

Politecnico di Torino

Master of Science in Electronic Engineering

Embedded Systems

**FPGA IMPLEMENTATION OF AN IMAGES  
CLASSIFIER**



**Politecnico  
di Torino**

**Supervisor**

Prof. Maurizio Martina

**Company Tutor Altran Italia**

Ing. Alessandro Bruscia

**Candidate**

Giulio Naggi

December 2021

## **Abstract**

Artificial intelligence is one of the main topic in nowadays scientific researches and it is starting to be appealing also for industrial productions. Due to the increase of interest of the industries, Xilinx, one of the major hardware supply companies, decided to create a development flow that allows to accelerate the production of artificial intelligence based systems. This thesis is centered on the study of the tool that allows to use the Xilinx new development flow, called Vitis AI. An overview on the tool functioning and its main component will be provided together with a project example whose main purpose is exploring the practical usage of the tool. In addition to that, the project has been extended to include the design of a hardware accelerator that allows to increase the performances of the developed application by transforming a software function into a hardware component.

# Contents

List of Figures . . . . .	III
List of Tables . . . . .	V
<b>1 Introduction</b>	<b>1</b>
1.1 The basis of Artificial Intelligence . . . . .	1
1.2 The importance of FPGA programming . . . . .	4
<b>2 Vitis AI overview</b>	<b>7</b>
2.1 The hardware core . . . . .	7
2.1.1 DPU architecture . . . . .	8
2.1.2 DPU configuration . . . . .	9
2.1.3 DPU integration . . . . .	12
2.2 The tools . . . . .	13
2.2.1 Vitis AI Quantizer . . . . .	14
2.2.2 Vitis AI Compiler . . . . .	15
2.2.3 Vitis AI Optimizer . . . . .	17
2.2.4 Vitis AI Profiler . . . . .	17
<b>3 The image classifier</b>	<b>20</b>
3.1 The AI model . . . . .	21
3.2 The application code . . . . .	27
3.3 The board preparation . . . . .	32
3.3.1 Hardware platform generation . . . . .	33
3.3.2 Software platform generation . . . . .	38
3.3.3 Vitis platform generation . . . . .	42
3.3.4 Application creation . . . . .	43
3.4 The ZCU102 connection . . . . .	46
3.5 The profiling process . . . . .	48
3.6 The kernel development . . . . .	51

## CONTENTS

---

3.7 The update of the application code . . . . .	61
<b>4 Conclusions</b>	<b>64</b>
<b>A Vitis AI flow scripts</b>	<b>66</b>
A.1 Test script . . . . .	66
A.2 TFRecord creation script . . . . .	69
A.3 Dataset Utility functions script . . . . .	74
A.4 Finetuning and quantization script . . . . .	78
A.5 Compilation script . . . . .	81
<b>B Application code</b>	<b>84</b>
<b>C Vitis DPU configuration file</b>	<b>88</b>
<b>D Pyopencl host application code</b>	<b>93</b>
<b>Bibliography</b>	<b>94</b>

# List of Figures

1.1	Model of a biological neuron [1]	2
1.2	Model of an artificial neuron	3
1.3	Organization of a neural network [7]	4
1.4	Supervised learning [1]	5
1.5	Internal diagram of an FPGA [6]	6
1.6	Example of an half adder implemented with a LUT.	6
2.1	DPU architecture [11]	8
2.2	DSP additional resources [11]	9
2.3	Configuration window inside the VIVADO software [11]	11
2.4	DPU IP with all its interfaces [11]	13
2.5	Vitis AI environment inside an Ubuntu terminal	14
2.6	Vitis AI Quantizer [14]	15
2.7	Vitis AI Compiler [14]	16
2.8	XIR structure [14]	17
2.9	Vitis AI Profiler [14]	18
3.1	Zynq UltraScale+ MPSoC [17]	20
3.2	Terminal output during the test procedure	23
3.3	Terminal view at the end of the quantization process	24
3.4	Terminal output after the fast finetuning method	25
3.5	View of the terminal during the QAT	26
3.6	Example of the whole pre-processing procedure on an image: the image is firstly read and then pre-processed	28
3.7	Diagram of a Vitis Platform development and usage	32
3.8	Zynq UltraScale+ MPSoC Block Automation effects	33
3.9	Block Diagram after the <i>Run Connection Automation</i> was run	35

## LIST OF FIGURES

---

3.10	Vivado menu for managing the communication interfaces between PS and PL . . . . .	36
3.11	Block Diagram at the end of the development process . . . . .	36
3.12	View of the PetaLinux terminal showing the <i>DTG</i> menu . . . . .	39
3.13	View of the PetaLinux terminal showing the <i>user packages</i> menu . . .	40
3.14	View of the PetaLinux terminal showing the <i>Image Features</i> menu . .	41
3.15	View of the PetaLinux terminal showing the <i>Kernel bootargs</i> menu . .	42
3.16	Tree representation of the platform output folder . . . . .	43
3.17	View of the Vitis IDE hardware linker . . . . .	45
3.18	View on the v++ option dialog box . . . . .	46
3.19	Vitis Analyzer view of the summary created at the end of the building process . . . . .	47
3.20	Directory tree of the folder loaded on the board . . . . .	48
3.21	Outcome of the python application when the execution is finished . .	48
3.22	Vitis Analyzer views of the data collected during the straight forward profiling procedure . . . . .	49
3.23	Vitis Analyzer views of the data collected during the straight forward profiling procedure with an application that uses two different DPU cores . . . . .	50
3.24	Vitis Analyzer view of the data collected during the profiling of the application with the python decorator applied to the functions . . . .	51
3.25	Schema of the component which converts the input integer ranging from 0 to 255 into the output floating-point ranging from -1 to 1 . . .	55
3.26	Schema of the AXI4 master interface provided by Capgemini engineering . . . . .	56
3.27	High level schema of the kernel logic . . . . .	58
3.28	View of the <i>Addressing and Memory</i> tab of the Vivado software during the IP packaging procedure . . . . .	59
3.29	Vitis Analyzer view of the summary created at the end of the building process of the project that includes the custom RTL kernel . . . . .	60
3.30	View of the terminal that shows the error that is encountered during the execution of the pyopencl code . . . . .	63

# List of Tables

3.1	MASATI dataset [4]	22
3.2	opencv-python library methods used in the pre-processing	29
3.3	XIR and VART libraries methods used in the managing of the DPU execution	31
3.4	Memport and SP Tag values for the slave interfaces	37
3.5	v++ compiler requirements for the RTL kernel developing	52
3.6	Registers inserted in the <i>Addressing and Memory</i> panel with their name, offset and size	59
3.7	Pyopencl methods used during the development of the RTL kernel host application	62

# INTRODUCTION

---

## 1.1 The basis of Artificial Intelligence

Artificial intelligence (AI) is a scientific field that has been evolving since 1931<sup>1</sup>, but in the last decades the studies on this topic have exponentially intensified. With the arrival of machines with a higher computational power, it has been possible to exploit more efficiently the potential of this kind of technology that can be applied to many different fields of use, allowing to increment the operational capability of algorithms.

In order to better understand the concept of artificial intelligence, it is a good choice to start from one of its definitions. According to the European Commission, AI “refers to systems that display intelligent behavior by analysing their environment and taking actions - with some degree of autonomy - to achieve specific goals” [2]. The emphasis is put on the concept of “intelligent behavior”, but there is not a real explanation on how a machine can achieve the complex task of being intelligent. It is therefore necessary to go a little bit deeper, analyzing at first the functioning of the human brain, which can be considered reference model for AI development, and then how it has been translated into something that can be executed by a machine.

The human brain is a very complex structure composed by billions of highly specialized cells known as neurons. Each neuron is made of three components: the body of the cell, called soma, and two types of filaments, called dendrites and axons (figure 1.1). The soma is the core of the cell, it is in charge of processing the stimuli received through the dendrites, which acts as inputs, generating new stimuli that are sent to nearby neurons through the axons, which can be considered the outputs. It is very important to inspect how the output is created: if the value of the input stimuli

---

<sup>1</sup>year in which Kurt Gödel built the foundations of Theoretical Computer Science and A.I. [10]

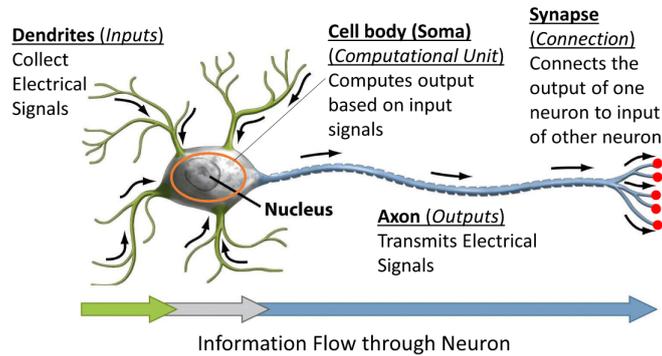


Figure 1.1: Model of a biological neuron [1]

is higher than a given threshold the output pulse is generated with full strength, otherwise the output pulse is null (i.e. the neuron results as it is not active). This particular kind of response is also known as all-or-none response.

From a mathematical point of view, the behavior of the neuron can be synthesized as follow:

$$\text{neuron output} = \begin{cases} 1 & \text{when } \sum x[n] \cdot w[n] + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1.1)$$

where the summation takes into account every input of the neuron to which is added the bias value ( $b$ ), that in this case can be considered as the activation threshold of the neuron itself.

The mathematical equation 1.1 can be easily implemented in software or in hardware with a sequence of simpler operations. The operator that can perform this task is called perceptron, or artificial neuron, and it is the essential block of artificial intelligence. In particular the perceptron requires three steps to obtain the wanted behavior:

1. a weighted sum of all the inputs of the artificial neuron (which requires a multiplication for evaluating the weight of each input and then a sum);
2. the addition of the bias value to the result of the previous step;
3. the application of a non-linear function to the result of the bias addition to emulate the activation behavior of the biological neuron.

A schematic representation of the perceptron steps is shown in figure 1.2

Since a single perceptron models a biological neuron, it is necessary to interconnect many different of them in order to emulate the human brain. This kind of

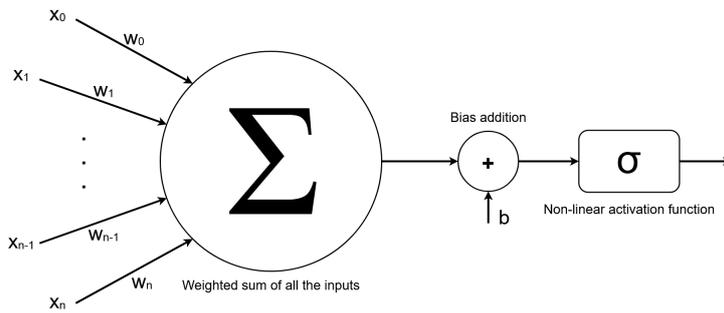


Figure 1.2: Model of an artificial neuron

structure is known as neural network and can be seen as a directed graph, where the nodes are the artificial neurons and the edges indicates which perceptrons are connected together. This graphs are usually organized in layers, that can be of three main types:

- **Input layer:** it corresponds to the set of perceptrons that takes the data from the external world, there is only one input layer in each neural network;
- **Output layer:** it indicates the group of artificial neurons whose output is directly given to the external world, thus representing the output of the whole neural network. Once again there is only one output layer for each neural network;
- **Hidden layer:** set of perceptrons that takes as input the values elaborated by the input layer, or by another hidden layer, and gives their own output either to another hidden layer or to the output layer. There could be more than one hidden layer in each neural network.

The structure that has just been introduced is very versatile since can be used with slightly modifications (such as the reorganization of hidden layers or the change of the interconnections between layers) for many different type of applications. This particular characteristic is given by the architecture of the artificial neuron: as it has been seen in the equation 1.1, the output of each perceptron depends on the values of the weights associated to each input and on the bias value, therefore if those numbers are changed the behavior of the whole artificial neuron is different. The process of tuning the values of each perceptron parameters in a neural network is called training and it can be done in an automatic way thanks to the introduction of machine learning.

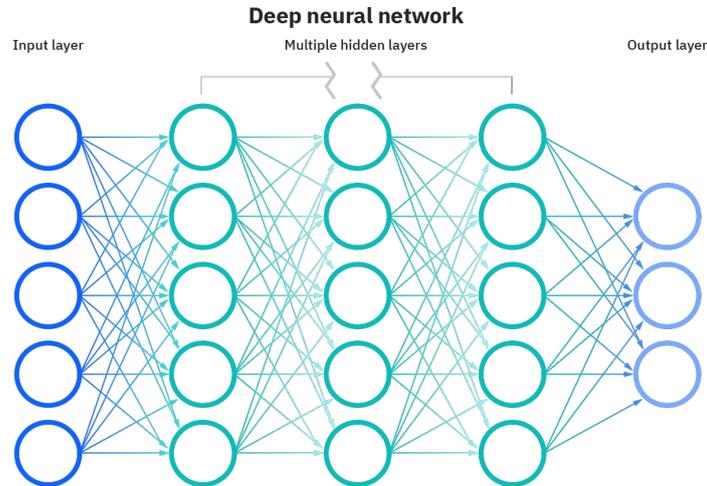


Figure 1.3: Organization of a neural network [7]

In order to clarify the training procedure it is useful to make an example. Assuming to have an image classifier (which is an AI that is able to recognize the subject of an image giving as output the probability that it belongs to one of the classes that the artificial intelligence knows), it is necessary to provide a large amount of data that has been previously labelled, i.e. it already contains the information about the right class the image should belong to. This set of data is then fed as input to the neural network that starts to make some predictions according to its actual parameter. The output is used to calculate a loss value by evaluating the difference between the actual guess and the expected output (which is indicated in the label), obtaining a number that indicates how far the prediction is from the correct result. The loss value is then used to modify the weight of the perceptrons inputs inside the neural network by using an algorithm that, starting from the outputs, tracks every artificial neuron that contributed to the wrong result and adjusts its parameters subtracting the loss value. This training system is known as supervised learning.

## 1.2 The importance of FPGA programming

The training process has been one of the key point that allowed to exploit the power and the potentiality of artificial intelligence thanks to the possibility of obtaining a customized neural network that could fulfill the requirements of many different applications. The opportunity of achieving a wide customization with a relatively

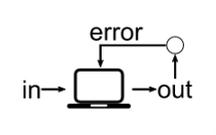
FEATURES	TASKS	APPLICATIONS
<ul style="list-style-type: none"> <li>• Labeled data</li> <li>• Direct feedback</li> </ul> 	<ul style="list-style-type: none"> <li>• Classification            </li> <li>• Regression            </li> </ul>	<ul style="list-style-type: none"> <li>• Image classification</li> <li>• Diagnostics</li> <li>• Market forecasting</li> <li>• Weather forecasting</li> </ul>

Figure 1.4: Supervised learning [1]

low cost granted to the AI a prominence role in industries that are now starting to intensely use it.

For some applications the artificial intelligence that can satisfy the required specifications can be very complex in terms of number of perceptrons, interconnections and layers. In this cases it can be very complicated to design a project to be run in software, i.e. that can be computed by a processor, without downgrading the performances. To solve the problem a good idea could be to move from a software implementation into a hardware one. With hardware the performances, in terms of timing, are usually increased, but the design of a hardware component could be very expensive for what concerns design time and costs. What is usually done to allow a faster and less expensive production is using an FPGA (Field Programmable Gate Array), which is a particular logic device that can be programmed in order to perform any custom logic function (as long as there is enough space to map it). An FPGA is composed by two main blocks: the Logic Element (LE), or Configurable Logic Block (CLB), and the switching block (figure 1.5). The former is composed by a Look-Up Table (LUT) and by a register. The LUT can be considered as a small memory where the stored data represents the output of the logic function which is addressed by the input of the function itself, thus allowing to create any combinational circuit (an example of a half adder is visible at figure 1.6). The register is added to the LE in order to be able to create sequential circuit by storing the purely combinational circuit provided by the LUT. The switching block is an element made of some transistors that are driven by 1-bit RAMs. In this way it is possible to configure how the various LEs are interconnected by simply writing a 0 or a 1 in the RAM memories. In fact, when a 1 is stored, the corresponding transistors acts like a closed switch enabling the interconnection, while if a 0 is stored the transistor can be considered as an open circuit.

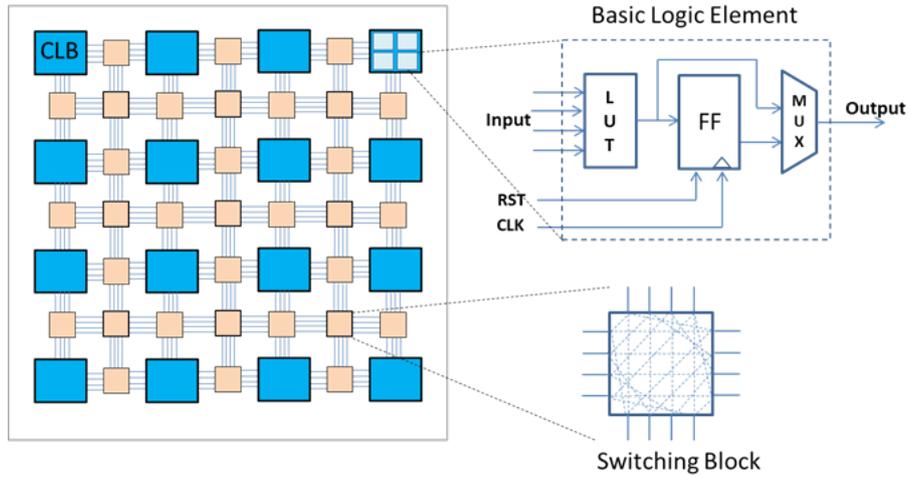


Figure 1.5: Internal diagram of an FPGA [6]

The potential of FPGA has been exploited also in the artificial intelligence field. Xilinx, one of the leading companies for what concerns FPGA production, designed a highly specialized hardware component that is able to perform in the most optimized way possible the most common operations required in neural networks. Since the component is general purpose, i.e. it is able to run many different AI models without changing its architecture, a compiler is necessary to transform the operations required by the software model into instructions that can be run by the component itself. In the next chapters an overview of the development flow necessary to exploit the Xilinx component is provided, as well as the step necessary to increase the performances of the obtained application.

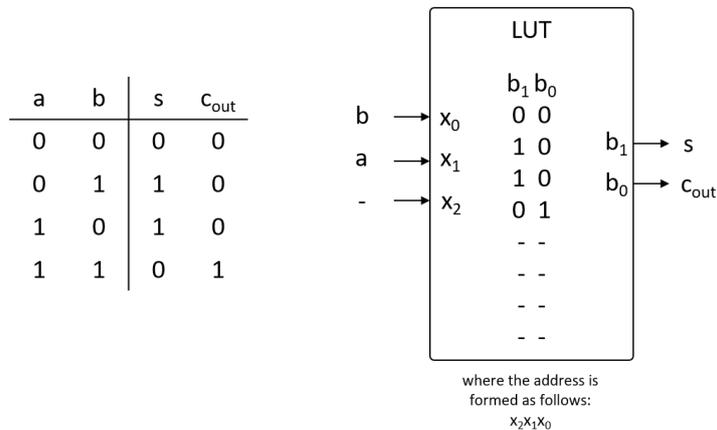


Figure 1.6: Example of an half adder implemented with a LUT.

## VITIS AI OVERVIEW

---

With the expansion of the artificial intelligence field, Xilinx decided to design an easy development flow that allows to accelerate the inference, i.e. the process of using and deploying an already trained neural network, of AI on its own hardware platforms. The main idea behind the flow is the hardware acceleration, which is a technique that can be used to increase the performances of a system by moving a function from a software implementation into a hardware one. In order to apply this approach to a neural network, Xilinx created a specialized hardware core that is capable of performing in an optimized way most of the common operations of convolutional neural network. Due to its high specialization, the hardware core can only execute instruction coming from an exclusive instruction set. For this reason it was necessary to create some tools that, combined with each others, allow to transform an input neural network into the series of instructions that emulate its behavior. In addition to these two elements, a run-time application was needed to manage the communication of the hardware core with the software environment. To simplify the task of creating such application, Xilinx developed some libraries, available in two different coding languages. The combination of all these components forms a development environment, which takes the name of Vitis AI, to which the possibility to use pre-trained artificial intelligence models to further accelerate the creation of a new application by avoiding to perform the training of the neural network was added.

### 2.1 The hardware core

The hardware core of the development environment is called Deep learning Processing Unit (shortened with the acronym DPU). It is a programmable engine that

is optimized for convolutional neural networks, which are particular AI networks that are well suited for tasks such as object detection and recognition since they allow to reduce the number of interconnections between perceptrons. It can be implemented in the programmable logic (PL), i.e. the hardware environment that is generated inside the FPGA, of different Xilinx boards and it is provided with some user-configurable options. The possibility to configure some parameters of its architecture allows to fulfill the performance requirements of many different applications reducing as much as possible the space occupied inside the FPGA, thus reducing the cost.

