POLITECNICO DI TORINO

Master's Degree in Electronic Engineer

Master's Degree Thesis

FPGA IMPLEMENTATION OF A VEHICLES DETECTOR

Supervisor

Prof. Maurizio MARTINA

Co-supervisor

Ing. Claudio SINISI

Candidate

Davide ALTAMORE

July 2024

Summary

The growing demand for self-driving vehicles in the market leads to investigate new optimized and higher performance technologies. Obstacle detection on the road is one of the key tasks for this purpose and it requires very short response times to avoid any possible risk.

This thesis work is focused on the development of an artificial intelligence model able to run on one of the over-mentioned feasible technologies, an FPGA-based device. The exploited model is YOLOv5 and it has been chosen to satisfy the object detection task. The expected final pipeline is made of an input sensor to acquire the traffic images and an output display to visualize them with the vehicles surrounded by rectangles generated by the inference process. This last operation is performed by the FPGA-based system that must be properly built by software and hardware perspectives.

The whole design is supported by the AMD software tools, such as Vitis AI, Vivado, Vitis and PetaLinux. While the target FPGA-based board is part of the AMD ones and it is known as Kria KV260 Vision AI Starter Kit.

The design is entirely organized into three macro-steps, as the diagram 1 shows.

Initially, two introductory chapters describe the theoretical background. The first one gives an overview about Artificial Intelligence and Autonomous Driving with an additional look at the hardware platforms capable to run AI models. The second chapter focuses on the AI model preparation. After having briefly explained the YOLOv5 neural network architecture, the training process is described together with the used dataset (Kitti), the analysed metrics and the validation of the final trained results.

All this concepts are a strong base for the next chapters. Indeed, the third one is about the **Model development** phase, that is the macro-step leading to a working compiled model. The Vitis AI environment with its main tools is presented. The *quantization* converts the 32-bit floating point model into a 8-bit integer one and, then, the *compilation* translates the model into the XIR-based format. This means having a deployable model, since its format is readable by the board's DPU, i.e.

the base IP of the FPGA-based AMD devices.

The fourth chapter is focused on the remaining two macro-steps. After a presentation about the entire pipeline, from the frames acquisition sensor to the visualization of the inference results, the **Platform development** is analysed. The FPGA-device boards typically need of a platform that enable all the necessary hardware elements available on the board. The advantage of the Kria KV260 board's use is that many ready applications are available and open-source. They gives the possibility to use their Vivado and Vitis base scripts as starting point for this thesis design. Vivado is used to build the Xilinx platform and Vitis to build the overlay and join together in a single xclbin file these two components. All these firmware elements are then loaded into a Linux image, properly built by the AMD PetaLinux tool.

Ready the hardware side of this design, the **Application Development** is the next step to directly run all the operations: pre-processing, inference and output display. It consists of easily writing an application code with the support of the GStreamer framework, a series of plug-ins to manage the multimedia pipelines as this one.

The last chapter analyses the final execution of the AI inference on the target. All the elements, i.e. the Linux image with platform firmware and the model with all the supporting configuration files, are deployed on the Kria KV260 board that, at this point, will be able to detect the vehicles on the input frames.

The several design layers, carried out in the whole thesis work, introduce critical issues that can compromise the final working. The choice of higher performance training machine, better quantization approaches and less corrupting pre-processing lead to more optimized results.

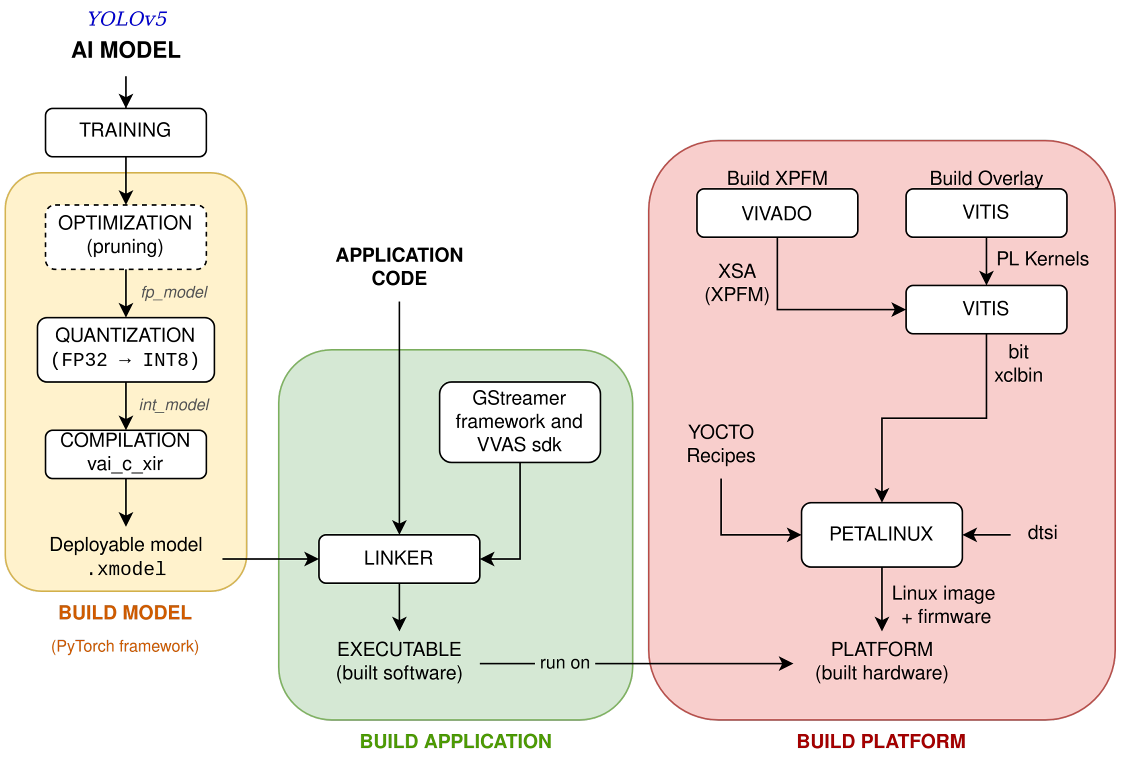


Figure 1: Summary of the three macro-steps of this project.

Acknowledgements

In conclusione di questo percorso sento la necessità di dedicare qualche parola alle persone che mi hanno supportato.

Ci tengo a ringraziare, innanzitutto, il mio relatore Maurizio Martina, per avermi dato l'opportunità di entrare in contatto con una nuova realtà con cui interfacciarmi per questa esperienza di tesi e per la disponibilità mostrata durante tutto il percorso.

Un grazie speciale va anche al mio tutor Claudio Sinisi per avermi guidato in questa nuova esperienza aziendale e a tutte le altre persone incontrate nel cammino.

Fondamentali tutta la mia famiglia e i miei amici, avete supportato le mie scelte fin dal primo giorno appoggiandomi e accompagnandomi fino alla fine. Senza di voi non sarei riuscito a portare a termine questa importante esperienza.

Table of Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence

ML Machine Learning

DL Deep Learning

AV Autonomous Vehicle

ADAS Advanced Driver Assistance Systems

SAE Society of Automotive Engineers

CV Computer Vision

NN Neural Network

DNN Deep Neural Network

CNN Convolutional Neural Network

FPGA Field Programmable Gate Array

ASIC Application-Specific Integrated Circuit

MAC Multiply-Accumulate

CLB Configurable Logic Block

PS Processing System

PL Programmable Logic

DPU	Deep learning Processing Unit
AXI	Advanced eXtensible Interface
SoC	System-on-Chip
ReLU	Rectified Linear Unit
YOLO	You Only Look Once
SiLU	Sigmoid Linear Unit
IP	Intellectual Property
VAI	Vitis AI
IoU	Intersection over Union
P	Precision
R	Recall
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
mAP	mean Average Precision
FPS	Frame per second
GD	Gradient Descent
OFA	Once-for-All
PTQ	Post-training quantization
QAT	Quantization-Aware Training

XIR Xilinx Intermediate representation

TRD Targeted reference design

ISP Image Signal Processor

XPFM Xilinx Platform

XSA Xilinx Support Archive

XO Xilinx Object

XSCT Xilinx Software Command-Line Tool

BSP Board Support Package

DTC Device Tree Compiler

VVAS Vitis Video Analytics SDK

Chapter 1

Introduction

This introductory chapter aims to contextualise the following thesis project by providing a historical and technical background to the underlying concepts. After clarifying what a *neural network* is and how it can be deployed in the context of *autonomous driving*, a brief discussion about the best possible *hardware platform* is carried out.

1.1 Artificial Intelligence

1.1.1 AI history

Artificial Intelligence does not have a single universally recognized definition, as it is a very broad concept and is still in the middle of its historical development. One of the most accredited definition is given by the European Commission[1]: "Artificial intelligence (AI) refers to systems that display intelligent behaviour by analysing their environment and taking actions - with some degree of autonomy - to achieve specific goals". This is a strongly general view that opens up the possibility of several application fields, [1]"AI-based systems can be purely software-based, acting in the virtual world (e.g. voice assistants, image analysis software, search engines, speech and face recognition systems) or AI can be embedded in hardware devices (e.g. advanced robots, autonomous cars, drones or Internet of Things applications)". The interest of this thesis work focuses on the latter devices.

"The brain's last stand" are the historical words used to describe the defeat of the grandmaster Garry Kasparov in 1997 by Deep Blue, a chess-playing expert system running on an IBM's computer. This is a milestone of the Artificial Intelligence history, from there on the rise of the AI was almost physiological in the scientific world and not only. Indeed when the AI systems started to spread even out of the research labs, particularly in the business environment, big societies like Google, IBM, Microsoft and Facebook have started to invest in these solutions. An example is the 2014 Google's acquisition of the British research lab DeepMind, which has developed a program, AlphaGo, able to defeat Lee Sedol, master of the ancient Asian board game Go.

For those years the Deep Blue and AlphaGo cases were a breakthrough, however, like frequently happening in AI history these events give the occasion to deeply understand the human intelligence working, leading to consider this computer AI skills as simple calculations.

The 1.1 figure shows as the AI gathers momentum outside the lab since 2010s. One of the main causes of this diffusion is the so called *Machine Learning* (ML). It is defined as an area of the artificial intelligence that "focuses on using data and algorithms to mimic the way humans learn, with the goal of steadily improving accuracy"[2]. It means that a ML model is not built directly to answer a task but it previously requires to learn from data which is the behaviour to achieve the best accuracy for its next predictions.

To further improve the features of these ML models, the *Deep Learning* (DL) was born. It is a subset of ML where the human intervention is almost removed. In the ML algorithms the humans act by pre-processing the data to make them

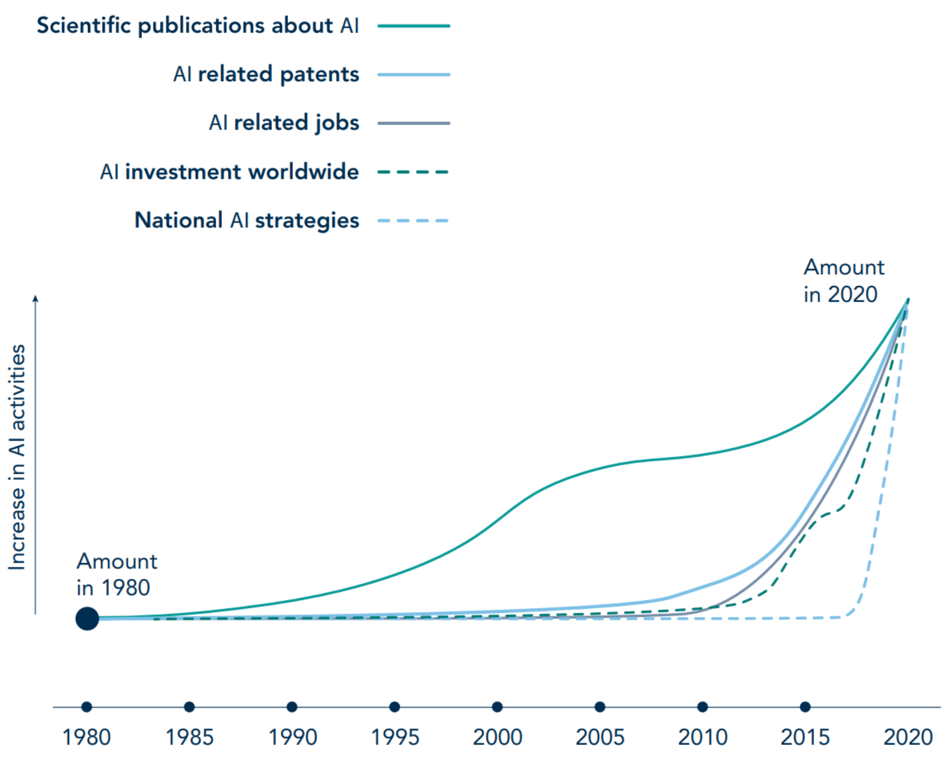


Figure 1.1: AI related activities trend[3].

structured (not necessarily completely labeled), while the DL ones automate the extraction of features' patterns through processes of backpropagation. This allows the parameters of the model itself to be adjusted and its accuracy to be improved[4].

1.1.2 Neural Networks fundamentals

As can be presumed by the previous section 1.1.1, the main task of a neural network is to emulate a human brain. To implement a so complex structure the best way is to build a simplified model able to emulate its behaviour. This AI model is called *neural network* and is arranged as follows. Its constituent element, known as *perceptron*, is a mathematical model of a neuron (figure 1.2):

- **Neuron** - Its cell body (*soma*) processes the signals received from other neurons via the *dendrites*. If the input voltage overcomes a threshold, a pulse is generated and ready to be transmitted to the other neurons. This happens with the support of an *axon* that drives the impulse away from the cell body to the *synapses*, connected to other neurons.

- **Perceptron** - By following the neuron behaviour, the perceptron weights each input signal ($x_i \cdot w_i$) and add together the results. The output is then biased adding an offset (b) and finally the single perceptron's result comes from the application of a so called *activation function* ($f(\cdot)$).

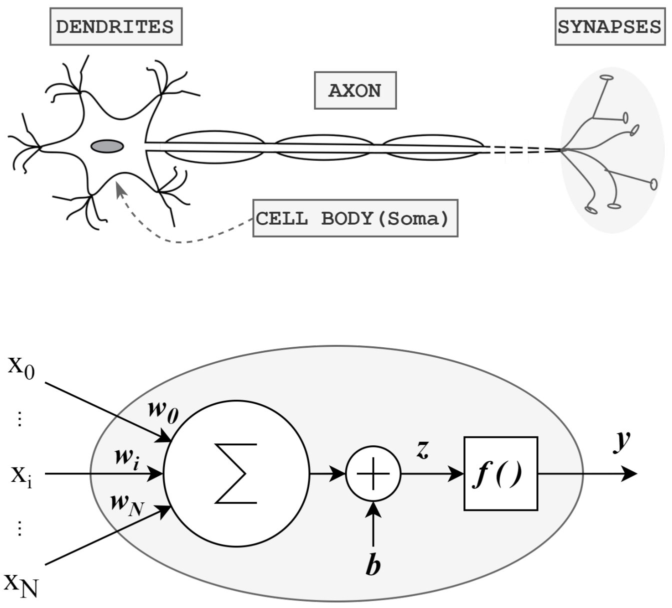


Figure 1.2: A simple scheme of a human neuron (up) compared to a perceptron, its mathematical model (down).

So the perceptron's behaviour can be mathematically expressed as:

$$z = \sum_{i}^{N}(w_i x_i) + b$$
$$y = f(z)$$

Once characterized its building blocks, it is possible to understand how a *neural network* is composed. It is a graph, where each node is a perceptron, exactly like a human brain is made by connected neurons. A neural network usually consists of more layers (group of concurrent nodes) that are distinguished into: input layer that collects the input data (known as features), output layer that generates the outputs (known as predictions) and the hidden layers that lie between the first two. If more than 3 hidden layers are placed, that one is a Deep Neural Network (example at figure 1.3) and can be further classified according to its composition.

Supervised Learning A neural network is able to perform the task for which it was designed by setting properly its parameters (weights, bias and activation

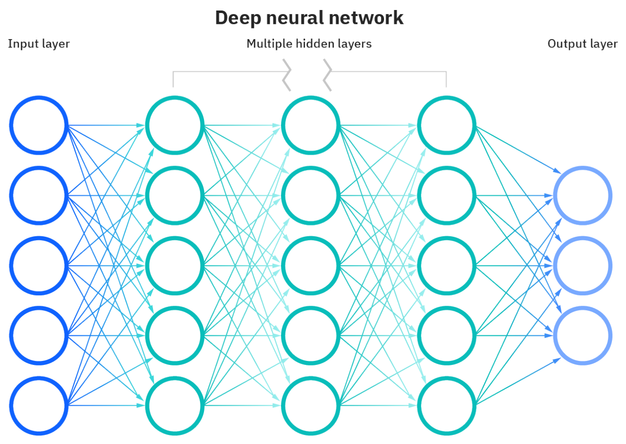


Figure 1.3: Deep Neural Network representation[5]

function). This phase is called *training* and it is the main step of the Machine Learning approach. In accordance with the level of human intervention a learning can be classified as Supervised, Unsupervised or Reinforcement. The used one in this thesis work and also the less computational-intensive is the first one.

The Supervised Learning, summarized at figure 1.4, requires the training data to be *labeled*. It means to provide together with the input data, also the correct corresponding predictions that the model must provide in that case. This approach allows to build a cost/error function, fundamental for the tuning mechanism of the model's parameters (like backpropagation and gradient-descent).

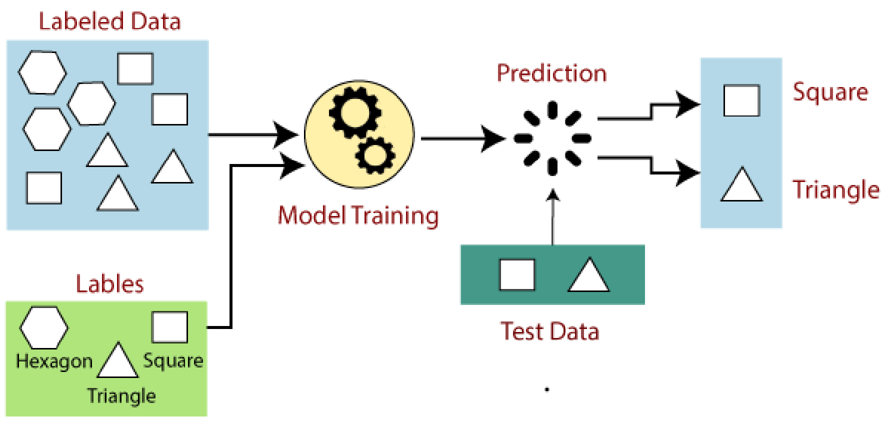


Figure 1.4: Supervised Machine Learning approach[6]

1.2 Autonomous Driving

A brief introduction about the Autonomous Vehicles (AVs) may be helpful considering the final objective of this project.

The AVs have emerged separately by the AI. The first developed systems were called ADAS (Advanced Driver Assistance Systems), they involve features such as automatic lane keeping, parking assistance and cruise control. Their main objective is to enhance safety by reducing errors associated with human drivers[7]. Almost all recently produced vehicles provide ADAS support, however this is just a very low level of driving autonomy. The Society of Automotive Engineers (SAE) has identified a taxonomy with 6 levels[1] that is the most cited source for driving automation. It was published on [8] and a brief summary is 1.5, showing as ADAS involve just the first 2 levels of autonomy.

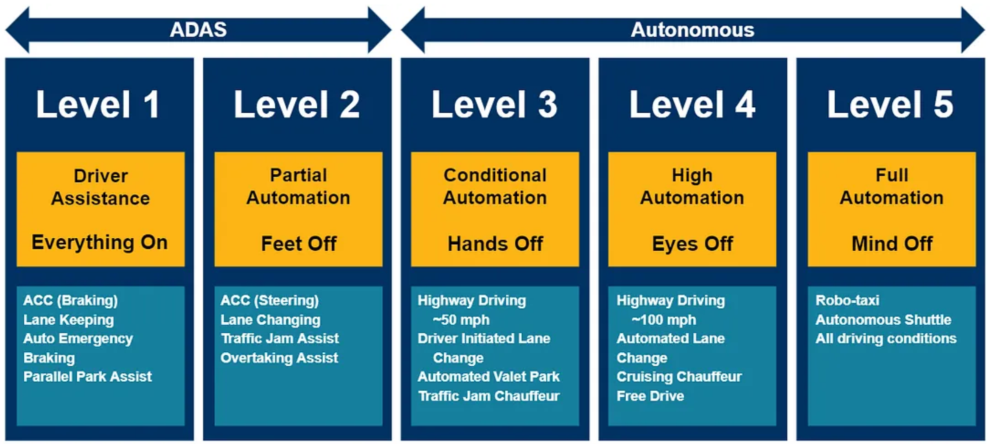


Figure 1.5: SAE's levels of automation[9].

The Artificial Intelligence starts to be used from level 3 and fully used at 5, where it must replace the human driver in every its task. As illustrated in the previous section 1.1.1, in recent years some cases (e.g. AlphaGo) have boosted the AI advancements leading to a natural cohesiveness between AI and specific application fields, like Autonomous Driving [7].

A foreseeable future is the use of *Full Automated* vehicles in well-defined environments, like industrial areas or airports[3]. Why not across the city roads? Many causes can be identified and they will be analysed later on. One of them is their unpredictability, it is still a problem too complex to be solved by an only-AI approach, especially because also human lives are involved. Hence, using AI as

[1]Level 0 means *No automation* - The driver has no support, thus he performs all driving tasks.

support and enhancement for the human capabilities today remains the most feasible solution.

Computer Vision, a field of AI, serves this purpose. It is a discipline focused on developing algorithms and models to analyze and extract insights from visual data, like images and videos. As for all the AI fields it tries to replicate a human capability, the visual perception[10]. Starting from these extracted information, the human driver or directly the car will be able to make consequent decisions, avoiding collisions and accidents.

Several functions are required by a self-driving car but just some of them are actually safety-critical, like steering, throttle and braking control. The AVs need mainly two information to handle these functions[11]:

- *Obstacle detection*: the surrounding area is analyzed to detect obstacles like pedestrians and vehicles.
- Traffic law enforcement: lane, traffic signs and traffic lights must be recognized to move around the roads respecting the traffic laws.

All these tasks, comprised other optional functions[2], are today implemented by Deep Learning techniques, particularly by *Convolutional Neural Networks* (CNNs). This implementation is the focus of this thesis work, so it will be explained then more in depth. However an important point can be observed now about its power requirements, that represents a further limitation to the *Full Automated* cars spreading.

MIT research on AVs power consumption The DL's computational complexity leads to a very high power consumption that could impact on driving range of AVs, i.e. on their buttery autonomy.

In a 2023 study of MIT researchers[12] some statistical models have been created to study the power consumption problem of self-driving cars. Supposed a scenario where the 95% of all vehicles are AVs, it is required a computer power consumption less than 1.2 kW to keep the global emissions under the quantity estimated for all existing data centers in 2018. This power efficiency is certainly not achievable with today's technology and it is estimated that it will not even be achievable by 2050 if 95% of cars become autonomous.

All this scenario is compounded by the use of deep neural networks. An explanatory comparison shows that Facebook's data centers execute 10^{18} inferences

[2]There are many less critical functions, like the windscreen swiping with a speed suitable with the detected rain, the door locking and the user information to warn about energy level, vehicle condition or also driver's drowsiness.

per day while one billions of autonomous vehicles[3] perform 21.6×10^{24} ones per day.

1.3 Hardware platform choice

1.3.1 Technology comparison

This section offers a brief comparison among the best hardware platforms for running neural networks:

- **CPU** - The *CPUs* are general-purpose processors typically used to handle most of the tasks of a computer. Their high flexibility and low costs make them widely used by ordinary users. However CPUs are not enough considering the DNNs training and inference requirements, tens of billions of MAC operations make these models extremely computer-intensive[13] for a general purpose processor with a low parallelism capability.
- **GPU** - The last mentioned point is the main reason that makes the *GPUs* preferable. They are special-purpose processors designed to render graphics and perform parallel operations on large data arrays. It means that they provide high parallelism and memory bandwidth, both features optimal for DNNs. GPUs are able to accelerate a lot of MAC operations with an higher energy efficiency than the CPUs. The MAC operations are fundamental for DL algorithms since matrix multiplications and convolutions are the basis of the DNNs.
- **FPGA** - As shown on the diagram 1.6, there are other two possible hardware platforms: FPGAs and ASICs. Paying with less flexibility a more energy efficiency is reached, mainly since the today's trend are IoT applications that involve a lot of sensor's data to process. The *FPGAs* can reach an energy efficiency of 10 times than GPUs, since they can parallelize the DL's concurrent tasks by using technique such as hardware partitioning and pipelining. Moreover the FPGA-based design have a deterministic latency, fundamental for car's systems[11].
- **ASIC** - Finally the *ASICs* are also more efficient than FPGAs but their less flexibility and higher NRE costs[4] cause a rarer and more thoughtful use.

[3]The AVs are considered driving for an hour per day and computing ten inferences at 60 Hz on each of the inputs of ten cameras.

[4]ASIC means application-specific integrated circuit, they are not re-programmable like the FPGAs and it causes Non-Recurring Engineering costs much higher than FPGA.

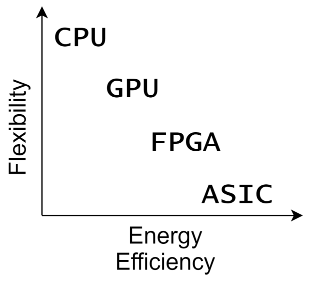


Figure 1.6: A simple view of Flexibility-Energy efficiency trade-off of hardware platforms[11].

Going more deeply, a discrimination (figure 1.7) can be made to emphasise a key aspect. The CPU and GPU can be labeled as *temporal architectures* since its processing elements (PEs) are not interconnected and perform just computational tasks (ALU), leaving control and memory centralized. While FPGA and ASIC are *spatial architectures*, each their PE performs both computational, memory and control task. However their main feature is that they are interconnected, allowing a data exchange among them. It happens differently from the temporal architectures where the data flow is temporally depending on the central memory[13].

Since the operations of each neural network's layer is known a priori, the spatial architectures can be made tailored and optimized for this application.

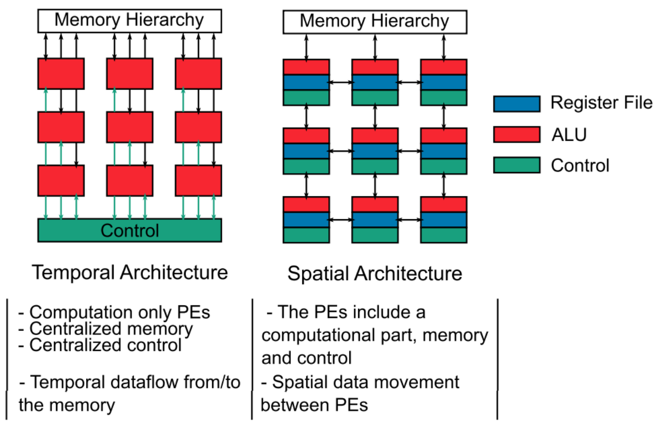


Figure 1.7: Temporal vs Spatial Architectures[13].

