another function that will be mentioned later but is called `deinitAccelerator` were all imported using CFFI.

```

1 out_ch = net['conv7']['output'][0]
2 out_dim = net['conv7']['output'][1]
3
4 conv_output = classifier.get_accel_buffer(out_ch, out_dim)
5 conv_input = classifier.prepare_buffer(conv0_output_quant*7);
6
7 classifier.inference(conv_input, conv_output)
8
9 conv7_out = classifier.postprocess_buffer(conv_output)

```

Listing 7.17: "Middle QNN layers"

The final steps of the QNN, shown in listing 7.18 show how the output of the middle layers is processed and run through the last layer. The results are now stored in the `net_darknet` created in listing 7.15.

```

1 conv7_out_reshaped = conv7_out.reshape(out_dim, out_dim, out_ch)
2 conv7_out_swapped = np.swapaxes(conv7_out_reshaped, 0, 1) # exp 1
3 conv7_out_swapped = conv7_out_swapped[np.newaxis, :, :, :]
4
5 conv8_output = utils.conv_layer(conv7_out_swapped, conv8_weights_correct,
6                                 b=conv8_bias_broadcast, stride=1)
7 conv8_out = conv8_output.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
8
9 darknet.lib.forward_region_layer_pointer_nolayer(net_darknet, conv8_out)

```

Listing 7.18: "Middle QNN layers post-processing"

The only thing left is processing the results. There are two useful functions included in Darknet to extract and sort the results. The process is straightforward: extract the detections from the result, sort them based on the probability of having found something and look for people on the detections that came out on top. The process is shown in listing 7.19 in a simplified way, but the whole process is available at the "Detection" Jupyter Notebook at [31]. In the example, `meta` contains the classes of the classifier and the names of the classes, e.g. 0 is a class and "person" is a name.

```

1 num = ctypes.c_int(0)
2 num_pointer = ctypes.pointer(num)
3
4 detections = darknet.get_network_boxes(net_darknet, QNN_SIZE, QNN_SIZE,
5                                       QNN_THRESHOLD, QNN_THRESHOLD_HIER,
6                                       None, 0, num_pointer)
7 n = num_pointer[0]
8
9 darknet.do_nms_sort(detections, n, meta.classes, QNN_THRESHOLD)
10
11 results = []
12 for i in range(n):
13     for j in range(meta.classes):
14         if detections[i].prob[j] > 0.0 and meta.names[j] == b"person":
15             results.append((
16                 meta.names[j],
17                 detections[i].prob[j],
18                 (get_box_coords(detections[i].bbox))

```

Listing 7.19: "Extraction of results from QNN"

The variable `results` now contains the probability and bounding box of each person in the processes image. It is important to take into account that memory was allocated with the bindings and is not automatically cleaned up by the Python garbage collector, thus `darknet.free_detections(detections, n)` must be called to manually release allocated memory. Unless more images are to be processed, the accelerator can be unloaded by running `classifier.deinit_accelerator()`.

7.4 Integration of Facial Recognition and Object Detection

This section goes through the attempt to use one of the facial recognition (BNN) overlays and object detection (QNN) overlays at the same time. The first thing that comes to mind is to just download both overlays, unfortunately, that is not possible as only one overlay can be downloaded at a time. The next idea is to just download them alternatively before using them. The benchmark notebooks [31] created for documenting this project show however that loading the overlay takes somewhere around 500 and 1000 milliseconds. Downloading the overlays interchangeably would be possible, but would harm the real-time potential as the download times would need to be compensated by processing images by batches instead of individually. The solution attempted in this project was to reconstruct the Vivado projects of both networks, integrate them in a single design and create an overlay from the integrated design. Both BNN-PYNQ and QNN-MO-PYNQ provide ways of generating Vivado projects from the sources included in the repositories. For the sake of simplicity, this section will refer to the BNN and QNN interchangeably repositories simply as *repository*. Both projects are reconstructed very similarly and thus only when reconstruction steps differ will the repository be specified.

The first step is to install Vivado and Vivado HLS. The networks were built using Vivado 2018.2 and that is the version that needs to be used for reconstruction to avoid compatibility errors. Once Vivado and HLS are installed, a script inside `repository/src/network` allows reconstruction of the projects. The `make-hw.sh` shell script receives the network type, target board and reconstruction mode as parameters and builds the project in a new folder at `repository/src/network/output`. The process takes a considerable amount of time to complete. The BNN used in this case is rebuilt with `./make-hw.sh lfcW1A1 ultra96 a`: the network type is "lfcW1A1", the target board is "ultra96" and "a" specifies to regenerate HLS and bitstream files. The Other BNN and QNN projects are reconstructed in the same way considering the three other network types used in this project are CNVW1A1, LFCW1A2 and W1A3. Once the project is rebuilt, four folders are found in the output folder:

- **bitstream.** Generated bitstreams are stored in this folder.
- **hls-syn.** The IP, source C/C++ files of the HLS synthesis and the VHDL and Verilog files generated as a result are stored in this folder.

- **report.** The Vivado and Vivado HLS reports are stored in this folder. The Vivado report contains the estimated FPGA utilization values.
- **vivado.** The vivado project is created on this folder. The `.xpr` file can be opened with Vivado to inspect and edit the designs.

After generating both projects, a new one is created to integrate the BNN and QNN IPs. While creating a project, “RTL” must be selected as the project type and “Ultra96” as the target board. Figure 7.5 shows the project confirmation window with the correct parameters.

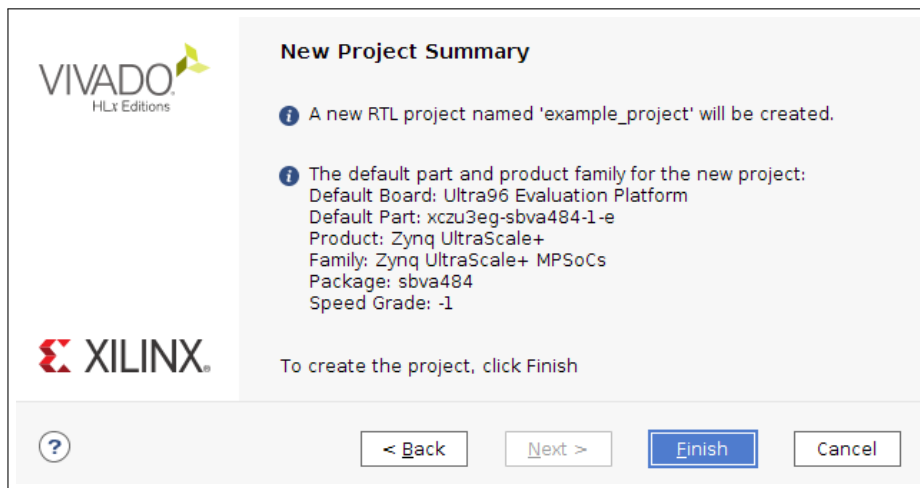


Fig. 7.5: Last step before creating the Vivado project

Once created, the IPs from the BNN and QNN repositories must be imported with the “Add Sources” option. The IPs that need to be imported are named “BlackBoxJam” and located in `repository/src/network/output/hls-syn/NETWORK-ultra96/sol1/impl/ip`, where `NETWORK` is the type of network: `W1A3`, `lfcW1A1`, `lfcW1A2` or `cnvW1A1`. A problem arises when importing the IPs: they overlap each other. To solve this, one of the IPs can be edited so its name changes from `xilinx.org:hls:BlackboxJam` to a custom one, in this case the BNN `BlackBoxJam` was edited to be `custom.org:hls:BlackBoxJamBNN`. Now, with both IPs imported the design can be created. Creating a block diagram and adding the Zynq UltraScale+ module is the first step to integrate the components in the design. Appendix A contains the block diagrams of BNN, QNN and the integration that can help follow this section. The Zynq module has two slave and two master AXI interfaces for the QNN IP and, one slave and one master AXI interface for the BNN one, as can be seen in the appendix. Next, an AXI Interconnect module should be loaded, this time with three slave AXI interfaces and two master ones. Now the `BlackBoxJam` IPs can be loaded and connected with the Zynq module via AXI SmartConnect modules. The few missing AXI connections that connect the Zynq, AXI Interconnect and `BlackBoxJam` modules are shown in the appendix. Vivado has an “auto-connect” feature that creates the missing connections, resulting finally in the design shown in the last block diagram of the appendix. The design needs to be validated, synthesized and finally implemented. The first two steps run without errors, however when implementing the design an error raises stating the design doesn’t fit the board. This while predictable as each of the overlays take more than half of the capacity, was the stopping point for the implementation. Efforts were

made to compress it by changing parameters in the steps previous to the implementation, but without much success. A question was posted to the PYNQ forum and quickly answered, however the problem remained unresolved. This was unfortunately the end of the integration phase, as any alternatives that came up seemed to require a deeper understanding of the components and were not in the project scope. It is, however, discussed in the future work section (section 9.1).