### 2.1.1 DPU architecture

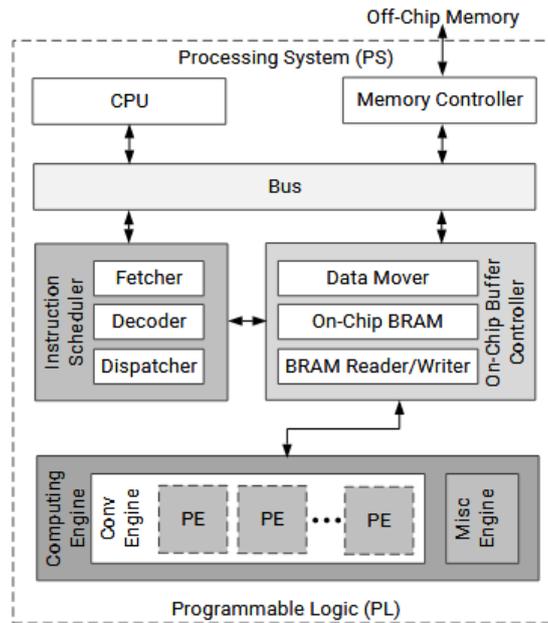


Figure 2.1: DPU architecture [11]

The detailed hardware architecture of a DPU, in particular the DPUCZDX8G which is the one optimized for the *Zynq Ultrascale+ MPSoCs*<sup>1</sup>, is visible at figure 2.1. It is composed by four main components:

- **Instruction scheduler:** it is the module that is in charge of managing the execution of the instructions, which belong to a specialized set that can be

<sup>1</sup>type of FPGA present on the ZCU102, board that has been used for this project

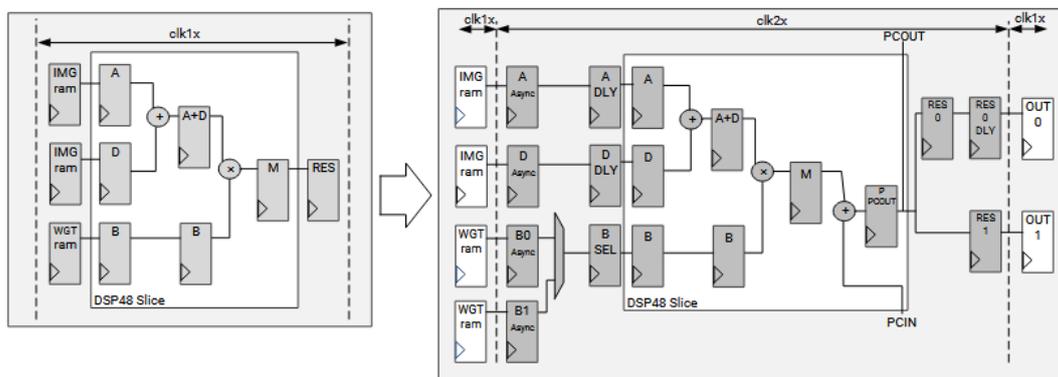


Figure 2.2: DSP additional resources [11]

used to execute in the most optimized way possible all the common operations needed in convolutional neural networks. The instructions are read from an off-chip memory and, after the decoding process, are scheduled to be run by one of the processing engine that form the computing engine;

- **On-Chip buffer controller:** component that handles the operations performed on the on-chip memory. This memory has been added to increase the performances of the whole DPU by storing the input data, the intermediate results and the outputs, thus avoiding to wait for the communication with the off-chip memory;
- **Computing engine:** it is the execution core of the DPU. It is composed by an array of processing engines (PE) which are designed with a deep pipelined architecture to increase the throughput as much as possible.
- **AXI interfaces:** communication interfaces that are necessary to exchange data through the bus with the off-chip memory and the processor, which will run an application that controls the execution of the DPU.

In order to further improve the performances, a DSP (Digital Signal Processing) Double Data Rate (DDR) technique is used. This addition increases the number of used resources, as can be seen in figure 2.2, and needs an additional clock.

### 2.1.2 DPU configuration

As it was said, the DPU is a configurable core whose configuration options allows the user to setup different parameters that can manage the usage of resources such

as DSP slices, LUT, block RAM and UltraRAM. The main possible options are:

- **Number of cores:** a single DPU instance can include internally up to 4 different cores. The more the number of the cores the more efficient the implementation is, but the amount of used resources is consequently increased.
- **Architecture:** the DPU IP can be implemented with different types of architecture that differ one from the other in the level of parallelism. The available designs are the following: B512, B800, B1024, B1152, B1600, B2304, B3136 and B4096. The numeric value that is used in the names indicates the number of operations that can be performed per clock cycle and it is directly correlated with the level of parallelism. In particular, the higher the parallelism, the higher the number of executable operations (as a consequence also the number of necessary resources increase with the increment of the performances).
- **RAM usage:** the on-chip memory that is used to increase the performances is a RAM memory that stores weights, bias and intermediate results. When instantiating the DPU module, it is possible to choose the amount of RAM that will be reserved by selecting between the *High RAM usage* and the *Low RAM usage* options.
- **Channel augmentation:** technique that exploits the fact that in some models the number of input channels is lower than the channel input parallelism of the architecture. When this condition is verified, the performances can be increased by enabling this option at the cost of extra resources usage.
- **Depth-wise Convolution:** with standard convolution each input channel needs to perform some operations with one specific kernel and then all the results of all the channels are combined to obtain the final result. By enabling the depth-wise convolution, the convolution operation is split in two parts: the depth-wise one and the point-wise one. The former allows to process all the input channels in parallel, while the second performs a convolution with a 1x1 kernel combining the results of the previous step to obtain the final value. This kind of approach allows to increase the performances since the parallelism of the depth-wise convolution is half of the pixel parallelism, thus more than one activation map is evaluated per clock cycle.

- **Average Pool:** the average pool is the operation in which a matrix is reduced in size by making an average of its element. The starting matrix is divided into smaller squares (the supported sizes of the squares goes from 2x2 to 8x8) and the average value is calculated by summing all the elements of the smaller matrix that are then divided by the number of summed elements. This kind of operation allows to reduce the size of the input data, thus increasing the performances.
- **ReLU type:** the ReLU (Rectified Linear Unit) is the operation that transforms the negative values into zeros by keeping unchanged the positive ones. It is possible to choose between: *ReLU*, *ReLU6* and *LeakyReLU*.
- **Softmax:** the softmax is a logistic regression function that allows to transform any obtained value into a probability in range  $0 \div 1$ . By enabling this option the hardware to perform this operation is added, thus avoiding to use the processor to perform this task.

By combining the above features it is possible to customize the DPU implementation reaching the best trade-off between resources and performances for each application. Once the configuration process is completed, it is necessary to save the enabled options in order to let the compiler to correctly select the instructions that will be executed. This step is automatically performed after the synthesis of the DPU and all the data is stored in a configuration file, called *arch.json*, that will be later given to the tools that compose the Vitis AI environment.

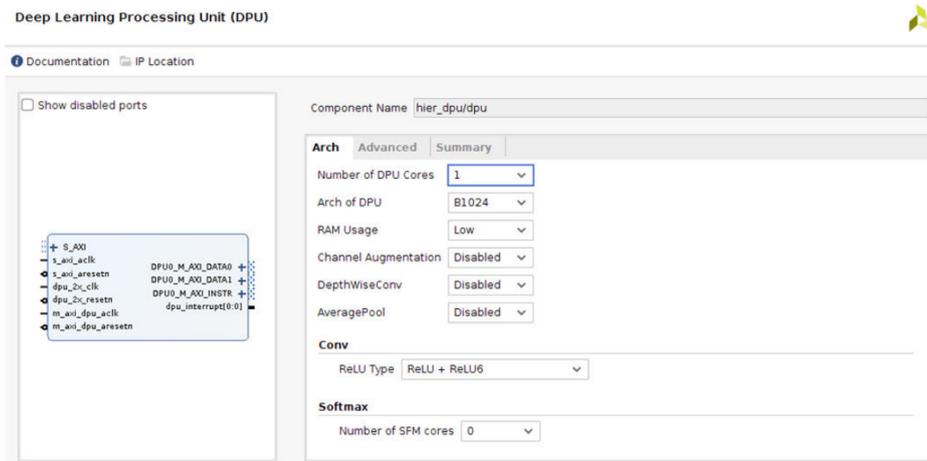


Figure 2.3: Configuration window inside the VIVADO software [11]

### 2.1.3 DPU integration

Now that the structure and the functionalities provided by the DPU core have been explained, it is important to understand how it is possible to integrate it inside a project. Vitis AI gives two possible solutions. The first one consists in creating a project with VIVADO (which is the Xilinx software that allows to create a hardware system that can then be synthesized and loaded into the FPGA), while the second option is using the Vitis IDE (which is a comprehensive tool that can be used to create an application that can include both a software and a hardware part).

In both the cases, it is important to take into account that the DPU needs to communicate with the processing system (PS), i.e. the software environment that runs on the processor, to receive the control signals, the instructions to be executed and to exchange data. To perform such a task, some communication interfaces must be added and a communication protocol must be selected. In the case of the DPU, the adopted protocol is the AXI (Advanced eXtensible Interface): a burst-based protocol that divides the communication in five different and independent channels (the read address, the read data, the write address, the write data and the write response) each one characterized by a handshake procedure. For what concerns the interfaces, the ones present in the architecture are the following:

- an AXI4-lite<sup>2</sup> slave used to receive the control data and the base addresses at which the instructions are stored in the memory. All the received data is stored inside some internal register so that it will be available at run-time;
- an AXI4 master to access the instruction memory according to the formerly received base addresses and read the instructions that must be performed;
- two AXI4 masters that allows to read the data that must be processed during execution. The presence of two different interfaces allows to read larger data, thus exploiting as much as possible the DPU parallelism;
- an optional AXI4 master that is automatically inserted if the softmax option has been enabled during the configuration process to handle the softmax data.

In addition to the communication interfaces, there are some other signals that must be taken into account when integrating the DPU: some interrupt signals, two

---

<sup>2</sup>variation of the standard AXI that simplifies the communication protocol for easier communications

clocks and two resets. The number of interrupt signals depends on the number of cores inside the DPU, in fact each core has its own interrupt signal. For what concerns the two clocks, one of them is necessary to manage the communication protocol, while the other one is used for the internal logic and it must have a frequency that is twice the one of the other clock. The two resets follows the same structure of the clocks: one is used to handle the communication, while the other one is used by the internal logic.

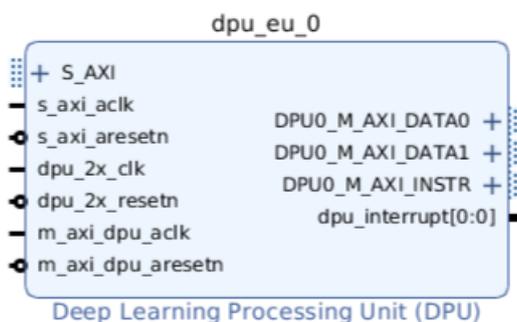
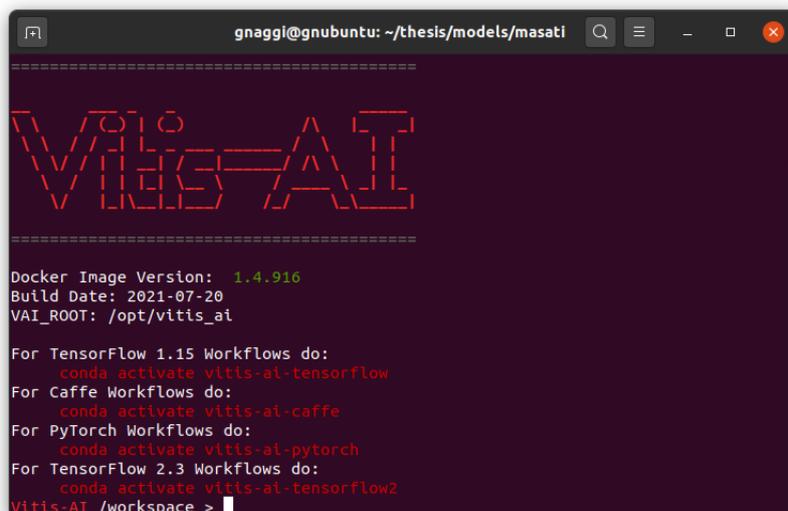


Figure 2.4: DPU IP with all its interfaces [11]

## 2.2 The tools

Vitis AI development environment makes available different tools that, when combined together, allow to create the sequence of instruction that model the input neural network. Before analyzing each one of the main tools, it is worth mentioning that Xilinx developer included inside the development environment additional feature to increase the flexibility and the usability of it. As an example, it is possible to train a new AI model written with the mainstream frameworks in the context of artificial intelligence implementation (such as TensorFlow, Caffe or PyTorch), which allows to start a project from its very basis. In the next sections all the main tools usage is explained in order to understand which is the flow that must be followed during the design. In order to be able to use the available tools it is necessary to underline that all the Vitis AI suite is available as a docker image, i.e. a standard unit of software that packages up code and all its dependencies so that applications can run easily and reliably in different computing environments [3], which can be installed only on a Linux-based operating system.



```
gnaggi@gnubuntu: ~/thesis/models/masati
=====
VITIS-AI
=====
Docker Image Version: 1.4.916
Build Date: 2021-07-20
VAI_ROOT: /opt/vitis_ai

For TensorFlow 1.15 Workflows do:
  conda activate vitis-ai-tensorflow
For Caffe Workflows do:
  conda activate vitis-ai-caffe
For PyTorch Workflows do:
  conda activate vitis-ai-pytorch
For TensorFlow 2.3 Workflows do:
  conda activate vitis-ai-tensorflow2
Vitis-AI /workspace >
```

Figure 2.5: Vitis AI environment inside an Ubuntu terminal

### 2.2.1 Vitis AI Quantizer

Artificial intelligence models usually stores all the weights and biases of their neural networks with a 32-bit floating-point representation, thus the inference of these type of models is very computing intensive and it requires a high memory bandwidth if a low-latency and high-throughput is desired. When designing the DPU, Xilinx developer discovered that by substituting the 32-bit fixed-point numbers with 8-bit integer values brings to an important boost in the performances with the cost of a little degradation in the performances. With this idea on mind, the Vitis AI quantizer was created. Its main goal is the automatic conversion of all the parameters of an input artificial intelligence trained model into 8-bit integer constants and, as output, it produces a new model file which results to be less heavy in terms of memory space (thanks to the previous conversion).

Vitis AI quantizer provides three different methods that can be used to correctly quantize the input neural network:

- **Post training quantization:** this method is the less expensive in terms of running time, which depends also on the complexity of the model. In order to process the input model correctly, it requires only a small set of unlabeled images to analyze the distribution of activation.
- **Quantization aware training:** with some networks, such as MobileNet, the

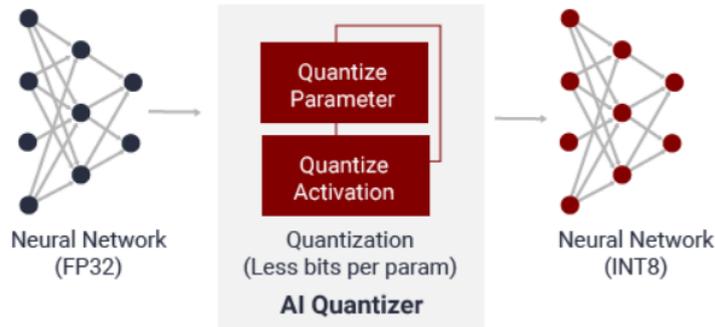


Figure 2.6: Vitis AI Quantizer [14]

previous method brings to a large accuracy loss. In this case it is possible to run the quantization aware training that requires a much higher running time (from minutes to hours), because it performs also part of the training process taking into account that the result will be quantized (thus it brings to different results with respect to a normal training procedure). In order to be able to use this method, the original training dataset must be provided.

- **Fast finetuning:** quantization algorithm have been evolving and the last ones are able to perform complex operations in a much efficient way. It is the case of the fast finetuning that allows to achieve a better quantization result with a slightly slower run time, but still requiring a small set of unlabeled data.

For all the methodologies listed above, the quantizer should run several iteration of inference to improve the accuracy of the quantized model, which, one the entire process is over, will be transformed into a DPU deployable model.

### 2.2.2 Vitis AI Compiler

Once the input model has been transformed into a deployable one, it must be mapped to a sequence of the highly optimized instructions that the DPU is able to execute. This operation is performed by the Vitis AI compiler in three subsequent steps: the parsing, the optimization and the code generation. Before going more into details, it is fundamental to introduce the intermediate representation that is used by the compiler, known as XIR (Xilinx Intermediate Representation).

XIR is a graph-based representation of the AI algorithms which is designed for simplifying the compilation and the deployment of the DPU on the FPGA. It is

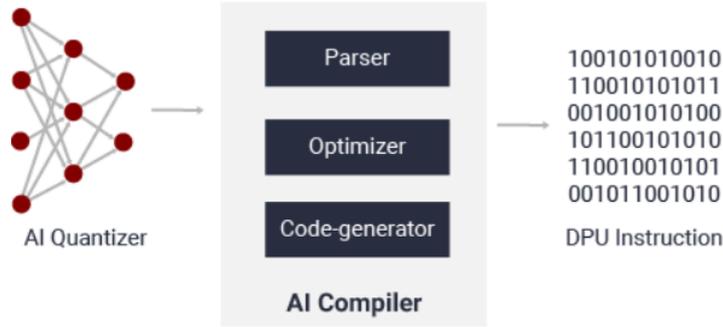


Figure 2.7: Vitis AI Compiler [14]

composed by four main libraries:

- **Graph:** it is the core component of XIR. It is formed by the OPs as vertexes and the relation between producer-consumer as the edges;
- **OP:** it correspond to the operator library, it defines a set of operator that are used to cover the most popular deep learning frameworks and all the built-in DPU operators. With the introduction of this library the differences within the mainstream frameworks are removed and a unified representation is provided;
- **Tensor:** unlike other frameworks tensor definition, in XIR it is only a description of the data block it represent and carries the information about the data type and the shape (no real data is associated to a tensor);
- **Subgraph:** it is the library that provides a tree-like hierarchy which divides the set of OPs into several sets which do not overlap.

When compiling a quantized model, the first step that the compiler performs is the parsing of the topology of the input model. The data retrieved during the parsing procedure is used to create the XIR graph of the input model on which different modifications are made. First of all there is an optimization phase, where some computation nodes are fused or the scheduling is modified according to the DPU parallelism or according to the data reuse. In this way it is possible to increase the performances without changing the behavior of the algorithm. The optimized graph is then divided in subgraphs according to the DPU capability of executing the operations and additional architecture-aware (i.e. dependent on the DPU architecture, which must then be known by the compiler) optimization are performed. The last step is the generation of the instruction stream for the DPU. This process

is performed only on the subgraph that contains all the operations that can be executed by the DPU itself, while all the other subgraphs are mapped on instructions that will be executed by the processor. As a final step, the whole graph is serialized in a *xmodel* file that will be used during the execution of the application (exploiting one of the XIR properties, which is the possibility to move flawlessly from the file format, the *xmodel* file, to the in-memory format, the XIR graph and vice versa).

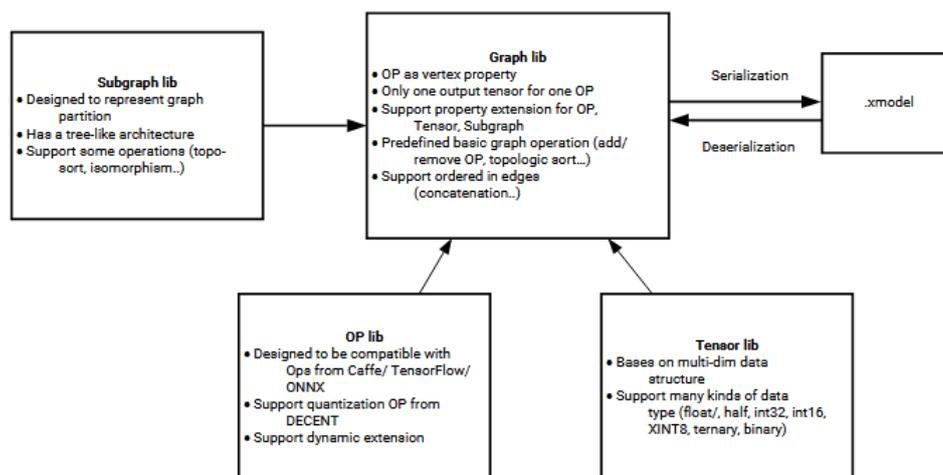


Figure 2.8: XIR structure [14]

### 2.2.3 Vitis AI Optimizer

Xilinx developers included the possibility to increase the performances of an application by modifying directly the initial neural network. This kind of operation is performed by a tool called Vitis AI optimizer, with a procedure called pruning. It consists in removing all the redundant connections of the neural network, thus reducing the overall number of operations involved. This type of optimization does not exclude the ones performed during quantization and compilation, thus it is a good choice to run the optimizer before the compiler and the quantizer. This tool is available only under a license.

### 2.2.4 Vitis AI Profiler

Before an application under development can be considered successfully concluded, it is necessary to verify if all the starting requirements have been met. In some cases

it can be very complex to complete this procedure, thus Xilinx inserted inside its development environment a set of tools that allows to facilitate this task. The whole set takes the name of Vitis AI profiler and it allows to easily gather performances information while the application is being executed and then to visualize them in a clear way.

The Vitis AI Profiler is an application level tool based on the VART (Vitis AI RunTime) library, which provides unified high-level run-time APIs to submit and collect asynchronously data to and from accelerators. In addition to the data coming from the accelerators, the profiler gathers also all the information about the performances of the CPU, where all the operations that can not be run by the DPU are executed. After having collected all the necessary data, it is necessary to display it in an user-friendly way. For this reason, all the retrieved information are stored inside one or more *csv* files that can than be read by the Vitis Analyzer software which provides a GUI to display them.