All these underlined aspects lead to a focus on the spatial architectures and mainly on FPGAs that allow a complete design workflow addressable without large non-recurring costs.

1.3.2 FPGA overview

An FPGA is an integrated circuit (figure 1.8) composed by *logic blocks*, known as CLBs (Configurable logic blocks), arranged in a two-dimensional array. They are interconnected via *wires* organized as horizontal and vertical routing channels. These wires are interrupted by *programmable switches* that, thanks to transistors driven by 1-bit RAMs, allow the CLBs to be interconnected in many ways[14], depending on 0/1 stored in those RAM cells. Moreover, to interface with the outside world *I/O blocks* are placed at the wires terminations.

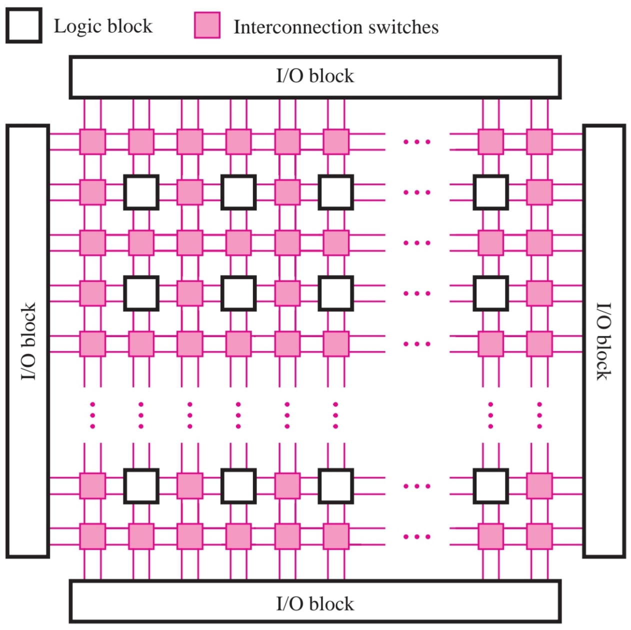


Figure 1.8: FPGA general scheme[14]

The logic blocks (CLBs) make possible to realize both combinational and sequential circuits. As shown on 1.9 figure, each block consists of a single *LUT* (Look Up Table) that works as a sort of memory containing a function. When the input bits are such that the function is activated, the output bit will be at 1, otherwise 0. It implements the digital electronics concept of the truth's table. A CLB has also a *D-FF*, it makes available the LUT's result at the next clock period, allowing a sequential behaviour of the FPGA.

The FPGAs, intended just as array of logic blocks, are not enough to implement the complexity of a DNN. A neural network only needs MAC operations, so an FPGA-based device can easily perform them. However the true bottleneck is caused by the memory accesses, since these simple operations act on a large quantity of

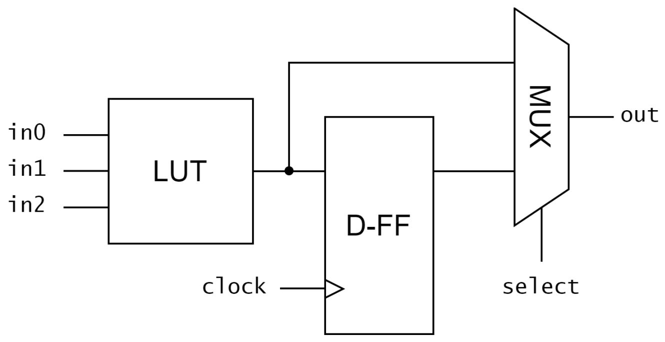


Figure 1.9: A simple scheme of a 3-input logic block (CLB) of an FPGA.

data that have to be fetched and stored multiple times[13]. Moreover the memory accesses require also an higher energy cost than the MAC operations. Known this issue, the major companies have developed more complex integrated systems to make possible an optimal memory management.

This thesis work is carried out by using an AMD development-ready kit, called *Kria KV260 Vision AI Starter Kit*. It belongs to the AMD Kria family, a series of products based on adaptive SOMs (System-On-Modules) that are hardware platforms designed for AI-acceleration with all performance optimizations, like low latency and low power consumption.

The KV260 board, as shown at 1.10 scheme, consists of an end-user-designed pcb (known as Carrier Card) that mounts the K26 SOM provided with a thermal solution and a variety of interfaces for integrating different peripherals. The K26 SOM has been developed to answer to current and future market demands for vision AI and video analytics[15]. It joins an adaptive SoC based on the Zynq UltraScale+ MPSoC architecture with all of the fundamental components required to support that SoC (such as memory and power).

Hence the hardware core of the KV260 board can be identified with the AMD Zynq UltraScale+ MPSoC. Its schematic is depicted in figure 1.11 and highlights two fundamental parts [15] [17]:

- **Processing system** - The PS part is in charge of booting the system and it has three major processing units: CPU, RPU and GPU. The first one is an ARM Cortex-A53 64-bit quad-core processor and is comprised in a more complex unit, called application unit (APU). Another main component is a DRAM memory controller that makes possible the communication between this SoC and the DRAM (4GB 64-bit DDR4) placed outside on the K26 SOM.

- **Programmable logic** - The PL is a configurable hardware resource and can provide many components, like on-chip memories, gates, clock structures,

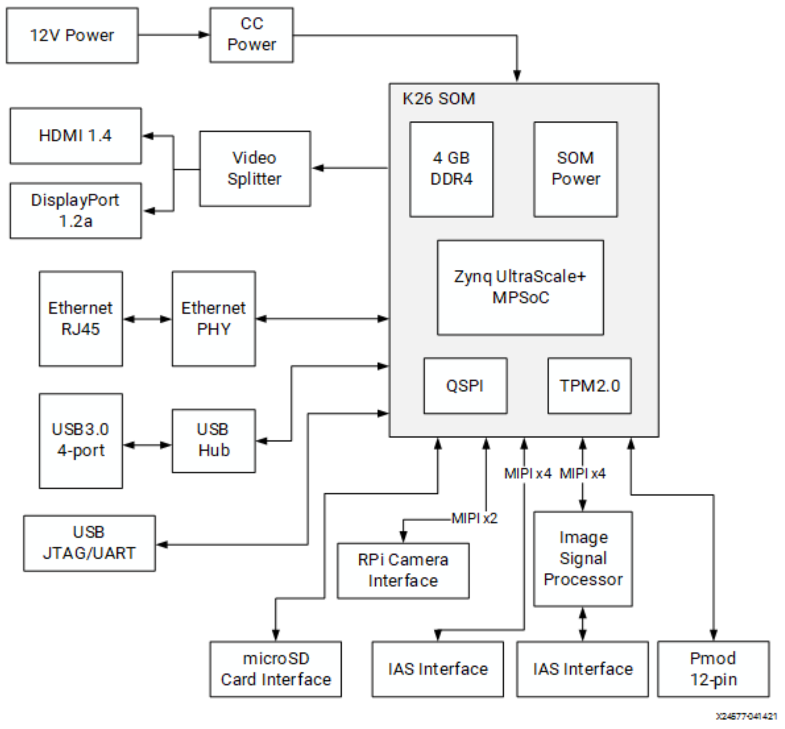


Figure 1.10: Kria KV260 Vision AI Starter Kit block scheme[16]

DSPs, LUTs and so on. All these components characterize the FPGA side of the SoC and they allow to implement the custom accelerators for computer vision.

To summarize, the Zynq architecture that will be used in this thesis work combines in a single SoC, both CPU (and GPU) and a traditional FPGA. All of this is supported by the Arm interface system based on the AMBA[5] *Advanced eXtensible Interface* (AXI) standard, which provides high bandwidth and low latency connections. To complete the discussion about this SoC, it is fundamental to mention the DPU (*Deep-learning Processing Unit*). Indeed, AMD provides this configurable computation engine, optimized to run CNNs, a sub-category of the DNNs called convolutional neural networks. Practically speaking, the DPU is an hardware accelerator comprising elements available in the programmable logic fabric mentioned before, such as DSP, BlockRAM, UltraRAM, LUTs, and Flip-Flops. Whereas at an higher level it is a micro-coded computation engine which has an optimized instruction set to support inference of most CNNs[19]. This

[5]The Arm Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in SoC designs.

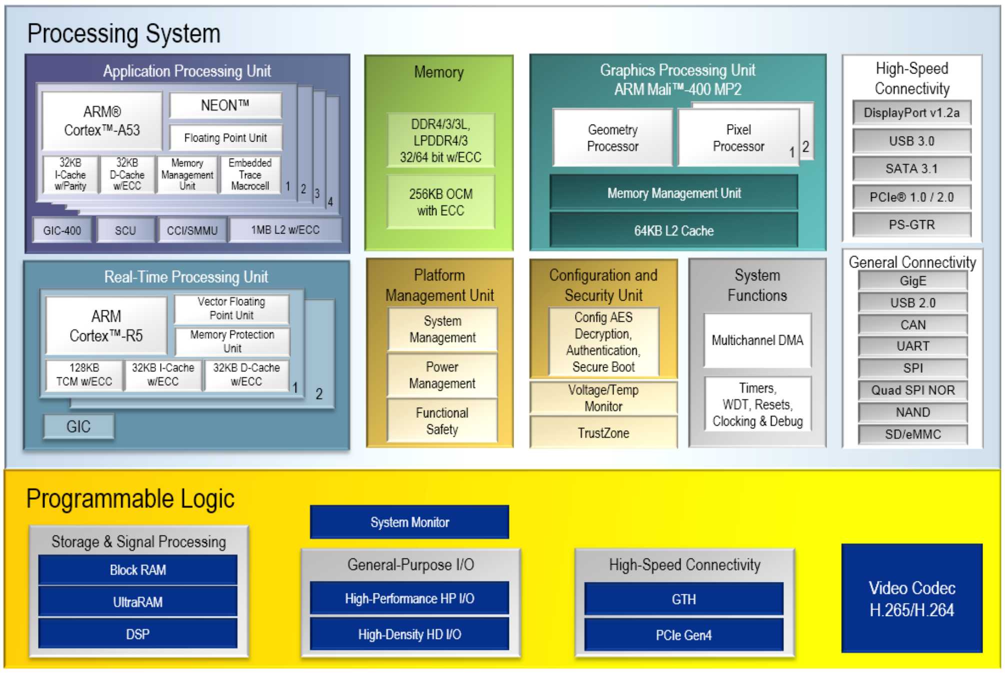


Figure 1.11: Zynq UltraScale+ EV MPSoC block diagram[18]

component will be described more in depth at the next chapters.

Chapter 2

The AI model

2.1 Neural Network architecture

The aim of this thesis work has been briefly introduced on section 1.2: a system of obstacles detection for autonomous driving to develop on a FPGA-based device.

The detection of obstacles in the surrounding environment corresponds to the Computer Vision task of *Object Detection*. Its complexity lies in having to handle actually two tasks: the identification and drawing of the *bounding box* around the object and the classification of its content. Several approaches have been developed to face this AI task with an increasing focus on speed and accuracy for each single detection.

The neural networks more suitable for this kind of behaviour are the previously mentioned *convolutional neural netowrks* (an example at figure 2.1). Being a deep neural network, a CNN has typically several hidden layers that may be chosen among:

- **Convolution layer**: it performs the *features extraction* thanks to a filtering system. Considering an RGB image as a 3D tensor ($w \times h \times 3$), a convolution kernel/filter is applied on all the image's pixels, generating a *convoluted feature map*, that is a modified version of the input image where a specific type of feature has been highlighted.
- **ReLU layer**: it only performs an *activation* operation where a non-linear activation function is used. The so called Rectified Linear Unit corresponds to the mathematical $\max(0, x)$ which performs easily the replacement of negative values (usually of the previously generated feature map) with 0.
- **Pooling layer**: its task is the *downsizing*. The image is divided into groups of pixels that are replaced by a single value (their maximum or average). This process has the consequence of reducing the computational cost of the network.

Finally, extracted the image features, the output layers work for classification. Following the right-side of the 2.1 diagram: the multi-dimensional result will be flatten into a single dimensional array; a fully connected layer outputs a vector of N dimensions[1] with the class probabilities; the final prediction is the result of a final activation function (e.g. Softmax)[20].

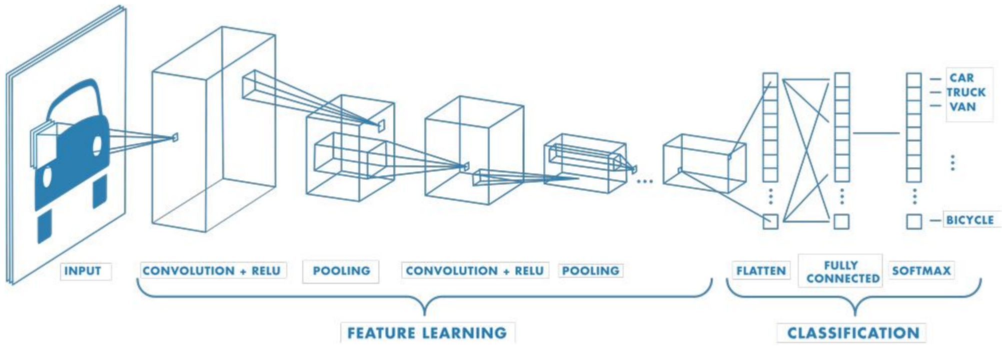


Figure 2.1: Example of a network with many convolutional layers[21]

Having analysed what a CNN is, the first step now is to find a model that is based on this architecture. A list of candidates is provided by the AMD software tool used in the following for the model development, *Vitis AI*. Its supported models are viewable at the Vitis AI Library's model list[22].

The decision falls on *YOLOv5*, belonging to the You Only Look Once (YOLO) family of computer vision models, proposed in 2016[23]. YOLO approach was a breakthrough for the object detection task. They are based on Convolution Neural Networks (CNNs) and are characterized by an innovative feature: unlike other models, such as Faster-RCNN that uses two-step algorithms, they are able to perform the two tasks of bounding box positioning and class determination at once (hence the name YOLO). The advantage of this implementation is that running a single CNN on the image makes much lighter and faster the whole approach, keeping high accuracy level.

The YOLOv5 architecture, shown at figure 2.2, is composed by 3 main parts: backbone, neck and head. The *backbone* is the CNN that aggregates and extracts the image features, the *neck* is a series of layers to mix and combine the extracted features and finally the *head* concludes with box and class predictions[24].

YOLOv5 is available in more versions depending on its weights dimension: from the nano variant *YOLOv5n*, having a limited accuracy but providing the possibility

[1]N is the number of classes able to be predicted

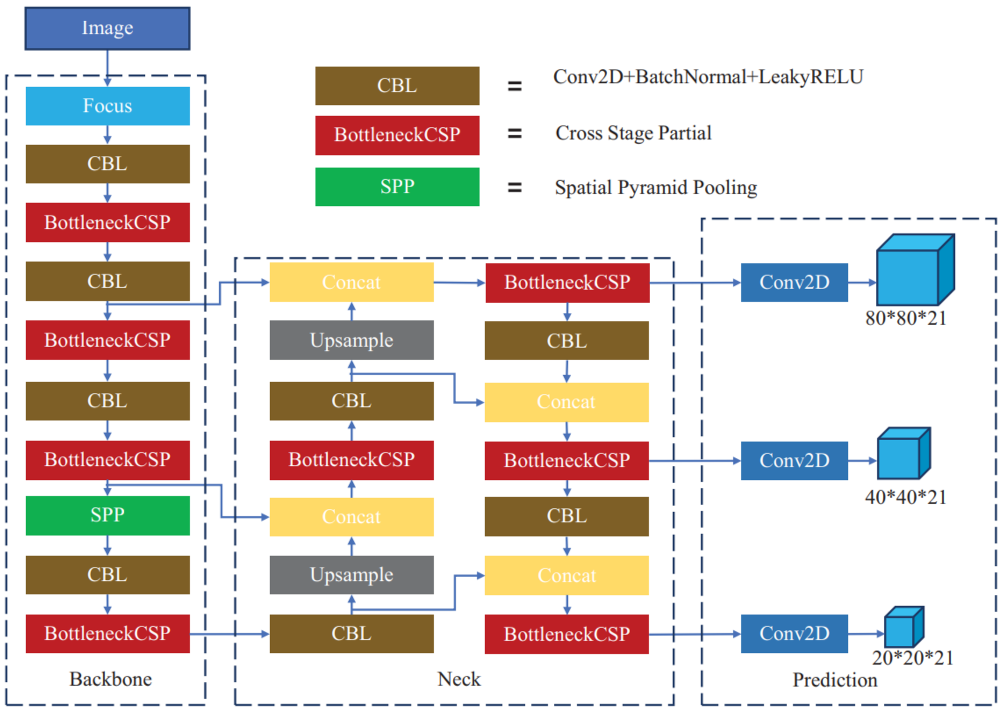


Figure 2.2: YOLOv5 architecture[25]

to use a very small-sized model (1.9 millions of parameters) with a good inference speed, to the extra-large YOLOv5x, which instead may reach an higher accuracy but paying with a slow and large model (86.7 millions of parameters). The model size is mainly referred to the width and depth of the BottleneckCSP modules[25] (figure 2.2). Among the model variants supported by Vitis AI Library the choice falls on the nano one since this model will run on an embedded device and a small size is a good advantage. In addition, YOLOv5n guarantees a much shorter training time than the others, which is very helpful in cases where hardware resources are limited (such as a CPU-only approach).

2.2 A dataset for obstacle detection

The Supervised Learning, described at 1.1.2, relies on the variety, size and comprehensiveness of the used dataset. Its quality is crucial to guarantee a trained model with high accuracy.

For this project one of the most credited dataset for the obstacles detection will be used. It is called *KITTI*[26] and it provides a large quantity of data collected by video cameras, laser scanners and a GPS/IMU localization units, all mounted on a proper vehicle. To train the YOLOv5 model mentioned before, the "2D object detection" data have been retrieved, which are coloured png images arranged in:

- 7481 training images with 80256 labeled objects, belonging to one of these 8 classes: pedestrian, truck, car, cyclist, misc, van, tram and sitting person.
- 7518 test images, so unlabeled.

Before the training, a fundamental step of *format conversion* is needed. All the datasets have their own format, characterized by:

1. A certain data organisation in the directories.
2. A specific representation of the detected object information in the label files.

The mandatory requirement of a training process is that this format matches that one accepted by the model. Therefore, a modification of these two aspects has been performed by a customized python script A.1, that can be run as:

```
python3 convertToYOLO.py <in_dataset> <train_perc>
```

The figure 2.3 helps to understand how it works. About the data organisation, the unlabeled testing images are copied without changes, while the training ones are subject to *data splitting*. This process consists of using a certain percentage of data for training and the remaining part for the next metrics evaluation step, named validation. Unlike the images, the labels must undergo a format conversion because their content strongly differ.

In the Supervised Learning approach, labelling an image means to identify all its contained objects annotating their characteristics inside textual files, called *labels*. Those ones needed by YOLO have: the object class index and the coordinates of the rectangular border fully enclosing it (i.e. *bounding box*). However KITTI has much more information for each object and also different formats: the object class is a string name instead of an integer index, so a mapping process is needed; while the bbox coordinates need this type of conversion:

$$(\text{left}, \text{top}, \text{right}, \text{bottom}) \Rightarrow (\overline{x_c}, \overline{y_c}, \overline{w}, \overline{h})$$

where the YOLO format coordinates are also normalized in (0,1) range by the image dimensions. A visual help is provided by the diagram 2.3 and all the other details are in the script code's comments A.1. Finally the building of a *data.yaml* file is performed. It has information on the dataset that the already-available YOLO scripts use by parsing it into a python dictionary.

After this script execution with a reasonable split ratio, like (train : val) = (80 : 20), the dataset will be perfectly ready to be used by the model for the training phase.

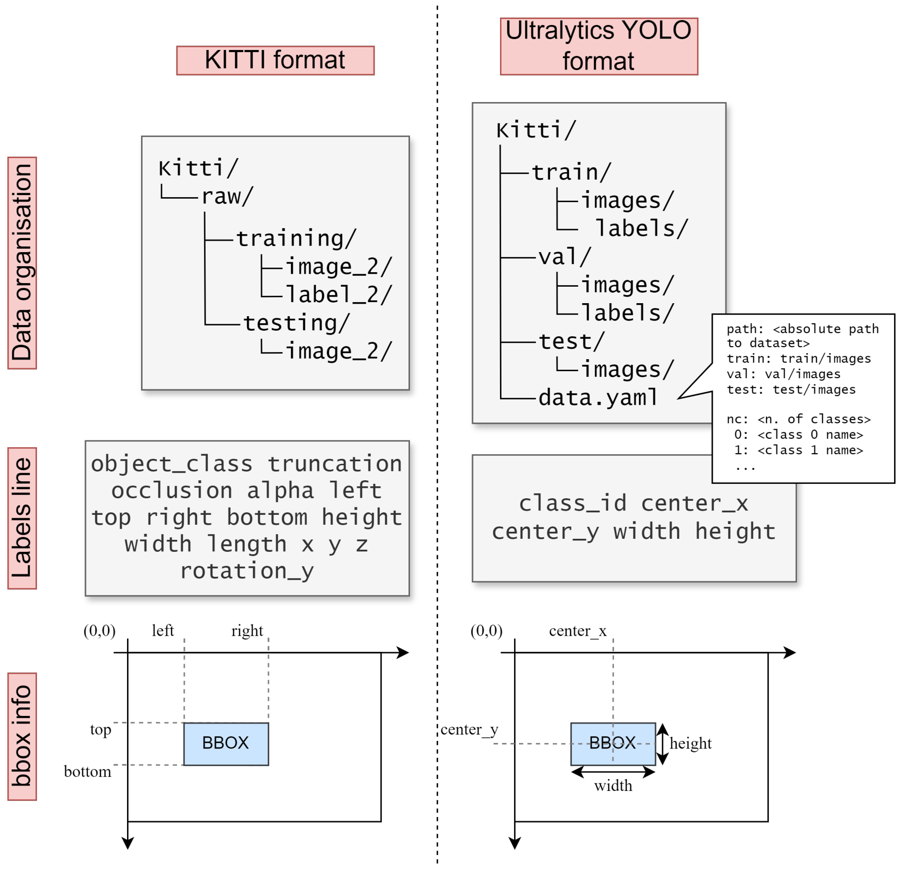


Figure 2.3: Data format comparison between KITTI and YOLOv5.

2.3 Training and validation

2.3.1 Metrics evaluation

An overview about the metrics used by the YOLO model's validation process is needed to have a comprehensive understanding of the training result. These metrics are supported by a comparative analysis[27] done about the object detection.

The most significant metric is the accuracy that the YOLO models measure as *Mean Average Precision at threshold IoU* (mAP@IoU). To properly define it, the object detection must be observed as two separate tasks: object localisation and classification[2].

[2]As before explained, YOLO performs the object detection all at once. However for this specific aim of measuring the model's metrics, the two tasks must be considered separately.

- **Classification** - The basic metrics are *Precision* P and *Recall* R. The figure 2.4 shows all the possible combinations of the model's output, where if the prediction matches the actual label the result is called *true*, otherwise *false*. So the above-mentioned metrics are defined:

$$P = \frac{TP}{TP + FP}$$
$$R = \frac{TP}{TP + FN}$$

  They are used to quantify the quality of a model classifier, graphically by the precision-recall curve 2.4 and mathematically by the area under this curve, named Average Precision (AP):

$$AP = \int_0^1 P(R)dR$$

  Finally the mean of this parameter among all the recognized classes is the mAP.

- **Localisation** - The predicted bounding boxes match almost never the ground-truth ones perfectly, so how to establish if a prediction is true or false? The *Intersection over Union* parameter is used:

$$IoU = A_I/A_U$$

  meaning the ratio between the area of the bounding boxes intersection and union.

Therefore, YOLO models measure their accuracy as mean average precision at a certain IoU threshold, typically at 50% (mAP@50) or as an average of equidistant values from 50% to 95% (mAP@50-95).

Finally another important metric can be identified in the *FPS* (Frame per second). It quantifies how fast is the object detection model to process the input images and generate the output. It depends also by the device where the model run the inference.

2.3.2 Model training

YOLOv5 is made available as open-source model by Ultralytics company. Its GitHub repository[28] can be cloned to have all the required code for the main AI operations, like training and inference. The provided python script for training is *train.py* and can be used as follows:

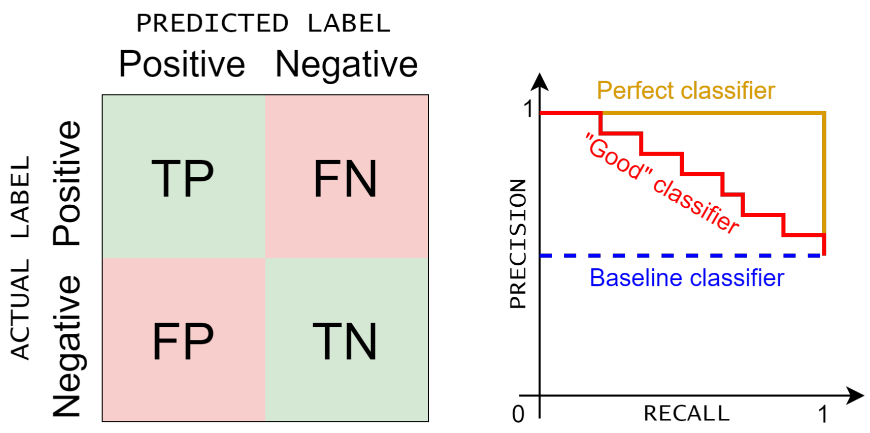


Figure 2.4: Precision-Recall curve for a classifier.

```
python3 train.py [options...]
```

The mentioned options are hyperparameters and configurations that have to be set properly, since they will lead to certain characteristics of the trained model (speed, accuracy but also the training duration). Their meaning is briefly explained here:

- **imgsz** - All input images are resized to this dimension (in pixel) before being fed into the model. Specifically if just one value is provided, it is considered the longest image's dimension (width in this case) and the aspect ratio is kept unchanged. This feature is important mainly for datasets like the used one, KITTI, that have all images with different sizes. The value has to be picked from multiples of 32, the *max* model's *stride*, and from values lower than all the widths of the images, to perform a downscaling. Considered that, the best matching size is *1216*.