7.5 Forked Projects

This section lists the modifications made to some projects used during development. They were all originally hosted on GitHub and have been forked to make modifications available to anyone. Changes were usually made in separate branches but have been merged to master for accessibility. This section is oriented not only to document the development process but also to keep future developers from investing time solving the issues presented during the development of the project.

Pylearn 2

Pylearn2 is a machine learning library used to train the models used for this project. Pylearn 2 has `six` as a dependency. The original, currently unmaintained project uses the version included in Theano (`from theano.compat.six import ...`) which raised errors during the model training phase and needed to be fixed. The solution was to fork the repository and replace those imports with the global installation of `six` (`from six import ...`). This uses the `six` module installed in the system instead of the one packed with Theano. The `pylearn2/datasets/mnist.py` file has assertions that need to be adjusted to the size of the datasets for the process to succeed. Finally, the output size was changed from 10 bits to 15 bits. The repository is hosted at <https://github.com/mikelsr/pylearn2>.

Darknet

Darknet is a neural network framework written in C and Cuda used by the QNN-MO-PYNQ repository for processing images. The version used by QNN-MO-PYNQ is a fork [34] or the original [33] that implements some functions required by QNN-MO-PYNQ. The fork is however lacking some features that are implemented in the original and are useful for processing the results. A new fork [35] of the original repository was created, the new functions and bindings from the fork included in QNN-MO-PYNQ were added, and new function was added to the Python bindings file to classify an image without sorting the results. This last function is used for the benchmarks shown in chapter 8. The repository needs to be rebuilt in the board after changing the sources so that the file `libdarknet.so`, required by the project, is updated. The forked repository is available at <https://github.com/mikelsr/darknet>.

BNN-PYNQ

The BNN-PYNQ repository is very relevant to this project, but the most modified parts are the model training scripts. The first modification was made to `mnist.py` so that the size of training, validation and test sets corresponded to the new dataset and the output was a 15-bit value instead of a 10-bit one. The second modified file was `gtsrb.py`. Some statements were added to show more information about the state of the program to better understand how the training should be done. In addition to that, dataset sizes were adjusted, the GTSRB junk files were disabled, some data-transforming statements were modified to fit the new dataset and the classification classes were updated.

After several iterations, the models were trained and resulting classes, header files and binary values were copied to the `params` directory so that the networks trained for this project are available to use in the Ultra96 just by cloning the repository found at <https://github.com/mikelsr/BNN-PYNQ>.

QNN-MO-PYNQ

QNN-MO-PYNQ is also one of the vital repositories for this project. The project setup file was modified so that the custom version of Darknet is installed when setting up the project instead of the Xilinx fork. This means the correct version of Darknet is automatically installed when installing the fork of QNN-MO-PYNQ, accessible at <https://github.com/mikelsr/QNN-MO-PYNQ>.

JPG-PNG-to-MNIST-NN-Format

This repository contains the scripts used to transform the PNG dataset to the same format as MNIST, which is the one expected by the training script. It was adapted for GNU/Linux systems as it was originally written for OS X and the size and color model of the images were changed to the ones used in BNN-PYNQ. The repository is available at <https://github.com/mikelsr/JPG-PNG-to-MNIST-NN-Format>.

Yale-To-GTSRB

Yale-To-GTSRB is a repository created for this project that contains scripts that convert the Yale Faces database to the format used by the GTSRB. It can be found at <https://github.com/mikelsr/yate-to-gtsrb> and contains four scripts:

- **genppm.sh** - Shell script that converts the original `.gif` images to `.ppm` files and organizes them in the same structure GTSRB images have.
- **create_dataset.py** - Python script that locates the faces of each image using a cascade classifier and creates a CSV file for each face containing a list of image names, dimensions, face coordinates and the classification. It can be configured so that the image size is set to

a specific value and the face coordinates are scaled to the new image size, which affects the CSV files but not the images.

- **resize.sh** - Shell script that resizes the dataset images to have specific values. This step is optional and should only be run if `create_dataset.py` has been given specific values.
- **run.sh** - Shell script that runs the previous scripts in sequential order.

Chapter 8

Results

The results of the project will be analyzed by benchmarking the accelerated versions of the applications and their pure-software implementations, as the goal of the project is to examine the PYNQ framework as an edge computing device. These devices usually have ARM processors similar to the one the Ultra96 has, therefore the results of the pure-software implementations will be similar to the ones of some edge computing devices. Even if those devices were equipped with better processors, the benchmarks below will show that the accelerated sections of the programs would still be far ahead. The benchmarks are available at the GitLab repository [31] containing the Jupyter Notebooks with examples of how to use the facial recognition and object detection tools.

The first benchmark compares each of the BNN topologies (LFCW1A1, LFCW1A2 and CNVW1A1) in their accelerated and non-accelerated versions. A random image from the Yale Faces dataset is loaded and run through the HW and SW versions of the classifier 100 times. The time is measured by subtracting the time right before the classification function is called to the time right before the classifier finishes. Measured times are stored in arrays so that averages, maximum and minimum values can be calculated. Figure 8.1 shows the results in the form of execution time and frames per second by comparing each of the topologies, while figure 8.2 compares the HW and SW implementations.

As expected, CNV networks take longer than LFC networks. It does however come as a surprise that LFCW1A2 networks are faster than LFCW1A1 networks in the accelerated versions but not in the software ones. The models were retrained, the program was checked several times for errors and more iterations were measured, but the results were the same every time and the benchmarks will therefore be accepted for this project. The results are comparable to the benchmarks that PYNQ-CV [24] made with the xfOpenCV implementation, reinforcing the stated point.

It can be appreciated that the execution times decreased one or several orders of magnitude in all cases. The execution of CNVW1A1 is feasible to process several frames per second in real-time, while the software-only implementation can only process about two frames per second. The LFC networks have trivialized classification to a point that went from being a potential bottleneck to being one of the least time-consuming segments of the program.

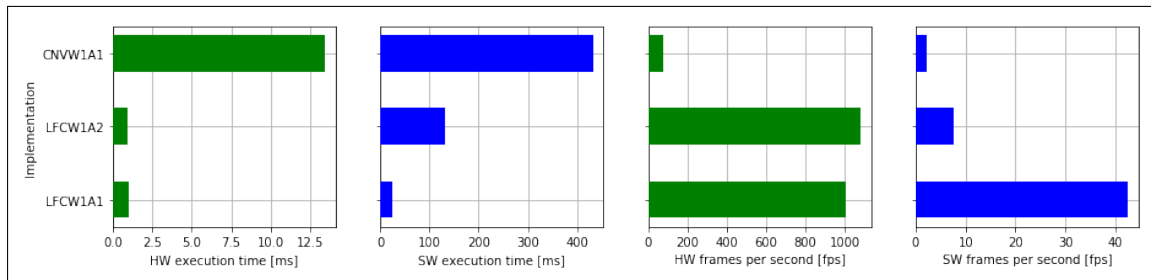


Fig. 8.1: Execution times of each BNN topology

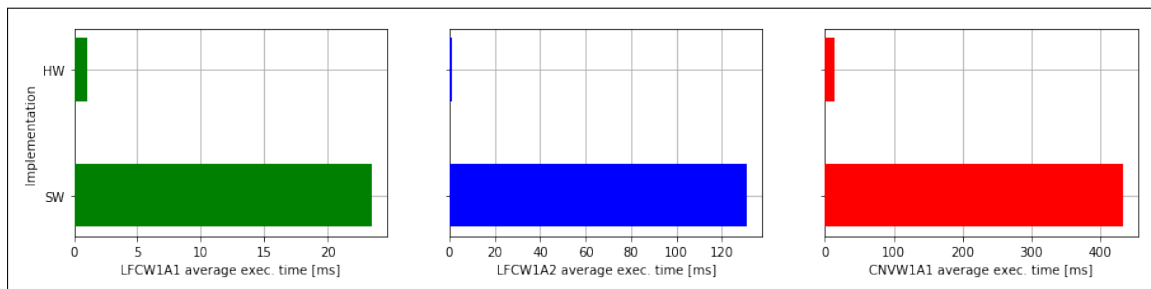


Fig. 8.2: Alternative view of the execution times of each BNN topology

Object detection also shows some very interesting results, although it should be noted that Darknet was used to run the SW tests as the software implementation was not available, perhaps due to an error on the corresponding shared library. Replacing the library did not, however, still resulted in the classifier not working. The Darknet network was loaded from the values used for the QNN-MO-PYNQ project, and only the performance of the classification function was measured. The Darknet fork [35] contains the benchmarked function, named `classify_nosort` and found in the Python bindings file. It is a modified version of the `classify` function that skips the sorting of results. Darknet is also implemented in C and is used in QNN-MO-PYNQ in both accelerated and non-accelerated versions. Running the Darknet binary, which takes additional steps such as loading the network configuration, took significantly longer than the classification-only measurement, thus the major factors that would contaminate the benchmark results have been discarded.