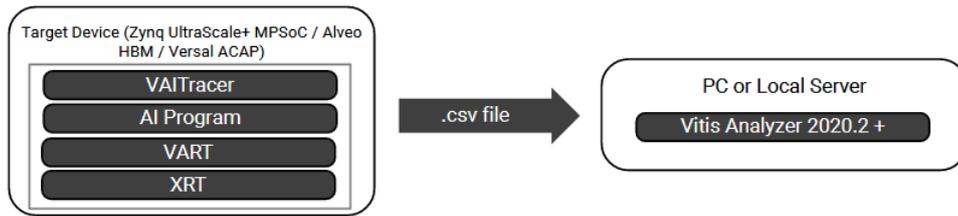


Figure 2.9: Vitis AI Profiler [14]

In order to correctly run the Vitis AI Profiler an operative system is necessary. In Xilinx applications the OS is usually created by using the PetaLinux Tool, an embedded Linux SDK (Software Development Kit) targeting FPGA-based systems. The generation of the operating system is performed through the Yocto SDK, which is integrated in the PetaLinux tool. With this kind of creation process it is very easy to modify the Linux version, creating a deep customized Linux distribution. One of the requirements that must be met to be able to run the Vitis AI Profiler is the enabling of some options inside the PetaLinux menu. In particular, inside the kernel configuration (accessible with the command `petalinux-config -c kernel` inside a PetaLinux project) the following must be on:

- General architecture-dependent options  $\rightarrow$  Kprobes
- Kernel hacking  $\rightarrow$  Tracers

- Kernel hacking → Tracers → Kernel Function Tracer
- Kernel hacking → Tracers → Enable kprobes-based dynamic events
- Kernel hacking → Tracers → Enable uprobes-based dynamic events

After having build the PetaLinux project and loaded the operating system inside the board, it is possible to start the profiling by inserting an option when the execution of the operation code is issued. In order to clarify the correct usage of the tool, the following are the two possibilities (the tool is available only for C++ and Python coded applications):

- **C++** : *vaitrace <path to the executable C++ file>*
- **Python** : *python3 -m vaitrace\_py <path to the Python code>*

## THE IMAGE CLASSIFIER

---

As it has been seen in the previous chapter, Vitis AI development environment introduces many different interesting features when dealing with artificial intelligence based projects. Since one of the most critical processes is putting into practice what the theory describes, it was necessary to create an example project that exploits as much as possible all the most important features of the Xilinx environment.

The project described in this thesis work implements a simple image classifier that receives as inputs some images stored in the memory of the running device, which is a ZCU102 board made available by Capgemini engineering. The ZCU102 is a general purpose evaluation board that mounts a Zynq UltraScale+ MPSoC (MultiProcessor System on Chip), which is a device that combines a 64-bit ARM processor, the processing system (PS) seen in the previous chapter, with an FPGA, the programmable logic (PL) cited in the previous chapter.

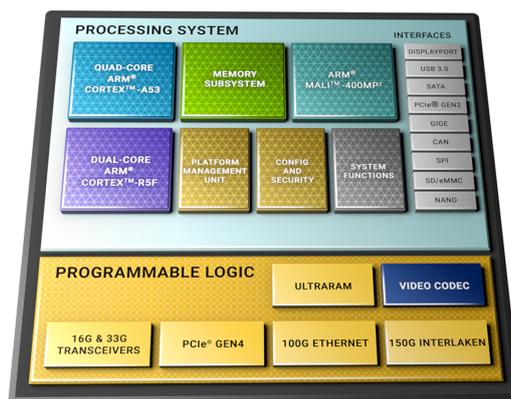


Figure 3.1: Zynq UltraScale+ MPSoC [17]

In addition to that, the board includes features such as a high speed DDR4 SODIMM, a multi-gigabit per second serial transceivers, a variety of peripherals and many others that can be seen in the board user guide [16].

In this chapter all the steps that were followed to reach the completion of the project are explained in details in order to allow to replicate the obtained results. Before starting the analysis of the project, it is important to notice that all the choices was made in order to enhance the possibility to use one or more tool features rather than developing a complex or optimized application.

### 3.1 The AI model

When dealing with an artificial intelligence based project, the first step is usually the development of an AI model capable of solving the requested problem. In the specific case of the project designed during this thesis work, there were not any problem to be solved, thus the AI model was chosen among the ones that were not very complex and already available.

Capgemini engineering provided the model of an image classifier based on the MobileNetV2 architecture, which is a neural network designed to efficiently run on mobile devices, and it was written with the TensorFlow2 framework (the corresponding programming language is python). The artificial intelligence was already trained on the MASATI (MAritime SATellite Imaginery) dataset, which provides several satellite images of marine environments. In particular there are seven possible classes inside the MASATI dataset, each one provided with a large number of images that can be used during the training process, as it is visible in table 3.1. All the images of the dataset are labeled with the correct category they belongs to through the name of the image, which is formed by an initial letter and a four digit number. The letter indicates in a unambiguous way the class, while the number is used to differentiate the images one over the other. The provided artificial intelligence model was trained only with the last three classes of the dataset, thus the neural network is able to categorize the input images only between land, sea or coast.

Before proceeding with the development flow seen in the previous chapter, an initial evaluation of the model efficiency was made. In order to perform this task, a Python script was created implementing an algorithm that at first loads the trained model through the TensorFlow2 APIs, then feeds the artificial intelligence with all the images inside an input folder. To increase the robustness of the test all the images are loaded into some lists and shuffled. In addition to that, the script takes

The image classifier

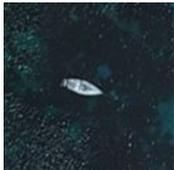
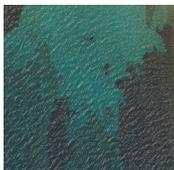
Class	Description	# of samples	Example image
Ship	Ship inside a portion of sea without coast	1027	
Detail	Zoom on a ship, showing some of its details	1789	
Multi	Multiple ships on water	304	
Coast & ship	Ship inside a portion of sea near a shore	1037	
Sea	Portion of water with no land	1022	
Coast	Section of a coast (both water and land are present)	1132	
Land	Portion of land with no water	1078	

Table 3.1: MASATI dataset [4]

the model output and compare it against the label of the input image to verify the correctness and the precision of the model itself. The comparison is made by checking if the first letter of the AI output is equal to the letter indicated in the

name of the image (since for the three class used the letter in the name can be a l, indicating land, a c, standing for coast, or a w, indicating water). For each image, the algorithm displays on the terminal the input image path, the AI guess (with its probability value) and the overall accuracy of the model. These information are colored according to the correctness of the guess, a green print indicates the correct guess while a red one indicates an error. At the end of the test procedure, the given

```

gnaggi@gnubuntu: ~/thesis/models/masati
|-----| 85.7% ETA: 00:00:16
Input: dataset/data/w0525.png
output class: water 0.886151
accuracy: 0.994320

|-----| 85.7% ETA: 00:00:16
Input: dataset/data/c0944.png
output class: coast 0.906442
accuracy: 0.994323

|-----| 85.8% ETA: 00:00:16
Input: dataset/data/c1122.png
output class: water 0.733382
accuracy: 0.993920

|-----| 85.8% ETA: 00:00:16
Input: dataset/data/w0315.png
output class: water 0.864153
accuracy: 0.993922

|-----| 85.8% ETA: 00:00:16
Input: dataset/data/w0944.png
output class: water 0.865881
    
```

Figure 3.2: Terminal output during the test procedure

model showed a very high accuracy since the percentage of rightly classified images is 0.994784. Since the provided model proved to be quite reliable, it was possible to perform all the steps that brings to the deployment of the application.

Basing on the tutorial made available by Xilinx, in particular the one targeting the TensorFlow2 development flow [13], that was used as starting point to design this thesis project, a good practice is using TFRecords when performing operations that involves large quantity of data in a TensorFlow2 environment. A TFRecord is a simple format for storing a sequence of binary records which allows to reduce the amount of space occupied by the data (without losing information), to increase the speed of I/O operations (can be read in parallel) and allows to read different data files from a single source. In order to follow the indications of the tutorial, the input data was converted into different TFRecords by modifying the script provided in the tutorial itself. Before launching the real conversion process, that is performed with the TensorFlow2 APIs, it is necessary to prepare the data. All the images inside the input directory are divided into the three categories, thanks to the name labelling

system of the dataset, shuffled and then inserted into two different lists. The first list contains all the images that will be used during the testing procedure, while the second all the data that is necessary to train in a reliable way the model. The number of images inside the first list is much smaller than the one in the second list and it corresponds to the 20% of the total images, since the training procedure requires a big amount of data to tune the neural network parameters. At the end of the TFrecords creation, three files was created (*test\_0.tfrecord*, *train\_0.tfrecord* and *train\_1.tfrecord*) which includes all the 3066 images of the dataset. The number of total images is given by the necessity of having the same amount of images in each class in order to correctly train an AI model monitoring its accuracy. For this reason the script that creates the TFrecords takes the class with the lowest amount of data and adapt the other classes to this number (it truncates the exceeding images).

Once all the data was available, the quantization process was the first operation that had to be performed on the model that was provided by Capgemini engineering. Looking at the quantization script provided by the Xilinx tutorial, the process is performed by loading the trained model (also known as float model) and using the *quantize\_model* method present in the Vitis AI APIs for the TensorFlow2 framework. As a first try, the script available on the tutorial was used by simply changing the input model, but some problem arose. When evaluating the output model of the quantization process, the accuracy dropped significantly from the starting 0.994784 to 0.315152 which, of course, is not acceptable. After a long troubleshooting, the

```

gnaggi@gnubuntu: ~/thesis/models/masatl
--batchsize      : 50
--tfrec_dir      : dataset/tfrecords/jpg
--evaluate       : True
-----

[VAI INFO] Start CrossLayerEqualization...
10/10 [=====] - 4s 379ms/step
[VAI INFO] CrossLayerEqualization Done.
[VAI INFO] Start Quantize Calibration...
10/10 [=====] - 14s 1s/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quantize Adjustment...
[VAI INFO] Post-Quantize Adjustment Done.
[VAI INFO] Quantization Finished.
Saved quantized model to models/quantized_photo.h5

-----

Evaluating quantized model..
-----

Quantized model accuracy: 31.5152 %

-----

(vitis-ai-tensorflow2) Vitis-AI /workspace >

```

Figure 3.3: Terminal view at the end of the quantization process

cause of the problem was found in the architecture that was chosen for the model. As it has been said in the previous chapter, there are some AI architectures that are not well suited for the quantization process and one of them is the MobileNet family. To correctly use this type of architecture it is necessary to substitute the quantization with the fine tuning procedure, which partially trains again the floating model taking into account that the internal parameters (weights and biases) will be expressed with 8 bit integer values. Inside the Vitis AI development environment, the finetuning technique is supported with two different methods: the fast finetuning and the quantization aware training (QAT). The former is an automatic procedure that is enabled with an option of the *quantize\_model* APIs (in particular the *include\_fast\_ft* and the *fast\_ft\_epochs*, which must be set to True and to an integer number respectively) that modify the quantization process adding the tuning of the inner parameters, while the second method requires the following steps:

- Instantiate the quantizer with the *quantize\_strategy* option set to *8bit\_tqt*;
- Generate a quantization aware model through the *get\_qat\_model* method of the quantizer class (the function requires two arguments that are *init\_quant*, which must be set to True, and the calibration dataset, portion of the data that can be used as a test during the tuning);
- Compile and train the model that has just been created.

```

g Naggi@gubuntu: ~/thesis/models/masati
ctor_block_15_project
[VAI INFO] Fast Finetuning(49/55): feat_extractor_block_16_expand -> feat_extractor_block_16_expand_relu
[VAI INFO] Fast Finetuning(50/55): feat_extractor_block_16_depthwise -> feat_extractor_block_16_depthwise_relu
[VAI INFO] Fast Finetuning(51/55): feat_extractor_block_16_project -> feat_extractor_block_16_project
[VAI INFO] Fast Finetuning(52/55): feat_extractor_conv_1 -> feat_extractor_output_relu
[VAI INFO] Fast Finetuning(53/55): decision_branch_preds_0 -> activation
[VAI INFO] Fast Finetuning(54/55): decision_branch_preds_1 -> activation_1
[VAI INFO] Fast Finetuning(55/55): preds -> preds
[VAI INFO] Fast Finetuning Done.
[VAI INFO] Quantization Finished.
Saved quantized model to models/quantized_photo.h5

-----
Evaluating quantized model..
-----

Quantized model accuracy: 32.7273 %
-----
(vitis-ai-tensorflow2) Vitis-AI /workspace >

```

Figure 3.4: Terminal output after the fast finetuning method

For the sake of simplicity, the first method was used and it led to a result that was better than the previous one, but still not acceptable since the accuracy was around 0.327273.

As a last attempt, the quantization aware training technique was used and the final outcome was the one that was expected from the theory: 0.973574, a little bit lower than the original one, but still very good, thus it was possible to proceed with the next, and final step, which was the compilation.

```

gnaggi@gubuntu: ~/thesis/models/masati
-----
Loading model
Model correctly loaded
-----

[VAI INFO] Start CrossLayerEqualization...
10/10 [=====] - 4s 391ms/step
[VAI INFO] CrossLayerEqualization Done.
[VAI INFO] Start Generation of Quantize-aware Training Model.
[VAI INFO] Generation of Quantize-aware Training Model Done.

Epoch 00001: LearningRateScheduler reducing learning rate to 0.001.
Epoch 1/100
 1/350 [.....] - ETA: 0s - loss: 1.2680 - accuracy: 0.
 2/350 [.....] - ETA: 5:11 - loss: 0.6503 - accuracy:
 3/350 [.....] - ETA: 6:53 - loss: 0.4342 - accuracy:
 4/350 [.....] - ETA: 7:47 - loss: 0.3258 - accuracy:
 5/350 [.....] - ETA: 8:17 - loss: 0.2606 - accuracy:
 6/350 [.....] - ETA: 8:36 - loss: 0.2172 - accuracy:
 7/350 [.....] - ETA: 8:49 - loss: 1.1112 - accuracy:
 8/350 [.....] - ETA: 8:59 - loss: 1.9386 - accuracy:
0.7050
    
```

Figure 3.5: View of the terminal during the QAT

According to the theory explained in the previous chapter, the compilation process requires all the information about the configuration options used to implement the DPU inside the project. In order to keep the project a little bit simpler, all the default options was selected. Given that, the compilation process was performed using the compilation script available on the tutorial, changing only the source model that was the one coming from the quantization step. The final result of the whole development flow is the *.xmodel* file containing the DPU instructions that will be loaded inside the ZCU102 memory in order to run it on the board.

In order to replicate the flow it is sufficient to start Vitis AI, activate the TensorFlow2 environment (with the command *conda activate vitis-ai-tensorflow2*) and then execute all the scripts, visible in appendix A, in the correct order by issuing them through the *python3* command (the general syntax of the command is the following:

```
python3 -u <path_of_script> <script_parameters>
```

## 3.2 The application code

The *.xmodel* file that was generated with the steps described in the previous section is one of the main component to accelerate the AI inference, but, given that it must run on a board that mounts an operating system, it needs a software application that handles the communication between the processing system and the hardware core. In addition to that, the small firmware should load the DPU instructions, prepare the data that will be fed to the DPU and read the results coming from the DPU. In order to facilitate the design of such algorithm, Xilinx inserted inside the Vitis AI development environment some libraries which are available in two coding languages: C++ and Python. For sake of simplicity the programming language chosen to design the firmware was Python, due to the fact that it is the same language used in the Xilinx tutorials.

Before describing the application code, it is important to see which are the part that compose it at a higher level. There are three main parts that are run in sequence:

- **Input preparation:** images are stored in memory as multidimensional arrays (usually in three dimensions: height, width and color channels) were each element of the arrays correspond to a pixel (usually with a value in range 0 to 255). The AI models that work on images expect as well a multidimensional array of integer as input, but the value of each integer must be in range 0 to 1 or in range -1 to 1 (it depends on the choice made during the training of the model itself). A linear transformation that converts the integer value from the image domain (0 to 255) to the AI one (either 0 to 1 or -1 to 1) is needed and it is known as pre-processing. Together with the pre-processing, some methods to retrieve the initial image must be added (it can come from a communication channel with a camera or read directly from memory, for example);
- **DPU handling:** since the Vitis AI flow is based on the hardware acceleration, it is necessary to introduce some piece of code that will manage the execution of the hardware (by setting some registers inside the DPU is possible to control

its execution). In addition to that, there must be some synchronization process to allow the software code to wait the DPU to finish the data processing;

- **Output registration:** once the DPU has finished its execution, the output data should be read back in order to be used inside the rest of the system.

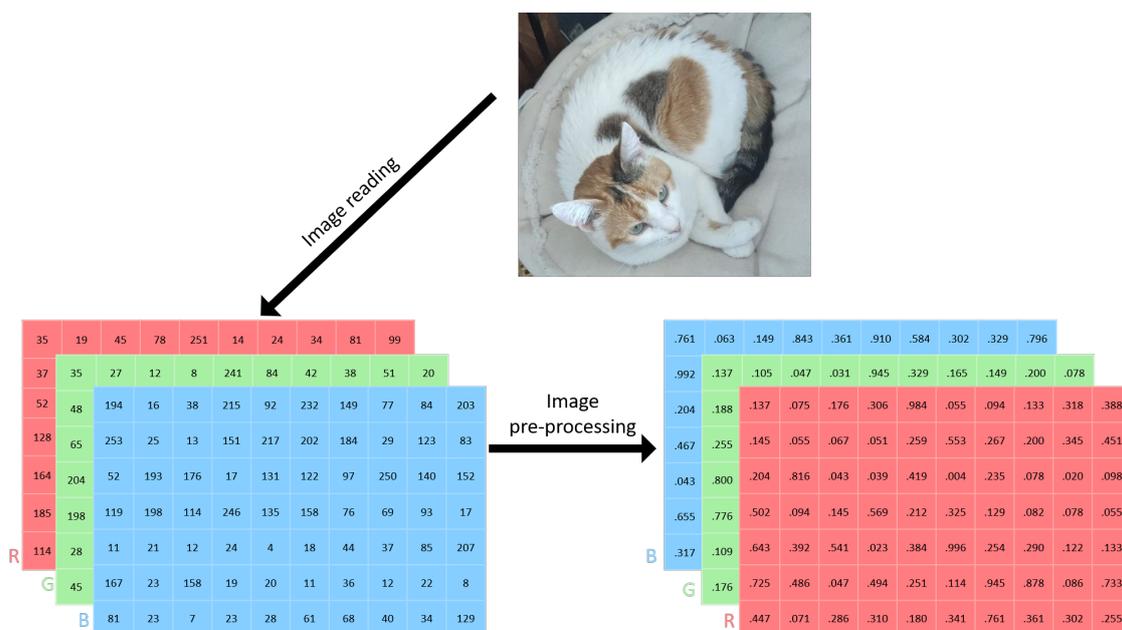


Figure 3.6: Example of the whole pre-processing procedure on an image: the image is firstly read and then pre-processed

The application code developed for this project implements all the steps above, taking the images to be pre-processed directly from the device memory and using the output of the AI model only for verifying the correctness of the guesses.

Starting from the pre-processing procedure, two possible methods were available during the design. The first one consists in pre-processing all the images at once, saving them into the memory and then pass the converted data to the DPU, while the second options is perform the pre-processing on a single image and send it directly to the DPU, starting then again for the next image. The last method was the one used, so that it could have been possible to exploit the presence of two DPU cores and performing the pre-processing while the other core is running. After saying that, the input data preparation was handled with the following steps:

1. Image read: the image is read and stored in memory as a multidimensional matrix using the *opencv-python* library with the *imread* method. The path of the image must be given as argument;

2. Image preparation: images can be stored in memory with different structures, thus it is necessary to reshape the input image into the structure that is expected by the DPU. In this particular case the reading function orders the image channels in the BGR standard (Blue channel first, Green channel second and Red channel last) while the used AI model requires the RGB standard. The channel shifting is done through the *cv2.cvtColor* function, where the second parameter indicates the type of conversion (for this project it is *cv2.COLOR\_BGR2RGB*). In addition it is necessary to match the size of the height and the width of the image, thus the *cv2.resize* function is used.
3. Pre-processing: the model provided by Capgemini engineering is able to process images whose pixel value is in range -1 to 1. The transformation is performed with the simple linear function visible in equation 3.1, which can be directly implemented in software.

$$pixel_{out} = \frac{pixel_{in}}{127.5} - 1 \quad (3.1)$$

For what concerns the DPU managing the Vitis AI libraries APIs were deeply used, in particular the VART, for run-time operations, and the XIR, for the managing of the compiled graph expressed in the Xilinx representation (see section 2.2.2). The first library introduces a software object associated with a DPU core, which is

Synopsys	Description
<b>cv2.imread</b> (image_path)	Returns the multidimensional array containing only the pixels values (all the decoding due to the image formatting is performed during the image opening)
<b>cv2.cvtColor</b> (image,conversion_mode)	Returns the multidimensional array of the image pixels with the color channels reordered according to the conversion_mode value
<b>cv2.resize</b> (image,(h,w))	Returns the multidimensional array of the image pixels resized according to the new height and width indicated by <i>h</i> and <i>w</i> respectively

Table 3.2: opencv-python library methods used in the pre-processing

called *runner*, that includes the main functions that can be used for the DPU execution. In order to correctly create this *runner*, it is necessary to pass as argument the XIR subgraph that must be executed, thus it must be extracted from the *.xmodel* file generated during compilation using the second library available. In addition to that, there is a second parameter that is used to indicate the mode in which the DPU should be run; the value that is usually associated to it is *run* and is passed as a string.