- **batch-size** - A key process of the training phase is the tuning of the neural network's parameters (weights and bias). To execute this task a cost/error function, tracking the difference between predictions and labels[3], is built and its minimum has to be found. The *gradient descent* (graphically shown at figure 2.5) is one of the most common algorithms that addresses this problem[29]. The training images are imposed through the hidden layers of the DNN to find the gradient of the function and then, the direction is chosen to minimize

[3]The actual output of the model can be called *prediction*, while the ideal and correct one is named *label*.

it. However at this point an issue arises: the use of all the dataset's samples together (Batch GD), to reach the minimum in one step, requires a huge quantity of memory, considering the large dataset used in DL. Whereas the opposite case of using just one sample per step (Stochastic GD) requires much less memory but a long time. A good trade-off is provided by the Mini Batch GD where a subset of the dataset (the *batch*) is used at each step, named *iteration*. This option is the size of each batch that goes from 1 (SGD) to the dataset size (BGD).

- **epochs** - During one epoch the entire dataset is processed, therefore a larger number of epochs guarantees typically a better achieved accuracy. It should be set to have a reasonable training time also based on the batch size. For instance, if the training dataset has 5984 images (80% of KITTI) and the batch size is 16, the resulting iterations for each epoch are $5984/16 = 374$, a large number that will extend the training time, mainly if also the number of epochs is high.

- **data** - this is the path where find the training and validation dataset to use (KITTI in this case).

- **weights** - to select the model variant to train (nano, small, etc.), as described at section 2.1.

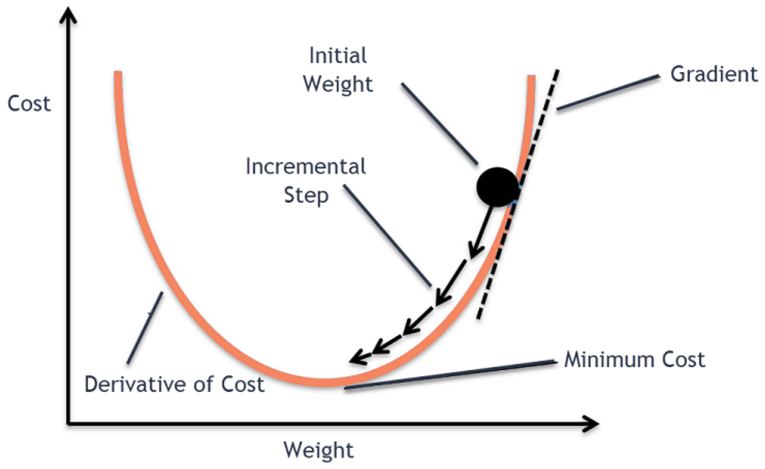


Figure 2.5: Gradient Descent algorithm[29].

The training must take into account its host machine configuration. In this thesis work the resources were limited by an integrated graphics unit, thus a CPU-only setup with lack of dedicated graphics memory[4]. As underlined at the previous "technologies comparison" discussion 1.3.1, this configuration is not optimal, leading to long training times. However, a feasible training process is reached by the options at table 2.1 where the use of a light-weighted model (nano) and a low batch size (16) has a reduced impact on the memory requirements. 59 training epochs have lasted a total of 71.496 hours (almost 3 days) and each of them has employed 374 iterations, that could be so justified:

$$\begin{aligned}
\text{Kitti dataset images} &= 7481 \\
\text{Split ratio} = \text{train:val} &= 80:20 \\
\text{Training images} &= 7481 \times 0.8 \simeq 5984 \\
\text{Validation images} &= 7481 - 5984 = 1497 \\
\text{Iterations per epoch} &= 5984/16 = 374
\end{aligned}$$

The achieved accuracy level is shown at table 2.2. Keeping the IoU threshold at 50% a pretty good result of mAP50 = 0.784 is reached and it is also visible by the precision-recall curve 2.6. The critical classes that cause a strong lowering of the accuracy are: *person_sitting*, *misc* and *pedestrian*. Whereas trying to consider higher IoU thresholds, as for mAP50-95 case, a worse result of just 0.433 is reached.

About the model's speed, the validation process measures some times:

```
Speed: 3.0 ms pre-process, 179.1 ms inference, 0.9 ms NMS per image
```

that correspond to these fps values: 333 fps just for the inference and 5.5 fps for all the steps (comprised pre and post-process).

Manually set		Automatically set	
imgsz	1216	Learning rate	0.01
epochs	59	Learning rate decay	0.937
batch-size	16	Weight rate decay	5e-4
weights	yolov5n.pt	Iterations	374
data	KITTI's data.yaml		

Table 2.1: Training options and hyperparameters set manually and automatically.

Supported operators check A last check has been performed before the training process. As already mentioned at 1.3.2, this model will be deployed on the KV260

[4]Specific setup made by: Processor - Intel Core i5-8250U CPU @ 1.60GHz x 8; Graphics - Mesa Intel UHD Graphics 620 (KBL GT2); Memory - 16GB

Class	Images	Instances	P	R	mAP50	mAP50-95
all	**1497**	**7991**	**0.81**	**0.709**	**0.784**	**0.433**
Pedestrian	1497	873	0.759	0.512	0.621	0.267
Truck	1497	220	0.933	0.873	0.95	0.595
Car	1497	5693	0.911	0.881	0.938	0.64
Cyclist	1497	292	0.779	0.688	0.754	0.348
Misc	1497	205	0.815	0.664	0.748	0.369
Van	1497	553	0.856	0.752	0.838	0.504
Tram	1497	106	0.856	0.811	0.918	0.515
Person_sitting	1497	49	0.573	0.49	0.501	0.227

Table 2.2: Accuracy results of the validation step on the trained model.

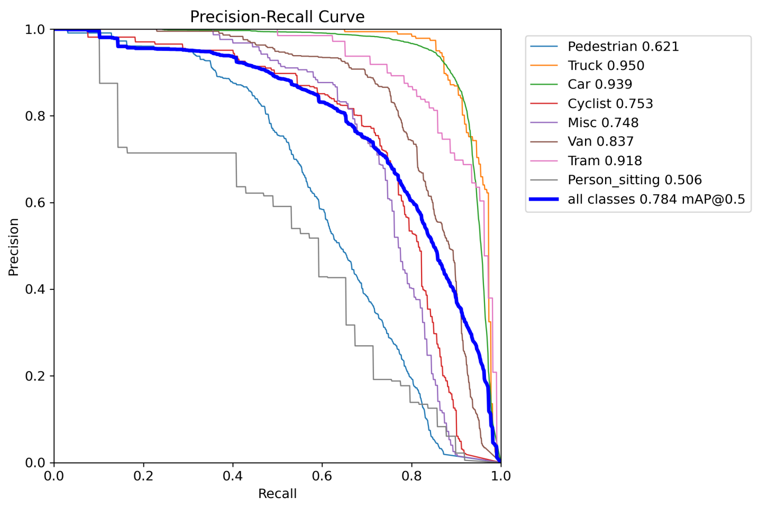


Figure 2.6: Precision-recall curve of the trained model (after 59 epochs).

board that is able to run a CNN, thanks to the acceleration provided by a DPU IP. However this acceleration unit has some limitations about the supported operators. To compile successfully the model for a certain DPU, the operators used in the model architecture has to be compared with the table of supported ones, provided by *Vitis AI* guide[30]. Otherwise, there is the risk that the unsupported operators will be assigned to the CPU, causing a degradation of the final inference on the board.

Going back to the YOLOv5 model in use, an unsupported operator is found on Convolution and BottleneckCSP layers. It is the *SiLU* (Sigmoid Linear Unit), an activation function that performs the multiplication among a sigmoid and its input. A valid alternative has been found following this LogicTronix tutorial [31] that replaces it with a *LeakyReLU* function with negative slope 25/256. After this modification to the model's code (*models/common.py* and *experimental.py*), the training process can be run with the assurance that, during the inference, all operators will be successfully accelerated by the DPU.

Chapter 3

Model Development

3.1 Vitis AI introduction

The previous chapter has outlined the training process reaching a model with properly tuned FP32 weights. This is the starting point of the workflow addressed by the AMD development environment that will be used in this thesis work, named *Vitis AI*.

It comprises optimized IPs, tools, libraries, models and example designs, all with the aim of accelerating DL inference applications on AMD hardware platforms. Its two main features are high efficiency and easy-of-use[30].

The entire Vitis AI environment is based on the Deep Learning Unit, mentioned at section 1.3.2 and that will be explained at next sections 3.3.1 in more depth. This chapter will proceeds with the analysis of the VAI tools. Each of them performs a specific task in the *model development*, i.e. in the process leading to a compiled model deployable in the target board.

Before to start with the Vitis AI workflow, it is important to know that its code is fully accessible by GitHub repository[32] and can be installed on the host machine as Docker container. The specific image to pull inside the docker depends on the device, CPU-only in this case, and on the used framework, PyTorch considering the YOLO code implementation.

3.1.1 VAI Optimizer

The *VAI Optimizer* has the aim to reduce the computational cost of the model's inference once deployed and running on the hardware platform. The only optimization performed is called *pruning* and is based on the concept of sparsity of a graph. Typically after training, the neural networks are still *dense graphs*, that means its nodes are almost all connected each others causing a number of edges close to the maximum one. The exploitable feature is that many of these nodes are redundant, so the pruning process can be applied to remove them and to obtain a *sparse graph* keeping the accuracy loss as low as possible.

This process can be analysed by subdividing it into some steps[33]:

- **Sensitivity analysis** - It is needed to understand how a convolutional channel afflicts the predictive behaviour of the model.
- **Coarse Pruning** - The previous analysis has found some channel's weights that can be removed/pruned. Here they will be zeroed in order to have an idea of the post-pruning accuracy of the model. If this step interests just the weights, the *sparse* (or fine-grained) pruning is its name, while if entire channels are interested the approach is named *channel* (or coarse-grained) pruning.
- **Finetune** - The training dataset is exploited to fine-tune the remaining network's parameters. In the *iterative* pruning this step is re-iterated until a satisfying accuracy level is reached. Otherwise a *one-step* pruning is addressed.
- **Transformation** - The zeroed nodes and the correlated edges are removed generating the optimized model.

The VAI Optimizer is able to perform different kinds of pruning as seen in the above description. One more type is called *Once-for-All* (OFA) and its main objective is to avoid the re-training of the networks for each hardware platform where they run. The OFA approach consists of training only one time the network and then performing hardware optimizations (leading to lower computational costs). This is possible since the network can be divided into subgraphs, each candidate for a different hardware configuration.

This tool was shown for completeness, however it is not used in this thesis work. The used Vitis AI version, 3.0, requires a license to purchase[1] for the optimizer and since this step is not mandatory for the final success of the project, its use has been avoided.

3.2 Model quantization

The common objective of all the Vitis AI tools is to develop a model capable to be run by an AMD hardware platform as the Zynq Ultrascale+ MPSoC. Following this purpose, the first steps are performed to reduce the computational complexity of the initial model before to be compiled for the specific hardware.

[1]The last Vitis AI release, 3.5, has made the VAI Optimizer free-of-charge.

After the optional pruning, a mandatory step is the *quantization*. It consists of converting weights, biases and activation values of the trained neural network from *32-bit floating point* to *8-bit integer* format. This process is needed by the DPU IPs used to accelerate the CNNs on the AMD platforms. Running a model with INT8 parameters leads to require less memory bandwidth and to have more power efficiency and speed, all features fundamental for embedded devices.

The quantization is practically performed by mapping FP32 values into INT8 ones. The challenge of this process is to keep a low accuracy loss. Although this may seem a considerable complexity, it is actually not true since the large range provided by a FP32 representation in real applications is typically not used. Hence integer parameters are enough and they can be obtained by easily applying simple scaling factors[33].

A summary of the quantization process is at diagram 3.1. Firstly the PTQ (*Post-training quantization*) is applied to generate an *INT8 model*. This is possible thanks to a *calibration* dataset, i.e. an unlabeled sub-set of the training one, that is forwarded through the network allowing to analyse the distribution of the activations at each layer.

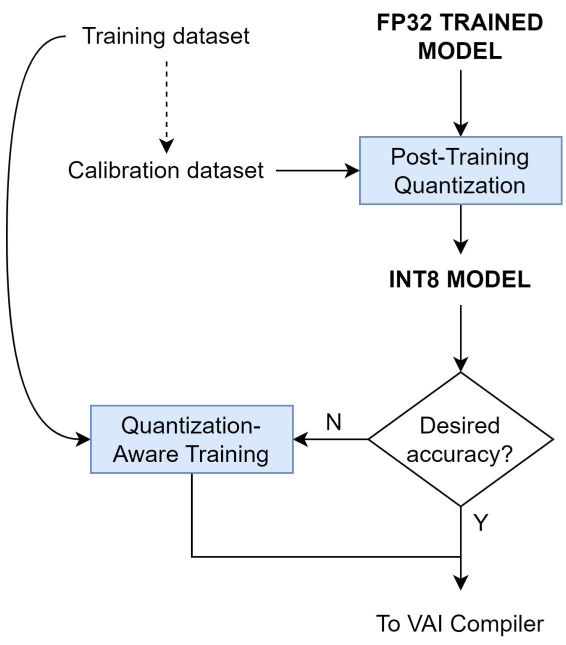


Figure 3.1: Vitis AI Quantizer workflow.

The quantization script A.2 has been written with the support of the LogicTronix tutorial [31]. The trained FP32 model is loaded by the *DetectMultiBackend* class of the YOLOv5 code and the VAI Quantizer is provided as *torch_quantizer* among the *pytorch_nndct* APIs. Finally to get the INT8 quantized model, the *quant_model()* method is run, after having established the input images dimension as 1216×320 (compliant with the training ones). As before analysed the KITTI dataset has all images with different size but the quantizer needs all the same dimensions, otherwise an error arises. Hence, the calibration dataset has been prepared before to run the script by selecting a subset of 1000 images and resizing all of them to 1216×320. Finally this dataset has been loaded by YOLOv5 methods in order to have a python structure compatible with that one used by the YOLOv5 validation process, which core is the *run()* method, get by *val.py* and modified to solve some arisen errors.

To reach the integer model export, two steps are needed but in separated run. On this purpose, the *quant_mode* parameter is used sequentially with these two options:

1. **"calib"** corresponds to the calibration step where, after the unlabeled images forwarding (performed by *run()* method), the VAI Quantizer method *export_quant_config()* is used to export the configuration files to can obtain the final quantized model.

2. **"test"** performs the *export_model()* quantizer's method able to generate the *XMODEL*[2] file containing the INT8 neural network.

At this point the accuracy has not been considered, so its degradation is possible. Following the diagram 3.1, the accuracy is evaluated before to decide if perform the QAT (*Quantization-Aware Training*) to fine-tune the parameters re-adjusting the model accuracy.

Pre-quantization modifications The Vitis AI user guide[30] illustrates some modifications needed to make the model quantizable. The only method accepted by the architecture's detection head (the *Detect()* class) is the *forward()* one. All the others are typically pre and post-processing operations. Therefore, before to quantize the model, *yolo.py* must be modified to match this requirement and the post-processing functions are moved in the *detect.py* code. The practical way to do it is get by the LogicTronix tutorial [31].

3.3 Model compilation

The last step is the compilation and its objective is to generate a *deployable model*. Specifically the VAI compiler is exploited to map the previously-quantized network

[2]XMODEL is typically the format of the models after compilation. A feature of the PyTorch framework's use is that this format is also exploited to store the quantization result.

into a highly optimized sequence of DPU instructions[30]. This process consists of some steps:

1. **Parsing** - The quantized model is generally still framework-dependent (i.e. PyTorch in this case), so the initial step will parse it and remove the framework dependencies. The result is a Xilinx intermediate representation (XIR) graph, consisting of independent control and data flow representations.

2. **Optimizations** - The XIR-based graph is partitioned into subgraphs and each of them is optimized considering the DPU where they will run. Information about the DPU is provided by a *arch.json* file as a fingerprint of some bits. The advantage is that to deploy the model on a different DPU, changing the json file and re-compiling is sufficient. A safe way to get the correct fingerprint is: interrupting here the model development, executing all the platform development addressed by 4.2 and, inside the Kria board's Linux CLI, executing the command *xdputil query* that generates the DPU-specific fingerprint. Obtained the json file, the compilation can be resumed and completed.

3. **Code generation** - The XIR graph object is finally serialized into a compiled *XMODEL* file.

The VAI Compiler is provided by the Docker and it is so executed:

```
vai_c_xir --xmodel <file.xmodel> --arch <arch.json>
```

generating a *XMODEL* file ready to be deployed on the hardware platform, i.e. the Kria KV260 in this thesis project.

3.3.1 The base hardware IP: DPUCZDX8G

The Deep Learning Unit is the core hardware accelerator used by the AMD platforms for AI inference. It can be seen by two perspective, the first is a matrix of heterogeneous processing elements (PEs) specialized on different tasks (like convolution) and comprising elements of the FPGA's PL fabric (like DSPs, LUTs, FFs and kinds of RAM). While by a higher-level view, a DPU is an engine able to run a set of microcoded instructions taken by the DPU specific instruction set.

Here the focus is on the *DPUCZDX8G*[19] that is the DPU designed for the core platfrom of the Kria KV260 board, i.e. the Zynq Ultrascale+ MPSoC. It is optimized for CNNs, like YOLOv5. The figure 3.2 shows the top-level block diagram where the main components supporting the PEs array are listed: high performance scheduler, instruction fetch unit and global memory pool. The DPU is inside the

PL part of the Zynq Ultrascale+ MPSoC and communicates by AXI Interconnect with the PS, particularly the APU is highlighted since it runs a program to service interrupts and to coordinate data transfers. Finally RAM memory locations are required to store input images, temporary and output data.

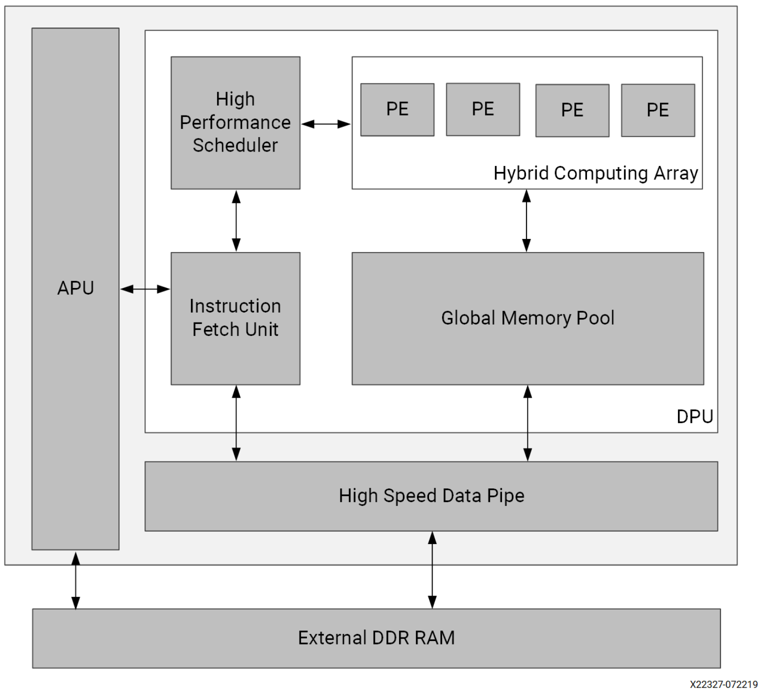


Figure 3.2: Top-level block diagram of DPUCZDX8G[19].

A deeper view with a clearer distinction between PS and PL sides is on figure 3.3. The *XMODEL* file, previously generated by the VAI Compiler, is placed in the *off-chip memory* that, thus, will contain the instructions to be fetched, decoded and dispatched by the scheduler. The computing engine is the execution core of the DPU and is placed in the PL side together with an *on-chip memory* buffering intermediate results to achieve high throughput and efficiency. This data reuse helps to reduce the external memory bandwidth requirements.

The DPU is provided by AMD as a TRD (targeted reference design) and both Vitis and Vivado can be used as software tools to integrate it in the design. The first one enables the integration as an acceleration kernel loaded at runtime in the form of an *xclbin* file, while Vivado does not provide it in the IP catalog but can be easily added separately[33]. All the details on the DPUCZDX8G configuration are explained in the next chapter 4 during the platform development description.

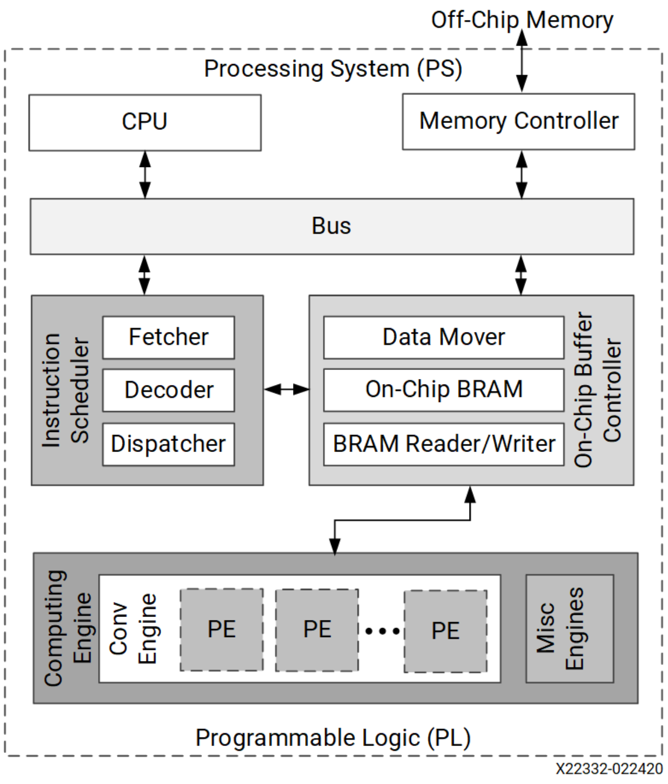


Figure 3.3: Hardware architecture of DPUCZDX8G[19].

Chapter 4

Kria Acceleration flow

The whole flow of this thesis project can be summarized as in figure 1. It consists of three macro-steps that, known the general specifications, can be addressed independently. One of them has already been discussed, it is the *model development* at chapter 3, where starting from a trained model, it led to a deployable one, compiled for the target DPU. The other two macro-steps involve the *Platform* 4.2.1 and *Application* 4.3 development and will be addressed in this chapter.

4.1 Video acquisition pipeline

Before to go deeper in the platform development 4.2.1, an introduction about the pipeline to enable and all the components involved of the board is fundamental.

One of the advantages of a Kria SOM board, like the KV260 Vision AI Starter Kit, is that it is provided together with many *pre-build accelerated applications*. They can be used as starting point to create the desired application by customizing it at any level, from software to FPGA design. The main goal of this thesis work is the obstacle detection in real-time, so an image sensor is fundamental to capture the input images. Among all the Kria pre-build applications the *Smartcam* seems to be the most suitable[34]. A deep analysis is provided in the following to have a comprehensive view of the end-to-end pipeline to design. The figure 4.1 shows this pipeline graphically and it also gives the idea of what is considered as part of the *platform* and of the *overlay*.

The end-to-end pipeline can be divided into sub-pipelines:

- **Capture pipeline** - An AR1335 image sensor (connected on the J7 IAS connector of the carrier card) is used as external video source. The captured video frames are sent to the AP1302 ISP (image signal processor) already

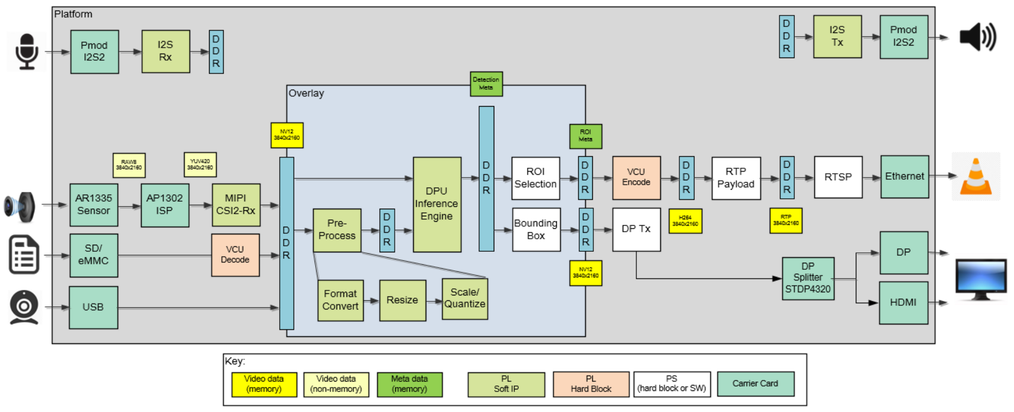


Figure 4.1: End-to-end pipeline of the Smartcam application[34].

integrated on the cc, able to process the input images[1]. Then a receiver is placed on the PL to get the video frames by a MIPI CSI-2 interface. This is the main capture pipeline for this project, however the other ones have been implemented too, like that one supporting the external memory and the USB camera. Each of them ends writing the captured data on a DDR memory.

- **Acceleration pipeline** - It is provided as an *overlay* that performs pre-processing, inference by the DPU and post-processing. The latter is the superposition of the bounding boxes on the detected objects of the initial captured images. All these middle steps store temporarily their output data on the DDR memory.

- **Output pipeline** - The input images with the final bounding boxes stored in the DDR memory are displayed through some possible ways: DisplayPort, HDMI or Ethernet.

4.2 Platform Development

As the diagram 1 shows one of the fundamental step to perform is the *platform development*. It consists of building some components, such as:

[1]An Image Signal Processor as the AP1302 ISP can perform auto white balancing (AWB), auto exposure (AE), auto focus (AF), etc.

- The **Xilinx platform** with all the hardware and software features for the design.
- The **overlay** with pre-processing and DPU IPs.
- The **Linux image** that will be the running OS image on the Zynq Ultrascale+ MPSoC.

Each of them is analysed in the following in separated sections of this chapter, moreover to have a more comprehensive view of the entire development flow, the Kria documentation provides an helpful diagram 4.2.

The practical starting point is the *kria-vitis-platforms* Xilinx GitHub repository[36] arranged essentially into two parts:

- **Platforms** - are Vivado projects to use as base design. They define physical interfaces to off-chip components, like the AR1335 sensor but also accelerator clock and memory interfaces.
- **Overlays** - are Vitis projects to overlay on the platforms. They include the DPU design as RTL kernel and the pre-processor bulit as HLS-based computer vision kernel supported by the Vitis Vision libraries.

This repository is locally cloned and its files are used as base for the next discussions.

4.2.1 Platform creation

In the AMD environment there are several types of *platform* concepts, however it is generally identified as a package of files and metadata containing both hardware and software features of the design. Here the focus is on the *Vitis extensible platform*, more easily identified as Xilinx Platform (XPFM). It is used by the Kria environment and its generation is supported by the use of Vivado and Vitis, two of the most common AMD software tools. Their 2022.2 version is previously installed taking into account the compatibility with the used Vitis AI release, 3.0.