The time, once again, is measured right before and right after running the classification functions of both HW and SW implementations. Figure 8.3 shows the results in execution times and frames per second. The accelerated version still processes the first and last layers of the network with a SW implementation, which as shown in figure 8.4 takes about twice the time to run than the pure-hardware section. This means the project has still a lot of room for performance improvement by accelerating the first and last layer.

The classification that takes an average of over eight seconds to run in the software implementation takes about a third of a second in the accelerated version including the first and last layers, but just 100 milliseconds if only the middle layers are examined. This means that even with the bottleneck of the software layers, several frames per second can be processed, making it capable of real-time applications.

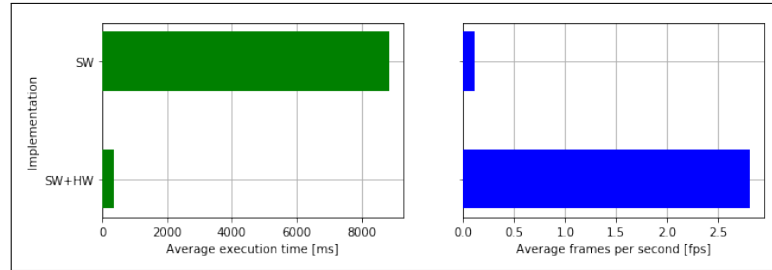


Fig. 8.3: Comparison of HW+SW and SW QNN execution times

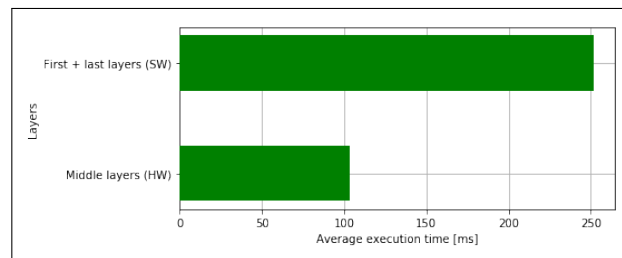


Fig. 8.4: Comparison of border (SW) and middle (HW) layers

While these benchmarks evaluate the bottlenecks of the applications, there is still room for performance improvements by developing new applications for PYNQ and by optimizing the ones that should be performed by the PS. There is also some time consumed loading the overlays, but that action has to be performed only once at the start of the program and is very well worth the half a second it takes.

This project has also produced some documentation resources. Jupyter Notebooks, mentioned at the beginning of the chapter and shown in figure 8.5 have been created showing the usage of the overlays and documenting the benchmarking methods and results. They may serve as the starting point for future projects based on this one, as they document the Python code used in each of the applications while using PYNQ and the libraries described in section 9.1. They might save time for future researchers as many of the problems encountered during development are already solved on the notebooks. The repository has been given a BSD-3 license, the same license PYNQ and other related projects have. The reader is encouraged to take a look at the notebooks, as they provide more insight into the results when combined with this document. They are accessible either with the citation or at the following link: <https://gitlab.com/mikelsr/pynq>. A recording of the live execution of each notebook is available at the following Google Drive link: https://drive.google.com/drive/folders/1zM5ILo1_jCNjf--qIyyCRUW2r89IW10n?usp=sharing.

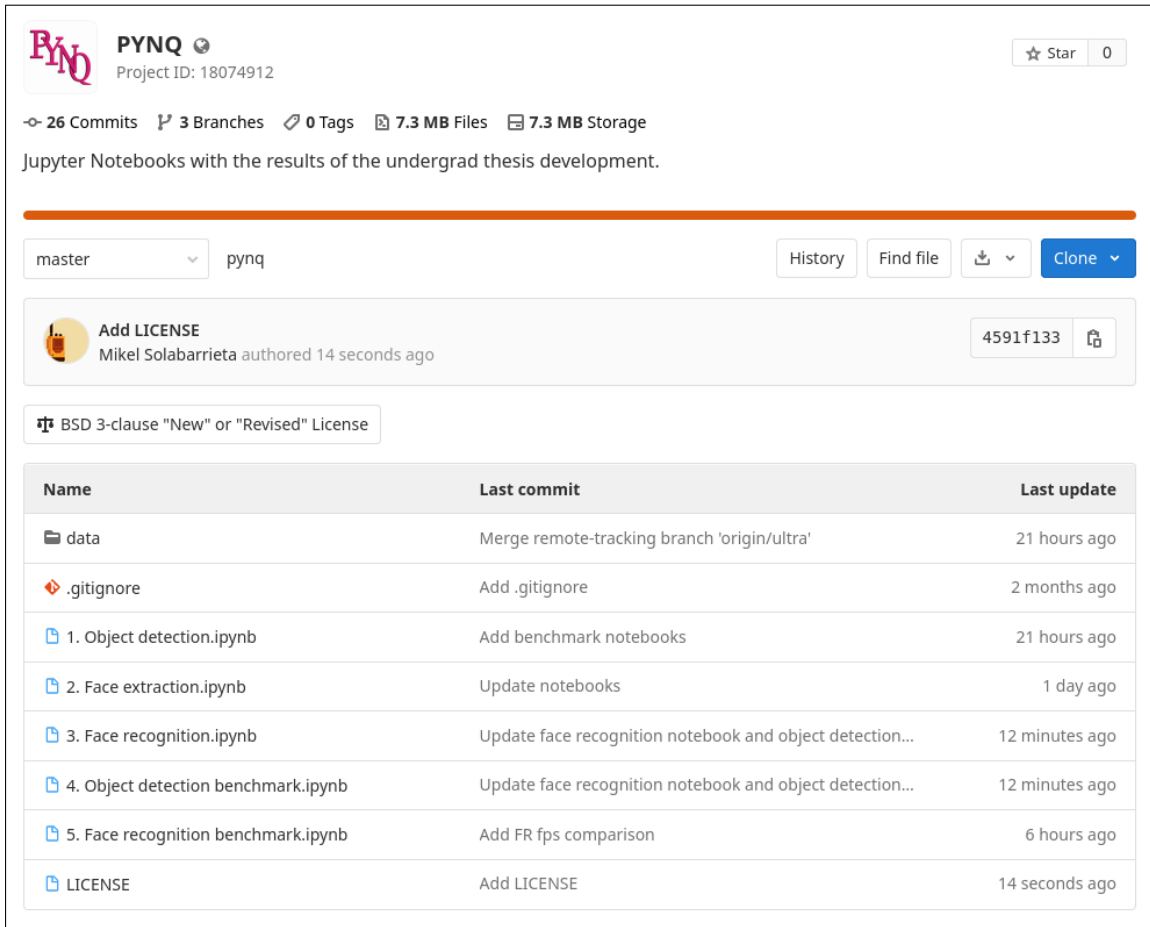
All the modified code libraries are publicly available in the web pages provided in section 7.5. In addition, the BNN-PYNQ library now includes the bitstreams to use the models trained in this project and the QNN-PYNQ-MO has been modified to install the custom version of Darknet. Both can be installed on an Ultra96 board running PYNQ with the Python package manager. Listing 8.1 shows how to install both of them.

```

1 sudo pip3 install git+https://github.com/mikelsr/BNN-PYNQ.git
2 sudo pip3 install git+https://github.com/mikelsr/QNN-MO-PYNQ.git
    
```

Listing 8.1: "BNN and QNN fork installation command"

Finally, although the project was successful in proving PYNQ has a lot of potential to transform edge computing by testing two applications, the implementation of both into a single overlay was achieved. This is not to say it is not possible; it is and it will be proposed as future work but it would have provided some interesting results and a very complete application ready to deploy on a real scenario.



Project ID: 18074912

☆ Star 0

26 Commits 3 Branches 0 Tags 7.3 MB Files 7.3 MB Storage

Jupyter Notebooks with the results of the undergrad thesis development.

master pynq History Find file Clone

Add LICENSE
Mikel Solabarrieta authored 14 seconds ago
4591f133

BSD 3-clause "New" or "Revised" License

Name	Last commit	Last update
data	Merge remote-tracking branch 'origin/ultra'	21 hours ago
.gitignore	Add .gitignore	2 months ago
1. Object detection.ipynb	Add benchmark notebooks	21 hours ago
2. Face extraction.ipynb	Update notebooks	1 day ago
3. Face recognition.ipynb	Update face recognition notebook and object detection...	12 minutes ago
4. Object detection benchmark.ipynb	Update face recognition notebook and object detection...	12 minutes ago
5. Face recognition benchmark.ipynb	Add FR fps comparison	6 hours ago
LICENSE	Add LICENSE	14 seconds ago

Fig. 8.5: GitLab repository containing the Jupyter Notebooks

Chapter 9

Conclusions

The Zynq architecture and PYNQ framework have proven to be the enablers of the adoption of new edge-computing applications on low-power devices. The heterogeneous dual processor architecture of Zynq allows performing parallelizable tasks much faster with considerable resource usage reductions. Having a PS-oriented system allows the development of software to be PS focused and using the PL in a way comparable to calling a software library, thanks to the overlay design. PYNQ has also proven to be able to provide developers the means to use FPGA accelerated software libraries without requiring deep understanding of FPGAs. Using well documented, existing overlays was easier than expected and quickly showed results that proved PYNQ was worth evaluating.

