DEBRECENI EGYETEM, INFORMATIKAI KAR

4028 Debrecen, Kassai út 26., 4002 Debrecen, Pf. 400.
fircc@inf.unideb.hu

Continental Automotive Hungary Kft. and DE IK final report of the joint scholarship agreement

By: Byron Fabricio Sarabia Morales – Computer Science Engineer MSc.

Design of a Hardware Accelerator Optimized for Neural Network Architecture

1. Description:

Machine learning applications typically have two main lifecycle phases: the training phase, during which the neural network is optimized and fine-tuned for a specific task, and the inference phase, when the trained network is used to solve the problem. Machine learning is usually performed on computers equipped with high-performance graphics cards, whereas inference may, in certain cases, be carried out on low-power, inexpensive, and low-computational-capacity embedded systems. These systems are typically equipped with specialized neural network accelerator modules.

Different accelerators have different hardware architectures, and therefore different characteristics and performance capabilities. They can execute certain layers or sub-operations of neural networks with varying levels of efficiency. As a result, different network architectures may require different execution units as the optimal choice.

When designing neural networks for inference on embedded devices, it is important to consider the hardware constraints to achieve the most optimal inference time (e.g., in convolutional layers: optimal kernel size, optimal number of channels). The task is to design an inference accelerator hardware running on an FPGA, which can be specifically optimized for a given neural network using parameters provided during synthesis. Figure 1 shows the overall process of the development of a machine learning application.

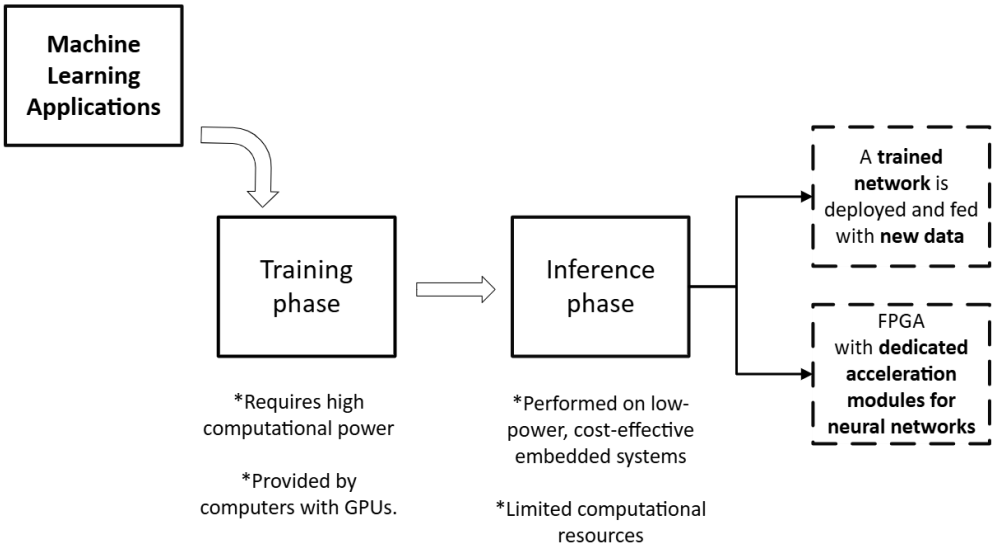


Figure 1. Stages of a Machine Learning system for training and inference.

2. Object Detection Task

Object detection plays a crucial role within the field of computer vision. Numerous machine learning (ML) and deep learning (DL) models are utilized to improve the effectiveness and accuracy of object detection and its associated tasks.
In earlier times, two-stage object detectors were widely used and demonstrated strong performance. However, recent advancements in single-stage object detection and its underlying algorithms have led to notable improvements, often surpassing many two-stage detectors. Additionally, the emergence of the YOLO family has driven numerous applications to adopt YOLO models for object detection and recognition across different contexts, where they have shown exceptional performance compared to their two-stage counterparts.