The objective of this section is to build the XPFM, considering it as a simple package of a hardware and a software platform. To immediately understand the following description the resulting directory can be taken into account, see `hw/` and `sw/` folders on figure 4.3.

The design starts by the **hardware platform**. The most complete way to proceed is to start with an empty vivado project and building manually by HDL or block diagram all the hardware specifications. However, using the pre-built Kria applications the design time and the possibility of serious errors are strongly reduced. Intending to exploit this advantage, the hardware platform building

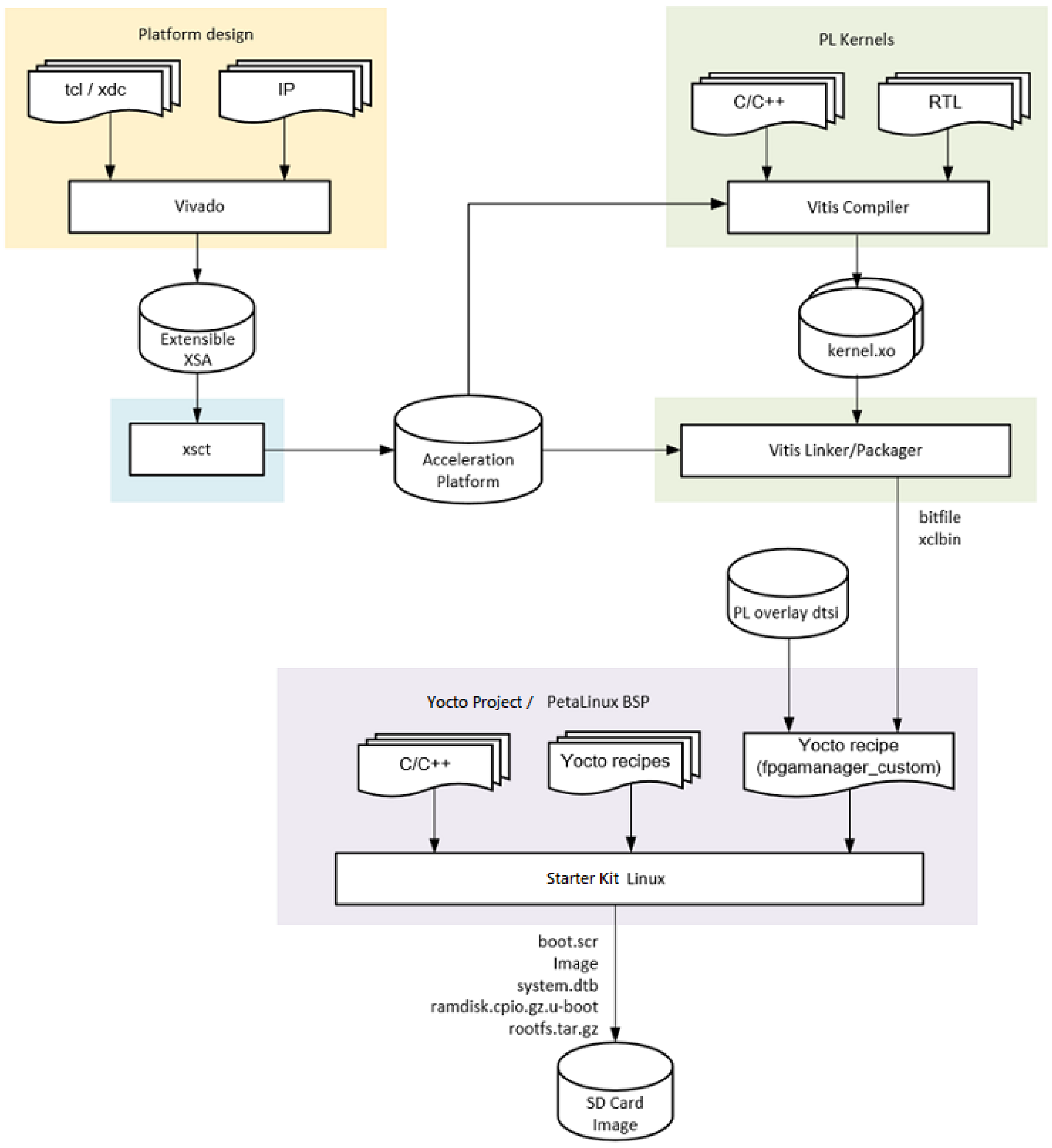


Figure 4.2: Platform Development flow[35].

starts by the choice of the pre-built platform among the available ones at the `platforms/vivado/` directory. That one matching the *Smartcamera* application and so, even all the main features useful for the final application of this project, is named:

```
kv260_ispMipiRx_vcu_DP
```

Vivado is a software for synthesis and analysis of hardware description language (HDL) designs with additional features for SoC development and high-level synthesis. Indeed, inside the above mentioned folder, there are: tcl scripts to build the vivado project with its block diagram and the xdc file with the constraints on the physical pins to enable. Hence, since most of the all hardware design is automatically built by these scripts, just some manual adjustments are required. Particularly the I2S audio pipeline is not needed for the thesis design project and so can be totally removed. The Vivado GUI has been opened and the modifications has been applied on the *Block Design*.

```
xilinx_kv260_ispMipiRx_vcu_DP_202220_1/
├── hw/
│   └── kv260_ispMipiRx_vcu_DP.xsa
├── sw/
│   ├── kv260_ispMipiRx_vcu_DP.spfm
│   └── kv260_ispMipiRx_vcu_DP/
│       ├── boot/
│       │   ├── generic.readme
│       │   └── linux.bif
│       ├── qemu/
│       │   ├── pmu_args.txt
│       │   └── qemu_args.txt
│       └── smp_linux/
│           ├── image/
│           └── qemu/
│               ├── pmu_args.txt
│               └── qemu_args.txt
└── kv260_ispMipiRx_vcu_DP.xpfm
```

Figure 4.3: Resulting folder of the *Vitis extensible platform* creation process.

After the audio support removal, the hardware specifications should match those ones described by the Kria documentation about the *Smartcamera* application [34]. The resulting platform is now described by separating it into some fundamental parts:

Capture pipeline The block scheme 4.4 illustrates how should be the capture pipeline of the project. The AR1335 sensor and the AP1302 ISP are part of the carrier card, indeed they are not considered in the Vivado project that will start with the input pin `mipi_phy_if`, i.e. MIPI physical interface.

The capture pipeline is grouped in the Vivado hierarchical block shown at 4.5. It starts with the *MIPI CSI-2 Rx Subsystem* that receives the video frames from the ISP in YUV 4:2:0 format and outputs them as *AXI4-Stream* 32-bit video data[2]. Then the *Subset Converter* takes these data and converts them to 48-bit, by adding zeros to the MSB of each data word. At the end of this pipeline the data are converted into AXI4-MM (memory mapped) format and written in the DDR memory by the *Video Frame Buffer Write* IP.

This hierarchical block is, then, placed in the PL side of the Vivado block design and connected to the PS as illustrated in the diagram 4.4 and explained at the next PS description.

Processing System The PS of the Zynq Ultrtascale+ MPSoC is provided by Vivado on its IPs catalog. It is shown on figure 4.6 where all its ports are listed. As described at section 1.3.2, the PS contains ARM Cortex-A53 processors that use the AMBA AXI4 standard interface for the communication, comprising multiple high-performance (HP) switches to connect the system resources[17]. The AXI buses can be distinguished in two power domains, full-power (FPD) and low-power (LPD), but also in Slave (input) and Master (output).

They define the communication between PL and PS as summarized at tables 4.1 and 4.2, with a particular note on `S_AXI_HP1_FPD`, `S_AXI_HP3_FPD`, `M_AXI_HPM0_FPD` and `M_AXI_HPM0_LPD`. These PS ports are connected to the other PL components through the *AXI Interconnect*, special IPs providing advanced routing capabilities, such as arbitration and prioritization to handle the cases where one signal should drive more than one, or vice versa.

VCU The Video Codec Unit is an IP capable to simultaneously compress and decompress video streams. It has been placed to give the possibility to use two further pipelines: using an external memory as input source with decoding before the inference, but also to visualize the results by a VCU-encoded stream through Ethernet.

Clock, Reset and Interrupt The PS has the limitation that maximum 4 clock signals can be generated and that their phase is not aligned[37]. To overcome it, the placed PS IP provides just a 100 MHz clock source `pl_clk0` and the other needed clock frequencies for the PL components are generated by a *Clocking Wizard* IP. They are listed at 4.7 with the related PL component that uses each of them.

Moreover the *Processor System Reset* IPs are also needed to create reset signals for each clock, as the clock export setup requires. The external input

[2]The MIPI CSI-2 Rx Subsystem outputs 32-bit *AXI4-Stream* video data at two pixels per clock (PPCS) and eight bits per pixel. AXI4-Stream is a protocol designed in the Vitis HLS programming context, for transporting arbitrary unidirectional data.

reset used for the clocking wizard and these processor system reset IPs is provided by the PS, named `resetn_0`. In addition to the synchronous reset signals[3], there are three `emio_gpio_o` reset used for the Frame Buffer Write IP, the AP1302 ISP and the VCU.

To conclude this design description, the interrupt signals should be considered. Most of all the PL components generates an interrupt that will be handled by the PS. However the PS input port `pl_ps_irq1` is set to accept an 8-bit signal. A concatenation IP is used to put together these signals and insert it into the PS[4]. Reset and clock connections are summarized at 4.8.

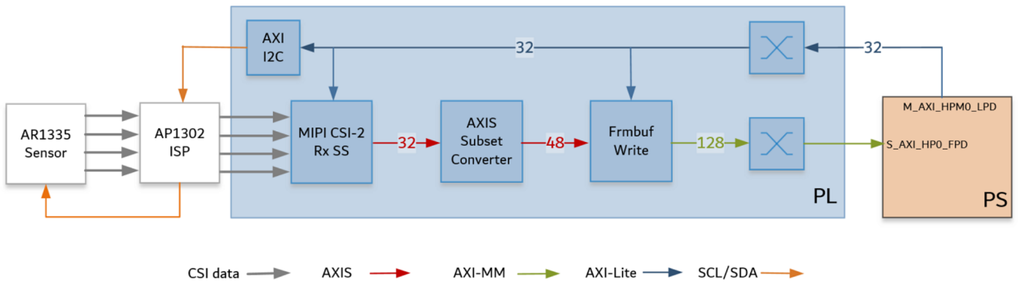


Figure 4.4: Capture pipeline of the *Smartcamera* application[34].

Established the block design organization and all the PL and PS configurations, it can be observed that some of the components' ports may not be properly connected, specifically those ones referred to DPU and pre-processing kernels that have been not yet inserted in the design. These connections are now ignored and then they will be added as overlay by Vitis, see section 4.2.2.

Finally, after Validation, Synthesis and Implementation with all the errors and warnings solved, the extensible hardware platform XSA (Xilinx Support Archive) has been generated. This archive contains the whole project, comprised:

- The bitstream, a .bit file including the description of the hardware logic, routing, and initial values for both registers and on-chip memory. It has bits and more human-readable fields, like assembly code.
- The hardware platform .hpfm, a metadata XML file describing the design hardware interfaces. Specifically the contained information are about: clock ports, AXI bus interfaces, AXI4-Stream bus interfaces and interrupts.

[3]Reset signals synchronous to the clock sources.

[4]The unused interrupt bit are imposed to 0

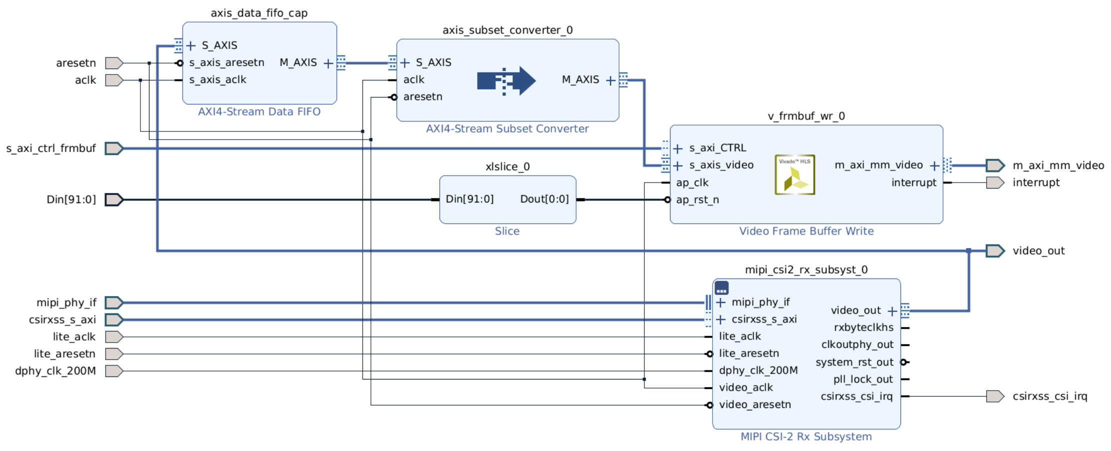


Figure 4.5: *capture_pipeline* hierarchical block of Vivado project.

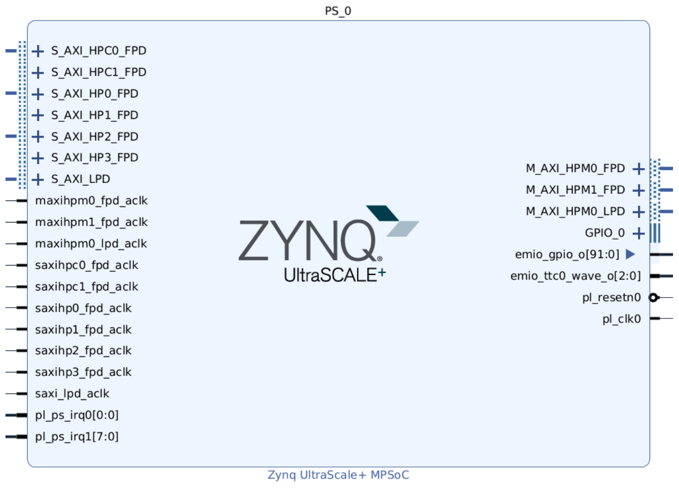


Figure 4.6: *PS* IP of the Zynq Ultrascale+ MPSoC in the Vivado project.

All these operations are supported by the Vivado GUI or by tcl commands and they lead to generate in the `kv260_ispMipiRx_vcu_DP/` folder the `project/` directory containing all the Vivado design files and the XSA above mentioned.

To complete the whole platform creation, the last step is the **software platform** building. It has to support a Linux OS running on it, so a series of software components need to be prepared in advance. However in this development flow these files are handled after, at the PetaLinux image building 4.2.3. Therefore, here, just the directories predisposition and the metadata files are generated thanks to the tcl scripts made available by the *kria-vitis-platforms* GitHub repository. A

PS slave	PL master
VCU	
S_AXI_HPC0_FPD	M00_AXI_VCU_EN
S_AXI_HP2_FPD	M00_AXI_VCU_DEC
S_AXI_LPD	M_AXI_VCU_MCU
DPU	
S_AXI_HPC1_FPD	M_AXI_HP2
S_AXI_HP1_FPD	M_AXI_GP0/HP0
Pre-processor	
S_AXI_HP3_FPD	m_axi_gmem0/1/2/3
Capture pipeline	
S_AXI_HP0_FPD	m_axi_mm_video

Table 4.1: PS slave - PL master interconnections in the Vivado project.

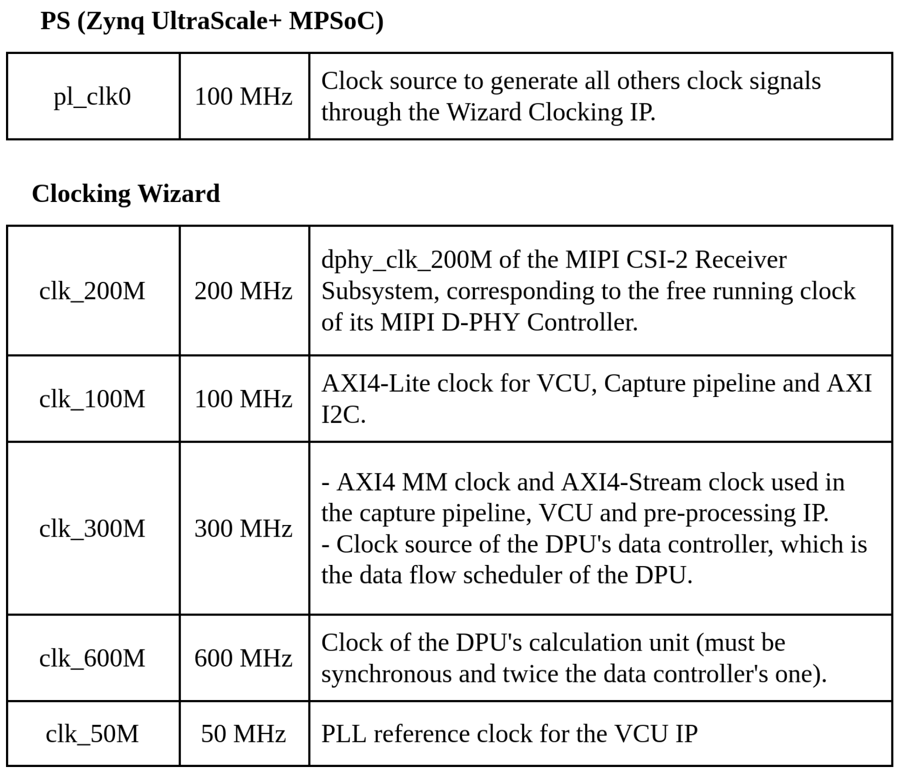
PS master	PL slave
M_AXI_HPM0_FPD	S_AXI_CONTROL (DPU) S_AXI (AXI Verification) S_AXI_CONTROL (Pre-proc)
M_AXI_HPM1_FPD	s_axi_ctrl_frmbuf (c.p.)
M_AXI_HPM0_LPD	S_AXI (I2C) csirxss_s_axi (c.p.) S_AXI_LITE (VCU)

Table 4.2: PS master - PL slave interconnections in the Vivado project.

makefile can be used to launch the *xsct* command[5] with the execution of the tcl script. The before-built hardware platform and the predisposition to the software one are put together and the result is the folder in figure 4.3 having:

- **Hardware platform** - The `hw/` folder has the before explained XSA archive and HPFM file.
- **Software platform** - The `sw/` folder has just the predisposition for the software components, like boot files and Linux images, and the SPFM file, that is a XML metadata file where to find the software components.
- **Xilinx platform** - The *.xpfm* file is the XML metadata file having the paths to the two previously described platforms.

[5]XSCT is a Vitis tool (Xilinx Software Command-Line Tool) that allows the user access to the full set of SDK tools from the command line.



PS (Zynq UltraScale+ MPSoC)

pl_clk0	100 MHz	Clock source to generate all others clock signals through the Wizard Clocking IP.

Clocking Wizard

clk_200M	200 MHz	dphy_clk_200M of the MIPI CSI-2 Receiver Subsystem, corresponding to the free running clock of its MIPI D-PHY Controller.
clk_100M	100 MHz	AXI4-Lite clock for VCU, Capture pipeline and AXI I2C.
clk_300M	300 MHz	- AXI4 MM clock and AXI4-Stream clock used in the capture pipeline, VCU and pre-processing IP. - Clock source of the DPU's data controller, which is the data flow scheduler of the DPU.
clk_600M	600 MHz	Clock of the DPU's calculation unit (must be synchronous and twice the data controller's one).
clk_50M	50 MHz	PLL reference clock for the VCU IP

Figure 4.7: Clock interconnections in the Vivado project.

The Vitis extensible platform is ready, its correctness can be verified by the Vitis tool:

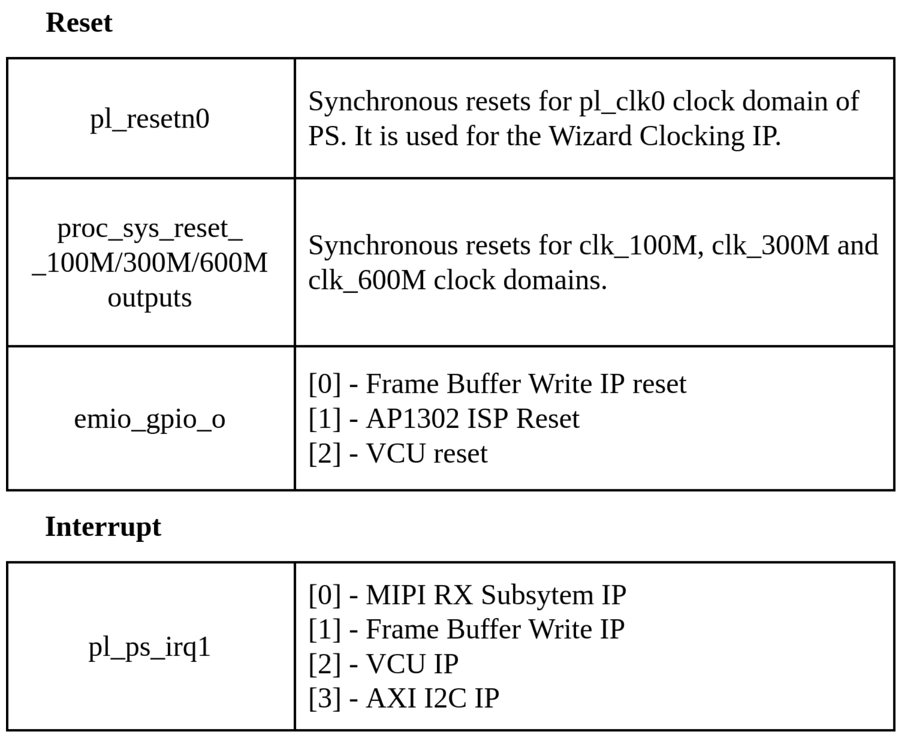
```
platforminfo kv260_ispMipiRx_vcu_DP.xpfm
```

that extracts the platform main features, comprised clock information, resource availability and the unconnected bus that will be used for the next step, the overlay.

4.2.2 Overlay creation

Following the development flow 4.2, after the *platform design* that led to the acceleration platform generation, the *overlay* has to be developed. It comprises a series of PL kernels that can be made by C/C++/Python code or by a RTL description. In this thesis project the overlay, as shown at figure 4.1 is composed of two kernels:

- **DPU** IP - As said before it was not comprised in the Vivado IPs catalog but is released embedded in a TRD (targeted reference design). It practically means



Reset

pl_resetn0	Synchronous resets for pl_clk0 clock domain of PS. It is used for the Wizard Clocking IP.
proc_sys_reset_ _100M/300M/600M outputs	Synchronous resets for clk_100M, clk_300M and clk_600M clock domains.
emio_gpio_o	[0] - Frame Buffer Write IP reset [1] - AP1302 ISP Reset [2] - VCU reset

Interrupt

pl_ps_irq1	[0] - MIPI RX Subsytem IP [1] - Frame Buffer Write IP [2] - VCU IP [3] - AXI I2C IP

Figure 4.8: Interrupt and reset signals in the Vivado project.

that a directory with all the needed files to implement can be downloaded by the AMD website. Then this TRD can be added to the Vivado block design or as PL kernel by Vitis (this case). The Kria documentation shows the ideal connection with the PS component and some of the configurations to set, see figure 4.10.

- **Pre-processing** IP - The input frames need to be modified before to run the inference process. The best way to create an IP able to act some pre-processing tasks is to exploit the Vitis Vision library. It contains C++ algorithms performing many processing on images. The most interesting for this application are: *color space conversion* from NV12 (output from MIPI camera) to BGR (expected by neural network), *resizing* to change the resolution of the input image to match that one used to train the network, 1216×320, and the *quantization*, i.e. a linear transformation (scaling and shifting) of each pixel of the BGR frame to satisfy the DPU input requirement. Its connection with the PS is summarized at the Kria documentation, see figure 4.9.

The DPU is already downloaded as TRD in the `overlays/` directory. It is the

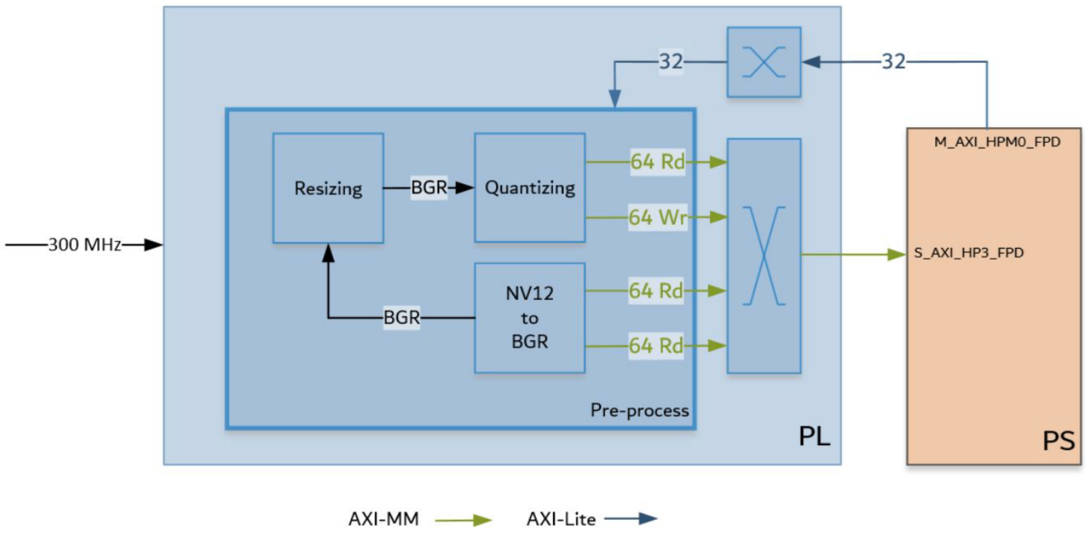


Figure 4.9: Pre-processing IP of the *Smartcamera* application[34].