The success of PYNQ may imply a shift in edge-computing application design paradigms that favors local processing done by edge-computing devices such as the Ultra96-v1 MPSoC studied in this project in contrast to the traditional centralized processing. Their open approach favoring community projects, free software and publicly available research might add to the potential of PYNQ, resulting in an even greater impact. It is also noticeable that Xilinx staff have made an effort to document the provided tools, and despite having room for improvement this effort is greatly appreciated and has made this project fairly reasonable. Documentation in form of interactive Jupyter Notebooks make it easier to get familiarized with the platform and try new ideas. This is why the steps to reproduce the results of the project have been made available in the form of Jupyter Notebooks. The community is also active and will, in many cases, be of invaluable help.

Both of the use cases: facial recognition and object detection, experienced performance improvements of one or several orders of magnitude in their classification stages. The utility of heterogeneous MPSoC devices has been proved, as even in the case that experienced the least improvement, performance was improved by 1100% in the classification stages. This means, however, that there is work to be done to optimize the rest of the programs either by taking advantage of the uniqueness of the Zynq architecture or by optimizing the functions that can't be accelerated.

Hopefully, the success of the current use cases will inspire more people to create new applications for PYNQ. As seen in chapter 8, both applications are now feasible in real-time, although there is a noticeable difference between the capacity of facial recognition and object detection, the former

being much faster than the latter. The latter can still process multiple frames per second, thus proving PYNQ has the potential to bring many intensive applications to edge computing. The attempt at implementing both functionalities into a single framework, while unsuccessful, seems to be on the correct path and might have been taken further with more resources, time or with another Zynq UltraScale+ device with more resources. Zynq devices will most surely evolve over time, and the integration will become easier when hardware advancements converge with the wider range of applications and documentation being developed.

In summary, the Zynq UltraScale+ architecture provides relatively low-energy edge computing devices unprecedented performance capacity and the PYNQ framework makes that capacity accessible not only to PL experts but to many other developers and engineers with a more shallow understanding of the topic. The PS-oriented design gives it flexibility and familiarity that will benefit both developers and the architecture itself.

9.1 Future Work

The failed attempt at integrating two overlays has left an itch for investigating the subject deeper until a solution is found. Given time, resources, and help from the PYNQ community it seems like an achievable goal worth following. The issue can surely be approached from many angles and there are likely different solutions that, while having little to do with each other, all provide more insight into what is currently possible with PYNQ. The simple solution based on the information available seems to be to obtain a PYNQ board with more capacity or find a way of optimizing the size of the design.

It would also be interesting to compare the applications of this project to other low-consumption edge computing devices. These devices could range from widely extended, cheaper Raspberry Pi to more complex, specialized, and more expensive ones such as the NVIDIA Jetson which has a high-end GPU and is specially build to AI applications such as the ones examined in this project. Comparing them would provide interesting data that could be used to decide what device should be used to solve each problem that may arise in edge computing.

As mentioned previously, now that the classifier has been accelerated the applications will have new bottlenecks. Iteratively accelerating or optimizing the bottlenecks could increase the performance of the programs to the point of being able to process more than one video stream in real-time. This is, however, more far fetched than integrating overlays and benchmarking more devices and is only proposed as a line of work to fully optimize an application for PYNQ.

Lastly, there is a very wide range of applications that could benefit from FPGA acceleration. Finding new use cases and either trying existing overlays or creating new ones would benefit PYNQ and be useful for future developers.

9.2 Personal Assessment

Experimenting with PYNQ has been a great closing experience for both Computer Engineering and Industrial Electronics and Automation Engineering degrees, despite being the final project of the latter. The project has benefited from many of the competencies developed during the degree, from higher level data manipulation to analyzing lower level Vivado designs and many more in between. As a total newcomer to PYNQ, the project involved learning about a topic from scratch and trying things until being able to achieve the proposed objectives. It has also involved many ours examining both Python and C code from multiple sources and examining project architectures and designs. A surprising amount was learned about AI and AI topologies while researching and developing the project. While challenging, solutions almost always ended up coming for every problem and it has been a really interesting experience. There is much to learn from PYNQ, it has ambitious goals and potential to really influence edge computing.

As a closing statement, PYNQ has proven its value and should be closely monitored for future advancements. The project was interesting from the initial research to the final stages of development, it required interdisciplinary knowledge, presented a varied set of problems to solve and has been an overall great learning experience.

Bibliography

- [1] Xilinx. “Zynq UltraScale+ MPSoC Data Sheet:Overview”. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. [Online; accessed 08-Jun-2020].
- [2] Xilinx. “Zynq UltraScale+ MPSoC - Product Tables and Product Selection Guide”. <https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf>. [Online; accessed 08-Jun-2020].
- [3] Xilinx. “Zynq Migration Guide”. https://www.xilinx.com/support/documentation/user_guides/ug1213-zynq-migration-guide.pdf. [Online; accessed 10-Jun-2020].
- [4] Xilinx. “Zynq-7000 SoC Data Sheet: Overview”. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Online; accessed 08-Jun-2020].
- [5] Xilinx. “PYNQ Homepage”. <http://www.pynq.io/>. [Online; accessed 10-Jun-2020].
- [6] Travis Oliphant (Original author). “NumPy”. <https://numpy.org>. [Online; accessed 10-Jun-2020].
- [7] Python Software Foundation. “asyncio — Asynchronous I/O”. <https://docs.python.org/3/library/asyncio.html>. [Online; accessed 10-Jun-2020].
- [8] Xilinx. “PYNQ Hello World”. <https://github.com/Xilinx/PYNQ>HelloWorld>. [Online; accessed 17-Jun-2020].
- [9] LinuxGizmos. “Ultra96-V2 SBC adds certified WiFi and industrial temp support”. <https://linuxgizmos.com/ultra96-v2-sbc-adds-certified-wifi-and-industrial-temp-support/>. [Online; accessed 20-Jun-2020].
- [10] Avnet. “Ultra96-V1 Single Board Computer Hardware User’s Guide”. https://zedboard.org/sites/default/files/documentations/Ultra96-HW-User-Guide-rev-1-0-V1_1.pdf. Version 1.1 [Online; accessed 15-May-2020].
- [11] Xilinx. “Introduction to FPGA Design with Vivado High-Level Synthesis”. https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf. [Online; accessed 11-Jun-2020].

- [12] Xilinx. “Introduction to High-Level Synthesis with Vivado HLS”. https://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf. [Online; accessed 20-Jun-2020].
- [13] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Visser. Finn: A framework for fast, scalable binarized neural network inference. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.
- [14] Xilinx. “FINN Documentation”. <https://finn.readthedocs.io>. [Online; accessed 10-Jun-2020].
- [15] Xilinx. “FINN-HLS”. <https://finn-hlslib.readthedocs.io/en/latest/>. [Online; accessed 10-Jun-2020].
- [16] Xilinx. “Binarized Neural Networks (BNNs) on PYNQ”. <https://github.com/Xilinx/BNN-PYNQ>. [Online; accessed 15-May-2020].
- [17] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.
- [18] Adiuvio. “PYNQ the coolest Python Framework you never heard about”. <https://files.devnetwork.cloud/DeveloperWeekAustin/presentations/2019/Adam-Taylor.pdf>. [Online; accessed 10-Jun-2020].
- [19] Xilinx. “Quantized Neural Networks (QNNs) on PYNQ”. <https://github.com/Xilinx/QNN-MO-PYNQ>. [Online; accessed 15-May-2020].
- [20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016.
- [21] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.
- [22] Amro Kamal. “YOLO, YOLOv2 and YOLOv3: All You want to know”. https://medium.com/@amrokamal_47691/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899, 2019. [Online; accessed 23-May-2020].
- [23] Joseph Redmon. “Darknet: Open Source Neural Networks in C”. <http://pjreddie.com/darknet/>, 2013–2016.
- [24] Xilinx. “Computer Vision Overlays on PYNQ”. <https://github.com/Xilinx/PYNQ-ComputerVision>. [Online; accessed 16-Jun-2020].
- [25] Xilinx. “Xilinx xfOpenCV Library”. <https://github.com/Xilinx/xfopencv>. [Online; accessed 16-Jun-2020].
- [26] D. Kriegman P. Belhumeur, J. Hespanha. “Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection, Ó IEEE Transactions on Pattern Analysis and Machine Intelligence”. <http://vision.ucsd.edu/content/yale-face-database>, 1997.