Computer vision has emerged as a dominant and highly adaptable field in the modern era, attracting extensive research efforts from scholars worldwide. It enables machines to interpret, comprehend, and analyze visual data at a high level. The field encompasses several subdomains, including scene and object recognition, object detection, video tracking, object segmentation, pose and motion estimation, scene modeling, and image restoration. Common deep learning models applicable to various computer vision tasks include Convolutional Neural Networks (CNNs), Deep Belief Networks (DBNs), Deep Boltzmann Machines (DBMs), Restricted Boltzmann Machines (RBMs), and Stacked Autoencoders.

Image classification involves assigning an image or an object within an image to one of several predefined categories. This task is typically addressed using supervised machine learning or deep learning techniques, where models are trained on large labeled datasets. Popular machine learning models used for image classification include Artificial Neural Networks (ANNs), Support Vector Machines (SVMs), Decision Trees, and K-Nearest Neighbors (KNN). In the realm of deep learning, Convolutional Neural Networks (CNNs) and their architectural extensions and variants have outperformed other deep learning models in image classification and related tasks.

Object localization refers to the process of identifying the position of an object, or multiple objects, within an image or frame by enclosing them in rectangular regions, commonly known as bounding boxes. [1] In Figure 2, it is possible to see the differences between image classification, localization, and object detection.

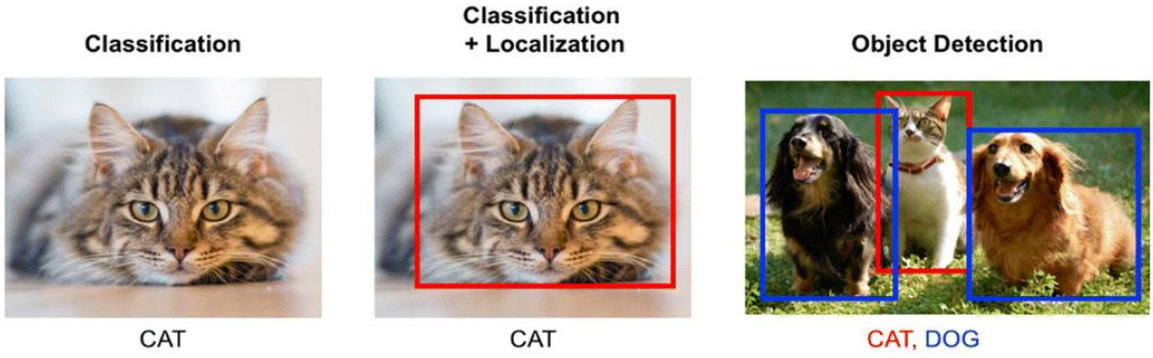


Figure 2. Classification, Localization and Object Detection in an image.

3. You Only Look Once (YOLO)

You Only Look Once (YOLO) is a widely adopted and popular algorithm known for its efficiency in object detection. At the heart of the YOLO detection framework is its compact model size and high computational speed. Its architecture is relatively simple, enabling the neural network to directly predict both the position and class of bounding boxes. YOLO achieves remarkable speed by requiring only a single pass of the image through the network to generate the final detection output, which also makes it suitable for real-time video detection. By analyzing the entire image at once, YOLO captures global context information, helping to minimize false detections of background regions as objects. [2]

3.1 YOLOv3

The You Only Look Once (YOLO) version 3 object detector is a multi-scale detection framework that leverages a feature extraction network along with multiple detection heads to perform predictions across various scales. YOLO v3 utilizes a deep convolutional neural network (CNN) to process the input image, generating predictions from multiple feature maps. These predictions are then aggregated and decoded by the detector to produce the final bounding boxes. In Figure 3 presents how Yolo V3 works. [3]