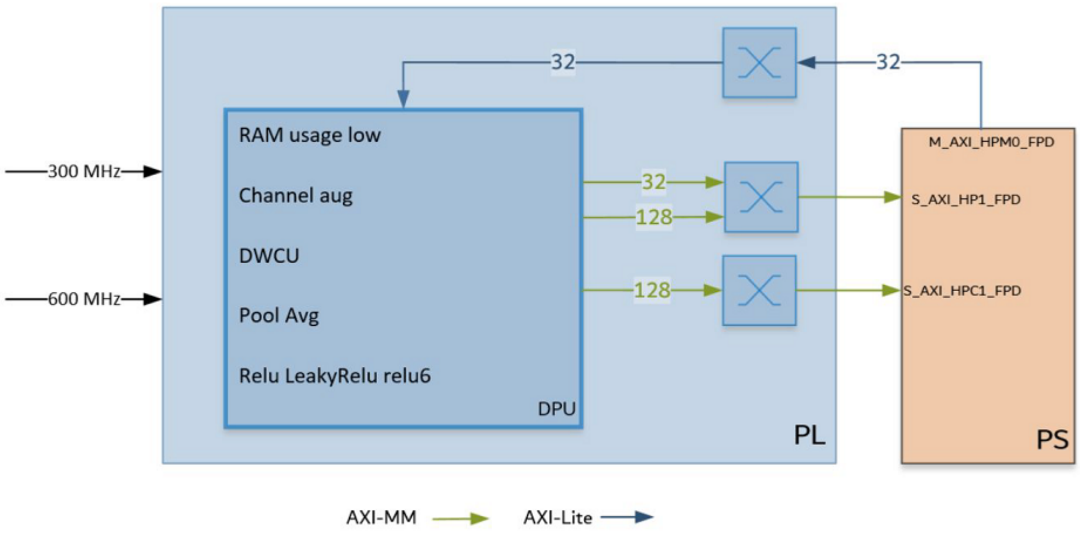


Figure 4.10: DPU IP of the *Smartcamera* application[34].

DPUCZCX8G version used by the Zynq Ultrascale+ MPSoC. Its configuration file *dpu_conf.vh* is contained in the `examples/smartcam/` directory and it can be properly modified to match this project's requirements. To have a comprehensive understanding of all the settings, the documentation can be consulted[19]. The chosen convolution architecture of the DPU is called *B3136*, it is characterized by a pixel parallelism of 8 and input/output channel parallelism of 14. The name comes from the resulting peak number of operations per cycle, thus $2 \times 8 \times 14 \times 14$, since in each clock cycle the convolution array performs a multiplication and an accumulation. The images and weights buffer use the UltraRAM and moreover the other configurations are: low RAM usage, channel augmentation, alu parallel = PP/2, conv: leaky ReLU + ReLU6, alu: ReLU6 features, and high DSP usage. The DPU configuration file, .vh formatted, is shown at B.1.

The whole overlay procedure is automated by the Makefile provided by the GitHub repository. All of the issued tasks are executed by the AMD software Vitis. It comprises Compiler, Linker and Packager tools, accessible by the `v++` command and able to generate different kinds of files useful to build the final deployable project. To follow this overlay building description, the `examples/smartcam/` folder is shown at figure 4.11.

1. **Pre-processing .xo file generation** - The Vitis Compiler is used to generate the pre-processing kernel. It is written in C++ (cpp and h files listed starting from B.2) and uses some functions get by the Vitis Vision Libraries. The key point is to set the right resizing dimensions 1216×320 to match the training images. The generated file is a Xilinx object kernel *.xo* that is actually an archive with the created IP as TRD and other configuration files.

2. **DPU .xo file generation** - It is generated by Vivado with the support of some pre-built tcl scripts, see them on figure 4.11.

3. **FPGA .xclbin and .bit files generation** - Built the PL kernels in .xo formt, they are now linked together into a FPGA executable binary file, the .xclbin. It typically comprises different parts of the compiled application, like the bitstream itself and some structured metadata to define memory topology, IP layout of instantiated peripherals and kernels, clocking details and kernel connectivity. A complete access to this kind of files is given by the *xclbinutil* tool, it can read, write and change xclbin files. Therefore, the `v++` command is used to make the Vitis compiler and linker generating this file. The command options are set in a well organized way through a properly formatted file shown at B.5. Besides the `v++` arguments, also clock and connectivity information are listed, they should match the unconnected ports leaved in the platform by Vivado (also visible at `platforminfo` output). Finally, also the updated bitstream file is generated.

4. **Final Vivado project generation** - Placed also the overlay, the entire Vivado block design is ready and in this overlay building a Vivado .xpr project is generated to visualize it. For sake of simplicity all the tables cited at the

previous platform section 4.2.1 are already updated with the overlay updates. See 4.2, 4.1, 4.7 and 4.8.

```
kria-vitis-platforms/kv260/overlays/examples/smartcam/
├── dpu_conf.vh
├── kernel_xml/
│   ├── dpu/
│   │   └── kernel.xml
│   └── sfm/
│       └── kernel.xml
├── Makefile
├── prj_conf/
│   ├── prj_config_1dpu
│   └── strip_interconnects.tcl
├── scripts/
│   ├── bip_proc.tcl
│   ├── gen_dpu_xo.tcl
│   ├── gen_sfm_xo.tcl
│   ├── package_dpu_kernel.tcl
│   └── package_sfm_kernel.tcl
├── xf_config_params.h
├── xf_pp_pipeline_accel.cpp
└── xf_pp_pipeline_config.h
```

Figure 4.11: Starting folder of the *overlay* building process.

4.2.3 PetaLinux image

As the platform development flow **??** shows, the last step leads to generate a SD card image based on a Linux operating system, with all the platform's components built until now. The AMD tool in charge of building this image is *PetaLinux*, able to customize, build and deploy embedded Linux solutions on AMD processing systems. The main components of this tool are [38]:

- **Yocto Extensible SDK** - Yocto is an open-source project helping to create custom Linux-based systems regardless of the hardware architecture. In this thesis work, the architecture is well defined, Zynq Ultrascale+ MPSoC, and its Yocto components are labeled by the PetaLinux environment as "*aarch64*".
- **XSCT** - Xilinx Software Command-Line Tool already mentioned at 4.2.1
- **PetaLinux CLI commands** - used to execute the macro-steps of the PetaLinux development flow. They will be mentioned in the following.

An optimal staring point for this flow is to create a PetaLinux project by a BSP (Board Support Package) [6]. It is a reference design specific for a certain hardware, Kria KV260 in this case, containing design and configuration files, pre-built and tested hardware and software images to be downloaded on the board.

The main objective now is to develop a so called PL application, that comes from the packaging of the previously built PL overlay. Then, it will be added to the target root file system thanks to the PetaLinux process, allowing the *xmutil* utility to load the overlay as an *accelerated application* after that Linux has booted on the Kria KV260 board.

After PetaLinux 2022.2 installation and the KV260-compatible BSP download, the project can be created by the CLI command:

```
petalinux-create -t project -s kv260-v2022.2.bsp
```

A directory[7] with all the starting files needed for the PetaLinux flow is generated. Now the description continues with the addition of some components needed for this project: the FPGA firmware containing the PL description built so far, some software packages for the application code running and the AP1302 firmware.

When the application is loaded on the board by issuing `xmutil loadapp`, the *DFX Manager* will be invoked. It is a Xilinx library implementing an on-target daemon that is used to handle the application's data model, to active the PL configuration, and to load/unload the corresponding bitstreams. The DFX Manager requires specific files that will be placed in the target Linux system at folder:

```
/lib/firmware/<company_name>/<app_name>
```

However to generate a fully complete Linux image, ready to run the target application, these necessary files must be prepared before into a Yocto recipe of the PetaLinux project:

- **xclbin and bit** - They are the two output files of the PL overlay building procedure of the previous section 4.2.2.
- **shell.json** - The DFX Manager requires information about the design, especially if it is slotted or flat. A simple .json metadata file with this information is contained in the kria applications firmware examples about the smartcamera one [39]. It has just this lines:

[6]The BSPs can be directly downloaded by the AMD download center.

[7]In the following this directory will be referred with the name of `<plnx_prj>`.

```
{
    "shell_type" : "XRT_FLAT",
    "num_slots": "1"
}
```

- **dtsi** - Vivado generates the hardware description of the custom PL IP design as XSA file. However, the PL IP is loaded in the KV260 after the Linux booting up, thus to be loaded dynamically, a Linux-understandable format is needed for this hardware description [40]. This is the reason of the necessity of a *device binary tree* file. A human-readable device tree file is the .dtsi format and it is generated by the Vitis DTG (Device Tree Generator). It is run by the XSCT tool:

  ```
   createdts -hw <XSA> -zocl -out <outdir> -platform-name <
  pfm_name> -git-branch xlnx_rel_v2022.2 -overlay -compile
  ```

  that writes the PL hardware description, get from the XSA archive, in .dtsi format. This description has not some fundamental design components, like AP1302 ISP, I2C Mux Connecting, ZOCL and mipi nodes, thus some modifications are done following the kria documentation's tutorial available at [34] and resulting in the file B.6. Finally to compile this file into a machine-readable .dtbo format the Vitis DTC (Device Tree Compiler) is used. This step is automatically done during the recipe creation of the PetaLinux building process.

Prepared into a generic folder, these files are processed by the *fpgamanager_dtg* tool of PetaLinux in charge of creating automatically the needed Yocto recipe for the PL overlay, i.e. for the accelerated application.

```
 petalinux-create -t apps --template fpgamanager -n kv260-ml-
accel --enable --srcuri "path/to/bitfile path/to/dtsi path/to/
xclbin path/to/shell.json"
```

After this command, the recipe *kv260-smartcam* is generated inside the folder visible at 4.12. It includes the above mentioned files and the Yocto recipe metadata file (.bb), see it at B.7.

The next point is to create a further recipe with the aim of adding to the embedded Linux image some software packages. These specific packages are required to support the ML inference on the board and so the application code

```
<plnx-prj>/project-spec/meta-user/recipes-firmware/kv260-smartcam/
├── files/
│   ├── kv260-smartcam.bit
│   ├── kv260-smartcam.dtsi
│   ├── kv260-smartcam.xclbin
│   └── shell.json
├── kv260-smartcam.bb
└── README
```

Figure 4.12: Resulting FPGA firmware recipe.

running, indeed its content will be clearer at the application development section 4.3. The directory

```
<plnx-prj>/project-spec/meta-user/recipes-apps/smartcam/
```

will be filled by a recipe metadata file, `smartcam.bb` at B.8, listing all these software packages.

The last recipe to include is for the AP1302 firmware. The Yocto process is able to get a file by a specified online source, exactly as this case where the AP1302 firmware is provided by the Xilinx GitHub repository [41]. The two needed recipe files are `ap1302-ar1335-single-firmware.bb` at B.9 and `ap1302-firmware.inc` at B.10, both of them to insert at the folder:

```
<plnx-prj>/project-spec/meta-user/recipes-firmware/ap1302-firmware/
```

The three main components have been prepared, now they are enabled by adding a final recipe to:

```
<plnx-prj>/project-spec/meta-user/recipes-core/packagegroups/
```

It is `packagegroup-kv260-ml-accel.bb` at B.11 and it lists the three prepared recipes. Finally the string `"CONFIG_packagegroup-kv260-ml-accel"` is appended to the file:

```
<plnx-prj>/project-spec/meta-user/conf/user-rootfsconfig
```

and the option `packagegroup-kv260-ml-accel` is selected among the user packages, accessible by the PetaLinux configuration command:

```
petalinux-config -c rootfs
```

Now the final image building, taking into account all the configurations previously described, can be issued easily typing:

```
petalinux-build
```

This process is not trivial, it requires to know the PetaLinux tool to set properly all the environment, for this reason it often requires a long troubleshooting to obtain a working result. Additionally it also requires many time and memory resources of the host machine, the RAM free memory should be monitored to avoid building failures. On the process completion three directories are generated:

- `build/` for the files generated during the building process, as the Yocto ones in `tmp/`
- `images/` for the bootable images.
- `components/` with the Yocto eSDK, generated at petalinux-config and petalinux-build execution.

The final deployable image is generated by a packaging step:

```
petalinux-package --wic --bootfiles "ramdisk.cpio.gz.u-boot
boot.scr Image system.dtb"
```

where the listed boot files, together with the default images inside *images/linux/*, are used to generate a *wic image*. The final `images/linux/petalinux-sdimage.wic` can be written into the SD card of the target platform, see 5.1 for details.

Fetching errors During the PetaLinux building process, some errors about the online fetching step (do_fetch Yocto action) often occur. In these cases a good solution is a manual intervention by downloading the files from the GitHub source and put in the correct final location. Two cases:

- AP1302 firmware - At the target deploying step 5.1 this file should be inside the `/lib/firmware/`. If it is not found, an easy way to solve the problem is to manually put it there, after having downloaded it from GitHub. This can be done because the default procedure does not return any error or warning actually.
- Smartcam files - This step arises in an error about the *do_fetch()* operation causing the inability to proceed. A manual way to solve it is to analyse the error and identify the Yocto project directory destination for this files. Then they are downloaded by Github and put manually. At the PetaLinux building process termination, these files will be found at `/opt/xilinx/kv260-smartcam/`.

4.3 Application Development

Platform and model have been properly developed so far. Following the entire work flow diagram 1, the final step is the *application code* development. It is a program written in a certain programming language, like C++ or Python, that executes all the required practical operations of this design. To have an easy understanding of its behaviour, the idea is that the application code has to enable all the steps of the end-to-end pipeline described above at section 4.1, thus from the images acquisition to the final display of the inference results.

Each application code's task will be analysed, in the following, simultaneously to the specific code function used for that purpose. Before to focus on them, the programming language and its used libraries must be chosen. The Vitis AI environment exploited for the model development step, provides the *Vitis AI Runtime* (VART) library. It is a set of APIs based on the XRT library (Xilinx Runtime) that allow to run the model inference on the DPU hardware architectures. The base element is, indeed, the *runner*, that is an application level runtime interface for DPU IPs based on XRT. It uses XIR graphs (i.e. the model format after compilation) as inputs and runs them on different targets. The VART library is provided as open-source at the `Vitis-AI/src/vai_runtime/` folder of the Vitis AI GitHub repository [32]. This library together with many other ready-made application examples are provided in the Vitis AI library. Although these examples are a good starting point to build an application, they represent a limitation to all the possibility that the single VART APIs give.

Going in a deeper VART library analysis is not an aim of this thesis work, since an alternative more efficient in multimedia pipelines is found in the GStreamer framework. It is suggested by analysing the Kria documentation and specifically the pre-built *Smartcam* application [34]. GStreamer is a library for constructing graphs of media-handling components. Its supported applications range from simpler audio and video streaming, to more complex audio mixing and video non-linear processing [42]. Since this thesis application consists of capturing and display images, the use of this framework is very suggested. It consists of many plug-ins to sequentially interconnect each others with the goal to build a *multimedia pipeline*. The Gstreamer framework is available as C++ or Python library, in both cases the key approach is to build a string with all the required components.

In this case, where the target is an AMD platform, some additional GStreamer plug-ins are provided from the *Vitis Video Analytics SDK* (VVAS). It helps with the NN inference step allowing to access: the DPU via the VAI Library and the accelerated functions from the Vitis Vision Library. Hence, using GStreamer instead of the common python packages is better by a performance view since there is hardware underlying all the steps (comprised the pre-processing).

So now, the best way to understand the application code is to subdivide its behaviour in the five main tasks and analyse each piece of the GStreamer pipeline string that is entirely shown at C.4.

Input video acquisition The input images are frames of a video captured by the MIPI AR1335 sensor, so a suitable plug-in for this purpose is *mediasrcbin*. It is actually provided by Xilinx which is built on top of a standard GStreamer plug-in, *v4l2src*. Its advantage is the automatic initialization and configuration of the input pipeline. Indeed, it allows to set a proper device topology able to capture the video frames starting from the MIPI AR1335 camera.

This mentioned topology is composed by some sub-devices that corresponds with the capture pipeline seen before: AR1335 camera - AP1302 ISP - MIPI CSI2 Rx Subsystem. All of these components have source (input) and sink (output) pads and their main features, like media bus format, dimensions and frame rate are properly set by the mediasrcbin plug-in.

Identified this first plug-in to add to the GStreamer pipeline string, the application code's task here is limited to check the presence of the correct acquisition pipeline described so far. This check is easily performed thanks to this v4l-utils[8] application:

```
media-ctl -d /dev/media0 -p
```

It uses the Linux Media Controller API to visualize the topology of the device node of interest in this case, found as */dev/media0/*. The topology shows as the last entity the `vcap_capture_pipeline_mipi_csi2` component, it has the sink pad to collect all the input frames acquired from the rest of the acquisition pipeline (AR1335 - AP1302 - Rx Subsystem).

Hence, verified the device presence, the pipeline can be filled by the line 1 of the pipeline C.4. The line 2 is added to filter the images with those specific features. This allows to introduce at the pre-processor just the proper-formatted data.

Pre-processing After the capture pipeline, the input frames are stored in DDR memory and, before to be processed by the ML inference, the previously described 4.2.2 pre-processing operations must be performed. Thanks to the presence of an hardware support, the pre-processor IP of the overlay part, an execution faster than a just software implementation is provided.

[8]v4l-utils is one of the software packages installed before by the PetaLinux building process.

On this purpose just the support of the VVAS plug-in[43] *vvas_xmultisrc* is needed. It requires the use of a configuration file, see *preprocess.json* at C.1, containing information about:

- The xclbin path - The plug-in needs to know the location of the *xclbin* file used to program the FPGA device. It will be downloaded and an XRT handle for memory allocation and programming kernels will be created.
- The VVAS library path where finding all the acceleration software libraries.
- The information about the kernels to implement - In this case the kernel is the pre-processor `pp_pipeline_accel` that is supported by the acceleration software library `libvvas_xpp.so`.

About the pre-processing kernel above mentioned, some configuration parameters are required. The Kria *Smartcam* documentation has some specific indications[44] about it:

- The mean R, G and B values are 0 since they must match the prototxt file provided by Vitis AI model zoo.
- The scale R, G and B values are 0.25 because they come by the multiplication between the scale value provided by the prototxt 0.00392156 and 2^x. The exponent is called *fixpos* and for the used model its value is 6. An easy way to consult the model's features is the Vitis AI command:

```
xdputil xmodel yolov5_kv260.xmodel -l
```

The GStreamer pipeline sub-string to add at this point is made by lines 3-6 of C.4, where the *tee* component will split the data in order to, then, attach them with the inference output metadata, i.e. the rectangular boxes surrounding the detected objects.

DPU Inference Having pre-processed the input frames, they are ready to be given at the input of the model. Very closely to the previous step, here, a single VVAS plug-in is used to run the AI inference on the images. It is the *vvas_xfilter*.

It works as interconnection between the high-level application interfacing with the user and the underlying Vitis AI library interfacing with the DPU. This action leads to the actual AI inference task, generating the bounding boxes metadata. Even this plug-in needs just of a configuration file to properly operate, see the *aiinference.json* at C.2.

In addition to the xclbin and vvas library locations, some other information are needed:

- The GStreamer element mode to operate - In this case the *inplace* mode is chosen because the intention is to alter the input buffer itself instead of producing new output ones.
- The information about the kernels to implement - In this case the kernel object is supported by the acceleration software library `libvvas_xdpuinfer.so` able to run the AI inference on the underlying DPU.

About the above mentioned kernel object, some configuration parameters are required to set the AI model to use for the inference. It has been called, after the compilation, *yolov5_kv260* and belongs to the YOLOV3 class. Indeed. analysing the Vitis AI Library examples, it can be noticed that all those ones referred to YOLOv5 used YOLOv3 as model class. Moreover the pre-processing is not needed because it has been implemented manually before.

Therefore the GStreamer pipeline string is now updated with the line 7 of C.4.

Output bounding box To display the output images with the bounding boxes around the detected vehicles, some steps must be performed. As seen before, the DPU inference acts on the pre-processed data that are converted and scaled compared with the original frames. This means that the inference output features will not match the original frames but the pre-processed ones.

To face out this issue, the *vvas_xmetaaffixer* plug-in can re-size the metadata to match again the original frames. It has two kinds of input ports: the sink_master that acquires the scaled metadata and the sink_slave that recover the original ones coming from the *tee* component. The scaling ratio is set by easily comparing the data between the slave and master sink pads.

The output drawing of the re-scalded metadata on the input images is performed by the *vvas_xfilter* plug-in. The configuration file *drawresult.json*, shown at C.3, beyond the previously cited settings, has information about the appearance of the rectangular boxes to display, like labels font, size, name and RGB color code.

The GStreamer pipeline string is updated by the lines 8-14 of C.4.

Output display Considering an HDMI or a DisplayPort to display the resulting video frames, the *kmssink* GStreamer plug-in is employed by setting all its configurations. The pipeline string is completed with the line 15 of C.4.

The whole pipeline, above analysed, is built by means of a Python script, *application.py* at C.5. The GStreamer libraries required to build and run the pipeline in a Python script are comprised in the Python module *gi*, i.e. GObject Introspection. After all the operations described in this section, the state of the GStreamer pipeline built so far, is set to *PLAYING* and the system enters in an infinite loop (the GLib MainLoop feature) until the process termination signal is sent by typing CTRL+C (SIGINT Linux signal). During this infinite loop the GStreamer pipeline runs. Its graphical view is provided at 4.13, based on the dot representation produced by a GStreamer debug feature.

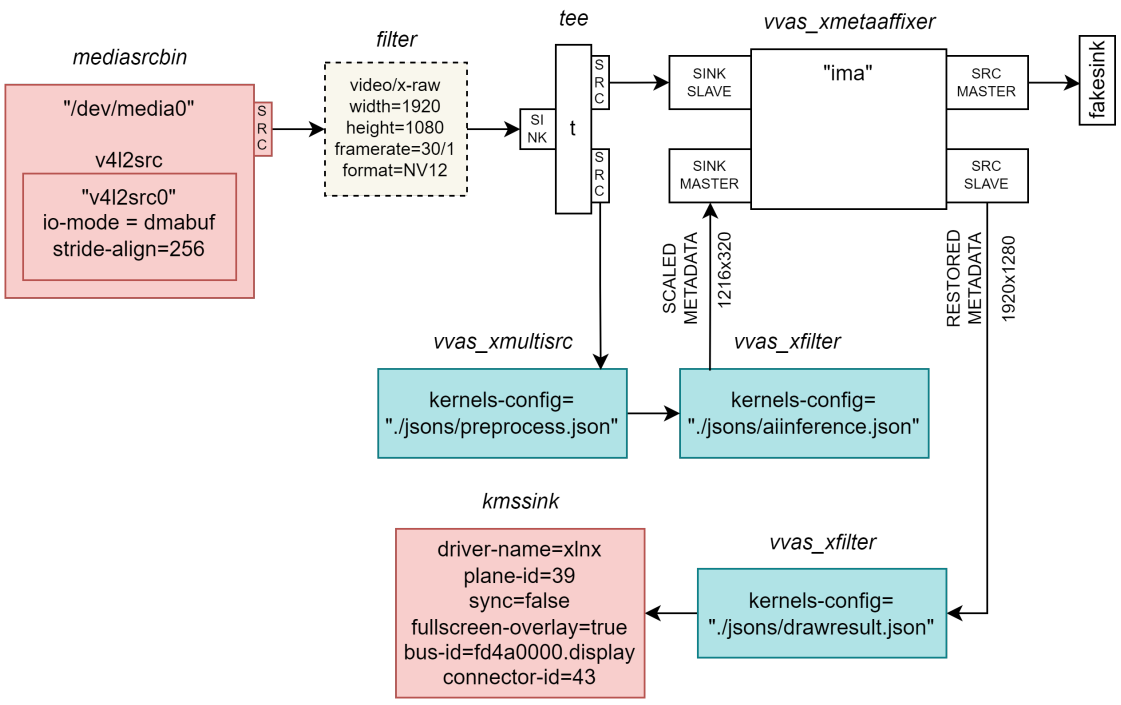


Figure 4.13: GStreamer pipeline's graphical view based on the debug dot file.

Chapter 5

Final Considerations

5.1 Deploy on target

The complete design flow, figure 1, has been fully detailed in the previous chapters. Here the final step is addressed: *deploying on target* the whole built design.

Starting on the host machine, it is used initially to write the *wic image* to the KV260 board's sd card. It is the final result of the platform development section 4.2 that contains the fully customized hardware and software platform to can run the AI model on the Zynq UltraScale+ MPSoC with its DPU support. Then, all the required files to run the application are collected: the application code and the compiled model with their configuration files. The figure 5.1 shows in details the directory content that will be copied into the sd card at the path `/home/root/`.

Before to turn on the FPGA device system, a board preparation is needed. The KV260 board scheme is on figure 5.2:

- Connect the AR1335 camera to the J7 IAS connector, since it is that one linked to the MPSoC through the ISP AP1302.
- Connect the J5 HDMI connector to a monitor through a HDMI cable.
- Insert the micro SD card into J11.
- Connect the J4 micro-usb connector to the USB port of the host machine to ensure an UART communication.

Now, after having identified the COM port name corresponding to the host machine's UART, the connection can be made by:

```
sudo putty /dev/ttyUSB1 -serial -sercfg 115200,8,n,1,N
```

```
target_kv260/
├── application.py
├── jsons/
│   ├── drawresult.json
│   ├── aiinference.json
│   └── preprocess.json
└── models/
    └── yolov5_kv260/
        ├── label.json
        ├── md5sum.txt
        ├── meta.json
        ├── yolov5_kv260.prototxt
        └── yolov5_kv260.xmodel
```

Figure 5.1: Directory to entirely deploy on the KV260 sd card.

Just connected the power supply to the board DC jack, the terminal shows the booting up messages and, after the password setting, the Linux environment is ready to be used. As said before, the application must be loaded by the DFX Manager and, on this purpose, the xmutil commands are exploited. Initially to list the existing application firmware available on the board, this command is issued:

```
sudo xmutil listapp
```

The active application firmware, initially the default one, will have the value *Active_slot* corresponding to 0. To enable the *kv260-smartcam* firmware, created at the PetaLinux section 4.2.3, the default application must be unloaded before:

```
sudo xmutil unloadapp
```

and the desired one is loaded then:

```
sudo xmutil loadapp kv260-smartcam
```

At the end, the `listapps` must be addressed again in order to call a rescan by the DFX Manager that will update definitively the firmware direcotry tree. It

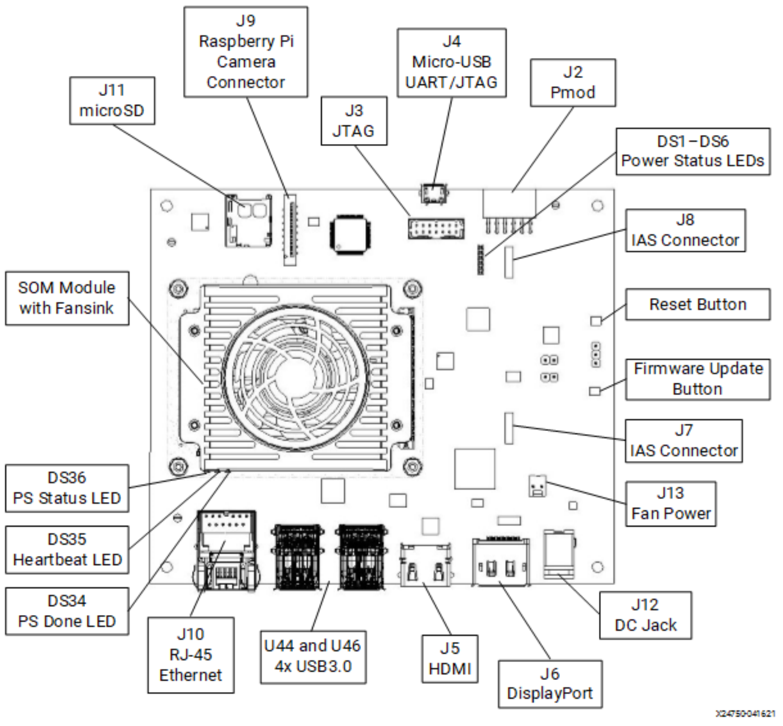


Figure 5.2: KV260 Vision Starter Kit Interfaces and connectors [16].

helps even to check that the desired application firmware has the *Active_slot* value turned into 0.