- [27] Wikipedia. “One-hot — Wikipedia, the free encyclopedia”. <https://en.wikipedia.org/wiki/One-hot>. [Online; accessed 14-May-2020].
- [28] Steve Golson. “One-hot state machine design for FPGAs”, 1993.
- [29] Gregory S. Kielian. “JPG and PNG to MNIST”. <https://github.com/ledinhphuong/JPG-PNG-to-MNIST-NN-Format>. [Online; accessed 13-May-2020].
- [30] Phuong Le. “JPG and PNG to MNIST (Fork)”. <https://github.com/ledinhphuong/JPG-PNG-to-MNIST-NN-Format/tree/wip>. [Online; accessed 14-May-2020].
- [31] Mikel Solabarrieta. “PYNQ Notebooks”. <https://gitlab.com/mikelsr/pynq>. [Online; accessed 22-May-2020].
- [32] Mikel Solabarrieta. “yale-to-gtsrb”. <https://github.com/mikelsr/yate-to-gtsrb>. [Online; accessed 16-Jun-2020].
- [33] Joseph Redmon. “Darknet”. <https://github.com/pjreddie/darknet>. [Online; accessed 22-May-2020].
- [34] Giulio Gambardella. “Darknet (Fork)”. <https://github.com/giuliogamba/darknet>. [Online; accessed 22-May-2020].
- [35] Mikel Solabarrieta. “Darknet (Fork)”. <https://github.com/mikelsr/darknet>. [Online; accessed 22-May-2020].

Acronyms

ACP	Accelerator Coherency Port
AI	Artificial Intelligence
AMBA	Advanced Micro-controller Bus Architecture
APU	Application Processor Unit
ARM	Advanced RISC Machine
AXI	Advanced Extensible Interface
BC	Bluetooth Classic
BLE	Bluetooth Low Energy
BNN	Binarized Neural Network
BSD	Berkeley Software Distribution
CFFI	C Foreign Function Interface
CLB	Configurable Logic Block
CIFRAR	Canadian Institute For Advanced Research
CPU	Central processing unit
CV	Computer Vision
DDR	Double Data Rate
DMA	Direct Memory Access
EMIO	Extended Multiplexed IO
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FPS	Frames per Second
GPU	Graphics Processing Unit
GTSRB	German Traffic Sign Recognition Benchmark

9. Acronyms

HLS	High-Level Synthesis
HPC	High Performance Computing
HW	Hardware
IO	Input/Output
IoT	Internet of Things
IP	Intellectual Property
JSON	JavaScript Object Notation
LUT	Look-Up Table
MIO	Multiplexed IO
MMU	Memory Management Unit
MNIST	Modified National Institute of Standards and Technology
MPU	Master Processing Unit
MPSoC	Multiprocessor System-on-Chip
OCM	On-Chip Memory
ONNX	Open Neural Network Exchange
PPM	Portable Pixmap
PS	Processing System
PL	Programmable Logic
QNN	Quantized Neural Network
QoS	Quality of Service
VHDL	Very High Speed Integrated Circuit Hardware Description Language
RoHS	Restriction of Hazardous Substances Directive
ROM	Read Only Memory
RPU	Real-Time Processing Unit
RTL	Register-Transfer Level
SoC	System-on-Chip
SRAM	Static Random Access Memory
SSH	Secure Shell
SW	Software
YOLO	You Only Look Once

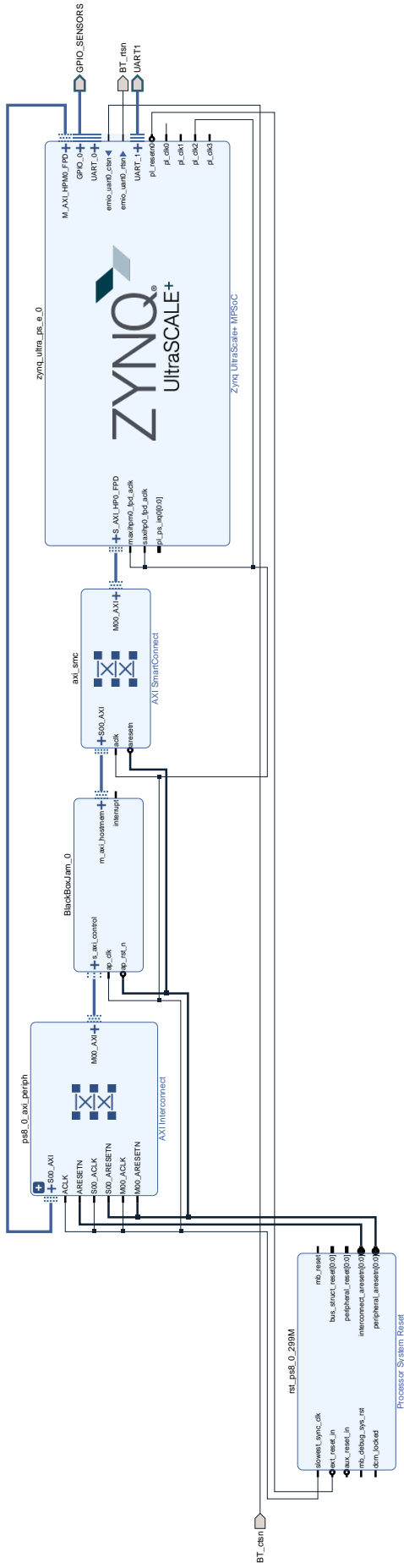
Acknowledgements

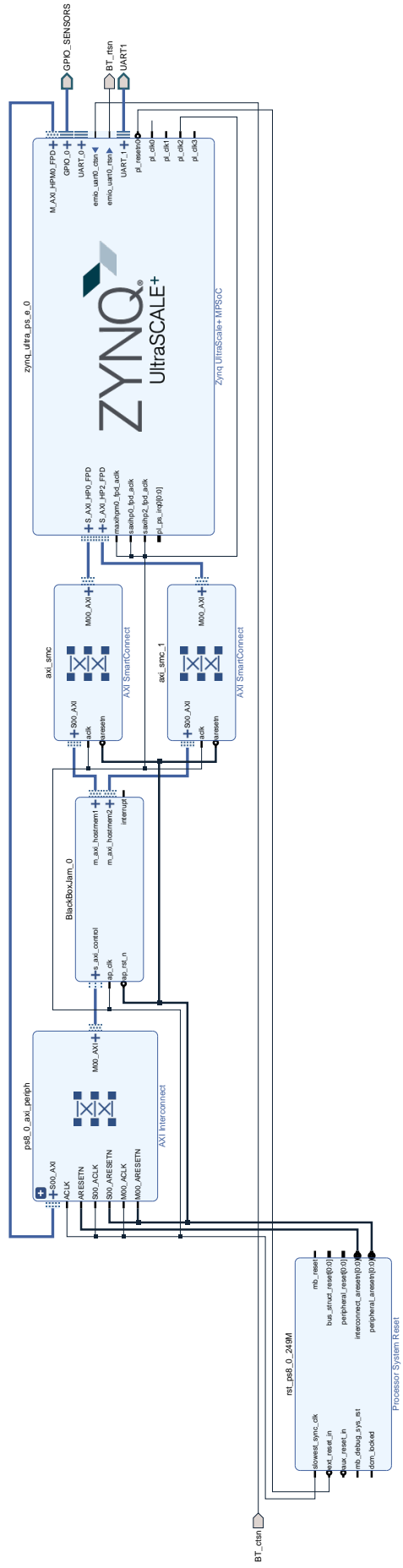
I would like to thank Ignacio Angulo Martínez for his continuous support and tutoring, proposing solutions when I was stuck on a problem, as well as introducing me to PYNQ and providing all the necessary resources and materials. The project would have not been possible without his help and advice. I would also like to thank Antonio David Masegosa for advising me on Artificial Intelligence and taking the time to answer my questions.