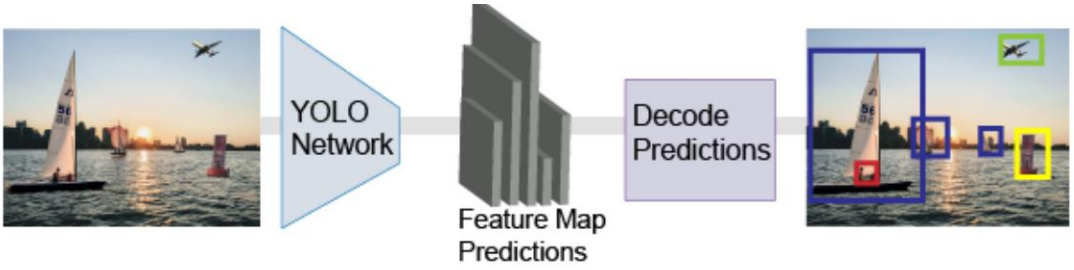


Figure 3. Yolo V3 methodology for object detection.

4. Technical Features and Comparison of FPGA Boards.

In this work, two FPGA development boards have been used: the PYNQ-Z2 board and the KRIA KV260 board, both from the manufacturer Xilinx. Although their price range, technical specifications, and target applications are different, a summary of both boards is presented below, followed by a comparative table.

4.1 Pynq Z2

PYNQ is an open-source project from AMD that makes it easier to use AMD platforms. Using the Python language and libraries, designers can exploit the benefits of programmable logic and micro-processors. A key feature is the use of Jupyter Notebook to use directly the Programable Logic (PL) of the FPGA and increase productivity. The PYNQ-Z2 board featuring the Zynq-700 FPGA. [4]

4.2 Kria KV260 Vision AI Starter Kit

The development platform for AMD Kria™ K26 SOMs, the KV260 Starter Kit is built for advanced vision application development without requiring complex hardware design knowledge. The Kria KV260 Vision AI Starter Kit is designed to provide customers a platform to evaluate their target applications for smart city and machine vision, security cameras, retail analytics, and other industrial applications. The Kria KV260 board featuring the Zynq™ UltraScale+™ MPSoC FPGA. [5]

4.3 Comparison of boards

Table 1. Comparison of Technical Features between boards.

	Pynq Z2	**Kria KV260**
FPGA	Zynq-7000 SoC XC7Z020-1CLG400C 85K logic cells (13300 logic slices, each with four 6-input LUTs and 8 flip-flops) 220 DSP slices	Zynq™ UltraScale+™ MPSoC 256K logic cells 1200 DSP slices
MEMORY	512 Mbyte DDR3 with 16-bit bus @ 1050 Mbps 128 Mbit Quad-SPI Flash Micro SD card connector	4 GB (4 x 512 Mb x 16 bit) 512 Mb QSPI SDHC card
I/O INTERFACES	USB-JTAG Programming circuitry USB OTG 2.0 /USB-UART bridge One 10/100/1G Ethernet HDMI Input / HDMI Output I2S interface with 24bit DAC with 3.5mm TRRS jack Line-in with 3.5mm jack	4 x USB® 3.0/2.0 Interface One 10/100/1000 Mb/s Ethernet Interface HDMI™ 1.4 DisplayPort™ 1.2a 2 x IAS MIPI Sensor Interfaces 1 x Raspberry Pi Camera Interface
EXPANSION PORTS	2 Pmod ports 1 Arduino Shield connector Raspberry Pi connector	1 Pmod 12-pin Interface

5. Methodology

With the project focus established, it is essential to emphasize that synthesis constitutes a specific phase within the broader process of developing a hardware accelerator for the inference stage of a neural network. To obtain measurable and meaningful results, it is first necessary to define both the neural network model to be used and its intended application. In parallel, the target FPGA platform must also be specified.

Based on the initial meeting with the Continental team, the selected application is real-time object detection. For this purpose, the YOLOv3 neural network has been chosen. This network will be trained using the COCO dataset, a large-scale dataset that includes 80 distinct object categories. Following the training phase, the PYNQ-Z2 and KRIA KV260 boards will be utilized to perform inference, due to its affordability, accessibility, and compatibility with USB cameras.

To support neural network inference on the FPGA, a Deep Learning Processing Unit (DPU) will be implemented. This hardware engine is specifically designed to execute trained neural network models efficiently. A camera will be connected to the system to capture video frames, which will then be processed by the neural network for object detection in real time.