After these commands the application firmware is enabled, so now the application can be run. Being a simple python script, the only operation to do is to move on the `/home/root/` directory where all the files collected before has been copied, and type:

```
 python3 application.py -s mipi -o display -W 1920 -H 1080 -n
False -f False
```

The meaning of these options is on the application code at C.5. When the application is lunched, the connected HDMI monitor starts to display the images captured by the MIPI sensor and when an object is detected, the bounding boxes should be printed surrounding it.

An additional note is done aboute the option `-f` that, if true, it enables the GStreamer plug-in *fpsdisplaysink*. It is fundamental to have an idea of the output performance and specifically of the output frame rate. Actually an optimal tool for the performance evaluation is the Vitis AI Profiler, however it works only if the inference is issued by the VART library. In this application case the GStreamer framework is used, so its plug-ins must be used on this purpose.

5.2 Issues and possible solutions

The followed work flow had several design layers, it went from the model development, with training, quantization and compilation, to the target platform design, comprising of software and hardware features. This means that critical design points could be found at many of these layers.

All the thesis work flow has been properly carried out but the final result, i.e. the visualization of the detection objects' bounding boxes, has not been fully achieved. Here, some possible reasons with viable solutions are discussed.

AI model The chosen model is YOLOv5n, that corresponds to the *nano version* of YOLOv5. It has the least number of convolutional layers and filters compared with all the other versions. It has been chosen since it is the best version for an embedded system and also because it does not require a high-performance training machine to reach reasonable accuracy levels. However it typically achieves a lower accuracy since less complex features can be learnt from the model's layers. This model has been *trained* in a non-optimized way due to the absence of a GPU, achieving a mAP@50-95 of just 0.433.

Another point that has contributed to the lowering of this accuracy is the *quantization* process. This step has translated the weights into integer leading to a leak of accuracy, on this purpose an improvement is the QAT (Quantization Aware Training) procedure that could be exploited to fine-tune the model reaching again an higher accuracy.

Pre-processing and images stretching The Kitti training dataset that has been used, is made of images all different among them but with an aspect-ratio almost equal to 19:5. This means that the input frames, captured by a mipi camera at 16:9, have been stretched too much by the pre-processing causing an harder inference task for the model. A solution is either using a dataset of images equal to the captured aspect ratio or using a camera sensor able to capture with the dataset aspect ratio. Finally a more comprehensive knowledge of the HLS APIs used to build the pre-processing kernel could be exploited to make changes like cropping the images to avoid a stretching and so a deformation.

Test equipment In addition with these possible critical points, the step of testing can not be exhaustively performed. Specifically it requires an equipment made by power suppliers for all the design components (FPGA board, display and host machine) and a camera easier to point on the objects, all mounted on a vehicle able to go across the city's roads. In absence of this equipment the testing has been made by showing some printed images to the camera sensor, however the image quality of the sensor appears to be low, making hard the inference task.

Moreover a deeper knowledge about the GStreamer and VVAS plug-ins could help to change the ways to capture and display the data. For instance to use as input source a video file an encoding system is required to decode the h264 frames of the file into raw frames. The decoder at hardware level (IP) is a VCU and the Gstreamer framework's plug-in is a omh264enc/dec. When this attempt has been done in the thesis projects, some unknown error about the plug-in have occurred.

5.3 Further Improvements

Beyond the improvements mentioned before, that are more like fixes, there are some directions to take for better technologies. An example is to boost the sensor equipment of the vehicle with LiDAR and Radar systems. The former guarantees a 3D mapping with high level of accuracy helping in the obstacles detection also at a very short distance. The RADAR systems work at longer wavelengths and this allows to detect objects at long distance and through fog or clouds. Hence an idea is to use a dataset as RADIATE[45] having a lot of traffic data in RADAR, LiDAR and camera formats in good and bad weather conditions.

Appendix A

Model development

Listing A.1: Conversion script from KITTI to YOLO format

```python
'''
USAGE:
    python3 convertToYOLO.py <in_dataset> <train_perc>
DESCRIPTION:
    Conversion from <in_dataset> format to
    Ultralytics YOLO format. <train_perc> of the input
    dataset is organised for training, the remaining
    part for validation.
LOCATION:
    Must be located at the same level of <in_dataset>/ folder.
AUTHOR:
    Davide Altamore
'''

import argparse
import os
from PIL import Image # pillow package with Image.size method

# Input dataset formats accepted
IN_DATASETS_LIST = ['Kitti']

# List for class id mapping.
# CLASSES has the class names and the class ids are the position
    in list
CLASSES = []

### Labels format conversion:
# Kitti input label = [0]object_class ... [4]left [5]top [6]right
    [7]bottom ...
# YOLO output label = object_class_id xc yc h w
def kitti2yolo_label(kitti_label, image_w, image_h):

  # return value = YOLO-formatted list of objects
  yolo_label = []

  yolo_line = ""

  for kitti_line in kitti_label:

    object_class = kitti_line.split()[0]
    # Kitti marks as 'DontCare' the unrecognized objects, YOLO does not mark them at all
    if object_class != 'DontCare':
      # class id mapping
      if object_class not in CLASSES:
        CLASSES.append(object_class)
      # index() method starts by 0 as YOLO requires
      object_class_id = CLASSES.index(object_class)

      # Kitti bbox parameters
      x1, y1, x2, y2 = kitti_line.split()[4:8] # get <left><top><right><bottom>
      x1, y1, x2, y2 = float(x1), float(y1), float(x2), float(y2)
      # YOLO bbox parameters (normalized)
      xc = ((x1 + x2) / 2 ) / image_w
      yc = ((y1 + y2) / 2 ) / image_h
      h = (y2 - y1) / image_h
      w = (x2 - x1) / image_w

      yolo_line = f"{object_class_id} {xc} {yc} {h} {w}"

    if yolo_line:
      yolo_label.append(yolo_line)
      yolo_line = ""

  return yolo_label

### Conversion from KITTI to Ultralytics YOLO format.
def kitti2yolo(train_perc):
  ## Source directories
  TRAIN_IM_PATH = 'Kitti/raw/training/image_2'
  TRAIN_LAB_PATH = 'Kitti/raw/training/label_2'
  TEST_IM_PATH = 'Kitti/raw/testing/image_2'

  ## TESTING directory is the same, it contains only images
  try:
    os.system(f"cp -r {TEST_IM_PATH}/* ../datasets/Kitti/test/images")
  except OSError as error:
    print(error)
```

```python

## TRAINING directory must be divided into train and val (based on train_perc)
train_images_lst = sorted(os.listdir(TRAIN_IM_PATH)) # comprised ".png"
# n. of images for TRAINING
train_images_N = int(len(train_images_lst) * train_perc)
# For each input training image...
i = 0
for image in train_images_lst:

  # Evaluate image dimesions for next evaluations
  image_w, image_h = Image.open(os.path.join(TRAIN_IM_PATH,image)).size
  # /!\ Kitti images are not all equally sized.

  outdir = 'train' if i < train_images_N else 'val' # set the image purpose

  # 1. Copy the image
  try:
    os.system("cp {} {}".format(os.path.join(TRAIN_IM_PATH,image),
                                os.path.join('../datasets/Kitti',outdir,'images')))
  except OSError as error:
    print(error)

  # 2. Read the Kitti label and write the new YOLO one
  label = image[:-4] + '.txt' # remove .png and add .txt
  with open(os.path.join(TRAIN_LAB_PATH,label),"r") as rfp:
    # A. read (in a list)
    kitti_label = rfp.readlines()
    # B. convert (return a list)
    yolo_label = kitti2yolo_label(kitti_label,image_w, image_h)

    # If at least one detected object: write a file, otherwise no file (by YOLO docs)
    if yolo_label:
      with open(os.path.join('../datasets/Kitti',outdir,'labels',label),"w") as wfp:
        # C. write (lines with breakline)
        wfp.write("\n".join(yolo_label))

  i = i + 1 # next image

if i == len(train_images_lst):
  retval = True
else:
    retval = False

  return retval

def main():

  ## Arguments parsing (see USAGE)
  parser = argparse.ArgumentParser(description='Convert a dataset to Ultralytics YOLO format.')
  parser.add_argument('in_dataset', help='Format of the input dataset')
  parser.add_argument('train_perc', help='Percentage of in data to use for training, the remaining for validation')
  args = parser.parse_args()
  # Check <in_dataset>
  if args.in_dataset not in IN_DATASETS_LIST:
    print(f"Invalid 1st argument {args.in_dataset} ! Must be one of these:\n{IN_DATASETS_LIST}")
    exit(-1)
  # Check <train_perc>
  try:
    train_perc = float(args.train_perc)
  except:
    print(f"Invalid 2nd argument {args.train_perc} ! Must be a number !")
  if train_perc < 0 or train_perc > 1:
    print(f"Invalid 2nd argument {args.train_perc} ! Must be a number in 0-1 range !")
    exit(-1)

  ## Check if a possible output is already existing
  abs_outpath = os.path.join(os.path.abspath("../datasets"),args.in_dataset)
  if os.path.isdir(abs_outpath):
    print(f"A possible output directory already exists, check:\n{abs_outpath}\n ...and eventually remove it !")
    exit(-1)

  ## Output directories for YOLO format
  OUT_DIRECTORIES = [f"{abs_outpath}/train/images",
                     f"{abs_outpath}/val/images",
                     f"{abs_outpath}/test/images",
                     f"{abs_outpath}/train/labels",
                     f"{abs_outpath}/val/labels"]
  try:
    for directory in OUT_DIRECTORIES:
      os.makedirs(directory)
    print("\nOutput train, val and test directories created !")
  except OSError as error:
    print(error)

  ## Run the conversion function specific for <in_dataset>
  if args.in_dataset == IN_DATASETS_LIST[0]:        # 'Kitti'
    print("\nConversion from Kitti dataset format to Ultralytics YOLO format...")
    is_complete = kitti2yolo(train_perc)
  #elif args.in_dataset = IN_DATASETS_LIST[1]:      # <other_dataset>
  else:
    print(f"Invalid 1st argument {args.in_dataset} ! Must be one of these:\n{IN_DATASETS_LIST}")
    exit(-1)

  if is_complete:
    print(f"\nAll the images of {args.in_dataset} dataset have been processed.")
  else:
    print(f"\nThere are some unprocessed images of {args.in_dataset} dataset. Check it!")
    exit(-1)

  ## Write the data.yaml file, needed for YOLO
  print(f"\n{abs_outpath}/data.yaml writing...")

  with open(f"{abs_outpath}/data.yaml","w") as fp:
    # absolute path to YOLO-format dataset
    fp.write(f"path: {abs_outpath}\n")
    # relative path to train, val and test images directories
    fp.write(f"train: train/images\nval: val/images\ntest: test/images\n")
    # class id mapping
    fp.write(f"\nnc: {len(CLASSES)}\nnames:")
    for obj in CLASSES:
      fp.write(f"\n {CLASSES.index(obj)}: {obj}")

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

Listing A.2: Quantization script

```
'''
USAGE:
  python3 quantize.py --build_dir <build>
                      --quant_mode <calib,test>
                      --fp_model <fp_model.pt>
                      --dataset <KITTI_yolo>
                      --batchsize <1>
DESCRIPTION:
  Quantization script for a pre-trained YOLOv5 model on Kitti
   dataset.
  It runs in the Vitis AI docker container, run as:
    /thesis/Vitis-AI$  ./docker_run.sh xilinx/vitis-ai-pytorch-cpu
   :latest
            [docker]$  conda activate vitis-ai-pytorch
LOCATION:
  Must be located in a Vitis-AI/YOLOv5_quant/ folder with all
   other YOLOv5 files.
AUTHOR:
  Edited by: Davide Altamore
  Based on: Vitis AI tutorials and YOLOv5 validation code
'''

import os
import time
import sys
import argparse
import numpy as np

# PyTorch imports
import torch
from torch.utils.data import Dataset
import torchvision

# YOLO imports
from models.common import DetectMultiBackend
from utils.general import (
    non_max_suppression,
    xywh2xyxy,
    check_dataset,
    colorstr
)
from utils.metrics import (
    ap_per_class
)
from utils.dataloaders import(
    create_dataloader
)
import val

# Vitis AI imports
from pytorch_nndct.apis import torch_quantizer, dump_xmodel
# torch_quantizer (vai_q_pytorch) is the VAI Quantizer for Pytorch
    framework
# (NNDCT Pytorch APIS located in Vitis-AI/src/vai_quantizer/
   vai_q_pytorch/pytorch_binding/pytorch_nndct/apis.py)

'''--------------------------------------
```

```
(*) Calibration/Testing images dimensions
-----------------------------------------
- Kitti images are not all equally sized:
   there are 1224 x 370, 1242 x 375, 1241 x 376, 1238 x 374 (avg AR = 3.3)
- To decide which dimensions using for the quantizer tensors, let's observe that:
   /!\ torch_quantizer() raises an error if these dimensions are not multiples of 64 !
- Reasonably choice: 1216 x 320 (AR=3.8) that are the multipes of 64 lower than all the original images.
'''
W_IMG = 1216 # ( NOTE: is also the size used for the training)
H_IMG = 320
IMGSZ = (W_IMG,H_IMG)

DIVIDER = '-'*50


def quantize(build_dir, quant_mode, fp_model, dataset, batchsize):

  # Path of the output integer model (quantized)
  quant_model = os.path.join(build_dir, 'quant_model')

  # Use GPU if available, otherwise CPU
  # easily: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
  # more exhaustively:
  if (torch.cuda.device_count() > 0):
    print('You have',torch.cuda.device_count(),'CUDA devices available')
    for i in range(torch.cuda.device_count()):
      print(' Device',str(i),': ',torch.cuda.get_device_name(i))
    print('Selecting device 0..')
    device = torch.device('cuda:0')
  else:
    print('No CUDA devices available..selecting CPU')
    device = torch.device('cpu')

  ## [1]. Load trained model
  print(f"Loading trained model {fp_model} ...")
  # YOLOv5 MultiBackend class for python inference on various backends, see models/common.py
  model = DetectMultiBackend(weights=fp_model).to(device)

  # Override batchsize if in test mode
  if (quant_mode=='test'):
    batchsize = 1

```

```python
## [2]. Create quantizer object instance (a torch_quantizer() is used)
# A dummy input tensor with the same shape of the real input images is needed.
# /!\ If the 2 last dimensions are not multiple of 64 an error occurs ! See (*) above.
# /!\ THE TENSOR DEFINITION IS HxW - PAY ATTENTION TO IT !
rand_in = torch.randn([batchsize, 3, H_IMG, W_IMG])
# Vitis AI quantizer used = the torch_quantizer() by pythorch_nndct library
quantizer = torch_quantizer(quant_mode, model, (rand_in), device=device, output_dir=quant_model)
# Run the quantization and obtain the quantized model
quantized_model = quantizer.quant_model.to(device)


## [3]. Forward step
# The process of propagating input data through the network's layers.
# It is needed for export_quant_config() and export_model() at the end !

# The validation process of yolo val.py is used to get an accuracy metrics comparable with the
# one obtained before on training (mAP@0.5-0.95). This function does all the fundamental steps:
# > Forwarding
# > Accuracy evaluation

# Get a dictionary of infos about the dataset from yaml file
#  this function performs a parsing of the yaml file (in yolo format)
yaml_path = os.path.join(dataset,'Kitti','data.yaml')
data_dict = check_dataset(yaml_path)

# Dataloader creation
dataloader, dataset = create_dataloader(
            data_dict['val'],
            imgsz=W_IMG,
            batch_size=batchsize,
            stride=model.stride, #shuold be 64
            prefix=colorstr(f"val: "),
            rect=True
)
print(f"\nThe dataset has been collected, infos from {yaml_path}\n{data_dict}")

results, maps, _ = val.run(
                      data_dict,
                                  quantize=True,
                                  batch_size=batchsize,
                                  imgsz=IMGSZ,
                                  model=quantized_model,
                                  dataloader=dataloader,
                                  verbose=True,
                                  plots=False,
                                  max_det=300,
                                  iou_thres=0.6
                            )

    print("Results")
    print(results)
    print("mAPs")
    print(maps)


    ## [4]. Output the quantization result and deploy the model
    if quant_mode == 'calib':
      # Export the quantization configuration obtained during
     calibration.
      # Fundamental for a correct deployment.
      quantizer.export_quant_config()
    if quant_mode == 'test':
      # Export the quantized model in xmodel format.
      quantizer.export_xmodel(deploy_check=False, output_dir=
     quant_model)
    return


def main():

    # Construct the argument parser and parse the arguments:
    ap = argparse.ArgumentParser()
    # --build_dir: where saving the quantized model (xmodel)
    ap.add_argument('-d', '--build_dir', type=str, default='build',
     help='Path to build folder. Default is build')
    # --quant_mode: how to handle the quantization result ?
    # 'calib' the quantizer is configured to firstly perform a model
       calibration.
    #          A process of statistics collection of the model's
     inputs (e.g. min and max activation values).
    # 'test' performs quantization testing and exports the quantized
       model in different formats
    #         (used also for validation before deployment).
    ap.add_argument('-q', '--quant_mode', type=str, default='calib',
       choices=['calib','test'], help='Quantization mode (calib or
     test). Default is calib')
```

```
  # --fp_model: .pt model with the weights to quantize (already
   trained)
  ap.add_argument('-w', '--fp_model', type=str, help='Path to yolo
    weights file')
  # --dataset: root path of the small dataset used during the
   forward pass for calibration.
  ap.add_argument('-s', '--dataset', type=str, help='Path to your
   calibration directory with subdirectories called "images" and "
   labels"')
  # --batchsize: batch size needed for the evaluation process.
  ap.add_argument('-b', '--batchsize', type=int, default=16, help=
   'Testing batchsize - must be an integer. Default is 16')

  args = ap.parse_args()

  print('\n'+DIVIDER)
  print('PyTorch version : ',torch.__version__)
  print(sys.version)
  print(DIVIDER)
  print(' Command line options:')
  print ('--build_dir    : ',args.build_dir)
  print ('--quant_mode   : ',args.quant_mode)
  print ('--fp_model     : ',args.fp_model)
  print ('--dataset      : ',args.dataset)
  print ('--batchsize    : ',args.batchsize)
  print(DIVIDER)

  # Run the quantization
  quantize(args.build_dir, args.quant_mode, args.fp_model, args.
   dataset,args.batchsize)

  return


if __name__ == '__main__':
  main()
```

Appendix B

Platform Development

Listing B.1: DPU configuration file

```
/*
* Copyright 2019 Xilinx Inc.
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*    http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
   software
* distributed under the License is distributed on an "AS IS" BASIS
   ,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
* See the License for the specific language governing permissions
   and
* limitations under the License.
*/

//Setting the arch of DPU, For more details, Please read the PG338


/*====== Architecture Options ======*/
// |------------------------------------------------------|
// | Support 8 DPU size
// | It relates to model. if change, must update model
// +------------------------------------------------------+
// | `define B512
// +------------------------------------------------------+
// | `define B800
// +-----------------------------------------------------+
// | `define B1024
// +-----------------------------------------------------+
// | `define B1152
// +-----------------------------------------------------+
// | `define B1600
// +-----------------------------------------------------+
// | `define B2304
// +-----------------------------------------------------+
// | `define B3136
// +-----------------------------------------------------+
// | `define B4096
// |-----------------------------------------------------|

`define B3136

// |-----------------------------------------------------|
// | If the FPGA has Uram. You can define URAM_EN parameter
// | if change, Don't need update model
// +-----------------------------------------------------+
// | for zcu104 : `define URAM_ENABLE
// +-----------------------------------------------------+
// | for zcu102 : `define URAM_DISABLE
// |-----------------------------------------------------|

`define URAM_ENABLE

//config URAM
`ifdef URAM_ENABLE
    `define def_UBANK_IMG_N          6
    `define def_UBANK_WGT_N          17
    `define def_UBANK_BIAS           1
`elsif URAM_DISABLE
    `define def_UBANK_IMG_N          0
    `define def_UBANK_WGT_N          0
    `define def_UBANK_BIAS           0
`endif

// |-----------------------------------------------------|
// | You can use DRAM if FPGA has extra LUTs
// | if change, Don't need update model
// +-----------------------------------------------------+
// | Enable DRAM  : `define DRAM_ENABLE
// +-----------------------------------------------------+
// | Disable DRAM : `define DRAM_DISABLE
// |-----------------------------------------------------|

`define DRAM_DISABLE

```

```
//config DRAM
`ifdef DRAM_ENABLE
    `define def_DBANK_IMG_N            1
    `define def_DBANK_WGT_N            1
    `define def_DBANK_BIAS             1
`elsif DRAM_DISABLE
    `define def_DBANK_IMG_N            0
    `define def_DBANK_WGT_N            0
    `define def_DBANK_BIAS             0
`endif

// |------------------------------------------------------|
// | RAM Usage Configuration
// | It relates to model. if change, must update model
// +------------------------------------------------------+
// | RAM Usage High : `define RAM_USAGE_HIGH
// +------------------------------------------------------+
// | RAM Usage Low  : `define RAM_USAGE_LOW
// |------------------------------------------------------|

`define RAM_USAGE_LOW

// |------------------------------------------------------|
// | Channel Augmentation Configuration
// | It relates to model. if change, must update model
// +------------------------------------------------------+
// | Enable  : `define CHANNEL_AUGMENTATION_ENABLE
// +------------------------------------------------------+
// | Disable : `define CHANNEL_AUGMENTATION_DISABLE
// |------------------------------------------------------|

`define CHANNEL_AUGMENTATION_ENABLE

// |------------------------------------------------------|
// | ALU parallel Configuration
// | It relates to model. if change, must update model
// +------------------------------------------------------+
// | setting 0  : `define ALU_PARALLEL_DEFAULT
// +------------------------------------------------------+
// | setting 1  : `define ALU_PARALLEL_1
// |------------------------------------------------------|
// | setting 2  : `define ALU_PARALLEL_2
// |------------------------------------------------------|
// | setting 3  : `define ALU_PARALLEL_4
// |------------------------------------------------------|
// | setting 4  : `define ALU_PARALLEL_8
// |------------------------------------------------------|

`define ALU_PARALLEL_DEFAULT
```

```

// +-------------------------------------------------------+
// | CONV RELU Type Configuration
// | It relates to model. if change, must update model
// +-------------------------------------------------------+
// | `define CONV_RELU_RELU6
// +-------------------------------------------------------+
// | `define CONV_RELU_LEAKYRELU_RELU6
// |-------------------------------------------------------|

`define CONV_RELU_LEAKYRELU_RELU6

// +-------------------------------------------------------+
// | ALU RELU Type Configuration
// | It relates to model. if change, must update model
// +-------------------------------------------------------+
// | `define ALU_RELU_RELU6
// +-------------------------------------------------------+
// | `define ALU_RELU_LEAKYRELU_RELU6
// |-------------------------------------------------------|

`define ALU_RELU_RELU6

// |-------------------------------------------------------|
// | argmax or max Configuration
// | It relates to model. if change, must update model
// +-------------------------------------------------------+
// | enable  : `define SAVE_ARGMAX_ENABLE
// +-------------------------------------------------------+
// | disable : `define SAVE_ARGMAX_DISABLE
// |-------------------------------------------------------|

//`define SAVE_ARGMAX_ENABLE

// |-------------------------------------------------------|
// | DSP48 Usage Configuration
// | Use dsp replace of lut in conv operate
// | if change, Don't need update model
// +-------------------------------------------------------+
// | `define DSP48_USAGE_HIGH
// +-------------------------------------------------------+
// | `define DSP48_USAGE_LOW
// |-------------------------------------------------------|

`define DSP48_USAGE_HIGH

// |-------------------------------------------------------|
// | Power Configuration
// | if change, Don't need update model
// +--------------------------------------------------------+
// | `define LOWPOWER_ENABLE
// +--------------------------------------------------------+
// | `define LOWPOWER_DISABLE
// |--------------------------------------------------------|

`define LOWPOWER_DISABLE

// |--------------------------------------------------------|
// | DEVICE Configuration
// | if change, Don't need update model
// +--------------------------------------------------------+
// | `define MPSOC
// +--------------------------------------------------------+
// | `define ZYNQ7000
// |--------------------------------------------------------|

`define MPSOC
```

Listing B.2: Pre-processing header file with parameters

```
/*
 * Copyright 2020 Xilinx, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License
    .
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
    software
 * distributed under the License is distributed on an "AS IS"
    BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
     implied.
 * See the License for the specific language governing permissions
     and
 * limitations under the License.
 */

// Max image resoultion

// Enable or disable channel Swap
#define BGR2RGB 0
// Enable or disable crop
#define CROP 0

static constexpr int WIDTH = 3840;
static constexpr int HEIGHT = 2160;

static constexpr int INPUT_PTR_WIDTH = 64;
static constexpr int OUTPUT_PTR_WIDTH = 64;

static constexpr int IN_TYPE = XF_8UC3;
static constexpr int OUT_TYPE = XF_8UC3;
// Pixels processed per cycle
static constexpr int NPC = XF_NPPC1;

// preprocess kernel params out = (in - a) * b
// a, b and out are fixed point values and below params are used
    to configure
// the width and integer bits
static constexpr int WIDTH_A = 8;
static constexpr int IBITS_A = 8;
static constexpr int WIDTH_B = 8;
static constexpr int IBITS_B = 4; // so B is 8-bit wide and 4-bits
     are integer bits
static constexpr int WIDTH_OUT = 8;
static constexpr int IBITS_OUT = 8;

// Resize configuration parameters
static constexpr int NEWWIDTH = 1216; // Training and Quantization
     width
static constexpr int NEWHEIGHT = 320; // Almost-Training and
    Quantization height

static constexpr int MAXDOWNSCALE = 9;

static constexpr int INTERPOLATION = 1;

static constexpr int XF_CV_DEPTH = 2;
```

Listing B.3: Pre-processing header file with inclusions

```
/*
 * Copyright 2020 Xilinx, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License
    .
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
    software
 * distributed under the License is distributed on an "AS IS"
    BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
 * See the License for the specific language governing permissions
   and
 * limitations under the License.
 */