Once all the basic objects have been created, the DPU execution can be started with the runner *execute\_async* method, which asks as parameters an input array, where the data must already be present, and an output array, where the processed data will be stored. If the operating system has to wait for the end of the DPU data processing, there is another method of the runner class that allows to perform such operation. This functions, called *wait*, works in a similarly way to the Unix *waitpid* function: it receives as a parameter the id of the running job, which is the return value of the *execute\_async* method, and waits until the process associated to the given id has finished all its operations. Table 3.3 lists all the methods used inside the code with a little description that should clarify their functionalities.

As a last step, the output must be retrieved and, in the specific case of this application, compared against the real class of the input image. Differently from how pre-processing is performed, the post-processing (i.e. the operations performed on the AI model output) is not performed on a single image at a time, but to all the images at once. For this reason, after the *wait* function described above ends, the index of the output array which contains the higher value, i.e. the class that was considered as most probable for the given input image, is stored inside a list. Once all the images have passed through the DPU processing, the list containing all the classes is iterated and each index is associated to the correspondent class name. At the same time, the input images are scanned in the same order they were scanned during the pre-processing operation to obtain the first letter of the name and check if it matches the first letter of the AI output class. If a match is found a *correct* counter is incremented, otherwise a *wrong* counter is increased. Those two values are used as output of the whole application that will indicate the number of correct guesses, the number of wrong guesses and the accuracy, i.e. correct guesses divided by the number of total images).

Synopsys	Description
<b>XIR</b>	
<b>xir.Graph.deserialize</b> (xmodel_path)	Returns the XIR model object starting from the <i>.xmodel</i> file
<b>mg.get_root_subgraph</b> ()	Returns the root subgraph of the original XIR graph ( <i>mg</i> is the output of the previous function)
<b>rs.is_leaf</b>	Checks if the root subgraph of the XIR graph is the leaf, i.e. no subgraphs are present under it ( <i>rs</i> is the object returned by the previous method)
<b>rs.topsort_child_subgraph</b> ()	Returns a list of all the subgraphs in the XIR graph, sorted from root to bottom
<b>cs.has_attr</b> (attribute)	Checks if a child subgraph has the given attribute ( <i>cs</i> is an iterable object that scans all the child subgraphs in the list obtained with the previous function)
<b>cs.get_attr</b> (attribute)	Returns the value of the requested attribute
<b>VART</b>	
<b>virt.Runner.create_runner</b> (sg,'run')	Function that returns the runner object. <i>Sg</i> is the subgraph that must be executed by the DPU.
<b>dpu.execute_async</b> (in,out)	Command to start the execution of the DPU. <i>In</i> is the input array already filled with data, while <i>out</i> is the output array that will be filled by the DPU. The returned value is the id of the just started job ( <i>dpu</i> is the runner object returned by the previous function)
<b>dpu.wait</b> (job_id)	Functions that synchronize the OS with the DPU execution by waiting the end of the job indicated as parameter ( <i>dpu</i> is the runner object)

Table 3.3: XIR and VART libraries methods used in the managing of the DPU execution

### 3.3 The board preparation

Before being able to finally deploy the whole application on the ZCU102, one last step is necessary. At this moment in time, all the software components were added to the project, but the implementation of the hardware core, the DPU, is needed. According to what has been said in section 2.1.3, there are two possible methods to include the DPU inside the board FPGA. During the development of this project, the Vitis approach was adopted, thus the concept of platform must be introduced.

An embedded platform is the starting point of any Vitis designs since applications are built on top of it. It is composed by two main elements: a hardware platform and a software platform. The former is the static part of the hardware development, thus it does not correspond to what will be inserted inside the FPGA, and is used to describe all the resources that will be available during the acceleration of some functions. An example of the possible resources are input and output interfaces, clocks, AXI buses and interrupts. For what concerns the software platform, it specifies the environment that runs the software used to control the acceleration kernels, thus it corresponds to a kind of operating system. Usually, on Xilinx systems a Linux domain with the Xilinx Run-time (XRT) enabled is chosen.

During the development of the project described in this thesis work, a custom platform for the ZCU102 board was generated following one of the official tutorials provided by Xilinx [15] with slightly modifications since it targets a ZCU104 board.

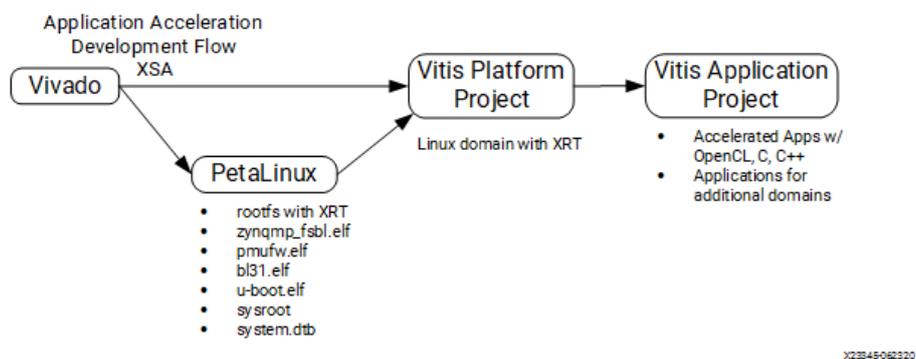


Figure 3.7: Diagram of a Vitis Platform development and usage

### 3.3.1 Hardware platform generation

The first step is the creation of the hardware platform that can be performed by generating the *.xsa* (Xilinx Support Archive) file exporting a VIVADO project as a platform. This project should be filled with all the resources available that can be useful during the development of an acceleration RTL (Register Transfer Level) component able to communicate with the rest of the system. When opening the VIVADO software and creating a new project, the project wizards open up and the following configurations were selected:

- **Type of project:** RTL project, with both the check-box flagged (*Do not specify sources at this time* and *Project is an extensible Vitis Platform*);
- **Board page:** select the ZCU102 from the board list (in the specific case of this project). Indicating the correct board at this moment is important since it allows to automate part of the process as it will be seen.

Once the project was set up, it was necessary to include all the following components inside the Block Design (that can be added by selecting the Create Block Design, under Project Manager → IP INTEGRATOR):

- **Zynq Ultrascale+ MPSoC:** it allows to insert both the PS and the PL inside the platform. After having included it, the *Run Block Automation* link was used to customize the IP according to the board presets, i.e. the default configurations of the given board (for this reason it is important to correctly select the board from the project wizard). Before proceeding with the block automation, all the check-boxes had to be flagged. Once the automation process ends, the IP will have additional signals with respect to the one freshly inserted.

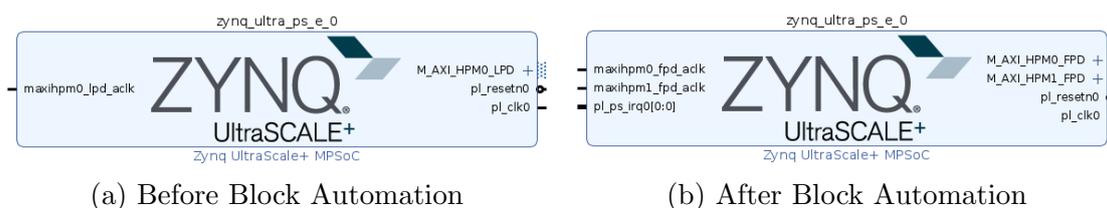


Figure 3.8: Zynq UltraScale+ MPSoC Block Automation effects

- **Clocking wizard:** elements that allows to insert different clocks inside the platform by double clicking the clocking wizard IP (after it has been added to the Block Design) and selecting the *Output Clocks* tab. From here it is possible to enable up to eight clocks, customizing the name, the output frequency, the phase and the duty cycle. Following the tutorial, three clocks were added:
  - clk\_out1 with a 100 MHz frequency;
  - clk\_out2 with a 200 MHz frequency;
  - clk\_out3 with a 400 MHz frequency.

At the bottom of the same tab, the *Reset Type* was set to *Active Low* by checking the corresponding radio box. The 100 MHz clock is intended to be used by the AXI lite interface, while the other two clocks corresponds to the ones for the DPU (the values can be modified to increase the performances if needed).

- **Processor System Resets:** it is the block that provides mechanisms to manage the reset conditions for the whole system. In the design one Processor Reset System was inserted for each clock.

The first basic elements were added, thus it was possible to connect them. This procedure is once again done automatically by clicking the *Run Connection Automation* link, that opens up a window where it is possible to configure the process. For this project the following options were specified:

- *All Automation* checkbox flagged;
- Under the *clk\_in1* of the *clk\_wiz\_0*, the */zynq-ultra-ps-e-0/pl\_clk0* option was selected from the dropdown box;
- The *slowest\_sync\_clk* under the *proc\_sys\_reset\_0* was set to */clk\_wiz\_0/clk\_out1*;
- The *slowest\_sync\_clk* under the *proc\_sys\_reset\_1* was set to */clk\_wiz\_0/clk\_out2*;
- The *slowest\_sync\_clk* under the *proc\_sys\_reset\_2* was set to */clk\_wiz\_0/clk\_out3*;
- On every *proc\_sys\_reset* instance, under *ext\_reset\_in* the *Board Part Interface* was set to *Custom* and the */zynq-ultra-ps-e-0/pl\_resetn0* option was selected for the *Select Manual Source*.

Once all the settings above were selected, the connection automation was run and after that all the *dcm\_locked* signal of every *proc\_sys\_reset* instance was manually connected to the *locked* signal of the *clk\_wiz\_0*.

Now that all the clocks were inserted, it was necessary to activate them. To perform

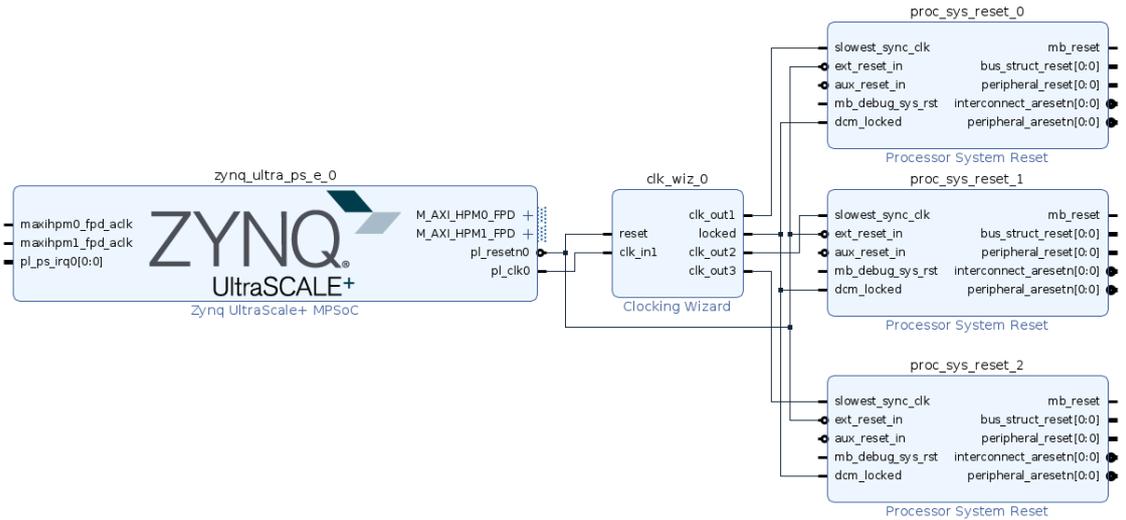


Figure 3.9: Block Diagram after the *Run Connection Automation* was run

such a task, the *Platform Tab* was selected and all the clocks under *clk\_wiz\_0* were enabled by flagging the corresponding checkbox. In addition to that, the *clk\_out2* was selected as the default one, checking the *Is Default* property, and all the ID were changed to 0, 1 and 2 to allow the Vitis IDE to automatically use them while building the application.

The next step is the insertion of all the interrupt handling system which includes the settings of some parameter inside the processor, i.e. the *Zynq UltraScale+ MPSoC*, as well as the addition of an *AXI Interrupt Controller*. For the first passage, it was necessary to enable the *AXI HPM0 LPD* option and disable the *AXI HPM0 FPD* and the *AXI HPM1 FPD* options that could be find by double clicking the MPSoC instance, in the *PS-PL Configuration* property expanding the *PS-PL interface* entry before and the *Master Interface* item after. This operation allows to reserve the first AXI interface to controlling purposes (it was set to have a 32 bit bus-width to avoid the automatic addition of unnecessary logic inside the PL) while the other two interfaces were disabled to avoid any automatic connection that could be made accidentally and reserving the interfaces for the DPU connection. To proceed with the interrupt system addition, it was necessary to include inside the *Diagram Block*

the *AXI Interrupt Controller*, to which the *Interrupt Output Connection* parameter was set to *Single* (the property can be reached by double clicking the instance of the element). For a last time inside the design of the hardware platform, the added element needed to be connected to the rest of the system. Once again this step was automatically performed through the *Run Connection Automation* link, where the *All Automation* was enabled and the *Clock Source for Slave Interface* and the *Clock Source for Master Interface* were set to */clk\_wiz\_0/clk\_out2*. To finally conclude the whole design, it was necessary to enable the interrupt signals for the platform ,by enabling the *intr* options that could be find under the *Interrupt* tab of the *Platform Setup* panel, and enable the AXI interfaces. This last step was

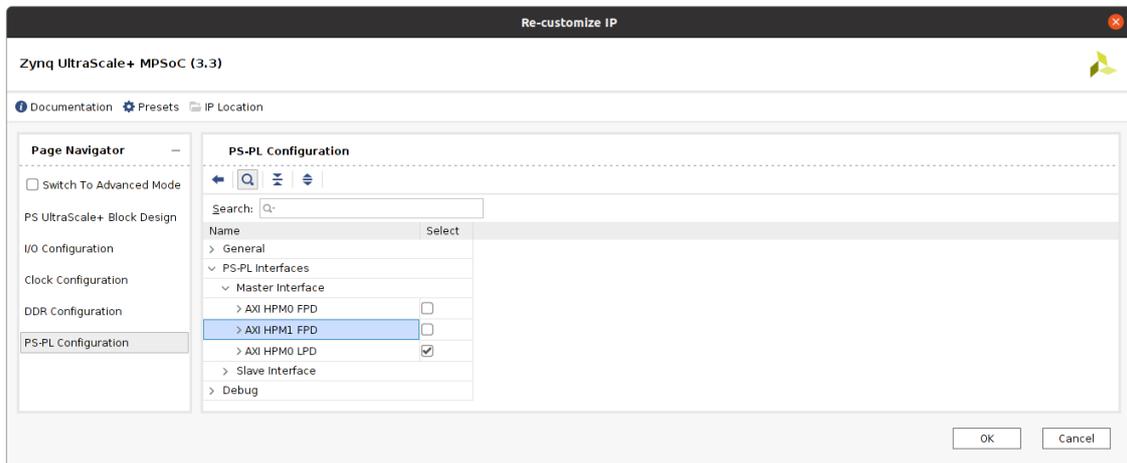


Figure 3.10: Vivado menu for managing the communication interfaces between PS and PL

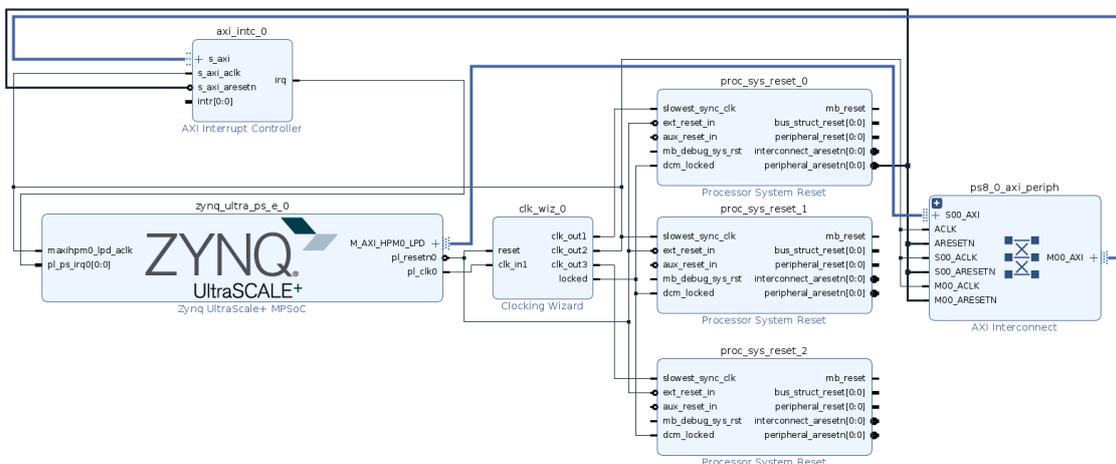


Figure 3.11: Block Diagram at the end of the development process

performed in the *AXI Port* tab inside the *Platform Setup* section where the masters from *M01\_AXI* to *M07\_AXI* of the *ps8\_0\_axi\_periph* were enabled as well as all the interfaces of the *zynq-ultra-ps-e-0* except for the *S\_AXI\_LPD*. In the tab there is the possibility to specify some values for the *Memport* and the *SP Tag* property of every interface (the former option determines the type of the interface, while the second gives tag that can be used while building to identify the interface), which was done for all the slave interfaces with the parameters visible in table 3.4, all the masters were left with the default options. Now that the whole hardware platform was

<b>AXI interface</b>	<b>Memport</b>	<b>SP Tag</b>
S_AXI_HPC0_FPD	S_AXI_HP	HPC0
S_AXI_HPC1_FPD	S_AXI_HP	HPC1
S_AXI_HP0_FPD	S_AXI_HP	HP0
S_AXI_HP1_FPD	S_AXI_HP	HP1
S_AXI_HP2_FPD	S_AXI_HP	HP2
S_AXI_HP3_FPD	S_AXI_HP	HP3

Table 3.4: Memport and SP Tag values for the slave interfaces

completely designed, it was possible to export the *.xsa* file. The *Block Diagram* is firstly validated to check the presence of possible errors (a critical warning relative to the missing connection of the interrupt raised, but it was okay to ignore it since the signal will be connected during the building of the application inside the Vitis IDE) and the wrapped with the *Create HDL Wrapper* function of Vivado. As a last step before exporting, the design was pre-synthesized by clicking the *Generate Block Design* link and selecting *Global* in the *Synthesis Option*. At the end of all these procedure, the *.xsa* file was exported through the *Export Hardware Platform* wizard that opens up when the user clicks the *Export Platform* button under the *Export* item of the *File* toolbox menu. The wizard requires different information in order to complete the export procedure: *Platform Type*, which was set to *Hardware and Hardware Emulation*, *Platform State*, that was selected to *Pre-Synthesis* with the *Include Bitstream* option enabled, and finally some details on the platform (such as name, vendor, board, version and description).

### 3.3.2 Software platform generation

Now that the hardware platform file was finally available, it was possible to proceed with the generation of the custom Linux-based operating system that fit the hardware that had just been generated. All the operations performed to achieve the final Linux distribution were made possible thanks to the PetaLinux tool, that allows to easily insert packages and functionalities to a default Linux-based OS which fits the needs of a Xilinx application. In order to further simplify the customization process, when a new PetaLinux project is created it is possible to specify a *.xsa* file that will be parsed to automatically modify some parts of the Linux distribution so that it will fit the hardware platform. From a practical point of view, the creation and the specification of the hardware platform is done with the following commands inside the Linux host terminal:

1. `source <petalinux_tool_installation_dir>/settings.sh` (activates the PetaLinux environment)
2. `petalinux-create --type project --template zynqMP --name <project_name>`
3. `cd <project_name>`
4. `petalinux-config --get-hw-description=<path_to_xsa_file>`

The last commands opens up a menu, where the *MACHINE\_NAME*, that can be found under *DTG settings*, must be changed in order to match the target board. In the case of this project, the name that had to be inserted was *zcu102-rev1.0*. As it has been said, the PetaLinux tool allows to enable different types of available packages, but to be able to do so they must be added to the *user-rootsconfig* file located in the */project-spec/meta-user/conf/* path inside the project folder. Following the tutorial all the packages below were added:

- Xilinx Run-Time support packages:
  - CONFIG\_xrt
- Easy system management packages:
  - CONFIG\_dnf
  - CONFIG\_e2fsprogs-resizefs

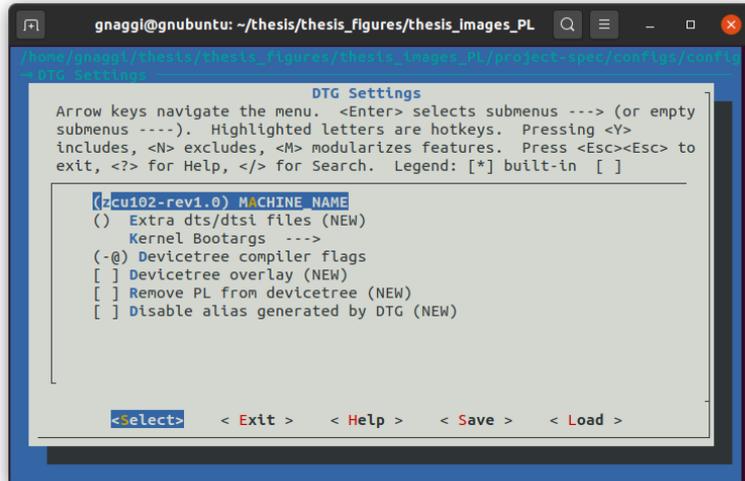


Figure 3.12: View of the PetaLinux terminal showing the *DTG* menu

- CONFIG\_parted
- CONFIG\_resize-part
- Vitis AI dependencies support package:
  - CONFIG\_packagegroup-petalinux-vitisai
- Natively building Vitis AI application packages (Optional):
  - CONFIG\_packagegroup-petalinux-self-hosted
  - CONFIG\_cmake
  - CONFIG\_packagegroup-petalinux-vitisai-dev
  - CONFIG\_xrt-dev
  - CONFIG\_opencl-clhpp-dev
  - CONFIG\_opencl-headers-dev
  - CONFIG\_packagegroup-petalinux-opencv
  - CONFIG\_packagegroup-petalinux-opencv-dev
  - CONFIG\_packagegroup-petalinux-python-modules
- Packages for running Vitis AI demo application with GUI (Optional):
  - CONFIG\_mesa-megadriver



under the *Image Features* entry of the menu.