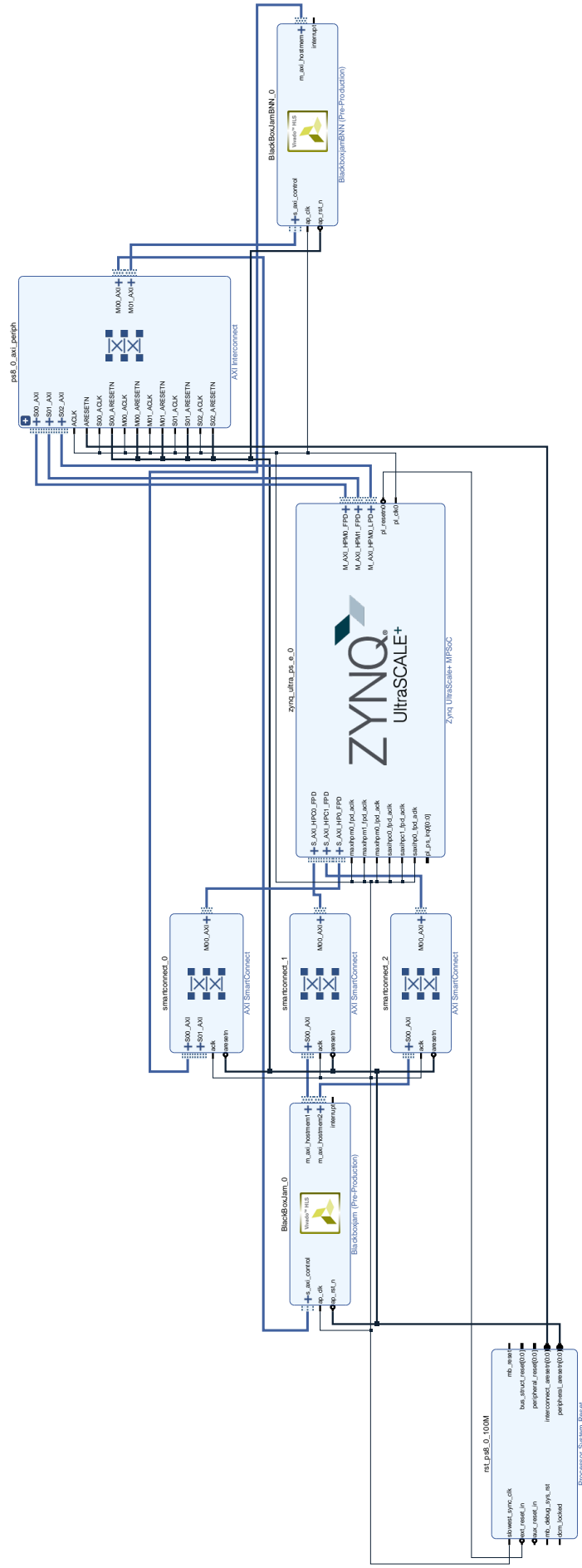
Appendix A

Vivado Block Designs

This annex includes, in order, the Vivado block diagrams for the LFCW1A2 (BNN), W1A3 (QNN) and the integration of both.







Appendix B

Integration Utilization Report

This annex includes the BNN and QNN BlackBoxJam utilization reports. The first document contains the utilization report for one of the BNN BlackBoxJam IPs, while the second one contains the utilization report of the QNN IP.

Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

```
-----
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date        : Tue May 12 23:26:16 2020
| Host       : vm running 64-bit Ubuntu 18.04.4 LTS
| Command    : report_utilization -file design_1_BlackBoxJamBNN_0_0_utilization_synth.rpt -
pb design_1_BlackBoxJamBNN_0_0_utilization_synth.pb
| Design     : design_1_BlackBoxJamBNN_0_0
| Device    : xczu3egsbva484-1
| Design State : Synthesized
-----
```

Utilization Design Information

Table of Contents

- ```

1. CLB Logic
1.1 Summary of Registers by Type
2. BLOCKRAM
3. ARITHMETIC
4. I/O
5. CLOCK
6. ADVANCED
7. CONFIGURATION
8. Primitives
9. Black Boxes
10. Instantiated Netlists
```

### 1. CLB Logic

```

```

| Site Type              | Used  | Fixed | Available | Util% |
|------------------------|-------|-------|-----------|-------|
| CLB LUTs*              | 47852 | 0     | 70560     | 67.82 |
| LUT as Logic           | 42986 | 0     | 70560     | 60.92 |
| LUT as Memory          | 4866  | 0     | 28800     | 16.90 |
| LUT as Distributed RAM | 4352  | 0     |           |       |
| LUT as Shift Register  | 514   | 0     |           |       |
| CLB Registers          | 32009 | 0     | 141120    | 22.68 |
| Register as Flip Flop  | 32009 | 0     | 141120    | 22.68 |
| Register as Latch      | 0     | 0     | 141120    | 0.00  |
| CARRY8                 | 780   | 0     | 8820      | 8.84  |
| F7 Muxes               | 1658  | 0     | 35280     | 4.70  |
| F8 Muxes               | 638   | 0     | 17640     | 3.62  |
| F9 Muxes               | 0     | 0     | 8820      | 0.00  |

```

```

\* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt\_design after synthesis, if not already completed, for a more realistic count.

### 1.1 Summary of Registers by Type

```

```

| Total | Clock Enable | Synchronous | Asynchronous |
|-------|--------------|-------------|--------------|
| 0     | -            | -           | -            |
| 0     | -            | -           | Set          |
| 0     | -            | -           | Reset        |
| 0     | -            | Set         | -            |

```

```

|       |     |       |       |
|-------|-----|-------|-------|
| 0     |     | Reset | -     |
| 0     | Yes | -     | -     |
| 0     | Yes | -     | Set   |
| 0     | Yes | -     | Reset |
| 65    | Yes | Set   | -     |
| 31944 | Yes | Reset | -     |

## 2. BLOCKRAM

| Site Type      | Used | Fixed | Available | Util% |
|----------------|------|-------|-----------|-------|
| Block RAM Tile | 110  | 0     | 216       | 50.93 |
| RAMB36/FIFO*   | 6    | 0     | 216       | 2.78  |
| RAMB36E2 only  | 6    |       |           |       |
| RAMB18         | 208  | 0     | 432       | 48.15 |
| RAMB18E2 only  | 208  |       |           |       |

\* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E2 or one FIFO18E2. However, if a FIFO18E2 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E2

## 3. ARITHMETIC

| Site Type    | Used | Fixed | Available | Util% |
|--------------|------|-------|-----------|-------|
| DSPs         | 4    | 0     | 360       | 1.11  |
| DSP48E2 only | 4    |       |           |       |

## 4. I/O

| Site Type  | Used | Fixed | Available | Util% |
|------------|------|-------|-----------|-------|
| Bonded IOB | 0    | 0     | 82        | 0.00  |

## 5. CLOCK

| Site Type            | Used | Fixed | Available | Util% |
|----------------------|------|-------|-----------|-------|
| GLOBAL CLOCK BUFFERS | 0    | 0     | 196       | 0.00  |
| BUFGCE               | 0    | 0     | 88        | 0.00  |
| BUFGCE_DIV           | 0    | 0     | 12        | 0.00  |
| BUFG_PS              | 0    | 0     | 72        | 0.00  |
| BUFGCTRL*            | 0    | 0     | 24        | 0.00  |
| PLL                  | 0    | 0     | 6         | 0.00  |
| MMCM                 | 0    | 0     | 3         | 0.00  |

\* Note: Each used BUFGCTRL counts as two global buffer resources. This table does not include

global clocking resources, only buffer cell usage. See the Clock Utilization Report (report\_clock\_utilization) for detailed accounting of global clocking resource availability.

## 6. ADVANCED

```

+-----+-----+-----+-----+
| Site Type | Used | Fixed | Available | Util% |
+-----+-----+-----+-----+
| PS8 | 0 | 0 | 1 | 0.00 |
| SYSMONE4 | 0 | 0 | 1 | 0.00 |
+-----+-----+-----+-----+

```

## 7. CONFIGURATION

```

+-----+-----+-----+-----+
| Site Type | Used | Fixed | Available | Util% |
+-----+-----+-----+-----+
BSCANE2	0	0	4	0.00
DNA_PORTE2	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE4	0	0	1	0.00
ICAPE3	0	0	2	0.00
MASTER_JTAG	0	0	1	0.00
STARTUPE3	0	0	1	0.00
+-----+-----+-----+-----+

```

## 8. Primitives

```

+-----+-----+-----+
| Ref Name | Used | Functional Category |
+-----+-----+-----+
FDRE	31944	Register
LUT6	20107	CLB
LUT5	9974	CLB
LUT4	9582	CLB
LUT3	7061	CLB
RAMS32	4352	CLB
LUT2	2600	CLB
MUXF7	1658	CLB
CARRY8	780	CLB
MUXF8	638	CLB
SRL16E	514	CLB
RAMB18E2	208	Block Ram
LUT1	121	CLB
FDSE	65	Register
RAMB36E2	6	Block Ram
DSP48E2	4	Arithmetic
+-----+-----+-----+

```

## 9. Black Boxes

```

+-----+-----+
| Ref Name | Used |
+-----+-----+

```

10. Instantiated Netlists

-----

| Ref Name | Used |
|----------|------|
|----------|------|

Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

```

| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date : Tue May 12 23:33:11 2020
| Host : vm running 64-bit Ubuntu 18.04.4 LTS
| Command : report_utilization -file design_1_BlackBoxJam_0_0_utilization_synth.rpt -pb
design_1_BlackBoxJam_0_0_utilization_synth.pb
| Design : design_1_BlackBoxJam_0_0
| Device : xczu3egsbva484-1
Design State : Synthesized
```

## Utilization Design Information

### Table of Contents

- ```
-----
1. CLB Logic
1.1 Summary of Registers by Type
2. BLOCKRAM
3. ARITHMETIC
4. I/O
5. CLOCK
6. ADVANCED
7. CONFIGURATION
8. Primitives
9. Black Boxes
10. Instantiated Netlists
```

1. CLB Logic

```
-----
```

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	36472	0	70560	51.69
LUT as Logic	33117	0	70560	46.93
LUT as Memory	3355	0	28800	11.65
LUT as Distributed RAM	2304	0		
LUT as Shift Register	1051	0		
CLB Registers	32918	0	141120	23.33
Register as Flip Flop	32918	0	141120	23.33
Register as Latch	0	0	141120	0.00
CARRY8	1172	0	8820	13.29
F7 Muxes	277	0	35280	0.79
F8 Muxes	125	0	17640	0.71
F9 Muxes	0	0	8820	0.00