#ifndef _XF_BLOBFROMIMAGE_CONFIG_
#define _XF_BLOBFROMIMAGE_CONFIG_

#include "common/xf_common.hpp"
#include "common/xf_utility.hpp"
#include "dnn/xf_preprocess.hpp"
#include "imgproc/xf_crop.hpp"
#include "imgproc/xf_cvt_color.hpp"
#include "imgproc/xf_cvt_color_1.hpp"
#include "imgproc/xf_duplicateimage.hpp"
#include "imgproc/xf_resize.hpp"
#include "xf_config_params.h"
#include <ap_int.h>
#include <hls_stream.h>

#define _XF_SYNTHESIS_ 1

#endif
```

Listing B.4: Pre-processing definition file

```
/*
 * Copyright 2020 Xilinx, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License
   .
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
   software
 * distributed under the License is distributed on an "AS IS"
   BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
 * See the License for the specific language governing permissions
   and
 * limitations under the License.
 */
#include "xf_pp_pipeline_config.h"
```

```
void pp_pipeline_accel(ap_uint<INPUT_PTR_WIDTH>* img_inp_y,  // Y
    Input image pointer
              ap_uint<INPUT_PTR_WIDTH>* img_inp_uv, // UV Input
    image pointer
                           ap_uint<OUTPUT_PTR_WIDTH>* img_out, //
    output image pointer
                           float params[2 * XF_CHANNELS(IN_TYPE, NPC
    )],
                           int in_img_width,
                           int in_img_height,
                           int in_img_linestride,
                           int out_img_width,       // Final Output
    image width
                           int out_img_height,      // Final Output
    image height
                           int out_img_linestride) { // Final Output
     image line stride
// clang-format off
#pragma HLS INTERFACE m_axi     port=img_inp_y   offset=slave
    bundle=gmem1
#pragma HLS INTERFACE m_axi     port=img_inp_uv   offset=slave
    bundle=gmem2
#pragma HLS INTERFACE m_axi     port=img_out   offset=slave bundle=
    gmem3
#pragma HLS INTERFACE m_axi     port=params   offset=slave bundle=
    gmem4
#pragma HLS INTERFACE s_axilite port=in_img_width
#pragma HLS INTERFACE s_axilite port=in_img_height
#pragma HLS INTERFACE s_axilite port=in_img_linestride
#pragma HLS INTERFACE s_axilite port=out_img_width
#pragma HLS INTERFACE s_axilite port=out_img_height
#pragma HLS INTERFACE s_axilite port=out_img_linestride
#pragma HLS INTERFACE s_axilite port=return
     // clang-format on
     xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, NPC,XF_CV_DEPTH>
    imgInput_y(in_img_height, in_img_width);
#pragma HLS stream variable = imgInput_y.data depth = 2
     xf::cv::Mat<XF_8UC2, HEIGHT/2, WIDTH/2, NPC,XF_CV_DEPTH>
    imgInput_uv(in_img_height/2, in_img_width/2);
#pragma HLS stream variable = imgInput_uv.data depth = 2
     xf::cv::Mat<XF_8UC3, HEIGHT, WIDTH, NPC,XF_CV_DEPTH> rgb_mat(
    in_img_height, in_img_width);
#pragma HLS stream variable = rgb_mat.data depth = 2

#if BGR2RGB
     xf::cv::Mat<XF_8UC3, HEIGHT, WIDTH, NPC,XF_CV_DEPTH>
    ch_swap_mat(in_img_height, in_img_width);
#endif
```

```
    xf::cv::Mat<XF_8UC3, NEWHEIGHT, NEWWIDTH, NPC,XF_CV_DEPTH>
   resize_out_mat(out_img_height, out_img_width);
#pragma HLS stream variable = resize_out_mat.data depth = 2
    xf::cv::Mat<OUT_TYPE, NEWHEIGHT, NEWWIDTH, NPC,XF_CV_DEPTH>
   out_mat(out_img_height, out_img_width);
// clang-format off
#pragma HLS stream variable = out_mat.data depth = 2
    xf::cv::accel_utils obj_iny, obj_inuv;
#pragma HLS DATAFLOW
    // clang-format on
    obj_iny.Array2xfMat<INPUT_PTR_WIDTH, XF_8UC1, HEIGHT, WIDTH,
   NPC,XF_CV_DEPTH>(img_inp_y, imgInput_y, in_img_linestride);
    obj_inuv.Array2xfMat<INPUT_PTR_WIDTH, XF_8UC2, HEIGHT/2, WIDTH
   /2, NPC,XF_CV_DEPTH> (img_inp_uv, imgInput_uv,
   in_img_linestride/2);
    xf::cv::nv122bgr<XF_8UC1, XF_8UC2, XF_8UC3, HEIGHT, WIDTH, NPC
   , NPC,XF_CV_DEPTH,XF_CV_DEPTH,XF_CV_DEPTH>(imgInput_y,
   imgInput_uv, rgb_mat);

#if BGR2RGB
    xf::cv::bgr2rgb<IN_TYPE, OUT_TYPE, HEIGHT, WIDTH, NPC,
   XF_CV_DEPTH,XF_CV_DEPTH>(rgb_mat, ch_swap_mat);
    xf::cv::resize<INTERPOLATION, IN_TYPE, HEIGHT, WIDTH,
   NEWHEIGHT, NEWWIDTH, NPC, MAXDOWNSCALE,XF_CV_DEPTH,XF_CV_DEPTH
   >(ch_swap_mat,

                                                   resize_out_mat);
#else
    xf::cv::resize<INTERPOLATION, IN_TYPE, HEIGHT, WIDTH,
   NEWHEIGHT, NEWWIDTH, NPC,MAXDOWNSCALE,XF_CV_DEPTH,XF_CV_DEPTH>(
   rgb_mat,

                                                   resize_out_mat);
#endif
    xf::cv::preProcess<IN_TYPE, OUT_TYPE, NEWHEIGHT, NEWWIDTH, NPC
   ,WIDTH_A, IBITS_A, WIDTH_B, IBITS_B, WIDTH_OUT,IBITS_OUT,
   XF_CV_DEPTH,XF_CV_DEPTH>(resize_out_mat, out_mat, params);
    xf::cv::xfMat2Array<OUTPUT_PTR_WIDTH, OUT_TYPE, NEWHEIGHT,
   NEWWIDTH, NPC,XF_CV_DEPTH>(out_mat, img_out, out_img_linestride
   );
}
```

Listing B.5: Overlay configuration file

```
# /*
# * Copyright 2019 Xilinx Inc.
# *
# * Licensed under the Apache License, Version 2.0 (the "License")
   ;
```

```
# * you may not use this file except in compliance with the
    License.
# * You may obtain a copy of the License at
# *
# *    http://www.apache.org/licenses/LICENSE-2.0
# *
# * Unless required by applicable law or agreed to in writing,
    software
# * distributed under the License is distributed on an "AS IS"
    BASIS,
# * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
    or implied.
# * See the License for the specific language governing
    permissions and
# * limitations under the License.
# */


[clock]

freqHz=300000000:DPUCZDX8G_1.aclk
freqHz=600000000:DPUCZDX8G_1.ap_clk_2
freqHz=300000000:pp_pipeline_accel_1.ap_clk

#id=0:DPUCZDX8G_1.aclk
#id=1:DPUCZDX8G_1.ap_clk_2

[connectivity]

sp=DPUCZDX8G_1.M_AXI_GP0:HP1
sp=DPUCZDX8G_1.M_AXI_HP0:HP1
sp=DPUCZDX8G_1.M_AXI_HP2:HPC1
sp=pp_pipeline_accel_1.m_axi_gmem1:HP3
sp=pp_pipeline_accel_1.m_axi_gmem2:HP3
sp=pp_pipeline_accel_1.m_axi_gmem3:HP3
sp=pp_pipeline_accel_1.m_axi_gmem4:HP3

[advanced]
misc=:solution_name=link
param=compiler.userPostSysLinkOverlayTcl=/home/davide/thesis/
    kv260_workspace/kria-vitis-platforms/kv260/overlays/examples/
    smartcam/prj_conf/strip_interconnects.tcl
param=compiler.addOutputTypes=hw_export
#param=compiler.addOutputTypes=sd_card
#param=compiler.skipTimingCheckAndFrequencyScaling=1

[vivado]
prop=run.impl_1.strategy=Performance_Explore
impl.jobs=3
```

```
synth.jobs=3
#prop=run.impl_1.strategy=Congestion_SpreadLogic_high
#prop=run.impl_1.strategy=Performance_NetDelay_high
#prop=run.impl_1.strategy=Performance_WLBlockPlacementFanoutOpt
#prop=run.impl_1.strategy=Performance_WLBlockPlacement
#prop=run.impl_1.strategy=Performance_ExploreWithRemap
#prop=run.impl_1.strategy=Performance_BalanceSLRs
#prop=run.impl_1.strategy=Performance_EarlyBlockPlacement
#prop=run.impl_1.strategy=Performance_ExtraTimingOpt
#prop=run.impl_1.strategy=Performance_NetDelay_low
#param=place.runPartPlacer=0
```

Listing B.6: Device binary tree of the PL design

```
/*
 * CAUTION: This file is automatically generated by Xilinx.
 * Version: XSCT 2022.2
 * Today is: Tue Feb 20 18:46:47 2024
 */

/*
 * Added features by Davide Altamore are marked with " ### ".
 * Today is: Tue Feb 20 2024 > Corrections: Fri Mar 01 2024
 * Purpose: to match the smartcamera application requirements.
 * Source: https://xilinx.github.io/kria-apps-docs/
   creating_applications/2022.1/build/html/docs/
   dtsi_dtbo_generation_smartcam_example.html
 */


/dts-v1/;
/plugin/;
/ {
    fragment@0 {
        target = <&fpga_full>;
        overlay0: __overlay__ {
            #address-cells = <2>;
            #size-cells = <2>;
            firmware-name = "kv260_smartcam.bit.bin";
            pid = <0x0>;
            resets = <&zynqmp_reset 116>;
            uid = <0x0>;
        };
    };
    fragment@1 {
        target = <&amba>;
        overlay1: __overlay__ {
            afi0: afi0 {
                compatible = "xlnx,afi-fpga";
                config-afi = < 0 0>, <1 0>, <2 0>, <3 0>, <4 0>,
   <5 0>, <6 0>, <7 0>, <8 0>, <9 0>, <10 0>, <11 0>, <12 2>, <13
   2>, <14 0x0>, <15 0x000>;
            };
            clocking0: clocking0 {
                #clock-cells = <0>;
                assigned-clock-rates = <99999001>;
                assigned-clocks = <&zynqmp_clk 71>;
                clock-output-names = "fabric_clk";
                clocks = <&zynqmp_clk 71>;
                compatible = "xlnx,fclk";
            };
        };
    };
    fragment@2 {
        target = <&amba>;
        overlay2: __overlay__ {
            #address-cells = <2>;
            #size-cells = <2>;
            axi_iic_0: i2c@80030000 {
                #address-cells = <1>;
                #size-cells = <0>;
                clock-names = "s_axi_aclk";
                clocks = <&misc_clk_0>;
                compatible = "xlnx,axi-iic-2.1", "xlnx,xps-iic
   -2.00.a";
                interrupt-names = "iic2intc_irpt";
                interrupt-parent = <&gic>;
                interrupts = <0 107 4>;
                reg = <0x0 0x80030000 0x0 0x10000>;

                /* ### i2c mux */
                i2c_mux: i2c-mux@74 {
                    compatible = "nxp,pca9546";
                    #address-cells = <1>;
                    #size-cells = <0>;
                    reg = <0x74>;
                    i2c@0 {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        reg = <0>;
                        ap1302: isp@3c {
                            compatible = "onnn,ap1302";
                            reg = <0x3c>;
                            #address-cells = <1>;
                            #size-cells = <0>;
                            reset-gpios = <&gpio 79 1>;
                            clocks = <&ap1302_clk>;
                            sensors {
```

```
                            #address-cells = <1>;
                            #size-cells = <0>;
                            onnn,model = "onnn,ar1335";
                            sensor@0 {
                                reg = <0>;
                                vdd-supply = <&ap1302_vdd>;
                                vaa-supply = <&ap1302_vaa>;
                                vddio-supply = <&ap1302_vddio
>;
                            };
                        };
                        ports {
                            #address-cells = <1>;
                            #size-cells = <0>;
                            port@0 {
                                reg = <2>;
                                isp_out: endpoint {
                                    remote-endpoint = <&
mipi_csi_incapture_pipeline_mipi_csi2_rx_subsyst_0>;
                                    data-lanes = <1 2 3 4>;
                                };
                            };
                        };
                    };
                };
            };
        };
        misc_clk_0: misc_clk_0 {
            #clock-cells = <0>;
            clock-frequency = <99999000>;
            compatible = "fixed-clock";
        };
        axi_vip_0: axi_vip@a0000000 {
            /* This is a place holder node for a custom IP,
user may need to update the entries */
            clock-names = "aclk";
            clocks = <&misc_clk_1>;
            compatible = "xlnx,axi-vip-1.1";
            reg = <0x0 0xa0000000 0x0 0x10000>;
            xlnx,axi-addr-width = <0x20>;
            xlnx,axi-aruser-width = <0x10>;
            xlnx,axi-awuser-width = <0x10>;
            xlnx,axi-buser-width = <0x0>;
            xlnx,axi-has-aresetn = <0x1>;
            xlnx,axi-has-bresp = <0x1>;
            xlnx,axi-has-burst = <0x1>;
            xlnx,axi-has-cache = <0x1>;
            xlnx,axi-has-lock = <0x1>;
            xlnx,axi-has-prot = <0x1>;
```

```
                xlnx,axi-has-qos = <0x1>;
                xlnx,axi-has-region = <0x0>;
                xlnx,axi-has-rresp = <0x1>;
                xlnx,axi-has-wstrb = <0x1>;
                xlnx,axi-interface-mode = <0x2>;
                xlnx,axi-protocol = <0x0>;
                xlnx,axi-rdata-width = <0x20>;
                xlnx,axi-rid-width = <0x10>;
                xlnx,axi-ruser-width = <0x0>;
                xlnx,axi-supports-narrow = <0x1>;
                xlnx,axi-wdata-width = <0x20>;
                xlnx,axi-wid-width = <0x10>;
                xlnx,axi-wuser-width = <0x0>;
            };
            misc_clk_1: misc_clk_1 {
                #clock-cells = <0>;
                clock-frequency = <299997000>;
                compatible = "fixed-clock";
            };
            capture_pipeline_mipi_csi2_rx_subsyst_0: mipi_csi2_rx_subsystem@80000000 {
                clock-names = "lite_aclk", "dphy_clk_200M", "video_aclk";
                clocks = <&misc_clk_0>, <&misc_clk_2>, <&misc_clk_1>;
                compatible = "xlnx,mipi-csi2-rx-subsystem-5.2", "xlnx,mipi-csi2-rx-subsystem-5.0";
                interrupt-names = "csirxss_csi_irq";
                interrupt-parent = <&gic>;
                interrupts = <0 104 4>;
                xlnx,csi-pxl-format = <0x18>; // ### Added
                reg = <0x0 0x80000000 0x0 0x2000>;
                xlnx,axis-tdata-width = <32>;
                xlnx,max-lanes = <4>;
                xlnx,en-active-lanes; // ### Added
                xlnx,ppc = <2>;
                xlnx,vfb;
                mipi_csi_portscapture_pipeline_mipi_csi2_rx_subsyst_0: ports {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    mipi_csi_port1capture_pipeline_mipi_csi2_rx_subsyst_0: port@1 {
                        /* Fill cfa-pattern=rggb for raw data types, other fields video-format and video-width user needs to fill */
                        reg = <1>;
                        /* ### Removed since no longer supported
                        xlnx,cfa-pattern = "rggb";
                        xlnx,video-format = <12>;
                        xlnx,video-width = <8>;*/
                    mipi_csirx_outcapture_pipeline_mipi_csi2_rx_subsyst_0: endpoint {
                            remote-endpoint = <&capture_pipeline_v_frmbuf_wr_0capture_pipeline_mipi_csi2_rx_subsyst_0>;
                        };
                    };
                    mipi_csi_port0capture_pipeline_mipi_csi2_rx_subsyst_0: port@0 {
                        /* Fill cfa-pattern=rggb for raw data types, other fields video-format,video-width user needs to fill */
                        /* User need to add something like remote-endpoint=<&out> under the node csiss_in:endpoint */
                        reg = <0>;
                        /* ### Removed since no longer supported
                        xlnx,cfa-pattern = "rggb";
                        xlnx,video-format = <12>;
                        xlnx,video-width = <8>; */
                        mipi_csi_incapture_pipeline_mipi_csi2_rx_subsyst_0: endpoint {
                            data-lanes = <1 2 3 4>;
                            // ### Connect the remote endpoint to camera serial out
                            remote-endpoint = <&isp_out>;
                        };
                    };
                };
            };
            misc_clk_2: misc_clk_2 {
                #clock-cells = <0>;
                clock-frequency = <199998000>;
                compatible = "fixed-clock";
            };
            capture_pipeline_v_frmbuf_wr_0: v_frmbuf_wr@b0010000 {
                #dma-cells = <1>;
                clock-names = "ap_clk";
                clocks = <&misc_clk_1>;
                compatible = "xlnx,v-frmbuf-wr-2.4", "xlnx,axi-frmbuf-wr-v2.2";
                interrupt-names = "interrupt";
                interrupt-parent = <&gic>;
                interrupts = <0 105 4>;
                reg = <0x0 0xb0010000 0x0 0x10000>;
                reset-gpios = <&gpio 78 1>;
                xlnx,dma-addr-width = <32>;
                xlnx,dma-align = <16>;
                xlnx,max-height = <2160>;
                xlnx,max-width = <3840>;
                xlnx,pixels-per-clock = <2>;
                xlnx,s-axi-ctrl-addr-width = <0x7>;
                xlnx,s-axi-ctrl-data-width = <0x20>;
                xlnx,vid-formats = "nv12";
                xlnx,video-width = <8>;
            };
            vcu_vcu_0: vcu@80100000 {
                #address-cells = <2>;
                #clock-cells = <1>;
                #size-cells = <2>;
                clock-names = "pll_ref", "aclk", "vcu_core_enc", "
    vcu_mcu_enc", "vcu_core_dec", "vcu_mcu_dec";
                clocks = <&misc_clk_3>, <&misc_clk_0>, <&vcu_vcu_0
     0>, <&vcu_vcu_0 1>, <&vcu_vcu_0 2>, <&vcu_vcu_0 3>;
                compatible = "xlnx,vcu-1.2", "xlnx,vcu";
                interrupt-names = "vcu_host_interrupt";
                interrupt-parent = <&gic>;
                interrupts = <0 106 4>;
                ranges ;
                reg = <0x0 0x80140000 0x0 0x1000>, <0x0 0x80141000
     0x0 0x1000>;
                reg-names = "vcu_slcr", "logicore";
                reset-gpios = <&gpio 80 0>;
                encoder: al5e@80100000 {
                    compatible = "al,al5e-1.2", "al,al5e";
                    interrupt-parent = <&gic>;
                    interrupts = <0 106 4>;
                    reg = <0x0 0x80100000 0x0 0x10000>;
                };
                decoder: al5d@80120000 {
                    compatible = "al,al5d-1.2", "al,al5d";
                    interrupt-parent = <&gic>;
                    interrupts = <0 106 4>;
                    reg = <0x0 0x80120000 0x0 0x10000>;
                };
            };
            misc_clk_3: misc_clk_3 {
                #clock-cells = <0>;
                clock-frequency = <49999500>;
                compatible = "fixed-clock";
            };
            zyxclmm_drm {
                compatible = "xlnx,zocl";
                status = "okay"; // ### Added
                interrupt-parent = <&gic>; // ### Added
                interrupts = <0 89  4>, <0 90  4>, <0 91  4>, <0 92  4>,
                             <0 93  4>, <0 94  4>, <0 95  4>, <0 96  4>; // ### Added
            };
            vcap_capture_pipeline_mipi_csi2_rx_subsyst_0 {
                compatible = "xlnx,video";
                dma-names = "port0";
                dmas = <&capture_pipeline_v_frmbuf_wr_0 0>;
                vcap_portscapture_pipeline_mipi_csi2_rx_subsyst_0: ports {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    vcap_portcapture_pipeline_mipi_csi2_rx_subsyst_0: port@0 {
                        direction = "input";
                        reg = <0>;
                        capture_pipeline_v_frmbuf_wr_0capture_pipeline_mipi_csi2_rx_subsyst_0: endpoint {
                            remote-endpoint = <&mipi_csirx_outcapture_pipeline_mipi_csi2_rx_subsyst_0>;
                        };
                    };
                };
            };

            /* ### ap1302 */
            ap1302_clk: sensor_clk {
                #clock-cells = <0x0>;
                compatible = "fixed-clock";
                clock-frequency = <0x48000000>;
            };

            ap1302_vdd: fixedregulator@0 {
                compatible = "regulator-fixed";
                regulator-name = "ap1302_vdd";
                regulator-min-microvolt = <2800000>;
                regulator-max-microvolt = <2800000>;
                enable-active-high;
            };

            ap1302_vaa: fixedregulator@1 {
                compatible = "regulator-fixed";
                regulator-name = "ap1302_vaa";
                regulator-min-microvolt = <1800000>;
                regulator-max-microvolt = <1800000>;
            };

```

```
            ap1302_vddio: fixedregulator@2 {
                compatible = "regulator-fixed";
                regulator-name = "ap1302_vddio";
                regulator-min-microvolt = <1200000>;
                regulator-max-microvolt = <1200000>;
            };
        };
    };
};
```

Listing B.7: Yocto BitBake recipe metadata file (.bb) with the FPGA firmware

```
#
# This file is the kv260-smartcam recipe.
#

SUMMARY = "Simple kv260-smartcam to use fpgamanager class"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835
    ade698e0bcf8506ecda2f7b4f302"

inherit fpgamanager_custom

SRC_URI = "file://kv260-smartcam.bit \
           file://shell.json \
           file://kv260-smartcam.dtsi \
           file://kv260-smartcam.xclbin \
           "
```

Listing B.8: Yocto BitBake recipe metadata file (.bb) with the software packages required by the Smartcam application

```
SUMMARY = "Example Smartcam application"

LICENSE = "Apache-2.0"
LIC_FILES_CHKSUM = "file://LICENSE;md5=
    a9c5ded2ac97b4ce01aa0ace8f3a1755"

#BRANCH = "xlnx_rel_v2022.1"
#SRC_URI = "git://github.com/Xilinx/smartcam.git;protocol=https;
    branch=${BRANCH}"
#SRCREV = "ad9523ee5f002141334698eb6ddc9a14679ac8d2"

inherit cmake

DEPENDS = "vvas-accel-libs glog gstreamer1.0-rtsp-server opencv"
RDEPENDS:${PN} = " \
    gst-perf \
    gstreamer1.0-omx \
    gstreamer1.0-plugins-bad-faac \
    gstreamer1.0-plugins-bad-mpegtsmux \
    gstreamer1.0-plugins-good-rtp \
    gstreamer1.0-plugins-bad-kms \
    gstreamer1.0-plugins-bad-mediasrcbin \
    gstreamer1.0-plugins-bad-videoparsersbad \
    gstreamer1.0-plugins-good-multifile \
    gstreamer1.0-plugins-good-rtpmanager \
    gstreamer1.0-plugins-good-udp \
    gstreamer1.0-plugins-good-video4linux2 \
    gstreamer1.0-python \
    gstreamer1.0-rtsp-server \
    vvas-accel-libs \
    libdrm-tests \
    v4l-utils \
    alsa-utils \
    python3-core \
    "

SOMAPP_INSTALL_PATH = "/"
EXTRA_OECMAKE += "-DCMAKE_BUILD_TYPE=Release -DCMAKE_SYSROOT=${
    STAGING_DIR_HOST} -DCMAKE_INSTALL_PREFIX=${SOMAPP_INSTALL_PATH}
    "

S = "${WORKDIR}/git"

FILES:${PN} += " \
    /opt/xilinx \
    "
```

Listing B.9: Yocto BitBake recipe metadata file (.bb) for the AP1302 firmware (1)

```
SUMMARY = "ap1302 ar1335-single firmware binary"

include ap1302-firmware.inc

FW_NAME = "ap1302_ar1335_single_fw.bin"
```

Listing B.10: Yocto BitBake recipe metadata file (.bb) for the AP1302 firmware (2)

```
LICENSE = "Proprietary"
LIC_FILES_CHKSUM = "file://LICENSE.txt;md5=9
    c13aad1aab42f76326f1beceafc40c4"

BRANCH ?= "xlnx_rel_v2022.1"
SRC_URI = "git://github.com/Xilinx/ap1302-firmware.git;protocol=
    https;branch=${BRANCH}"
```

```
SRCREV ?= "63e20752dc8b1e91fc6d6d518ebeb76f65e9f738"

S = "${WORKDIR}/git"

FW_NAME ?= ""

do_configure[noexec] = "1"
do_compile[noexec] = "1"

do_install() {
    install -d ${D}/lib/firmware  # create /lib/firmware
    install -m 0644 ${FW_NAME} ${D}/lib/firmware/${FW_NAME} # copy
    firmware binary to /lib/firmware
}

FILES:${PN} = "/lib/firmware/${FW_NAME}"
```

Listing B.11: Yocto BitBake recipe metadata file (.bb) for the final application *kv260-ml-accel*

```
DESCRIPTION = "ML Acceleration Smartcam related Packages"

inherit packagegroup

EXAMPLE_PACKAGES = " \
    ap1302-ar1335-single-firmware \
    kv260-smartcam \
    smartcam \
    "

RDEPENDS:${PN} = "${EXAMPLE_PACKAGES}"

COMPATIBLE_MACHINE = "^$"
COMPATIBLE_MACHINE:k26-kv = "${MACHINE}"
PACKAGE_ARCH = "${MACHINE_ARCH}"
```

Appendix C

Application Development

Listing C.1: Pre-processing configuration file for the *vvas_xmultisrc* plug-in

```
{
    "xclbin-location":"/lib/firmware/xilinx/kv260-smartcam/kv260-
  smartcam.xclbin",
    "vvas-library-repo": "/opt/xilinx/kv260-smartcam/lib",
    "element-mode": "transform",
    "kernels":
    [
        {
            "kernel-name": "pp_pipeline_accel:{pp_pipeline_accel_1
  }",
            "library-name": "libvvas_xpp.so",
            "config": {
                "debug_level" : 1,
                "mean_r": 0,
                "mean_g": 0,
                "mean_b": 0,
                "scale_r": 0.25,
                "scale_g": 0.25,
                "scale_b": 0.25
            }
        }
    ]
}
```

Listing C.2: Inference configuration file for the *vvas_xfilter* plug-in

```
{
  "xclbin-location":"/lib/firmware/xilinx/kv260-smartcam/kv260-
   smartcam.xclbin",
  "vvas-library-repo": "/usr/lib/",
  "element-mode":"inplace",
  "kernels" :[
    {
      "library-name":"libvvas_xdpuinfer.so",
      "config": {
        "model-name" : "yolov5_kv260",
        "model-class" : "YOLOV3",
        "model-path" : "/home/root/target_kv260/models",
        "run_time_model" : false,
        "need_preprocess" : false,
        "performance_test" : false,
        "debug_level" : 0
      }
    }
  ]
}
```

Listing C.3: Post-processing configuration file for the *vvas_xfilter* plug-in