At this point in time, the rootfs was fully customized, thus it was possible to move to the kernel menu where it is a good practice to disable the CPU IDLE in order to facilitate the debug. The menu can be accessed with the `petalinux-config -c kernel` command and under the *CPU Power Management* entry is possible to disable the *CPU idle PM support*, under the *CPU idle* sub-menu, and the *CPU Frequency scaling* option, under the *CPU Frequency scaling* sub-menu. The last configuration that was performed before being able to build the operating system is the addition of the EXT4 format support for the root file system. The relative menu is opened with the `petalinux-config` command and under *Image Packaging Configuration* is possible to select as *Root File System Type* the EXT4 value and add the `ext4` value to the list named *Root filesystem formats*, if it is not already present. Lastly, to

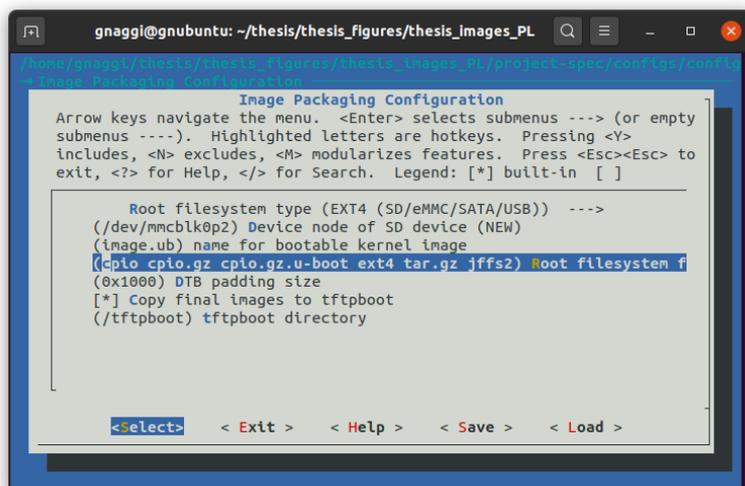


Figure 3.14: View of the PetaLinux terminal showing the *Image Features* menu

let Linux know which root file system should be used it is necessary to modify the *User Set Kernel Bootargs* to `earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk0p2 rw rootwait cma=512M` after having disabled the *generate boot args automatically* options (both the parameters can be found under *DTG settings* → *Kernel Bootargs*).

Now that the whole configuration is completed it was possible to build the entire operating system by issuing the `petalinux-build` command and, once the build is over, the `petalinux-build -sdk` command to create a system root self-installer.

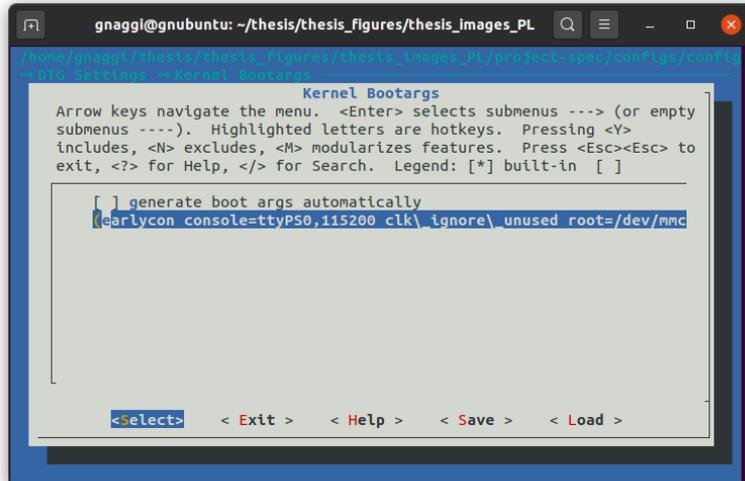


Figure 3.15: View of the PetaLinux terminal showing the *Kernel bootargs* menu

### 3.3.3 Vitis platform generation

At this point, all the elements of a platform are ready to be packed into a file that can then be used in any Vitis application. Before opening the Vitis IDE where the platform can be created, it is necessary to install the sdk that was generated with the last command issued during the PetaLinux procedure in order to have all the needed components available. When the Vitis IDE is launched, a new platform project had to be created by clicking the *Platform Project* button under *File* → *New* and in the platform wizard page the following options had to be inserted:

- Name of the platform;
- XSA file generated during the hardware platform procedure;
- Operating system had to be selected to Linux;
- psu\_cortexa53 with a 64-bit architecture as processor;
- The *Generate boot components* disabled.

Since no boot component was automatically generated, they had to be specified in the *Platform Settings* view under the *Setup software settings*. In particular the following files or directory had to be selected for the *linux in psu\_cortexa53* domain:

- BIF file: set to *Generate BIF*;

- **Boot Components Directory:** set to a folder where the *zynqmp\_fsbl.elf*, the *pmufw.elf*, the *u-boot-dtb.elf* and the *system.dtb* files must be copied from the /images/linux sub-directory of the directory where the PetaLinux project was built.
- **FAT32 Partition Directory:** set to a folder where the *boot.scr* and the *system.dtb* files must be copied from the /images/linux sub-directory of the directory where the PetaLinux project was built.

With this parameters correctly set, the platform is ready to be built and it will be possible to use it for any Vitis application. The output is composed by a folder that includes an hardware sub-folder, named hw, a software sub-folder, named sw, and a file with extension *.xpfm*. For sake of simplicity it is a good choice to create a folder where to store all the petalinux created files that will be useful during the Vitis application generation as it will be seen in the following section (all these files are stored inside the /images/linux sub-directory of the directory where the PetaLinux project was built). In addition to that, it could also be useful to save the sysroot of the Linux distribution.

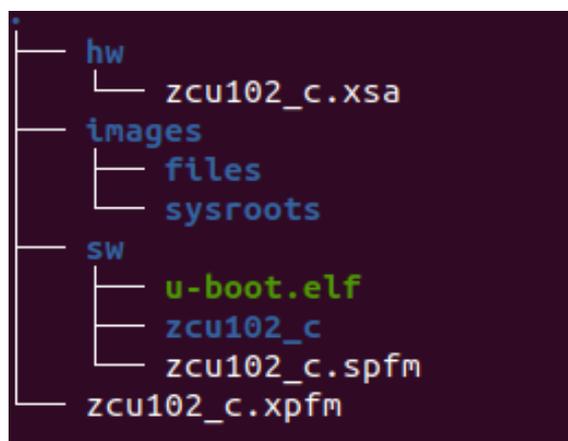


Figure 3.16: Tree representation of the platform output folder

### 3.3.4 Application creation

Once the Vitis platform is ready it was possible to finally create a Vitis project, whose main objective is the generation of an image that can be flashed on a SD card and loaded inside the ZCU102 in order to start the operating system and to configure

at boot the FPGA. As it was seen at the beginning of this chapter, the platform that had just been created is used to provide all the static hardware component as well as the operating system, thus the hardware accelerators, i.e. the logic that will be implemented inside the FPGA and used by a software application at run-time, have still to be added and this step is performed inside the Vitis IDE. All the accelerators, that are usually called RTL kernels inside the Vitis suite, comes in files with the *.xo* (Xilinx Object) extension, therefore a *.xo* file containing the DPU had to be created in order to load it into Vitis.

Fortunately, this procedure is totally automated since Xilinx set up a project (named DPU-TRD and included inside the Vitis AI github folder) where the only configurations that must be modified are the ones that allows to customize the DPU (see chapter 2.1.2) and the ones that allows to specify how the DPU should be connected to the rest of the system. For the former it is sufficient to edit the *dpu\_conf.vh* file that can be found inside the *prj/Vitis* folder of the DPU-TRD directory. The configuration file, whose default version is visible in appendix C, will be parsed during the build of the DPU and all the *define* command issued inside determine one different parameter for the DPU that will be synthesized. For what concerns the DPU interconnection, there is once again a file that must be edited to set up all the parameters. It can be find in the *prj/Vitis/config.file* directory and it is named *prj\_config*. Inside this file all the argument of the *v++* compiler, which can be seen as the Vitis synthesizer, are stored, thus changing values inside will change the outcome of the synthesis.

For what concerns this particular project, none of the two files were modified and the generation of the whole kernel was done with the following command: `make binary_container1/dpu.xo DEVICE=zcu102_c` (where the parameter after the device tag is the name of the platform and the argument after the make indicates the output folder).

At this point, the development of the application was the last thing to be done in order to finally deploy the entire system on the ZCU102 board. A new project from the Vitis IDE was created, but this time the *Application Project* button was clicked instead of the *Platform Project* of the previous section. Once again a wizard opens up requiring some additional information that were the following:

- Project name;
- The target platform, which in this case is the one created before;

- The software Domain, which was set to *linux on psu\_cortexa53*;
- The Sys\_root path, which is the folder that was copied inside the platform final directory;
- Empty application, to avoid any template to load.

When the IDE loads all the resources of the project, it is possible to import all the *.xo* files needed for acceleration purposes by right clicking the *src* folder and selection the *Import Sources* entry. In addition, there is the possibility to include any software code that manages the execution of the RTL kernels. In the specific case of this project, the DPU *.xo* file generated with the Xilinx scripts was added, but no python code was loaded inside the Vitis IDE since it was directly loaded on the device through the ssh protocol that had been configured during the PetaLinux customization. Now that the dpu kernel is available, it must be added to the project by selecting the *hw.link.prj* file in the explorer on the left of the IDE. A setting page opens, where there is a section called *Hardware Functions* with some icon buttons. The icon showing a blue circle and a white thunderbolt is the button that allows to select a RTL kernel and add it to the system. In this project, the DPU was added

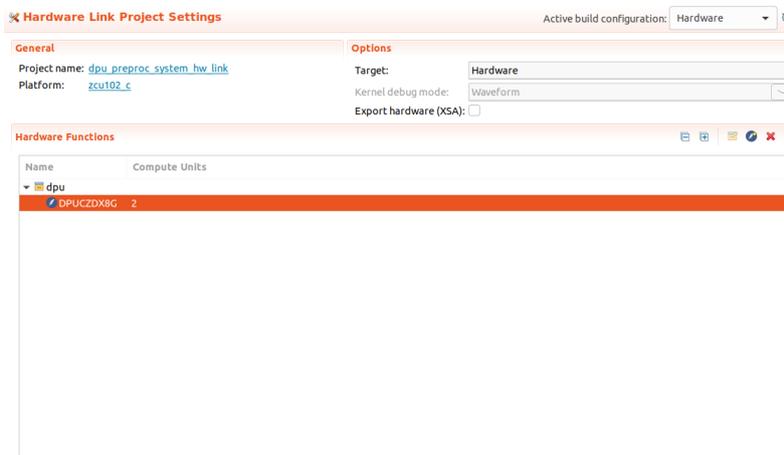


Figure 3.17: View of the Vitis IDE hardware linker

and the compute units number was changed to two to include two hardware cores. Finally the configuration file containing all the *v++* arguments needs to be linked to Vitis, operation that can be performed selecting the *Edit v++ options* entry in the context menu of the binary container on top of the added hardware function. For the case of the DPU, all the needed parameters were stored inside the file seen at the

beginning of this section, thus the data inserted was: `--config <path_to_config_file>`. The final build can be run now and when it ends the SD image will be generated. During the build, a bug was reported since the packaging phase (which is one of the main phases that are run to synthesize the entire system) failed due to a wrong parameter that is set automatically by Vitis. To bypass this issue, the `v++` command was issued directly inside a Linux terminal with the command:

```
v++ --package --config package.cfg ../../dpu_preproc_system_hw_link/Hardware/dpu.
xclbin -o dpu.xclbin
```

where the `package.cfg` file has the last line commented (it was the source of the problem due to a wrong concatenation of directory path strings). At the end of

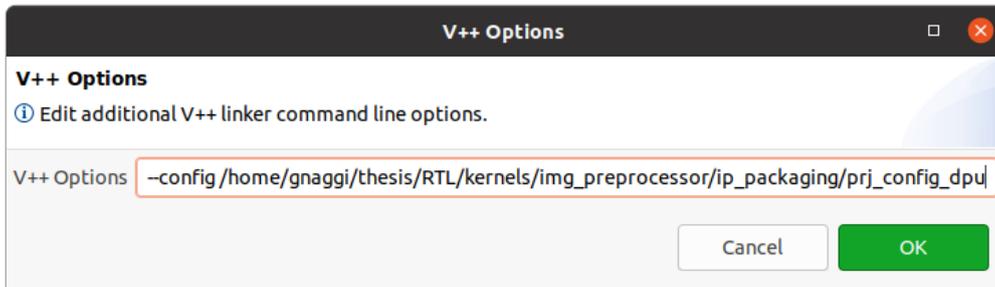


Figure 3.18: View on the `v++` option dialog box

the process, the image was correctly generated and report file containing all the information is produced. From figure 3.19 it is possible to notice that both the DPU cores were correctly inserted and connected to the PS, thus it was possible to flash the obtained image file inside the SD card through the usage of the *Balena etcher* software. Once the flash process is finished it is possible to load the SD on the board and connect to it with the host computer.

### 3.4 The ZCU102 connection

As it was said during the creation of the operating system, one of the most used ways to connect to the ZCU102 board is the ssh protocol, which requires the IP address of the target (in this case the board). To retrieve it it is possible to use the UART micro-usb interface of the board which can be used to communicate with a software like Putty. To establish a connection with the board through the Putty software, the following command was used inside an Ubuntu terminal:

## The image classifier

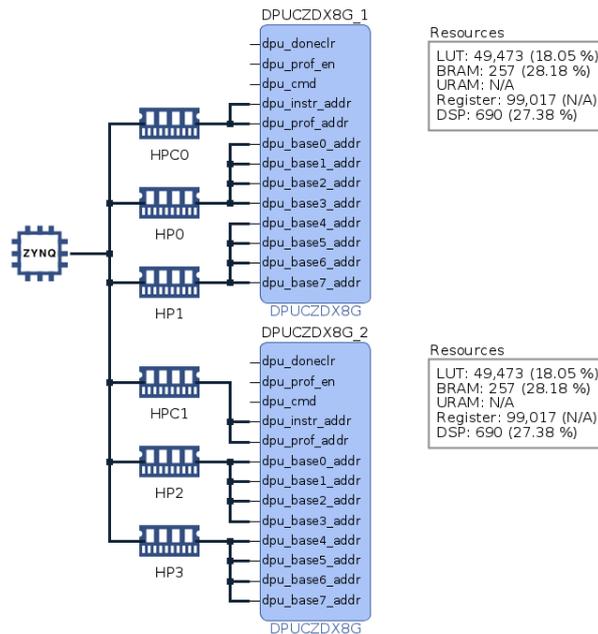


Figure 3.19: Vitis Analyzer view of the summary created at the end of the building process

```
sudo putty -serial /dev/ttyUSB0 -sercfg 115200,8,n,1,N
```

The `/dev/ttyUSB0` indicates the driver of the USB to which the board is connected, while the values that follow the `-sercfg` argument are the parameters for the connection. In particular:

- 115200 determines the Baud Rate
- 8 indicates the number of data bits for each frame
- n specifies that no parity bit is used
- 1 indicates the number of stop bits
- N determines that no CRC (Cyclic Redundancy Checking) is used

Once the connection is established, a putty window opens up and the command `ifconfig` is used to retrieve the IP address of the board. From now on, all the communications with the boards are possible with the ssh protocol with the username `root` and the password `root`. During the entire flow of this project, the ssh protocol was used both to enter inside the board operating system and launch all the applications that must be run on the ZCU102, but also to load the data from the host to the board memory through the `scp` command.

At this point of the development flow, it was necessary to check the correctness of what was designed so far. After turning on the ZCU102 and discovering the board IP, a folder with the directory tree in figure 3.20 was loaded into the memory of the board with the *scp* command. It contains the python application, the compiled AI model and some test images. Entering inside the operating system was then possible

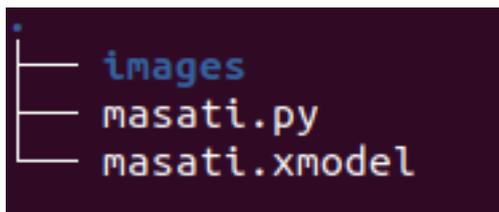


Figure 3.20: Directory tree of the folder loaded on the board

to run the python application and the obtained result was the one visible in figure 3.21. As it is possible to see, the final result was in line with the expected one, with an effective accuracy that is lower than the one evaluated after the model quantization (which was 0.973574). Since the final result was considered as reasonable, it was

```
root@zcu102_c:~/target# python3 masati.py
-----
Command line options:
  --image_dir : images
  --model     : masati.xmodel
Post-processing 300 images..
Correct:286, Wrong:14, Accuracy:0.9533
-----
```

Figure 3.21: Outcome of the python application when the execution is finished

possible to continue with the profiling that will be explained in the next chapter.

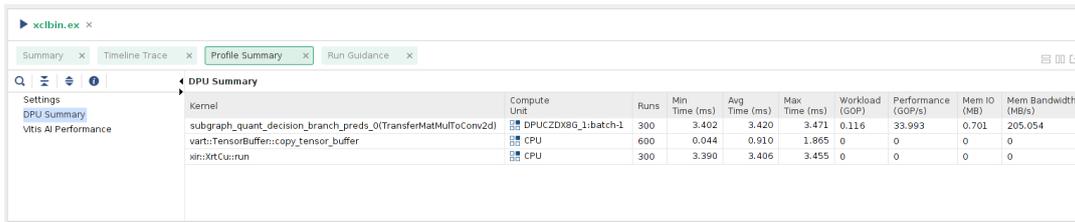
## 3.5 The profiling process

From a behavioral point of view the application was demonstrated to work correctly, but, in order to simulate all the processes that needs to be done on a real project, an additional step was performed. It consists in the profiling procedure that is useful to check the performances of all the software and hardware components in order to detect the bottlenecks to increase their speed, if necessary. By default, the command seen in section 2.2.4 automatically saves the data relative to the DPU, the VART library and the XIR library. All the other functions inside the software application

are not tracked, but it is possible to include them inside the profile procedure by adding the decorator *vai\_tracepoint* to each function.

Different profiling were made in order to test the decorator functionality, but also to verify the real behavior of the system when increasing the number of the executing cores.

In the first attempt, the profiling command was issued on the code that handles



(a) List of the tracked functions with all the relative information



(b) Timeline of the execution

Figure 3.22: Vitis Analyzer views of the data collected during the straight forward profiling procedure

the execution with a single DPU core and the results are reported in figure 3.22 that displays the Vitis Analyzer GUI that allows to visualize in a clearer way the data. In particular, from the figure 3.22a it is possible to notice a list of functions with some relative data. The information that were considered as the most important inside this project were: the Compute Unit, that indicates where the function is executed, the number of runs and the time taken for executing each run, the Max Time was taken into account to consider the worst case scenario. The last two information were also useful to calculate the impact of the function on the whole application: multiplying the number of runs by the time taken from each run it was possible to obtain the total time needed to completely execute the given function, thus it allowed to evaluate the impact on the execution of the entire system. For what

concerns the timeline shown in figure 3.22b, it reports how functions are executed in time and it allows to verify that the behavior is the expected one since the DPU completed a task every few milliseconds, thus each time a single image was pre-processed.

In a second attempt, a test on the differences when running two DPU cores was carried on. The application code was slightly modified in order to commit all the images in a even position in the list of all the images to the first core, while all the others where issued to the second core. In addition the *wait* function was called only after the first odd figure was dispatched and it was set to wait the end of the core that manages the even figures (which should allow to reduce the time since the DPU execution should overlap with the pre-processing of the following image). From figure 3.23a it is possible to notice that the number of runs that the DPU had to handle in figure 3.22a is split into two different cores and the same behavior is confirmed in figure 3.23b, even tough the final execution time seems to be higher than the previous one.

Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Workload (GOP)	Performance (GOP/s)	Mem IO (MB)	Mem Bandwidth (MB/s)
subgraph_quant_decision_branch_preds_0(TransferMatMulToConv2d)	DPUCZDX8G_1:batch-1	150	3.398	3.419	3.448	0.116	34.005	0.701	205.125
subgraph_quant_decision_branch_preds_0(TransferMatMulToConv2d)	DPUCZDX8G_2:batch-1	150	3.403	3.420	3.464	0.116	33.991	0.701	205.041
virt:TensorBuffer::copy_tensor_buffer	CPU	600	0.044	0.904	1.866	0	0	0	0
xin:xtcu:run	CPU	300	3.387	3.407	3.453	0	0	0	0

(a) List of the tracked functions with all the relative information



(b) Timeline of the execution

Figure 3.23: Vitis Analyzer views of the data collected during the straight forward profiling procedure with an application that uses two different DPU cores

The final attempt aimed to add custom functions inside the profiling report by using the python decorator. Since the process that is expected to be more time consuming is the image pre-processing, it was chosen to track it to see which are the most

critical parts. Unfortunately the decorator can be applied only on functions and not on instructions, thus the pre-processing function was split into several different functions that included a single instruction in order to be able to profile each single instruction. For this reason the code was updated with additional functions, each one of them decorated with the *vai\_tracepoint* decorator. The results obtained are visible in figure 3.24, where only the profiling view is reported since the timeline is the same as figure 3.23b. Analyzing the obtained data, it is possible to notice

Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Workload (GOP)	Performance (GOP/s)	Mem IO (MB)	Mem Bandwidth (MB/s)
subgraph_quant_decision_branch_preds_0(TransferMatMulToConv2d)	DPU_C2DX8G_1batch-1	150	3.402	3.421	3.472	0.116	33.981	0.701	204.981
subgraph_quant_decision_branch_preds_0(TransferMatMulToConv2d)	DPU_C2DX8G_2batch-1	150	3.400	3.421	3.466	0.116	33.985	0.701	205.004
vart::TensorBuffer::copy_tensor_buffer	CPU	600	0.044	0.916	1.957	0	0	0	0
xrt::XrtCu::run	CPU	300	3.388	3.408	3.461	0	0	0	0
openImage	CPU	300	13.159	17.528	21.950	0	0	0	0
convertImage	CPU	300	0.284	0.587	1.904	0	0	0	0
rearrangeImage	CPU	300	4.479	4.629	23.998	0	0	0	0
rearrangeImage	CPU	300	4.991	5.423	6.670	0	0	0	0
imagePreprocessing	CPU	300	23.750	28.402	48.672	0	0	0	0

Figure 3.24: Vitis Analyzer view of the data collected during the profiling of the application with the python decorator applied to the functions

that the pre-processing procedure takes up to 78% of the total execution time as it was expected. In particular the most time consuming function is the *openImage* which decodes the input image to transform it into the pixels multidimensional array. At this point of the development flow, all the information needed to choose which software function must be accelerated to increase the performances are available. In a project that has no demonstration purposes as this one, the most logic choice would be to implement in hardware the decoding of the image since it takes up to 21.95 ms for each image. Since the aim of this project is not the development of a hardware core, it was chosen to accelerate the *rearrangeImage* which appeared to be the second slowest function in terms of average execution time and it was very easily implemented in hardware since it only converts the pixel integer value, which is in range 0 to 255, into a floating value in range -1 to 1.

### 3.6 The kernel development

Before proceeding with the creation of the RTL kernel that substitutes the *rearrangeImage* function, it is necessary to introduce some requirements that must be met in order to correctly connect the developed kernel with a Vitis application and,

more in general, to a Xilinx board. Those requirements comes from two different reasons: the first one is that the v++ compiler must be able to recognize and correctly connect the RTL kernel to the rest of the system, while the second is that XRT is the library which usually manages the kernel, thus it is necessary to be compliant to its communication protocol.

For what concerns the v++ compiler requirements, they are listed in table (3.5) and principally specify the signals that may or may not be used while designing the kernel and the relative names that can be used.

Port or Interface	Description
Clock	At least one clock must be inserted in the kernel. It can have any name, but it must be packaged with a bus interface.
Reset	It is an optional port, but if it is present it must be associated with a Clock signal through the <i>ASSOCIATED_RESET</i> property. There are not any restrictions on the name.
interrupt	It is an optional port, but when it is present must be named exactly as shown on the left (it is case sensitive).
s_axi_control	Mandatory port that is necessary for the control data. It may not be present when the kernel is not controlled by a software application. When it is present must have the same name as shown on the left (case sensitive).
AXI4 Memory Mapped Interfaces (m_axi)	It is an optional interface which must have 64 bit addresses and it will be used to access the global memory. Must not use neither the Wrap nor the Fixed burst types of the AXI protocol and no sub-size (narrow) bursts can be used too.
AXI4 Stream (axis)	Optional interface that may be used to transfer data between kernels or between host application and kernel. It must be used as a mono-directional interface.

Table 3.5: v++ compiler requirements for the RTL kernel developing

Additionally, if the kernel is not user-managed but it needs to interact with XRT, there are some mapped registers that must be added inside the kernel that

will be used by XRT to store some useful data during the execution. There are two main protocols that can be used when dealing with XRT managed kernels: the *ap\_ctrl\_chain* and the *ap\_ctrl\_hs*. The former exploits a pipelined like execution feeding the kernel with new data at each clock cycle, while the latter forces the host application to wait for the end of the kernel execution before providing new data to it. In both the cases, the kernel is started through an API call.

The list of all the reserved addresses with the correspondent register structure is the following:

- Address 0x0: Control register

Bit	Name	Description
0	ap_start	Its value is 1 when the kernel can start processing data. It should be cleared when the ap_done is being asserted.
1	ap_done	Asserted when the kernel finishes the execution. When the value is read it must be cleared and set to 0.
2	ap_idle	Its value is 1 when the kernel its waiting for the execution to start.
3	ap_ready	Asserted by the kernel when it can receive new data.
4	ap_continue	Asserted by the XRT in order to keep the kernel running.
7	auto_restart	When its value is 1, the kernel automatically restarts a new execution.
5:31	Reserved	Reserved

- Address 0x4: Global Interrupt Enable Register

Bit	Name	Description
0	Global Interrupt Enable	When its value is 1 and the IP Interrupt enable bit has value 1, the interrupt is enabled.
1:31	Reserved	Reserved

- Address 0x8: IP Interrupt Enable Register

Bit	Name	Description
0	Interrupt Enable	When its value is 1 and the Global Interrupt enable bit has value 1, the interrupt is enabled.
1:31	Reserved	Reserved

- Address 0xC: IP Interrupt Status

Bit	Name	Description
0	Interrupt Status	Its value is toggled on write.
1:31	Reserved	Reserved

All the addresses after 0xC are reserved to registers that stores the values of the kernel arguments. There are two different types of arguments: the scalar ones and the pointer ones. The former consists in a simple value expressed on 32 bits, passed through the AXI4-lite interface and directly stored into the registers, while the latter consists in a base address, passed again through the AXI4-interface but in two different bursts since it is on 64 bits, that will be used by the kernel to access the global memory. It is in charge to the kernel developer to decide the number of arguments, thus the number of registers that must be inserted.

Now that the requirements have been specified it is possible to detail how the RTL kernel was developed. The first element that was created is the component that transforms the input integer value into the output transformed float value. Taking as an example the width of the DPU data bus, which was on 128 bits, the same

width was chosen for the component that must then be able to compute more than one value at each execution. In particular, the number of inputs and outputs that must be processed each cycle is four, value that comes from the division between the data bus width and the number of bits needed in the floating point representation, which is 32. In order to keep the things easy, also the input integers are considered as represented on 32 bits (which is usually the standard size in different programming languages) even if only the lowest 8 bits were used since they are sufficient to store all the values from 0 to 255. The solution that was adopted to obtain a fast result was the creation of a Look-Up Table with 256 entries, where each address is considered the input value and the stored data corresponds to the converted floating-point. As an example, the entry addressed with the 0 index will store the binary value 0xBF800000 that correspond to the decimal -1 (which is the correct result of the equation 3.1). With this type of solution, the LUT can be accessed by different values at each time outputting all the needed results, with the drawback of a larger amount of FPGA resources used to store the whole table. In addition, a validation signal is used in order to avoid to compute data when it is not necessary and an output validation signal is used to tell to the rest of the system that the value on the output bus is correct and can be used. Once the component was fully developed

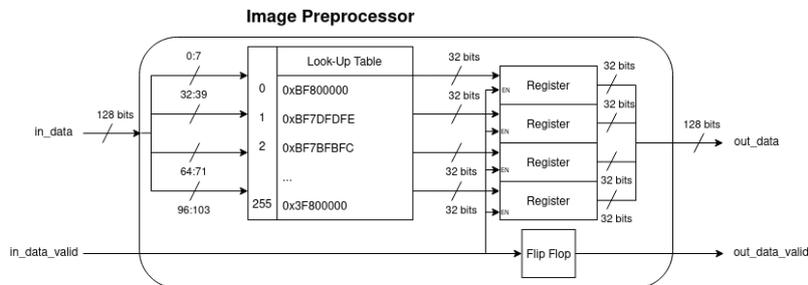


Figure 3.25: Schema of the component which converts the input integer ranging from 0 to 255 into the output floating-point ranging from -1 to 1

and tested (the validation procedure was done with a testbench that loads all the possible values since the range of possible data is very small) the AXI4-lite slave interface that meet the *ap\_ctrl\_hs* requirements was created. The whole design of this component was based on the translation into VHDL of the code posted in the Xilinx tutorial [12], which was written in Verilog. The last component that had to be added was the master interface, but before performing this operation it was necessary to decide how the data should be passed to the kernel in order to create the

appropriate number of masters. Since the developed kernel was very simple, it was chosen to use two arguments: a scalar one that indicates the total number of pixels that must be processed and a pointer one that is used to receive the base address of the location of the pixel multidimensional array inside the global memory in order to retrieve the data. In order to accelerate the design process, the AXI master was provided by Capgemini Engineering and it had only to be slightly modified to fit the needs of the kernel. A high level schematic of the provided interface is visible in figure 3.26. Now that all the components were available, they had to be connected

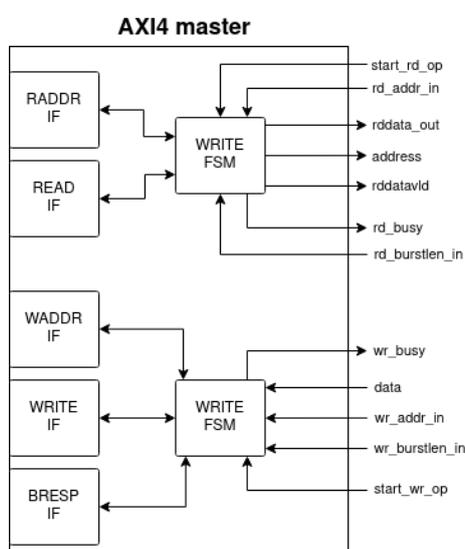


Figure 3.26: Schema of the AXI4 master interface provided by Capgemini engineering

one with each other and some additional logic had to be added to manage the correct execution of the kernel. In particular the following elements were included in the design:

- **Rising edge detector:** it is used to detect when the start signal goes high, allowing to start the first memory reading and resetting the counter;
- **Counter:** element used to keep track of the execution advancement. It allows to check when the kernel has finished to process the data, i.e. when the counter value is equal to the scalar argument divided by 4 (because 4 pixels are processed every time), and to calculate the memory offset needed to both read and write the memory. For this last operation, the value of the counter must be shifted left by four bits since at each read/write operation 16 bytes are read/written.

- **Register chain:** used to delay the address and the signal that indicates the end of the image in order to have the correct values when performing the write operation. There is a 4 clock cycle delay due to the time taken by the AXI4 master to read the data (3 clock cycles) plus the time taken by the image preprocessor to compute the output value (1 clock cycle).

In addition to these elements, some sparse logic was also necessary. Particularly important are the logics that manage the start of the reading and of the writing operation. The reading should start whenever the rising edge of the `ap_start` signal is detected as well as when the write operation is started (in this way write and reading are performed simultaneously and are synchronized to the slower operation, which is the writing, avoiding timing problems) and there are still pixels to be processed. For what concerns the write operation, it should be started whenever the data coming from the image preprocessor is valid, the master write interface is free, the kernel should be run (`ap_start` asserted) and the image is not ended.

As it was seen in section 3.3.4, the kernel has to be packaged into a `.xo` file in order to be included inside a Vitis project. Therefore, after testing with Vivado that the kernel was working as expected, it was packaged still using the Vivado software. The procedure that was followed is the one described below:

1. A Vivado blank project was created, specifying on the project wizard the name, the location, the type of project (which was RTL) and the target board (ZCU102 in this case). No sources were included when requested since this operation was done later.
2. From the source view, the plus button was clicked to add new design sources to the project.
3. From the *Tools* menu, the *Create and Package New IP* was clicked in order to start the wizard that guides the user through the packaging process. When required the *Package your current project* option was selected. A *Package IP* tab opens up and it includes many different sub-menus that must be filled with the correct information. In the *Identification* sub-menu all the information about the vendor and the IP can be inserted (optional but recommended). The two most important tabs are the *Compatibility* and the *Ports and interfaces* ones. The former allows the user to specify the type of execution protocol inside the Vitis application (such as *user\_managed*, *ap\_ctrl\_chain* or *ap\_ctrl\_hs*),

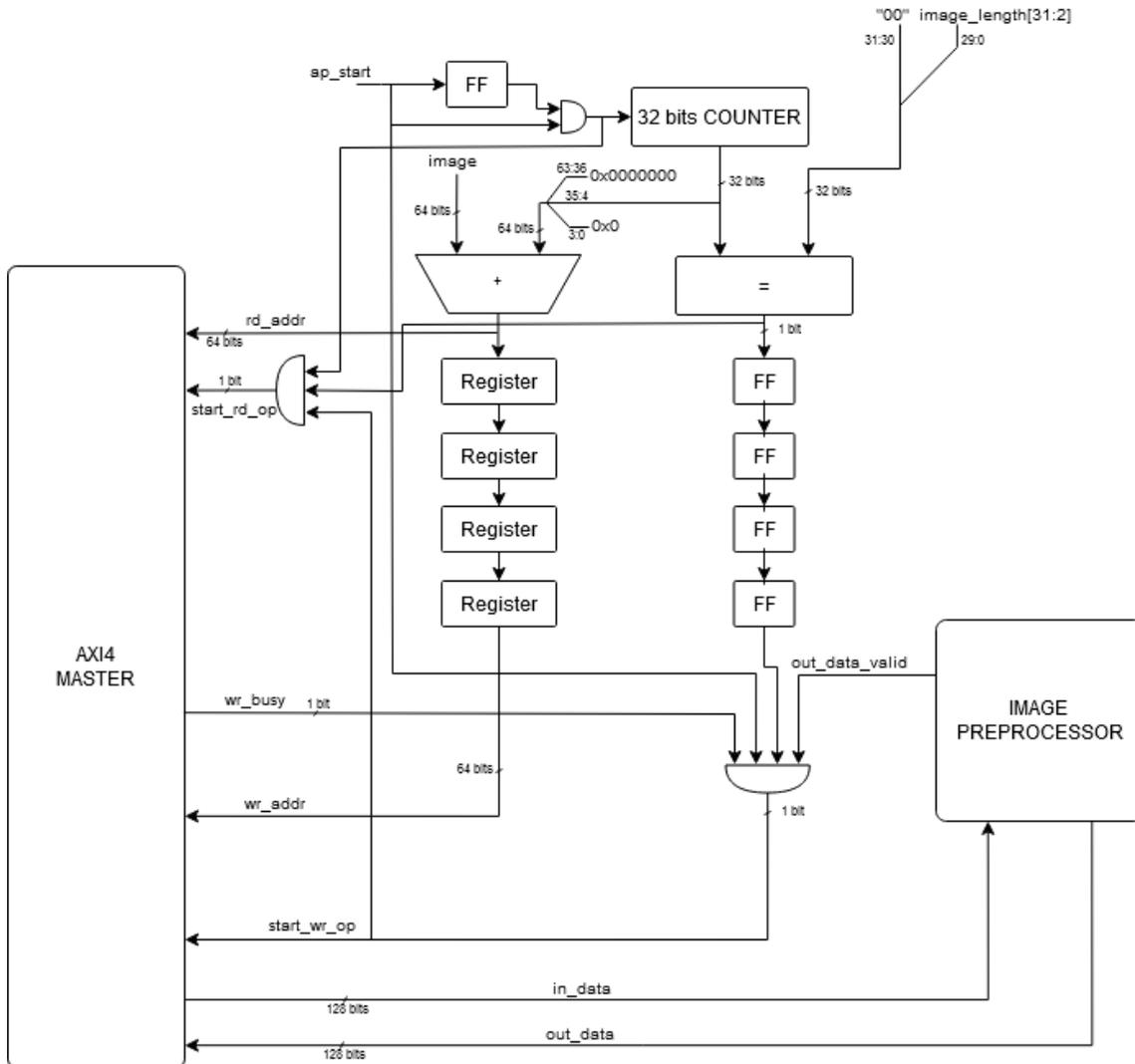


Figure 3.27: High level schema of the kernel logic

while the latter let the user to associate the clock to the AXI interface (which was one of the requirements for the `v++` compiler). In this project, the *Control protocol* option in the *Compatibility* tab was set to `ap_ctrl_hs` while the *ACLK* signal was associated to the `m00_axi` and to the `s_axi_control` interface (the association was performed by right clicking on the interface name, select the *Associate Clocks* option and specifying *ACLK* in the dialog box that opens up). Another important sub-menu inside the *Package IP* tab is the *Addressing and Memory* where the inner registers of the kernel must be inserted. In order to add a register the `reg0` instance must be right clicked and the *Add Reg* entry selected. The required information for each register are Name, Offset and Size

(in bits). For this project, all the inserted data is visible in table 3.6 After

Name	Offset	Size
CTRL	0x000	32
GIER	0x004	32
IP_IER	0x008	32
IP_ISR	0x00C	32
image_length	0x010	32
image	0x018	64

Table 3.6: Registers inserted in the *Addressing and Memory* panel with their name, offset and size

the addition of all the registers, a very important step is to associate each pointer argument register with its corresponding master interface. In this case there was only one register that had to be connected and in order to do so was sufficient to right click the register (the *image* one), select *Add Register Parameter* and add the *ASSOCIATED\_BUSIF* parameter. In the Vivado GUI an additional table with a line appeared under the *image* register and from there it was possible to set the *Value* to *m00\_axi*.

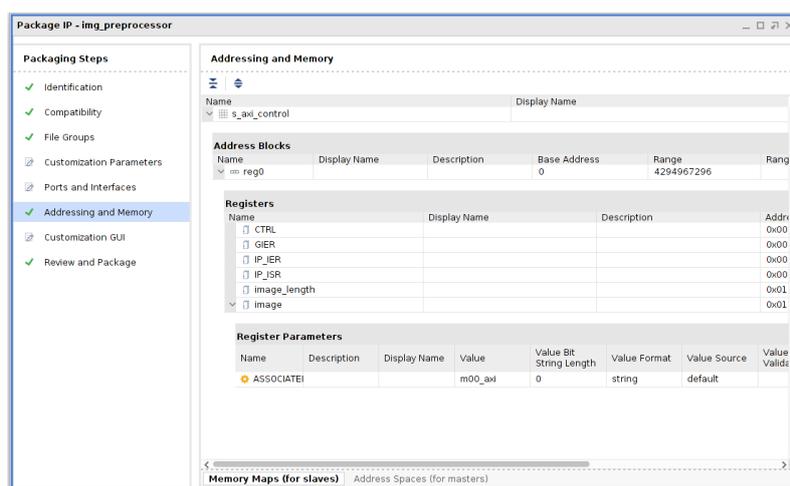


Figure 3.28: View of the *Addressing and Memory* tab of the Vivado software during the IP packaging procedure

4. At this point of the flow, the kernel should be ready to be packaged. To perform this operation it was necessary to click on the *Review and Package* tab and

click on the *Package IP* button. A message showed up in the console at the bottom of Vivado, indicating that the procedure was completed correctly.

Now that also the designed RTL kernel is fully packaged, it was possible to insert it inside the Vitis application and create the image that should be flashed on the SD card of the ZCU102. The procedure followed to create the Vitis project and to build it is the same seen in section 3.3.4, with the only difference that also the RTL kernel that had just been created had to be added among the hardware functions in the *hw.link.prj* configuration window. Immediately after the build process finished, the link report was checked to verify if the RTL kernel was correctly inserted. As can be seen in figure 3.29, the kernel seemed to be correctly inserted inside the system and also the connection with the AXI interface appeared to be correct, but no estimation of the utilized resources was indicated, which was not a normal behavior. To eliminate any doubt, the Vivado suite (which is the engine called by the Vitis

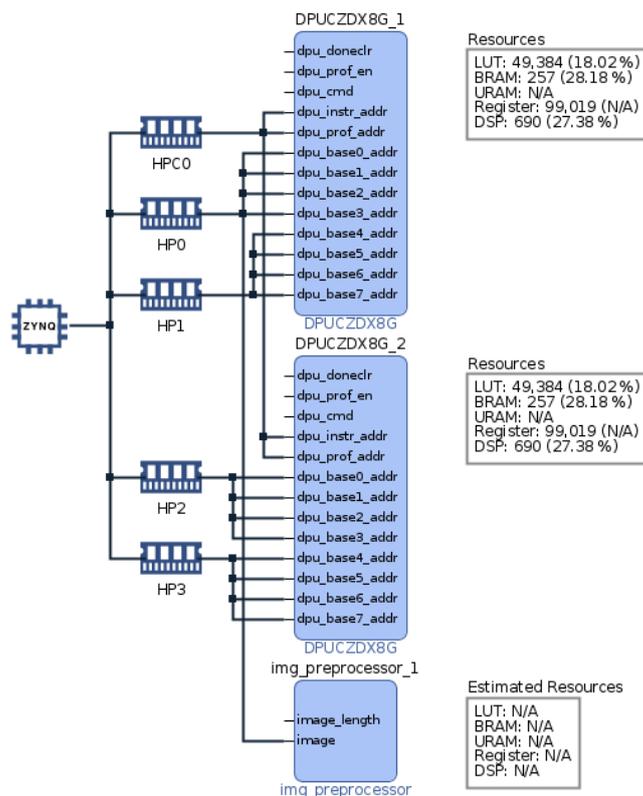


Figure 3.29: Vitis Analyzer view of the summary created at the end of the building process of the project that includes the custom RTL kernel

v++ compiler to perform the synthesis) was used to control the report and from

there everything seemed to be correctly synthesized. For this reason the process was considered concluded and the new image was flashed inside the SD to load it into the ZCU102.