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

```
-----
```

Total	Clock Enable	Synchronous	Asynchronous
0	-	-	-
0	-	-	Set
0	-	-	Reset
0	-	Set	-

0		Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
292	Yes	Set	-
32626	Yes	Reset	-

2. BLOCKRAM

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	190.5	0	216	88.19
RAMB36/FIFO*	187	0	216	86.57
RAMB36E2 only	187			
RAMB18	7	0	432	1.62
RAMB18E2 only	7			

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E2 or one FIFO18E2. However, if a FIFO18E2 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E2

3. ARITHMETIC

Site Type	Used	Fixed	Available	Util%
DSPs	53	0	360	14.72
DSP48E2 only	53			

4. I/O

Site Type	Used	Fixed	Available	Util%
Bonded IOB	0	0	82	0.00

5. CLOCK

Site Type	Used	Fixed	Available	Util%
GLOBAL CLOCK BUFFERS	0	0	196	0.00
BUFGCE	0	0	88	0.00
BUFGCE_DIV	0	0	12	0.00
BUFG_PS	0	0	72	0.00
BUFGCTRL*	0	0	24	0.00
PLL	0	0	6	0.00
MMCM	0	0	3	0.00

* Note: Each used BUFGCTRL counts as two global buffer resources. This table does not include

global clocking resources, only buffer cell usage. See the Clock Utilization Report (report_clock_utilization) for detailed accounting of global clocking resource availability.

6. ADVANCED

```

+-----+-----+-----+-----+
| Site Type | Used | Fixed | Available | Util% |
+-----+-----+-----+-----+
| PS8       | 0    | 0     | 1         | 0.00  |
| SYSMONE4  | 0    | 0     | 1         | 0.00  |
+-----+-----+-----+-----+

```

7. CONFIGURATION

```

+-----+-----+-----+-----+
| Site Type | Used | Fixed | Available | Util% |
+-----+-----+-----+-----+
| BSCANE2   | 0    | 0     | 4         | 0.00  |
| DNA_PORTE2 | 0    | 0     | 1         | 0.00  |
| EFUSE_USR  | 0    | 0     | 1         | 0.00  |
| FRAME_ECCE4 | 0    | 0     | 1         | 0.00  |
| ICAPE3    | 0    | 0     | 2         | 0.00  |
| MASTER_JTAG | 0    | 0     | 1         | 0.00  |
| STARTUPE3 | 0    | 0     | 1         | 0.00  |
+-----+-----+-----+-----+

```

8. Primitives

```

+-----+-----+-----+
| Ref Name | Used | Functional Category |
+-----+-----+-----+
| FDRE     | 32626 | Register            |
| LUT6     | 11684 | CLB                 |
| LUT4     | 9545  | CLB                 |
| LUT5     | 8245  | CLB                 |
| LUT3     | 8136  | CLB                 |
| LUT2     | 5545  | CLB                 |
| RAMD32   | 4032  | CLB                 |
| CARRY8   | 1172  | CLB                 |
| SRL16E   | 1050  | CLB                 |
| RAMS32   | 576   | CLB                 |
| LUT1     | 437   | CLB                 |
| FDSE     | 292   | Register            |
| MUXF7    | 277   | CLB                 |
| RAMB36E2 | 187   | Block Ram           |
| MUXF8    | 125   | CLB                 |
| DSP48E2  | 53    | Arithmetic          |
| RAMB18E2 | 7     | Block Ram           |
| SRLC32E  | 1     | CLB                 |
+-----+-----+-----+

```

9. Black Boxes

```

+-----+-----+

```



```
| Ref Name | Used |  
+-----+-----+
```

10. Instantiated Netlists

```
+-----+-----+  
| Ref Name | Used |  
+-----+-----+
```


Appendix C

HLS Example

Some of the files used to generate the LFCW1A1 bitstream are included in this annex. They are located in the original BNN-PYNQ repository (<https://github.com/Xilinx/BNN-PYNQ>). The first one contains the HLS description of the network, the second one contains the configuration of the network layers and the final one contains the software implementation of the network. The files depend on two other repositories, “finn-hls” (<https://github.com/Xilinx/finn-hlslib>) for the most part of the HLS definition and “xilinx-tiny-cnn” (<https://github.com/Xilinx/xilinx-tiny-cnn>) for the neural network implementation.

```
/*
 * Copyright (c) 2016, Xilinx, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the copyright holder nor the names of its
 *    contributors may be used to endorse or promote products derived from
 *    this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION). HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
/

/*
 *
 * @file top.cpp
 *
 * HLS Description of the LFC BNN, with axi-lite based parameter loading (DoMemInit)
 * and dataflow architecture of the image inference (DoCompute)
 *
 */
/

#include "config.h"
#include "bnn-library.h"
#include "dma.h"
#include "fclayer.h"
#include "weights.hpp"
#include "activations.hpp"
#include "interpret.hpp"

static BinaryWeights<L0_SIMD,L0_PE,L0_WMEM> weights0;
static BinaryWeights<L1_SIMD,L1_PE,L1_WMEM> weights1;
static BinaryWeights<L2_SIMD,L2_PE,L2_WMEM> weights2;
static BinaryWeights<L3_SIMD,L3_PE,L3_WMEM> weights3;

static ThresholdsActivation<L0_TMEM,L0_PE,L0_API,ap_int<16>,ap_uint<L0_API>> threshs0;
static ThresholdsActivation<L1_TMEM,L1_PE,L1_API,ap_int<16>,ap_uint<L1_API>> threshs1;
static ThresholdsActivation<L2_TMEM,L2_PE,L2_API,ap_int<16>,ap_uint<L2_API>> threshs2;
static ThresholdsActivation<L3_TMEM,L3_PE,L3_API,ap_int<16>,ap_uint<L3_API>> threshs3;
```

```

unsigned int paddedSizeHW(unsigned int in, unsigned int padTo) {
    if(in % padTo == 0) {
        return in;
    } else {
        return in + padTo - (in % padTo);
    }
}

void DoMemInit(unsigned int targetLayer, unsigned int targetMem, unsigned int targetInd,
unsigned int targetThresh, ap_uint<64> val) {
    switch(targetLayer) {
        case 0:
            weights0.m_weights[targetMem][targetInd] = val;
            break;
        case 1:
            threshs0.m_thresholds[targetMem][targetInd][targetThresh] = val;
            break;
        case 2:
            weights1.m_weights[targetMem][targetInd] = val;
            break;
        case 3:
            threshs1.m_thresholds[targetMem][targetInd][targetThresh] = val;
            break;
        case 4:
            weights2.m_weights[targetMem][targetInd] = val;
            break;
        case 5:
            threshs2.m_thresholds[targetMem][targetInd][targetThresh] = val;
            break;
        case 6:
            weights3.m_weights[targetMem][targetInd] = val;
            break;
        case 7:
            threshs3.m_thresholds[targetMem][targetInd][targetThresh] = val;
            break;
    }
}

void DoCompute(ap_uint<64> *in, ap_uint<64> *out, const unsigned int numReps) {

    hls::stream<ap_uint<64>> memInStrm("DoCompute.memInStrm");
    hls::stream<ap_uint<L0_PE * (L0_API + L0_APF)>> inter0("DoCompute.inter0");
    hls::stream<ap_uint<L1_PE * (L1_API + L1_APF)>> inter1("DoCompute.inter1");
    hls::stream<ap_uint<L2_PE * (L2_API + L2_APF)>> inter2("DoCompute.inter2");
    hls::stream<ap_uint<64>> memOutStrm("DoCompute.memOutStrm");

    // TODO: These values are just for comparability and produce the same amount
    // of FIFO buffer as in the preceding design.
    // If the resources allow, these depths should eventually go up to
    // Lx_DEPTH = Lx_MH/Lx_PE for a smooth operation.
    unsigned const L0_DEPTH = 512 / L0_PE;
    unsigned const L1_DEPTH = 512 / L1_PE;
    unsigned const L2_DEPTH = 512 / L2_PE;

#pragma HLS DATAFLOW
#pragma HLS stream depth=1024 variable=memInStrm // mask memory latency
#pragma HLS stream depth=L0_DEPTH variable=inter0
#pragma HLS stream depth=L1_DEPTH variable=inter1
#pragma HLS stream depth=L2_DEPTH variable=inter2
#pragma HLS stream depth=1024 variable=memOutStrm // mask memory latency