```
{
    "xclbin-location":"/lib/firmware/xilinx/kv260-ml-accel/kv260-
  ml-accel.xclbin",
    "vvas-library-repo": "/opt/xilinx/kv260-smartcam/lib",
    "element-mode":"inplace",
    "kernels" :[
        {
            "library-name":"libvvas_airender.so",
            "config": {
                "fps_interval" : 10,
                "font_size" : 2,
                "font" : 3,
                "thickness" : 2,
                "debug_level" : 0,
                "label_color" : { "blue" : 0, "green" : 0, "red" :
    255 },
                "label_filter" : [ "class", "probability" ],
                "classes" : [
                    {
                    "name" : "Pedestrian",
                    "blue" : 255,
                    "green" : 0,
                    "red" : 0
                    },
                    {
                    "name" : "Truck",
                    "blue" : 0,
                    "green" : 255,
                    "red" : 0
                    },
                    {
                    "name" : "Car",
                    "blue" : 0,
                    "green" : 0,
                    "red" : 255
                    },
                    {
                    "name" : "Cyclist",
                    "blue" : 0,
                    "green" : 255,
                    "red" : 255
                    },
                    {
                    "name" : "Misc",
                    "blue" : 255,
                    "green" : 0,
                    "red" : 255
                    },
                    {
                    "name" : "Van",
                    "blue" : 255,
                    "green" : 255,
                    "red" : 0
                    },
                    {
                    "name" : "Tram",
                    "blue" : 0,
                    "green" : 0,
                    "red" : 0
                    },
                    {
                    "name" : "Person_sitting",
                    "blue" : 128,
                    "green" : 128,
                    "red" : 128
                    }
                ]
            }
        }
    ]
}
```

Listing C.4: Summary of used GStreamer pipeline

```
mediasrcbin media-device=/dev/media0 v4l2src0::io-mode=dmabuf
    v4l2src0::stride-align=256 \
! video/x-raw,width=1920,height=1080,framerate=30/1,format=NV12 \
! tee name=t \
! queue \
! vvas_xmultisrc kconfig="./jsons/preprocess.json" \
! queue \
! vvas_xfilter kernels-config="./jsons/aiinference.json" \
! ima.sink_master vvas_xmetaaffixer name=ima ima.src_master \
! fakesink t. \
! queue max-size-buffers=1 leaky=2 \
! ima.sink_slave_0 ima.src_slave_0 \
! queue \
! vvas_xfilter kernels-config="./jsons/drawresult.json" \
! queue \
! kmssink driver-name=xlnx plane-id=39 sync=false fullscreen-
    overlay=true bus-id=fd4a0000.display connector-id=43 '
```

Listing C.5: Python script with the application code

```
'''
-------------------------------------------------------------------------

Edited by: Davide Altamore

APPLICATION CODE for pre-processing, DPU inference and post-
    processing.
USAGE: see options in the parse argument section
NOTES: all the application code is based on the use of GStreamer
    framework.
       It provides plug-in to handle the video capture and
    processing.
       Also the VVAS GStreamer plug-ins are used.
-------------------------------------------------------------------------

'''


'''
    Imports and Initializations
'''

import os
import glob
import subprocess
import re
import sys
import argparse

# [DEBUG] Create a directory for saving the pipeline graph as dot
    file.
dotdir = "./gst-dot/"
if not os.path.isdir(dotdir):
    os.makedirs(dotdir)
# ...and set it as environment variable
os.environ["GST_DEBUG_DUMP_DOT_DIR"] = dotdir
# Display dot file library
import pydot
```

```
from IPython.display import Image, display, clear_output

# Add some util path
pathv="{}:/usr/sbin:/sbin".format(os.environ.get("PATH"))
os.environ["PATH"] = pathv

# Import GStreamer-related libraries
# "gi" is the Python API for GObject Introspection.
# "gi.repository" is related to the repository of bindings
    available via GObject Introspection.
# Importing bindings via this method is what replaces the old
    straight Python bindings for gobject, glib, gtk and similar
    libraries.
import gi
gi.require_version('Gst', '1.0')
gi.require_version("GstApp", "1.0")
gi.require_version('GstVideo', '1.0')
gi.require_version('GstRtspServer', '1.0')
gi.require_version('GIRepository', '2.0')
from gi.repository import GObject, GLib, Gst, GstVideo,
    GstRtspServer, GIRepository
# And initialize GStreamer
Gst.init(None)
mainloop = GLib.MainLoop().new(None,False)
# Let's set the debug level of each  plug-in (wildcards are
    allowed)
#Gst.debug_set_threshold_from_string('*:1', True)

# Training/Calibration dataset = Kitti (where all images have
    arbitrary dimensions)
# To obtain good results from inference the captured dimensions
    must match those ones
# used for training and calibration.
#img_w = 1216
#img_h = 320

'''
    The application code's main core is the construction of
    the String Representation of a GStreamer Pipeline to run
'''

# Search the device presence and return the device name string
# e.g. typically the media dev is /dev/media0
def get_media_dev_by_name(src):
    sources = {
        'usb' : 'uvcvideo',
        'mipi' : 'vcap_capture_pipeline_mipi_csi2', # specific for
     our case (use regex to generalize, string with 'csi' and 'vcap
    ' words)
```

```python
    }
    devices = glob.glob('/dev/media*')
    for dev in devices:
        proc = subprocess.run(['media-ctl', '-d', dev, '-p'], capture_output=True, encoding='utf8')
        for line in proc.stdout.splitlines():
            if sources[src] in line:
                return dev

# e.g. typically the media dev is /dev/video0
def get_video_dev_of_mediadev(src):
    proc = subprocess.Popen(['media-ctl', '-d', src, '-p'], stdout=subprocess.PIPE)
    output = subprocess.check_output(('awk', '/^driver\s*uvcvideo/ {u=1} /device node name *\/dev\/video/ {x=$4;f=1;next} u&&f&&/ pad0: Sink/ {print x; x=""} f {f=0}'), stdin=proc.stdout).decode('utf8').splitlines()
    if len(output) > 1:
        return output[0]

def app(vidsrc, vidout, img_w, img_h, not_inf, fps):

    # Get the mediasrc index
    if vidsrc == 'mipi' or vidsrc == 'usb':
        media_device = get_media_dev_by_name(vidsrc)
        if media_device is None:
            raise Exception('Unable to find video source ' + vidsrc + '. Make sure the device is plugged in, powered, and the correct platform is used.')
        else:
            print("Found the video source at " + media_device)

    # INPUT
    # - Mipi: mediasrcbin
    #         Xilinx specific plug-in which is a bin element on top of `v4l2src` (video4linux2).
    #     It parses and configures the media graph of a media v4l2 device automatically (i.e. captures video from it).
    if vidsrc == "mipi":
        src = "mediasrcbin media-device=" + media_device
        if vidout == "display":
            src += " v4l2src0::io-mode=dmabuf v4l2src0::stride-align=256 "
    # - Usb: v4l2src is directly used without mediasrcbin
    elif vidsrc == "usb":
        usbmedia=media_device
        usbvideo=get_video_dev_of_mediadev(usbmedia)
        src = "v4l2src name=videosrc device={usbvideo} io-mode=mmap stride-align=256 ".format(usbvideo=usbvideo)
    # - File: all other strings are interpreted as video paths (filesrc, h264parse and omxh264dec plug-ins)
    else:
        src = "filesrc location={vidsrc} ! h264parse ! queue ! omxh264dec ".format(vidsrc=vidsrc)

    # For all the application code steps (pre-process, AI inference and bbox drawing)
    # some configuration files are required (json) and placed in ...
    confdir = "./jsons"

    # If the video source does not support the NV12 format, the pipeline can be adjusted
    if vidsrc=="usb":
        pipeline = src + ' ! video/x-raw, width={img_w}, height={img_h} ! videoconvert ! video/x-raw, format=NV12 '.format(img_w=img_w,img_h=img_h)
    else: # mipi or file
        pipeline = src + ' ! video/x-raw, width={img_w}, height={img_h}, format=NV12, framerate=30/1 '.format(img_w=img_w,img_h=img_h)

    if not(not_inf):
        '''
        PRE-PROCESS
        Performing the pre-processing of the input images by hardware is faster than doing it by software.
        Indeed a dedicated accelerator IP has been generated, so here the only step to do is to provide
        a config file through the VVAS plugin, vvas_xmultisrc.'''
        pipeline += ' ! tee name=t ! queue ! vvas_xmultisrc kconfig="{confdir}/preprocess.json" ! queue '.format(confdir=confdir)

        '''
        AI INFERENCE
        Inside the aiinference.json file the model path is set.'''
        pipeline += ' ! vvas_xfilter kernels-config="{confdir}/aiinference.json" '.format(confdir=confdir)

        '''
        DRAWING BBOX
        Accept and scale the original AI inference meta info. As the previous step, the meta info is pass down to here,
        the original buffer from t. is linked to the sink_slave_0, and get the scaled meta at the corresponding src_slave_0.
        Finally draw the bbox on the buffer.'''
```

```
     pipeline += ' ! ima.sink_master vvas_xmetaaffixer name=ima ima.src_master ! fakesink t. ! queue max-size-buffers=1 leaky=2 ! ima.sink_slave_0 ima.src_slave_0 ! queue  '
     pipeline += ' ! vvas_xfilter kernels-config="{confdir}/drawresult.json" ! queue  '.format(confdir=confdir)

 # DisplayPort/HDMI
 if vidout == "display":
     if fps:
         pipeline += ' ! fpsdisplaysink text-overlay=true sync=false\
          video-sink="kmssink driver-name=xlnx plane-id=39 sync=false fullscreen-overlay=true bus-id=fd4a0000.display connector-id=43"'
     else:
         pipeline += ' ! kmssink driver-name=xlnx plane-id=39 sync=false fullscreen-overlay=true bus-id=fd4a0000.display connector-id=43 '
     # The pipeline is entirely concluded, set to a PLAYING state by using OpenCV
     gst_pipeline = Gst.parse_launch(pipeline)
     print("The Gstreamer pipeline has been correctly parsed.")
     gst_pipeline.set_state(Gst.State.PLAYING)
     print("The Gstreamer pipeline's state has been set to PLAYING.")

     # [DEBUG] Generate pipeline dot file
     # ("tmp" is the name for the dotfile that will be saved in the location shown above, see the first lines)
     Gst.debug_bin_to_dot_file(gst_pipeline, Gst.DebugGraphDetails.ALL, "tmp")
     dotfile = dotdir + "tmp.dot"
     graph = pydot.graph_from_dot_file(dotfile, 'utf-8')
     print("A pipeline graph in dot format has been generated for debug purposes. See {dotfile}".format(dotfile=dotfile))


     def on_bus_message(bus, message):
         if message.type == Gst.MessageType.EOS:
             print("EOS message received.")
             gst_pipeline.seek_simple(Gst.Format.TIME, Gst.SeekFlags.FLUSH, 0)

     # Set up a pipeline bus watch to catch errors
     bus = gst_pipeline.get_bus()
     bus.add_signal_watch()
     bus.connect('message', on_bus_message)

     print('Running mainloop...\n')
     print("Press CTRL+C to stop the pipeline...")
     # Let's run the pipeline in an infinite loop until the SIGINT is sent
     try:
         mainloop.run()
     except KeyboardInterrupt:
         print(" >>CTRL+C has been pressed. End of application.<< ")
     finally:
         gst_pipeline.set_state(Gst.State.NULL)
         mainloop.quit()

 # RSTP (Ethernet connection)
 elif vidout == "rtsp":
     server = GstRtspServer.RTSPServer.new()
     server.props.service = "5000"
     mounts = server.get_mount_points()
     serverid=server.attach(None)
     factory = GstRtspServer.RTSPMediaFactory()
     # Then pass the frame with bbox to do the VCU encoding (with bbox info as encoding ROI):
     # - ROI info for VCU encoding generation, vvas_xroigen.
     pipeline += ' ! queue ! vvas_xroigen roi-type=1 roi-qp-delta=-10 roi-max-num=10   '
     # - VCU encoding, omxh264enc.
     pipeline += '! queue ! omxh264enc qp-mode=1 num-slices=8 gop-length=60 \
                   periodicity-idr=270 control-rate=low-latency \
                   gop-mode=low-delay-p gdr-mode=horizontal cpb-size=200 \
                   initial-delay=100  filler-data=false min-qp=15 \
                   max-qp=40  b-frames=0  low-bandwidth=false target-bitrate=3000 \
                   ! video/x-h264, alignment=au '
     # RTP payloading
     pipeline += '! queue ! rtph264pay name=pay0 pt=96'
     # Start the RTSP server with the pipeline string
     factory.set_launch('( ' + pipeline + ' )')
     factory.set_shared(True)
     mounts.add_factory("/test", factory)
     out=subprocess.check_output("ifconfig | grep inet", shell=True)
     for line in out.decode("ascii").splitlines():
         m = re.search('inet *(.*?) ', line)
         if m:
             found = m.group(1)
             if found != "127.0.0.1":
```

```python
                    break
        uri="rtsp://{}:{}/test".format("127.0.0.1" if (found=="") else found, server.props.service)
        print ("Video is now streaming from {src} source. \n\
                        Run the command \"ffplay {uri}\" in another PC which have network access to the SoM board to view the video.\n".format(src=vidsrc, uri=uri))
        # File
    else :
        # Pass the frame with bbox to do the VCU encoding (with bbox info as encoding ROI):
        # - ROI info for VCU encoding generation, vvas_xroigen.
        pipeline += ' ! queue ! vvas_xroigen roi-type=1 roi-qp-delta=-10 roi-max-num=10 '
        # - VCU encoding, omxh264enc.
        pipeline += '! queue ! omxh264enc qp-mode=1 num-slices=8 gop-length=60 \
                     periodicity-idr=270 control-rate=low-latency \
                     gop-mode=low-delay-p gdr-mode=horizontal cpb-size=200 \
                     initial-delay=100  filler-data=false min-qp=15 \
                     max-qp=40  b-frames=0  low-bandwidth=false target-bitrate=3000 \
                     ! video/x-h264, alignment=au '

        pipeline += '! filesink location={} async=false'.format(vidout)
        print("The output file is {}".format(vidout))
        # The pipeline is entirely concluded, set to a PLAYING state
        gst_pipeline = Gst.parse_launch(pipeline)
        gst_pipeline.set_state(Gst.State.PLAYING)

        # [DEBUG] Generate pipeline dot file
        # ("tmp" is the name for the dotfile that will be saved in the location shown above, see the first lines)
        Gst.debug_bin_to_dot_file(gst_pipeline, Gst.DebugGraphDetails.ALL, "tmp")
        dotfile = dotdir + "tmp.dot"
        graph = pydot.graph_from_dot_file(dotfile, 'utf-8')
        print("The pipeline graph in dot format is ready at {dotfile}".format(dotfile=dotfile))


def main():

    # Argument parsing
    ap = argparse.ArgumentParser()
    ap.add_argument('-s', '--video_src', type=str, default='mipi', help='Type of input sensor (mipi, usb or a h264 file). Default is mipi, i.e. Mipi Camera sensor.')
    ap.add_argument('-o', '--video_out', type=str, default='display', help='Type of output device (display, rtsp or file). Default is display, i.e. DisplayPort/HDMI.')
    ap.add_argument('-W', '--video_width', type=int, default=1920, help='Width of the input video frames. Default is 1920 (working for mipi and HDMI output).')
    ap.add_argument('-H', '--video_height', type=int, default=1080, help='Height of the input video frames. Default is 1080 (working for mipi and HDMI output).')
    ap.add_argument('-n', '--not_inf', type=bool, default=False, help='No perform the model inference. Default is false, the inference is performed.')
    ap.add_argument('-f', '--profiling', type=bool, default=False, help='Framerate profiling is enabled. Default is false.')
    args = ap.parse_args()

    print ('Command line options:')
    print (' --video_src : ', args.video_src)
    print (' --video_out : ', args.video_out)
    print (' --video_width : ', args.video_width)
    print (' --video_height : ', args.video_height)
    print (' --not_inf : ', args.not_inf)
    print (' --profiling : ', args.profiling)

    # Arguments checks
    errors = 0
    if args.video_src != 'mipi' and args.video_src != 'usb' and not os.path.isfile(args.video_src):
        errors = errors + 1
        print("Invalid --video_src {}".format(args.video_src))
    if args.video_out != 'display' and args.video_out != 'rtsp':
        print("The string --video_out is considered as a file path {}".format(args.video_out))
    if args.video_width != 1920:
        print("You are using a input frame width different by 1920. Pay attention if {} is compatible with you devices.".format(args.video_width))
    if args.video_height != 1080:
        print("You are using a input frame height different by 1080. Pay attention if {} is compatible with you devices.".format(args.video_height))
    if args.not_inf:
        print("You are not performing the AI model inference. It is a good test to verify input and output working.")
    if args.profiling:
        print("You are performing the framerate evaluation (fps). The performance should be affected by it.")

    # If no errors the application code can be executed
    if errors == 0:
        app(args.video_src, args.video_out, args.video_width, args.video_height, args.not_inf, args.profiling)

if __name__ == '__main__':
    main()
```

Bibliography

[1] Content European Commission Directorate-General for Communications Networks and Technology. *COMMUNICATION FROM THE COMMISSION - Artificial Intelligence for Europe.* `https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:52018DC0237`. 2018 (cit. on p. 2).

[2] Dr. Meghna Utmal. «Taxonomy on Machine Learning Algorithms». In: *International Journal of Recent Development in Engineering and Technology* 10 (2021). ISSN: 2347-6435 (cit. on p. 2).

[3] Erik Schrijvers Haroon Sheikh Corien Prins. *Mission AI - The New System Technology.* `https://link.springer.com/book/10.1007/978-3-031-21448-6`. Springer Cham, 2023. DOI: `10.1007/978-3-031-21448-6` (cit. on pp. 3, 6).

[4] IBM. *What is deep learning?* `https://www.ibm.com/topics/deep-learning` (cit. on p. 3).

[5] IBM Data and AI Team. *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the difference?* `https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/`. 2023 (cit. on p. 5).

[6] javatpoint.com. *Supervised Machine Learning.* `https://www.javatpoint.com/supervised-machine-learning` (cit. on p. 5).

[7] Yifang Ma, Zhenyu Wang, Hong Yang, and Lin Yang. «Artificial intelligence applications in the development of autonomous vehicles: a survey». In: *IEEE/CAA Journal of Automatica Sinica* 7.2 (2020), pp. 315–329. DOI: `10.1109/JAS.2020.1003021` (cit. on p. 6).

[8] SAE International. «Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles». In: (2021). `https://www.sae.org/standards/content/j3016_202104/`, p. 41. DOI: `10.4271/J3016_202104` (cit. on p. 6).

[9] Surya Gutta. *Benefits of Autonomous Vehicles*. `https://medium.com/geekculture/benefits-of-autonomous-vehicles-e90ebfd324e`. 2021 (cit. on p. 6).

[10] Debiprasad Bandopadhyay. *Autonomous Cars - How Computer Vision is Revolutionizing the Automotive Industry*. `https://www.linkedin.com/pulse/autonomous-cars-how-computer-vision-revolutionizing-bandopadhyay`. 2023 (cit. on p. 7).

[11] David Castells-Rufas, Vinh Ngo, Juan Borrego-Carazo, Marc Codina, Carles Sanchez, Debora Gil, and Jordi Carrabina. «A Survey of FPGA-Based Vision Systems for Autonomous Cars». In: *IEEE Access* 10 (2022), pp. 132525–132563. DOI: `10.1109/ACCESS.2022.3230282` (cit. on pp. 7–9).

[12] Soumya Sudhakar, Vivienne Sze, and Sertac Karaman. «Data Centers on Wheels: Emissions From Computing Onboard Autonomous Vehicles». In: *IEEE Micro* 43.1 (Jan. 2023), pp. 29–39. ISSN: 1937-4143. DOI: `10.1109/MM.2022.3219803` (cit. on p. 7).

[13] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. «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` (cit. on pp. 8, 9, 11).

[14] Zvonko Vranesic Stephen Brown. *Fundamentals of Digital Logic with VHDL Design, 3rd edition*. McGraw-Hill, 2009 (cit. on p. 10).

[15] AMD Xilinx. *Kria K26 SOM: The Ideal Platform for Vision AI at the Edge*. `https://docs.amd.com/v/u/en-US/wp529-som-benchmarks`. 2021 (cit. on p. 11).

[16] AMD. *Kria KV260 Vision AI Starter Kit User Guide*. `https://docs.amd.com/r/en-US/ug1089-kv260-starter-kit`. 2023 (cit. on pp. 12, 57).

[17] AMD. *Zynq UltraScale+ Device Technical Reference Manual*. `https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual`. 2023 (cit. on pp. 11, 37).

[18] Xilinx. *Zynq™ UltraScale+™ MPSoC*. `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html` (cit. on p. 13).

[19] AMD. *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide*. `https://docs.amd.com/r/en-US/pg338-dpu`. 2023 (cit. on pp. 12, 29–31, 43).

[20] Antonius Freenergi. *Convolutional Neural Network for Object Recognition and Detection*. `https://medium.com/@ringlayer/convolutional-neural-network-for-object-recognition-and-detection-126a22af8975`. 2019 (cit. on p. 15).

[21] Mathworks. *What Is a Convolutional Neural Network?* https://uk.mathworks.com/discovery/convolutional-neural-network.html (cit. on p. 15).

[22] AMD. *Vitis AI Library User Guide 3.0.* https://docs.amd.com/r/3.0-English/ug1354-xilinx-ai-sdk/Introduction. 2023 (cit. on p. 15).

[23] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.* 2016. arXiv: 1506.02640 [cs.CV] (cit. on p. 15).

[24] Jacob Solawetz. *What is YOLOv5? A Guide for Beginners.* https://blog.roboflow.com/yolov5-improvements-and-evaluation. 2020 (cit. on p. 15).

[25] Fangbo Zhou, Huailin Zhao, and Zhen Nie. «Safety Helmet Detection Based on YOLOv5». In: *2021 IEEE International Conference on Power Electronics, Computer Applications (ICPECA).* 2021, pp. 6–11. DOI: 10.1109/ICPECA51329.2021.9362711 (cit. on p. 16).

[26] Andreas Geiger, Philip Lenz, and Raquel Urtasun. «Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite». In: *Conference on Computer Vision and Pattern Recognition (CVPR).* 2012 (cit. on p. 16).

[27] Rafael Padilla, Wesley L. Passos, Thadeu L. B. Dias, Sergio L. Netto, and Eduardo A. B. da Silva. «A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit». In: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030279. URL: https://www.mdpi.com/2079-9292/10/3/279 (cit. on p. 18).

[28] Glenn Jocher. *YOLOv5 by Ultralytics.* Version 7.0. May 2020. DOI: 10.5281/zenodo.3908559. URL: https://github.com/ultralytics/yolov5 (cit. on p. 19).

[29] Sushant Patrikar. *Batch, Mini Batch Stochastic Gradient Descent.* https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. 2019 (cit. on pp. 20, 21).

[30] AMD. *Vitis AI User Guide 3.0.* https://docs.amd.com/r/3.0-English/ug1414-vitis-ai. 2023 (cit. on pp. 23, 25, 28, 29).

[31] LogicTronix [FPGA Design + Machine Learning Company]. *YOLOv5 Quantization Compilation with Vitis AI 3.0 for Kria.* https://www.hackster.io/LogicTronix/yolov5-quantization-compilation-with-vitis-ai-3-0-for-kria-7b005d#toc-quantizing-yolov5-pytorch-with-vitis-ai-3-0-5. 2023 (cit. on pp. 23, 27, 28).

[32] AMD Xilinx. *Vitis-AI.* Version 3.0. 2023. URL: https://github.com/Xilinx/Vitis-AI/tree/3.0 (cit. on pp. 25, 50).

[33] AMD. *Vitis AI 3.0.* `https://xilinx.github.io/Vitis-AI/3.0/html/index.html`. 2023 (cit. on pp. 26, 27, 30).

[34] AMD. *Smartcam Application for Kria KV260 - 2022.1.* `https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/smartcamera_landing.html`. 2022 (cit. on pp. 32, 33, 36, 38, 43, 47, 50).

[35] AMD. *Kria SOM Accelerator and Custom Carrier Card Firmware Development - 2022.1.* `https://xilinx.github.io/kria-apps-docs/creating_applications/2022.1/build/html/index.html`. 2022 (cit. on p. 35).

[36] AMD. *KRIA SOM VITIS PLATFORMS AND OVERLAYS - 2022.2.* `https://github.com/Xilinx/kria-vitis-platforms/tree/xlnx_rel_v2022.2`. 2022 (cit. on p. 34).

[37] AMD. *Vitis Custom Embedded Platform Creation Example on KV260 - 2022.2.* `https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Vitis_Platform_Creation/Design_Tutorials/01-Edge-KV260`. 2022 (cit. on p. 37).

[38] AMD. *PetaLinux Tools Documentation: Reference Guide 2022.2.* `https://docs.amd.com/r/2022.2-English/ug1144-petalinux-tools-reference-guide/Overview`. 2022 (cit. on p. 45).

[39] AMD Xilinx. *Kria Starter Kits Application Firmware.* Version xlnx$_r el_v$2022.2. 2022. URL: `https://github.com/Xilinx/kria-apps-firmware/tree/xlnx_rel_v2022.2` (cit. on p. 46).

[40] AMD. *Generating DTSI and DTBO Overlay Files - 2022.1.* `https://xilinx.github.io/kria-apps-docs/creating_applications/2022.1/build/html/docs/dtsi_dtbo_generation.html#example`. 2022 (cit. on p. 47).

[41] AMD. *AP1302 Firmware - 2022.1.* `https://github.com/Xilinx/ap1302-firmware/tree/xlnx_rel_v2022.1`. 2022 (cit. on p. 48).

[42] AMD. *Multimedia User Guide - 1.7.* `https://docs.amd.com/r/en-US/ug1449-multimedia/Document-Scope`. 2023 (cit. on p. 50).

[43] AMD. *VVAS Plug-ins - 2.0.* `https://xilinx.github.io/VVAS/2.0/build/html/docs/common/common_plugins.html`. 2023 (cit. on p. 52).

[44] AMD. *Kria KV260 Smartcam - Customizing the AI Models Used in the Application - 2022.1.* `https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/customize_ai_models.html`. 2022 (cit. on p. 52).

[45] Marcel Sheeny, Emanuele De Pellegrin, Saptarshi Mukherjee, Alireza Ahrabian, Sen Wang, and Andrew Wallace. «RADIATE: A Radar Dataset for Automotive Perception». In: *arXiv preprint arXiv:2010.09076* (2020) (cit. on p. 59).