### 3.7 The update of the application code

The very last step before being able to test the accelerated system was the upgrade of the code explained in section 3.2 in order to include the managing of the custom RTL kernel execution. This kind of operation is performed with the OpenCL library, which has an equivalent Python version called *pyopencl*. The OpenCL library defines an execution model which is based on two distinct execution units: the kernels and the host program. The former corresponds to where the computation occurs and are run inside a precise environment, defined as context, which includes different resources:

- **Devices:** a collection of compute units;
- **Kernel objects:** the encapsulation of a kernel function declared inside a program and the argument value needed and used during execution;
- **Program objects:** the encapsulation of a program source (or binary), a reference to the associated context, the number of kernels objects currently attached, the list of devices for which the program was built and the latest successfully built program executable;
- **Memory objects:** reference to a counted region of Global Memory.

The host program is the part of the execution in charge of creating and manage the entire context. There are some OpenCL APIs that allow the host program to communicate with the kernels to enqueue execution commands, manage memory objects and synchronize the host with the kernel.

This is the basic functionality of the OpenCL library which must be applied to the custom RTL kernel. In order to easily test if the kernel was working correctly, before uploading the previous application a new one was created with the aim of managing only the RTL kernel. The main command used inside this code are the ones defined in table 3.7.

Synopsis	Description
<code>cl.get_platforms()</code>	Returns a list with all the OpenCL platforms available on the running board.
<code>platform.get_devices()</code>	Returns a list of the devices present inside the given platform (where platform is the first element of the list returned by the previous command).
<code>cl.Context(devices)</code>	Creates an OpenCL context and attaches to it all the devices present in the devices argument, which corresponds to a list.
<code>cl.Program(context,devices,binaries)</code>	Creates an OpenCL program object to which all the arguments are attached. The binaries files correspond to the <i>.xclbin</i> file (i.e. the file that is loaded on the FPGA) that must be read inside the Python application.
<code>program.build()</code>	Builds the program object returned by the previous command
<code>cl.CommandQueue(context)</code>	Creates and attaches a command queue to the specified context.
<code>cl.Buffer(context,mem_flags,hostbuf)</code>	Creates a memory object related to the context specified as argument. The <code>mem_flags</code> argument is used to specify some additional properties of the memory object, while the <code>hostbuf</code> parameter is used to link to the buffer a memory structure of the host program.
<code>program.kernel_name(kernel_args)</code>	Starts the execution of the kernel with the specified arguments.

Table 3.7: Pyopencl methods used during the development of the RTL kernel host application

The execution of this piece of code inside the ZCU102, arose an error with the *source* of the kernel, but it was not possible to understand which was the source. To eliminate any doubt, the c++ code provided by Xilinx as demo was used, but once again a problem arose (different from the one due to pyopencl) and it was again not possible to identify the cause. The Python code that was developed is visible in appendix D.

```
root@192.168.1.16's password:
Last login: Fri Oct 22 00:38:12 2021
root@zcu102_c:~# python3 app.py
XRT build version: 2.11.0
Build hash: 0dc9f505a3a910dea96166db7b5df8530b9ae38e
Build date: 2021-06-05 12:31:54
Git branch: 2021.1
PID: 881
UID: 0
[Fri Oct 22 01:37:54 2021 GMT]
HOST: zcu102_c
EXE: /usr/bin/python3.8
[XRT] ERROR: kernel '_source' not found
```

Figure 3.30: View of the terminal that shows the error that is encountered during the execution of the pyopencl code

## CONCLUSIONS

---

Even if some unresolved problems arose at the end of the development of this thesis project, it can be considered a success if the initial objective is considered. For what concerns the validation of the Vitis AI tools, everything was proved to be perfectly functioning since the AI model provided by Capgemini engineering was fully accelerated inside the ZCU102 FPGA. There is a lot of space for improvement in two different scopes:

- **A deeper DPU customization:** The DPU was kept in its default version during the whole project, but, as it has been seen, it is possible to change many different parameters. An interesting test could be to deeply exploit this high customization and tracking all the relative performance results. In addition to that, the ZCU102 makes available a very large FPGA with plenty of space and resources, but in different real projects there may be some constraints in terms of costs (thus the ZCU102 FPGA is too expensive) or physical space available (for which the ZCU102 FPGA may be too large). It could be interesting to try to fit the DPU and all the Vitis AI development flow inside a smaller FPGA.
- **Hardware acceleration:** the last development process was the one that brought to the unresolved errors. One of the possible improvements is to make the hardware acceleration work and combine it with the Vitis AI flow as intended. The resolution of these problems could allow to further increase the performances of the project in many different ways since any software function can be turned into a hardware one.

# Appendices

# VITIS AI FLOW SCRIPTS

---

## A.1 Test script

```
# -*- coding: utf-8 -*-
2 """
Created on Fri Apr 9 18:53:02 2021
4 Modified on Tue Dec 2 2021

6 @author: emack
modified by: gnaggi
8 """
import numpy as np
10 from PIL import Image
import os
12 import timeit
import tensorflow.keras as K
14 import argparse
import random
16 from tensorflow_model_optimization.quantization.keras import vitis_quantize

18 os.environ ['TF_CPP_MIN_LOG_LEVEL'] = '3'

20 CLASSES = ['coast', 'land', 'water']

22 def printProgressBar (iteration, total, print_color, prefix = '', suffix = '',
    decimals = 1, length = 100, fill = '#', printEnd = "\r"):
    percent = ("{0:." + str(decimals) + "f}").format(100 * (iteration / float(
        total)))
24    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
```

```
26     print(f"\033[1;{print_color};48m {prefix} |{bar}| {percent}% ETA: {suffix} \n
      \033[00m", end = printEnd)
28 def startTest(model_path,data_path):
      # load model
30     MODEL_ROOT_DIR = model_path
      DATA_ROOT_DIR = data_path
32     print('loading the model')
      tStart = timeit.default_timer()
34     model = K.models.load_model(os.path.join(MODEL_ROOT_DIR))

36     tElapsed = timeit.default_timer()-tStart
      print('model loaded in %f [s].' % tElapsed)
38

40     # testing the images
      print('testing input images')
42     test_cnt = 0
      tStart = timeit.default_timer()
44     total_accuracy = 0
      total_correct = 0

46     imageList = os.listdir(DATA_ROOT_DIR)
48     random.shuffle(imageList)

50     for test_filename in imageList:
          test_cnt+=1

52

54     test_image_blob = Image.open(os.path.join(DATA_ROOT_DIR,test_filename)).
          convert('RGB')

56     test_image_blob_res = test_image_blob.resize((224,224), Image.BILINEAR)
      test_image_blob_res = np.asarray(test_image_blob_res, dtype='float32')

58

      test_input_tensor = test_image_blob_res/127.5
60     test_input_tensor -= 1

62

      [test_output_class] = model.predict(test_input_tensor[np.newaxis,:])
64
```

```

total_accuracy += np.max(test_output_class)
66 if CLASSES[np.argmax(test_output_class)][:1] == test_filename[:1] :
    total_correct += 1
68
70 if CLASSES[np.argmax(test_output_class)][:1] == test_filename[:1] :
    print_color = 32
72 else:
    print_color = 31
74
eta = (timeit.default_timer() - tStart) * (len(imageList) / test_cnt - 1)
76 hours, rem = divmod(eta, 3600)
minutes, seconds = divmod(rem, 60)
78 eta = "{:0>2}:{:0>2}:{:02.0f}".format(int(hours),int(minutes),seconds)
printProgressBar(test_cnt, len(imageList), print_color, length = 50,
    suffix = eta)
80 print(f"\033[1;{print_color};48m input: %s \n output class: %s %f\n
    accuracy: %f \n\033[00m"%(os.path.join(DATA_ROOT_DIR,test_filename),
    CLASSES[np.argmax(test_output_class)], np.max(test_output_class),
    total_correct/test_cnt))

82
tElapsed = timeit.default_timer() - tStart
84 print('testing done in %f [s] per image'%(tElapsed / test_cnt))
print('model uncertainty %f'%(total_accuracy / test_cnt))
86 print('model accuracy %f'%(total_correct / test_cnt))

88 def main():
    ap = argparse.ArgumentParser()
90 ap.add_argument('-m', '--model_path', type=str, default='./model/quantized/
    quantized_model_jpg.h5', help='Full path of floating-point model. Default
    is ./model/quantized/quantized_model_jpg.h5')
ap.add_argument('-d', '--data_path', type=str, default='./dataset/data/jpg/',
    help='Path of the images directory. Default is ./dataset/data/jpg/')
92 args = ap.parse_args()

94 print('\n-----')
print ('Command line options:')
96 print (' --model_path : ', args.model_path)
print (' --data_path : ', args.data_path)
98 print('-----\n')

```

```
100     startTest(args.model_path,args.data_path)
102 if __name__ == "__main__":
    main()
```

## A.2 TFRecord creation script

```
'''
2   Copyright 2020 Xilinx Inc.
   Licensed under the Apache License, Version 2.0 (the "License");
4   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at
6       http://www.apache.org/licenses/LICENSE-2.0
   Unless required by applicable law or agreed to in writing, software
8   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
10  See the License for the specific language governing permissions and
   limitations under the License.
12
   Author: Mark Harvey
14  Modified by: gnaggi
'''
16
18  import os
   import argparse
20  import zipfile
   import random
22  import shutil
   import math
24  from tqdm import tqdm
26
   # Silence TensorFlow messages
   os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
28  import tensorflow as tf
30
   DIVIDER = '-----'
32
```

```
34 def _bytes_feature(value):
    '''Returns a bytes_list from a string / byte'''
36     if isinstance(value, type(tf.constant(0))):
        value = value.numpy() # BytesList won't unpack a string from an
            EagerTensor.
38     return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

40
42 def _float_feature(value):
    '''Returns a float_list from a float / double'''
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))
44

46 def _int64_feature(value):
    ''' Returns an int64_list from a bool / enum / int / uint '''
48     return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))

50
52 def _calc_num_shards(img_list, img_shard):
    ''' calculate number of shards'''
    last_shard = len(img_list) % img_shard
54     if last_shard != 0:
        num_shards = (len(img_list) // img_shard) + 1
56     else:
        num_shards = (len(img_list) // img_shard)
58     return last_shard, num_shards

60
62 def write_tfrec(tfrec_filename, image_dir, img_list, image_type):
    ''' write TFRecord file '''
64
    with tf.io.TFRecordWriter(tfrec_filename) as writer:
66
        for img in img_list:
68
            class_name = img[:1]
70             if class_name == 'c':
                label = 0
72             elif class_name == 'l':
                label = 1
```

```
74     else:
75         label = 2
76     filePath = os.path.join(image_dir, img)
77
78     # read the JPEG source file into a tf.string
79     image = tf.io.read_file(filePath)
80
81     # get the shape of the image from the JPEG file header
82     if image_type == "png":
83         image_shape = tf.image.decode_png(image).shape
84     elif image_type == "jpg":
85         image_shape = tf.io.extract_jpeg_shape(image, output_type=tf.
86             dtypes.int32)
87
88     # features dictionary
89     feature_dict = {
90         'label' : _int64_feature(label),
91         'height': _int64_feature(image_shape[0]),
92         'width' : _int64_feature(image_shape[1]),
93         'chans' : _int64_feature(image_shape[2]),
94         'image' : _bytes_feature(image)
95     }
96
97     # Create Features object
98     features = tf.train.Features(feature = feature_dict)
99
100    # create Example object
101    tf_example = tf.train.Example(features=features)
102
103    # serialize Example object into TFRecord file
104    writer.write(tf_example.SerializeToString())
105
106    return
107
108
109
110 def make_tfrec(dataset_dir,tfrec_dir,img_shard, image_type):
111     # make a list of all images
112     imageList = os.listdir(os.path.join(dataset_dir,''))
```

```
114 # make lists of images according to their class
116 coastImages=[]
118 landImages=[]
118 waterImages=[]
120 for img in imageList:
120     class_name = img[:1]
122     if class_name == 'c':
122         coastImages.append(img)
124     elif class_name == 'l':
124         landImages.append(img)
126     else:
126         waterImages.append(img)
128 # define train/test split as 80:20
128 lengths = [len(coastImages), len(landImages), len(waterImages)]
130 split = int(min(lengths) * 0.2)
130 equilizer = min(lengths)
132
132 random.shuffle(landImages)
134 random.shuffle(coastImages)
134 random.shuffle(waterImages)
136
136 landImages = landImages[:equilizer]
138 coastImages = coastImages[:equilizer]
138 waterImages = waterImages[:equilizer]
140
140 assert(len(coastImages)==len(landImages)==len(waterImages)), 'Number of
142     images in each class do not match'
142
144
144 testImages = coastImages[:split] + waterImages[:split] + landImages[:split]
146 trainImages = coastImages[split:] + waterImages[split:] + landImages[split:]
146
148 print('Train images: ',len(trainImages))
148 print('Test images: ',len(testImages))
150
150 ''' Test TFRecords '''
152 print('Creating test TFRecord files...')
```

```
154 # how many TFRecord files?
    last_shard, num_shards = _calc_num_shards(testImages, img_shard)
156 print (num_shards,'TFRecord files will be created.')

158 if (last_shard>0):
    print ('Last TFRecord file will have',last_shard,'images.')
160

# create TFRecord files (shards)
162 start = 0
    for i in tqdm(range(num_shards)):
164         tfrec_filename = 'test_'+str(i)+'.tfrecord'
            write_path = os.path.join(tfrec_dir, tfrec_filename)
166         if (i == num_shards-1):
            write_tfrec(write_path, dataset_dir+'', testImages[start:],image_type)
168         else:
            end = start + img_shard
170             write_tfrec(write_path, dataset_dir+'', testImages[start:end],
                image_type)
            start = end

172

''' Training TFRecords '''
174 print('Creating training TFRecord files...')

176 # how many TFRecord files?
    last_shard, num_shards = _calc_num_shards(trainImages, img_shard)
178 print (num_shards,'TFRecord files will be created.')
    if (last_shard>0):
180         print ('Last TFRecord file will have',last_shard,'images.')

182 # create TFRecord files (shards)
    start = 0
184     for i in tqdm(range(num_shards)):
        tfrec_filename = 'train_'+str(i)+'.tfrecord'
186         write_path = os.path.join(tfrec_dir, tfrec_filename)
            if (i == num_shards-1):
188                 write_tfrec(write_path, dataset_dir+'', trainImages[start:],image_type
                    )
            else:
190                 end = start + img_shard
                    write_tfrec(write_path, dataset_dir+'', trainImages[start:end],
                        image_type)
```

```

192         start = end

194     print('\nDATASET PREPARATION COMPLETED')
194     print(DIVIDER,'\n')

196

198     return

200

202 def run_main():

204     # construct the argument parser and parse the arguments
204     ap = argparse.ArgumentParser()
204     ap.add_argument('-d', '--dataset_dir', type=str, default='dataset', help='
204         path to dataset images')
206     ap.add_argument('-t', '--tfrec_dir', type=str, default='tfrecords', help='
206         path to TFRecord files')
206     ap.add_argument('-s', '--img_shard', type=int, default=1000, help='Number of
206         images per shard. Default is 1000')
208     ap.add_argument('-i', '--img_type', type=str, default='jpg', help='Type of
208         image codification (jpg or png). Default is jpg')
208     args = ap.parse_args()

210

210     print('\n'+DIVIDER)
212     print('DATASET PREPARATION STARTED..')
212     print('Command line options:')
214     print (' --dataset_dir : ',args.dataset_dir)
214     print (' --tfrec_dir : ',args.tfrec_dir)
216     print (' --img_shard : ',args.img_shard)
216     print (' --img_type : ',args.img_type)

218

218     make_tfrec(args.dataset_dir,args.tfrec_dir,args.img_shard, args.img_type)

220

222 if __name__ == '__main__':
    run_main()

```

### A.3 Dataset Utility functions script

```

'''
2   Copyright 2020 Xilinx Inc.
   Licensed under the Apache License, Version 2.0 (the "License");

```

```

4 | you may not use this file except in compliance with the License.
   | You may obtain a copy of the License at
6 |     http://www.apache.org/licenses/LICENSE-2.0
   | Unless required by applicable law or agreed to in writing, software
8 | distributed under the License is distributed on an "AS IS" BASIS,
   | WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
10 | See the License for the specific language governing permissions and
   | limitations under the License.
12 | '''
   |
14 | '''
   | Utility functions for tf.data pipeline
16 | '''
   |
18 | '''
   | Author: Mark Harvey, Xilinx Inc
20 | '''
   |
   | import os
22 | import sys
   | import cv2
24 |
   | # Silence TensorFlow messages
26 | os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
   |
28 | import tensorflow as tf
   |
30 |
   | def parser(data_record):
32 |     ''' TFRecord parser '''
   |
34 |     feature_dict = {
   |         'label' : tf.io.FixedLenFeature([], tf.int64),
36 |         'height': tf.io.FixedLenFeature([], tf.int64),
   |         'width' : tf.io.FixedLenFeature([], tf.int64),
38 |         'chans' : tf.io.FixedLenFeature([], tf.int64),
   |         'image' : tf.io.FixedLenFeature([], tf.string)
40 |     }
   |     sample = tf.io.parse_single_example(data_record, feature_dict)
42 |     label = tf.cast(sample['label'], tf.int32)
   |
44 |     h = tf.cast(sample['height'], tf.int32)

```

```

w = tf.cast(sample['width'], tf.int32)
46 c = tf.cast(sample['chans'], tf.int32)
image = tf.io.decode_image(sample['image'], channels=3)
48 image = tf.reshape(image, [h,w,3])

50 return image, label

52
def resize_crop(x,y,h,w):
54     '''
Image resize & random crop
56 Args: Image and label
Returns: augmented image and unchanged label
58     '''
rh = int(h *1.2)
60 rw = int(w *1.2)
x = tf.image.resize(x, (rh,rw), method='bicubic')
62 x = tf.image.random_crop(x, [h, w, 3], seed=42)
return x,y
64

66 def augment(x,y):
'''
68 Image augmentation
Args: Image and label
70 Returns: augmented image and unchanged label
'''
72 x = tf.image.random_flip_left_right(x, seed=42)
x = tf.image.random_brightness(x, 0.1, seed=42)
74 x = tf.image.random_contrast(x, 0.9, 1.1, seed=42)
x = tf.image.random_saturation(x, 0.9, 1.1, seed=42)
76 return x, y

78
def normalize(x,y):
80     '''
Image normalization
82 Args: Image and label
Returns: normalized image and unchanged label
84     '''
# Convert to floating-point & scale to range -1.0 -> 1.0

```

```
86     x = tf.cast(x, tf.float32) * (1 / 127.5) - 1
      return x, y
88
90
91 def input_fn_trn(tfrec_dir, batchsize, height, width):
92     '''
93     Dataset creation and augmentation for training
94     '''
95     tfrecord_files = tf.data.Dataset.list_files('{}/*train_*.tfrecord'.format(
96         tfrec_dir), shuffle=True)
97     dataset = tf.data.TFRecordDataset(tfrecord_files)
98     dataset = dataset.map(parser, num_parallel_calls=tf.data.experimental.
99         AUTOTUNE)
100     dataset = dataset.map(lambda x,y: resize_crop(x,y,h=height,w=width),
101         num_parallel_calls=tf.data.experimental.AUTOTUNE)
102     dataset = dataset.batch(batchsize, drop_remainder=False)
103     dataset = dataset.map(augment, num_parallel_calls=tf.data.experimental.
104         AUTOTUNE)
105     dataset = dataset.map(normalize, num_parallel_calls=tf.data.experimental.
106         AUTOTUNE)
107     dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
108     dataset = dataset.repeat()
109     return dataset
110
111 def input_fn_test(tfrec_dir, batchsize, height, width):
112     '''
113     Dataset creation and augmentation for test
114     '''
115     tfrecord_files = tf.data.Dataset.list_files('{}/*test_*.tfrecord'.format(
116         tfrec_dir), shuffle=False)
117     dataset = tf.data.TFRecordDataset(tfrecord_files)
118     dataset = dataset.map(parser, num_parallel_calls=tf.data.experimental.
119         AUTOTUNE)
120     dataset = dataset.map(lambda x,y: resize_crop(x,y,h=height,w=width),
121         num_parallel_calls=tf.data.experimental.AUTOTUNE)
122     dataset = dataset.batch(batchsize, drop_remainder=False)
123     dataset = dataset.map(normalize, num_parallel_calls=tf.data.experimental.
124         AUTOTUNE)
125     dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

```

118     return dataset

120

121 def _parse_function(filename,label):
122     image_string = tf.io.read_file(filename)
123     image_decoded = tf.image.decode_png(image_string,channels=3)
124     image = tf.cast(image_decoded,tf.float32)
125     return image,label
126

127
128 def input_fn_quant(tfrec_dir,batchsize,height,width):
129     '''
130     Dataset creation and augmentation for quantization
131     The TFRecord file(s) must have > 1000 images
132     '''
133
134     tfrecord_files = tf.data.Dataset.list_files('{}/*test_0.tfrecord'.format(
135         tfrec_dir), shuffle=False)
136     dataset = tf.data.TFRecordDataset(tfrecord_files)
137     dataset = dataset.map(parser, num_parallel_calls=tf.data.experimental.
138         AUTOTUNE)
139     dataset = dataset.map(lambda x,y: resize_crop(x,y,h=height,w=width),
140         num_parallel_calls=tf.data.experimental.AUTOTUNE)
141     dataset = dataset.batch(batchsize, drop_remainder=False)
142     dataset = dataset.map(normalize, num_parallel_calls=tf.data.experimental.
143         AUTOTUNE)
144     dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
145     return dataset

```

## A.4 Finetuning and quantization script

```

import argparse
2 import os
import sys
4
os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
6
import tensorflow as tf
8 from tensorflow.keras.optimizers import Adam
from tensorflow_model_optimization.quantization.keras import vitis_quantize
10 from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler #

```