```

```

const unsigned int inBits = 28 * 28;
const unsigned int inBitsPadded = 832; // paddedSizeHW(inBits, 64)
const unsigned int inBytesPadded = inBitsPadded / 8;
const unsigned int outBits = 64;
const unsigned int outBitsPadded = 64; // paddedSizeHW(outBits, 64)
const unsigned int outBytesPadded = outBitsPadded / 8;
const unsigned int inWordsPerImg = inBitsPadded / 64;
const unsigned int outWordsPerImg = outBitsPadded / 64;

Mem2Stream_Batch<64, inBytesPadded>(in, memInStrm, numReps);
StreamingFCLayer_Batch<L0_MW, L0_MH, L0_SIMD, L0_PE, Recast<XnorMul>, Slice<ap_uint<1>>>
    (memInStrm, inter0, weights0, threshs0, numReps, ap_resource_lut());
StreamingFCLayer_Batch<L1_MW, L1_MH, L1_SIMD, L1_PE, Recast<XnorMul>, Slice<ap_uint<1>>>
    (inter0, inter1, weights1, threshs1, numReps, ap_resource_lut());
StreamingFCLayer_Batch<L2_MW, L2_MH, L2_SIMD, L2_PE, Recast<XnorMul>, Slice<ap_uint<1>>>
    (inter1, inter2, weights2, threshs2, numReps, ap_resource_lut());
StreamingFCLayer_Batch<L3_MW, L3_MH, L3_SIMD, L3_PE, Recast<XnorMul>, Slice<ap_uint<1>>>
    (inter2, memOutStrm, weights3, threshs3, numReps, ap_resource_lut());
Stream2Mem_Batch<64, outBytesPadded>(memOutStrm, out, numReps);
}

void BlackBoxJam(ap_uint<64> *in, ap_uint<64> *out, bool doInit,
                unsigned int targetLayer, unsigned int targetMem,
                unsigned int targetInd, unsigned int targetThresh, ap_uint<64> val, unsigned
int numReps) {
// pragmas for MLBP jam interface
// signals to be mapped to the AXI Lite slave port
#pragma HLS INTERFACE s_axilite port=return bundle=control
#pragma HLS INTERFACE s_axilite port=doInit bundle=control
#pragma HLS INTERFACE s_axilite port=targetLayer bundle=control
#pragma HLS INTERFACE s_axilite port=targetMem bundle=control
#pragma HLS INTERFACE s_axilite port=targetThresh bundle=control
#pragma HLS INTERFACE s_axilite port=targetInd bundle=control
#pragma HLS INTERFACE s_axilite port=val bundle=control
#pragma HLS INTERFACE s_axilite port=numReps bundle=control
// signals to be mapped to the AXI master port (hostmem)
#pragma HLS INTERFACE m_axi offset=slave port=in bundle=hostmem depth=256
#pragma HLS INTERFACE s_axilite port=in bundle=control
#pragma HLS INTERFACE m_axi offset=slave port=out bundle=hostmem depth=256
#pragma HLS INTERFACE s_axilite port=out bundle=control

//partition PE arrays
#pragma HLS ARRAY_PARTITION variable=weights0.m_weights complete dim=1
#pragma HLS ARRAY_PARTITION variable=threshs0.m_thresholds complete dim=1
#pragma HLS ARRAY_PARTITION variable=weights1.m_weights complete dim=1
#pragma HLS ARRAY_PARTITION variable=threshs1.m_thresholds complete dim=1
#pragma HLS ARRAY_PARTITION variable=weights2.m_weights complete dim=1
#pragma HLS ARRAY_PARTITION variable=threshs2.m_thresholds complete dim=1
#pragma HLS ARRAY_PARTITION variable=weights3.m_weights complete dim=1
#pragma HLS ARRAY_PARTITION variable=threshs3.m_thresholds complete dim=1

    if (doInit) {
        DoMemInit(targetLayer, targetMem, targetInd, targetThresh, val);
    } else {
        DoCompute(in, out, numReps);
    }
}

```

```
/**
 * Finnthesizer Config-File Generation
 *
 **/

#ifndef __LAYER_CONFIG_H_
#define __LAYER_CONFIG_H_

/**
 * Fully-Connected Layer L0:
 *   MatW = 832 MatH = 1024
 *   SIMD = 64 PE = 32
 *   WMEM = 416 TMEM = 32
 *   #Ops = 1703936 Ext Latency = 416
 **/

#define L0_SIMD 64
#define L0_PE 32
#define L0_WMEM 416
#define L0_TMEM 32
#define L0_MW 832
#define L0_MH 1024
#define L0_WPI 1
#define L0_API 1
#define L0_WPF 0
#define L0_APF 0

/**
 * Fully-Connected Layer L1:
 *   MatW = 1024 MatH = 1024
 *   SIMD = 32 PE = 64
 *   WMEM = 512 TMEM = 16
 *   #Ops = 2097152 Ext Latency = 512
 **/

#define L1_SIMD 32
#define L1_PE 64
#define L1_WMEM 512
#define L1_TMEM 16
#define L1_MW 1024
#define L1_MH 1024
#define L1_WPI 1
#define L1_API 1
#define L1_WPF 0
#define L1_APF 0

/**
 * Fully-Connected Layer L2:
 *   MatW = 1024 MatH = 1024
 *   SIMD = 64 PE = 32
 *   WMEM = 512 TMEM = 32
 *   #Ops = 2097152 Ext Latency = 512
 **/

#define L2_SIMD 64
#define L2_PE 32
#define L2_WMEM 512
#define L2_TMEM 32
#define L2_MW 1024
#define L2_MH 1024
#define L2_WPI 1
#define L2_API 1
```

```
#define L2_WPF 0
#define L2_APF 0

/**
 * Fully-Connected Layer L3:
 *   MatW = 1024 MatH = 64
 *   SIMD = 8 PE = 16
 *   WMEM = 512 TMEM = 4
 *   #Ops = 131072 Ext Latency = 512
 */

#define L3_SIMD 8
#define L3_PE 16
#define L3_WMEM 512
#define L3_TMEM 4
#define L3_MW 1024
#define L3_MH 64
#define L3_WPI 1
#define L3_API 1
#define L3_WPF 0
#define L3_APF 0

#endif // __LAYER_CONFIG_H_
```



```
extern "C" void load_parameters(const char* path) {
#include "config.h"
    FoldedMVInit("lfcW1A1-pynq");
    network<mse, adagrad> nn;
    makeNetwork(nn);
    cout << "Setting network weights and thresholds in accelerator..." << endl;
    FoldedMVLoadLayerMem(path, 0, L0_PE, L0_WMEM, L0_TMEM, L0_API);
    FoldedMVLoadLayerMem(path, 1, L1_PE, L1_WMEM, L1_TMEM, L1_API);
    FoldedMVLoadLayerMem(path, 2, L2_PE, L2_WMEM, L2_TMEM, L2_API);
    FoldedMVLoadLayerMem(path, 3, L3_PE, L3_WMEM, L3_TMEM, L3_API);
}

extern "C" int inference(const char* path, int results[64], int number_class, float
*usecPerImage) {
    std::vector<vec_t> test_images;
    std::vector<int> class_result;
    float usecPerImage_int;
    vec_t image;

    FoldedMVInit("lfcW1A1-pynq");
    network<mse, adagrad> nn;
    makeNetwork(nn);
    parse_mnist_images(path, &test_images, -1.0, 1.0, 0, 0);
    image = test_images[0];
    class_result=testPrebinarized_nolabel(test_images, number_class, usecPerImage_int);

    if(results) {
        std::copy(class_result.begin(),class_result.end(), results);
    }
    if (usecPerImage) {
        *usecPerImage = usecPerImage_int;
    }
    return (std::distance(class_result.begin(),std::max_element(class_result.begin(),
class_result.end())));
}

extern "C" int* inference_multiple(const char* path, int number_class, int *image_number,
float *usecPerImage, unsigned int enable_detail = 0) {
    std::vector<vec_t> test_images;
    std::vector<int> all_result;
    float usecPerImage_int;
    int* result;

    FoldedMVInit("lfcW1A1-pynq");
    network<mse, adagrad> nn;
    makeNetwork(nn);
    parse_mnist_images(path, &test_images, -1.0, 1.0, 0, 0);
    all_result=testPrebinarized_nolabel_multiple_images(test_images, number_class,
usecPerImage_int);

    result = new int [all_result.size()];
    std::copy(all_result.begin(),all_result.end(), result);
    if (image_number) {
        *image_number = all_result.size();
    }
    if (usecPerImage) {
        *usecPerImage = usecPerImage_int;
    }
    return result;
}

extern "C" void free_results(int * result) {
```

```
    delete[] result;
}

extern "C" void deinit() {
    FoldedMVDeinit();
}

extern "C" int main(int argc, char** argv) {
    if (argc != 5) {
        cout << "4 parameters are needed: " << endl;
        cout << "1 - folder for the binarized weights (binparam-**) - full path " << endl;
        cout << "2 - path to image to be classified" << endl;
        cout << "3 - number of classes in the dataset" << endl;
        cout << "4 - expected result" << endl;
        return 1;
    }

    float execution_time = 0;
    int class_inference = 0;
    int scores[64];

    load_parameters(argv[1]);
    class_inference = inference(argv[2], scores, atol(argv[3]), &execution_time);

    cout << "Detected class " << class_inference << " in " << execution_time << " microseconds"
<< endl;
    deinit();
    if (class_inference != atol(argv[4])) {
        return 1;
    } else {
        return 0;
    }
}
```