```
TensorBoard
12 from dataset_utils import input_fn_trn, input_fn_test
14 DIVIDER = '-----\n'
16 def fine_tuning_model(float_model, batchsize, tfrec_dir, chkpt_dir,
    fine_tuned_model, learnrate, epoch):
18     def step_decay(epoch):
19         """
20         Learning rate scheduler used by callback
21         Reduces learning rate depending on number of epochs
22         """
23         lr = learnrate
24         if epoch > 200:
25             lr /= 100
26         elif epoch > 10:
27             lr /= 10
28         return lr
30     print("Loading model")
31     model = tf.keras.models.load_model(float_model)
32     print("Model correctly loaded\n")
33     print(DIVIDER)
34
35     height = model.input_shape[1]
36     width = model.input_shape[2]
37
38     test_dataset = input_fn_test(tfrec_dir, batchsize, height, width)
39     train_dataset = input_fn_trn(tfrec_dir, batchsize, height, width)
40
41     #Training Call backs
42
43     chkpt_call = ModelCheckpoint(filepath=os.path.join(chkpt_dir, 'ft_model.h5'),
44                                 monitor='accuracy',
45                                 verbose=1,
46                                 save_best_only=True)
47
48     lr_scheduler_call = LearningRateScheduler(schedule=step_decay,
49                                               verbose=1)
```

```
50     callbacks_list = [chkpt_call, lr_scheduler_call] #tb_call
52     '''
54     End Training callbacks
56     '''
58     quantizer = vitis_quantize.VitisQuantizer(model,quantize_strategy='8bit_tqt')
59     model = quantizer.get_qat_model(init_quant=True,calib_dataset = test_dataset)
61
62     model.compile(
63         optimizer = Adam(learning_rate=learnrate),
64         loss='sparse_categorical_crossentropy',
65         metrics=['accuracy'])
66
67     model.fit( train_dataset,
68               epochs=epoch,
69               steps_per_epoch=17500//batchsize,
70               #validation_data=test_dataset,
71               #validation_steps=None,
72               callbacks=callbacks_list,
73               verbose=1)
74
75     print("Start model post-evaluation")
76     model.evaluate(test_dataset)
77
78     deployable_model = quantizer.get_deploy_model(model)
79     deployable_model.save(fine_tuned_model)
80
81 def main():
82
83     # construct the argument parser and parse the arguments
84     ap = argparse.ArgumentParser()
85     ap.add_argument('-f', '--float_model', type=str, default='float_model/f_model
86                     .h5', help='Full path of floating-point model. Default is float_model/
87                     f_model.h5')
88     ap.add_argument('-ft', '--fine_tuned_model', type=str, default='
89                     fine_tuned_model/ft_model.h5', help='Full path of quantized model.
90                     Default is quant_model/q_model.h5')
91     ap.add_argument('-b', '--batchsize', type=int, default=50, help='Batchsize
92                     for quantization. Default is 50')
```

```

86 ap.add_argument('-tfdir', '--tfrec_dir', type=str, default='tfrecords', help='
    Full path to folder containing TFRecord files. Default is tfrecords')
ap.add_argument('-c', '--chkpt_dir', type=str, default='fine_tuning', help='
    Full path to folder where to save fine tuning checkpoints. Default is
    fine_tuning')
88 ap.add_argument('-lr', '--learnrate', type=float, default=0.001, help='
    optimizer learning rate. Must be floating-point value. Default is 0.0001'
    )
ap.add_argument('-e', '--epochs', type=int, default=100, help='number of
    training epochs. Must be an integer. Default is 100.')
```

```

90 args = ap.parse_args()

92 print('\n-----')
93 print('TensorFlow version : ', tf.__version__)
94 print(sys.version)
95 print('-----')
```

```

96 print ('Command line options:')
97 print (' --float_model : ', args.float_model)
98 print (' --fine_tuned_model : ', args.fine_tuned_model)
99 print (' --batchsize : ', args.batchsize)
100 print (' --tfrec_dir : ', args.tfrec_dir)
101 print (' --chkpt_dir : ', args.chkpt_dir)
102 print (' --learnrate : ', args.learnrate)
103 print (' --epochs : ', args.epochs)
104 print('-----\n')
```

```

106
107 fine_tuning_model(args.float_model, args.batchsize, args.tfrec_dir, args.
    chkpt_dir, args.fine_tuned_model, args.learnrate, args.epochs)
108
```

```

110 if __name__ == "__main__":
    main()

```

## A.5 Compilation script

```

#!/bin/sh
2
# Copyright 2020 Xilinx Inc.
4 #
# Licensed under the Apache License, Version 2.0 (the "License");

```

## Vitis AI flow scripts

---

```
6 # you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
8 #
# http://www.apache.org/licenses/LICENSE-2.0
10 #
# Unless required by applicable law or agreed to in writing, software
12 # distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 # See the License for the specific language governing permissions and
# limitations under the License.
16
# Author: Mark Harvey, Xilinx Inc
18
if [ $1 = zcu102 ]; then
20     ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/ZCU102/arch.json
    echo "-----"
22     echo "COMPILING MODEL FOR ZCU102.."
    echo "-----"
24 elif [ $1 = u50 ]; then
    ARCH=/opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json
26     echo "-----"
    echo "COMPILING MODEL FOR ALVEO U50.."
28     echo "-----"
else
30     echo "Target not found. Valid choices are: zcu102, u50 ..exiting"
    exit 0
32 fi

34 if [$2 = "" ]; then
    echo "Model not found. Specify the quantized model to be compiled"
36     exit 0
fi

38
40 if [$3 = "" ]; then
    echo "Output directory not found. Specify the directory where to save the
        compiled model"
    exit 0
42 fi

44 if [$4 = "" ]; then
    echo "Model name not found. Specify the name of the model, it will be used"
```

```
        as name of the compiled model"
46     exit 0
fi
48
MODEL=$2
50 OUTPUT=$3
NAME=$4
52
compile() {
54     vai_c_tensorflow2 \
        --model $MODEL \
56     --arch $ARCH \
        --output_dir $OUTPUT \
58     --net_name $NAME
}
60
62 compile #2>&1 | tee compile.log
64
echo "-----"
66 echo "MODEL COMPILED"
echo "-----"
```

## APPLICATION CODE

---

```
import cv2
2 import numpy as np
import vart
4 import os
import xir
6 import sys
import argparse

8

from vaitrace_py import vai_tracepoint
10 import math

12

14 divider = "-----"

16 @vai_tracepoint
def imagePreprocessing(img_path):
    image = openImage(img_path)
18     image = convertImage(image)
    image = resizeImage(image)
20     image = rearrangeImage(image)
    return image

22

24 @vai_tracepoint
def openImage(img_path):
    image = cv2.imread(img_path)
26     return image

28 @vai_tracepoint
def convertImage(image):
30     img = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

```
    return img
32
@vai_tracepoint
34 def resizeImage(image):
    img = cv2.resize(image, (224,224))
36     return img

38 @vai_tracepoint
def rearrangeImage(image):
40     img = image / 127.5 - 1
    return img
42

44 def app(image_dir,model):
    test_images = os.listdir(image_dir)
46     total_run = len(test_images)

48     global out_q
    out_q = [None]*total_run
50

    model_graph = xir.Graph.deserialize(model)
52

    assert model_graph is not None, "Graph should not be None."
54     root_subgraph = model_graph.get_root_subgraph()
    assert root_subgraph is not None, "Failed to get root subgraph of input
        Graph"
56     if root_subgraph.is_leaf:
        subgraphs = []
58     child_subgraphs = root_subgraph.toposort_child_subgraph()
    assert (child_subgraphs is not None and len(child_subgraphs)) > 0
60     subgraphs = [
        cs
62         for cs in child_subgraphs
        if cs.has_attr('device') and cs.get_attr('device').upper() == "DPU
            "
64     ]

66     dpu_runner = [None]*2
    dpu_runner[0] = vart.Runner.create_runner(subgraphs[0], "run")
68     dpu_runner[1] = vart.Runner.create_runner(subgraphs[0], "run")
```

```
70     dpu_job_id = [None]*2

72     input_tensors = [None]*2
       output_tensors = [None]*2
74     input_ndim = [None]*2
       output_ndim = [None]*2
76     batchsize = [None]*2
       input_data = [None]*2
78     output_data = [None]*2

80     for img in range(total_run):
           path = os.path.join(image_dir, test_images[img])
82           image = imagePreprocessing(path)

84           input_tensors[img%2] = dpu_runner[img%2].get_input_tensors();
           output_tensors[img%2] = dpu_runner[img%2].get_output_tensors();
86           input_ndim[img%2] = tuple(input_tensors[img%2][0].dims)
           output_ndim[img%2] = tuple(output_tensors[img%2][0].dims)

88           batchsize = input_ndim[img%2][0]

90           input_data[img%2] = np.empty(input_ndim[img%2], dtype=np.float32,
           order='C')
           input_data[img%2][0,...] = image.reshape(input_ndim[img%2][1:])
92           output_data[img%2] = np.empty(output_ndim[img%2], dtype=np.float32
           , order='C')

94           dpu_job_id[img%2] = dpu_runner[img%2].execute_async(input_data[img
           %2], output_data[img%2])

96           if img % 2 == 0 and img != 0:
           dpu_runner[1].wait(dpu_job_id[1])
           else:
100            dpu_runner[0].wait(dpu_job_id[0])

102           out_q[img] = np.argmax((output_data[img%2][0]))

104

106     #Post processing
       classes = ['coast', 'land', 'water']
       correct = 0
```

## Application code

---

```
108     wrong = 0
109     print('Post-processing', len(out_q), 'images..')
110     for i in range(len(out_q)):
111         prediction = classes[out_q[i]]
112         ground_truth = test_images[i][:1]
113         if (ground_truth==prediction[:1]):
114             correct += 1
115         else:
116             wrong += 1
117     accuracy = correct/len(out_q)
118     print('Correct:%d, Wrong:%d, Accuracy:%.4f' %(correct,wrong,accuracy))
119     print (divider)
120
121
122 def main():
123     # construct the argument parser and parse the arguments
124     ap = argparse.ArgumentParser()
125     ap.add_argument('-d', '--image_dir', type=str, default='images', help='
126         Path to folder of images. Default is images')
127     ap.add_argument('-m', '--model', type=str, default='masati.xmodel', help='
128         Path of xmodel. Default is masati.xmodel')
129
130     args = ap.parse_args()
131
132     print(divider)
133     print ('Command line options:')
134     print (' --image_dir : ', args.image_dir)
135     print (' --model : ', args.model)
136
137     app(args.image_dir,args.model)
138
139 if __name__ == '__main__':
140     main()
```

# VITIS DPU CONFIGURATION FILE

---

```
2 //Setting the arch of DPU, For more details, Please read the PG338
4 /*===== Architecture Options =====*/
6 // |-----|
8 // | Support 8 DPU size
   // | It relates to model. if change, must update model
10 // +-----+
12 // | 'define B512
   // +-----+
14 // | 'define B800
   // +-----+
16 // | 'define B1024
   // +-----+
18 // | 'define B1152
   // +-----+
20 // | 'define B1600
   // +-----+
22 // | 'define B2304
   // +-----+
24 // | 'define B3136
   // +-----+
26 // | 'define B4096
   // +-----+
28 // |-----|
   // | If the FPGA has Uram. You can define URAM_EN parameter
30 // | if change, Don't need update model
```

## Vitis DPU configuration file

---

```
32 // +-----+
// | for zcu104 : 'define URAM_ENABLE
// +-----+
34 // | for zcu102 : 'define URAM_DISABLE
// |-----|
36
'define URAM_DISABLE
38
//config URAM
40 'ifndef URAM_ENABLE
    'define def_UBANK_IMG_N 5
42    'define def_UBANK_WGT_N 17
    'define def_UBANK_BIAS 1
44 'elsif URAM_DISABLE
    'define def_UBANK_IMG_N 0
46    'define def_UBANK_WGT_N 0
    'define def_UBANK_BIAS 0
48 'endif
50 // |-----|
// | You can use DRAM if FPGA has extra LUTs
52 // | if change, Don't need update model
// +-----+
54 // | Enable DRAM : 'define DRAM_ENABLE
// +-----+
56 // | Disable DRAM : 'define DRAM_DISABLE
// |-----|
58
'define DRAM_DISABLE
60
//config DRAM
62 'ifndef DRAM_ENABLE
    'define def_DBANK_IMG_N 1
64    'define def_DBANK_WGT_N 1
    'define def_DBANK_BIAS 1
66 'elsif DRAM_DISABLE
    'define def_DBANK_IMG_N 0
68    'define def_DBANK_WGT_N 0
    'define def_DBANK_BIAS 0
70 'endif
```

## Vitis DPU configuration file

---

```
72 // |-----|
// | RAM Usage Configuration
74 // | It relates to model. if change, must update model
// +-----+
76 // | RAM Usage High : 'define RAM_USAGE_HIGH
// +-----+
78 // | RAM Usage Low : 'define RAM_USAGE_LOW
// |-----|
80
'define RAM_USAGE_LOW
82
// |-----|
84 // | Channel Augmentation Configuration
// | It relates to model. if change, must update model
86 // +-----+
// | Enable : 'define CHANNEL_AUGMENTATION_ENABLE
88 // +-----+
// | Disable : 'define CHANNEL_AUGMENTATION_DISABLE
90 // |-----|
92 'define CHANNEL_AUGMENTATION_ENABLE
94 // |-----|
// | DepthWiseConv Configuration
96 // | It relates to model. if change, must update model
// +-----+
98 // | Enable : 'define DWCV_ENABLE
// +-----+
100 // | Disable : 'define DWCV_DISABLE
// |-----|
102
'define DWCV_ENABLE
104
// |-----|
106 // | Pool Average Configuration
// | It relates to model. if change, must update model
108 // +-----+
// | Enable : 'define POOL_AVG_ENABLE
110 // +-----+
// | Disable : 'define POOL_AVG_DISABLE
112 // |-----|
```

## Vitis DPU configuration file

---

```
114 'define POOL_AVG_ENABLE
116 // |-----|
116 // | support multiplication of two feature maps
118 // | It relates to model. if change, must update model
118 // +-----+
120 // | Enable : 'define ELEW_MULT_ENABLE
120 // +-----+
122 // | Disable : 'define ELEW_MULT_DISABLE
122 // |-----|
124
124 'define ELEW_MULT_DISABLE
126
126 // +-----+
128 // | RELU Type Configuration
128 // | It relates to model. if change, must update model
130 // +-----+
130 // | 'define RELU_RELU6
132 // +-----+
132 // | 'define RELU_LEAKYRELU_RELU6
134 // |-----|
136
136 'define RELU_LEAKYRELU_RELU6
138
138 // |-----|
138 // | DSP48 Usage Configuration
140 // | Use dsp replace of lut in conv operate
140 // | if change, Don't need update model
142 // +-----+
142 // | 'define DSP48_USAGE_HIGH
144 // +-----+
144 // | 'define DSP48_USAGE_LOW
146 // |-----|
148
148 'define DSP48_USAGE_HIGH
150
150 // |-----|
150 // | Power Configuration
152 // | if change, Don't need update model
152 // +-----+
```

## Vitis DPU configuration file

---

```
154 // | 'define LOWPOWER_ENABLE
// +-----+
156 // | 'define LOWPOWER_DISABLE
// |-----|
158
'define LOWPOWER_DISABLE
160
// |-----|
162 // | DEVICE Configuration
// | if change, Don't need update model
164 // +-----+
// | 'define MPSOC
166 // +-----+
// | 'define ZYNQ7000
168 // |-----|
170 'define MPSOC
```

# D

## PYOPENCL HOST APPLICATION CODE

---

```
import pyopencl as cl
2 from pyopencl import array
import numpy as np
4
if __name__ == "__main__":
6
    input_data = np.zeros(256,np.intc)
8    for i in range(256):
        input_data[i] = i
10
    ## Step #1. Obtain an OpenCL platform.
12    platform = cl.get_platforms()[0]
14
    ## Step #2. Obtain a device id for at least one device (accelerator).
    device = platform.get_devices()[0]
16
    ## Step #3. Create a context for the selected device.
18    context = cl.Context([device])
20
    ## Step #4. Create the accelerator program from source code.
    ## Step #5. Build the program.
22    ## Step #6. Create one or more kernels from the program functions.
    xclbin = open("/media/sd-mmcblk0p1/dpu.xclbin","rb",buffering=0).read()
24    program = cl.Program(context, [device], [xclbin]).build()
26
    ## Step #7. Create a command queue for the target device.
    queue = cl.CommandQueue(context)
28
    ## Step #8. Allocate device memory and move input data from the host to the
        device memory.
```

## Pyopencl host application code

---

```
30 mem_flags = cl.mem_flags
input_buf = cl.Buffer(context, mem_flags.READ_WRITE | mem_flags.COPY_HOST_PTR
    , hostbuf=input_data)
32
## Step #9. Associate the arguments to the kernel with kernel object.
34 ## Step #10. Deploy the kernel for device execution.
program.img_preprocessor(queue, input_data.shape, None, cl.cltypes.uint(64),
    input_buf)
36
print(input_data)
```

# BIBLIOGRAPHY

---

- [1] Maurizio Capra et al. “Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead”. In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/ACCESS.2020.3039858.
- [2] European Commission. *A definition of AI: Main Capabilities and Disciplines*. 2021. URL: [https://ec.europa.eu/newsroom/dae/document.cfm?doc\\_id=60419](https://ec.europa.eu/newsroom/dae/document.cfm?doc_id=60419) (visited on 10/15/2021).
- [3] Docker. *Use containers to Build, Share and Run your applications*. 2021. URL: <https://www.docker.com/resources/what-container> (visited on 11/15/2021).
- [4] Antonio-Javier Gallego, Antonio Pertusa, and Pablo Gil. “Automatic Ship Classification from Optical Aerial Images with Convolutional Neural Networks”. In: *Remote Sensing* 10.4 (2018). ISSN: 2072-4292. DOI: 10.3390/rs10040511.
- [5] gewuek. *Vitis AI Custom Platform Development*. June 5, 2020. URL: [https://github.com/gewuek/vitis\\_ai\\_custom\\_platform\\_flow](https://github.com/gewuek/vitis_ai_custom_platform_flow).
- [6] David H. K. Hoe. *FPGA research*. 2021. URL: <https://evergreen.loyola.edu/dhhoe/www/HoeResearchFPGA.htm> (visited on 10/28/2021).
- [7] IBM. *Neural Networks*. 2021. URL: <https://www.ibm.com/cloud/learn/neural-networks#toc-neural-net-u3voPJVU> (visited on 10/20/2021).
- [8] Keras. *Creating TFRecords*. Feb. 27, 2021. URL: [https://keras.io/examples/keras\\_recipes/creating\\_tfrecords/](https://keras.io/examples/keras_recipes/creating_tfrecords/).
- [9] Khronos. *The OpenCL™ Specification*. Version 3.0.10. Nov. 19, 2021. URL: [https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL\\_API.pdf](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf).

## BIBLIOGRAPHY

---

- [10] Dusko Lukac, Miljana Milic, and Jelena Nikolic. “From Artificial Intelligence to Augmented Age An Overview”. In: *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. 2018, pp. 100–103. DOI: 10.1109/ZINC.2018.8448793.
- [11] Xilinx. *DPUCZDX8G for Zynq Ultrascale+ MPSoCs*. Version 3.3. July 22, 2021. 62 pp. URL: [https://www.xilinx.com/support/documentation/ip\\_documentation/dpu/v3\\_3/pg338-dpu.pdf](https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_3/pg338-dpu.pdf).
- [12] Xilinx. *Getting Started with RTL Kernels*. Nov. 19, 2021. URL: [https://github.com/Xilinx/Vitis-Tutorials/tree/2021.2/Hardware\\_Acceleration/Feature\\_Tutorials/01-rtl\\_kernel\\_workflow](https://github.com/Xilinx/Vitis-Tutorials/tree/2021.2/Hardware_Acceleration/Feature_Tutorials/01-rtl_kernel_workflow).
- [13] Xilinx. *TensorFlow2 and Vitis AI design flow*. Version 1.4. Oct. 20, 2021. URL: [https://github.com/Xilinx/Vitis-AI-Tutorials/tree/master/Design\\_Tutorials/08-tf2\\_flow](https://github.com/Xilinx/Vitis-AI-Tutorials/tree/master/Design_Tutorials/08-tf2_flow).
- [14] Xilinx. *Vitis AI user guide*. Version 1.4. July 22, 2021. 154 pp. URL: [https://www.xilinx.com/support/documentation/sw\\_manuals/vitis\\_ai/1\\_4/ug1414-vitis-ai.pdf](https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_4/ug1414-vitis-ai.pdf).
- [15] Xilinx. *Vitis Custom Embedded Platform Creation Example on ZCU104*. Nov. 15, 2021. URL: [https://github.com/Xilinx/Vitis-Tutorials/blob/2021.2/Vitis\\_Platform\\_Creation/Introduction/02-Edge-AI-ZCU104/README.md](https://github.com/Xilinx/Vitis-Tutorials/blob/2021.2/Vitis_Platform_Creation/Introduction/02-Edge-AI-ZCU104/README.md).
- [16] Xilinx. *ZCU102 Evaluation Board*. Version 1.6. June 12, 2019. 125 pp. URL: [https://www.xilinx.com/support/documentation/boards\\_and\\_kits/zcu102/ug1182-zcu102-eval-bd.pdf](https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf).
- [17] Xilinx. *Zynq UltraScale+ MPSoC*. 2021. URL: <https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html> (visited on 11/18/2021).