A key metric in this process is the inference time per frame, which will serve as a baseline for evaluating the impact of different synthesis configurations. By analyzing how synthesis parameters influence inference performance, it will be possible to determine whether they contribute to reductions or increases in processing time. Figure 4 presents the general schematic of the implemented system.

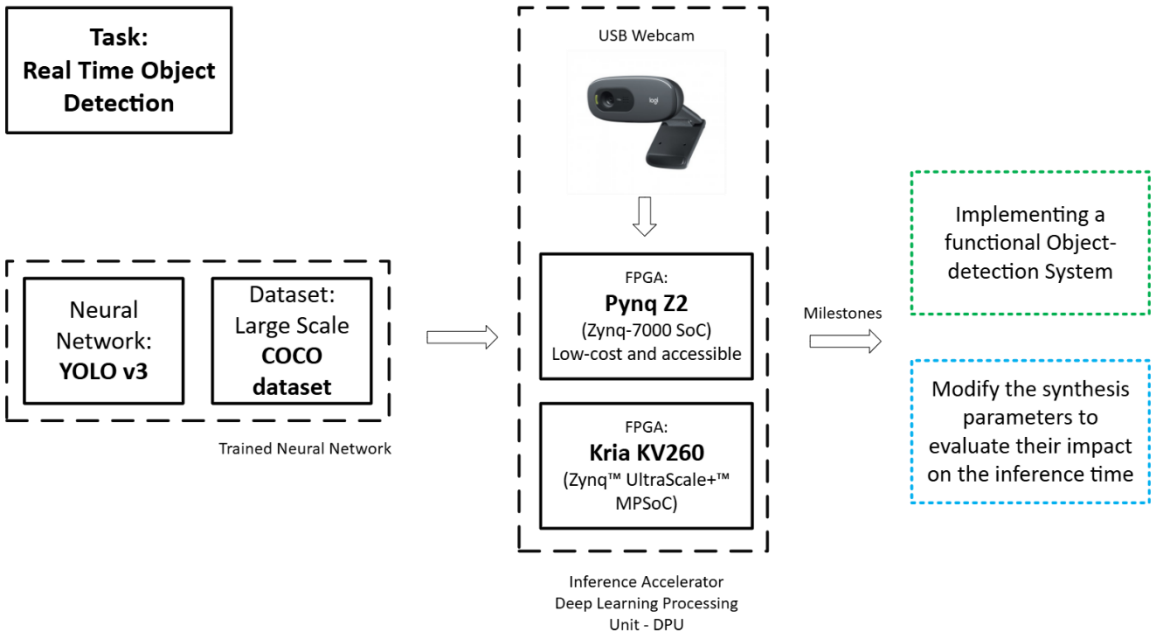


Figure 4. System structure for object detection.

6. Experimental Results on FPGA-Based Accelerators

6.1 Pynq Z2

The workflow for developing the previously described object detection system is illustrated in the Figure 5. While the overall process is extensive, one of the most critical steps is the generation of the Deep Learning Processing Unit (DPU) using the VIVADO Design Suite. The resulting hardware design is packaged as an SD card image and deployed to the FPGA board. This step involves the use of a virtual machine running Ubuntu, as it requires tools such as PetaLinux for the creation of the bootable image.

In parallel, the model preparation phase includes the optimization and compilation of the YOLOv3 object detection model. This includes reducing the model's complexity and generating the necessary files for deployment on the DPU. These steps are constrained by the resource limitations of the PYNQ-Z2 board, which needs a careful balance between model size and hardware capabilities.

The process begins with the conversion of the YOLOv3 model, originally implemented in the Darknet framework, into the TensorFlow format. Following this, a quantization procedure is applied to reduce the bit-width of the model's weights, thereby decreasing its size and computational requirements. Lastly, the compilation step transforms the model into a binary format compatible with the DPU, enabling real-time inference on the FPGA. [4]

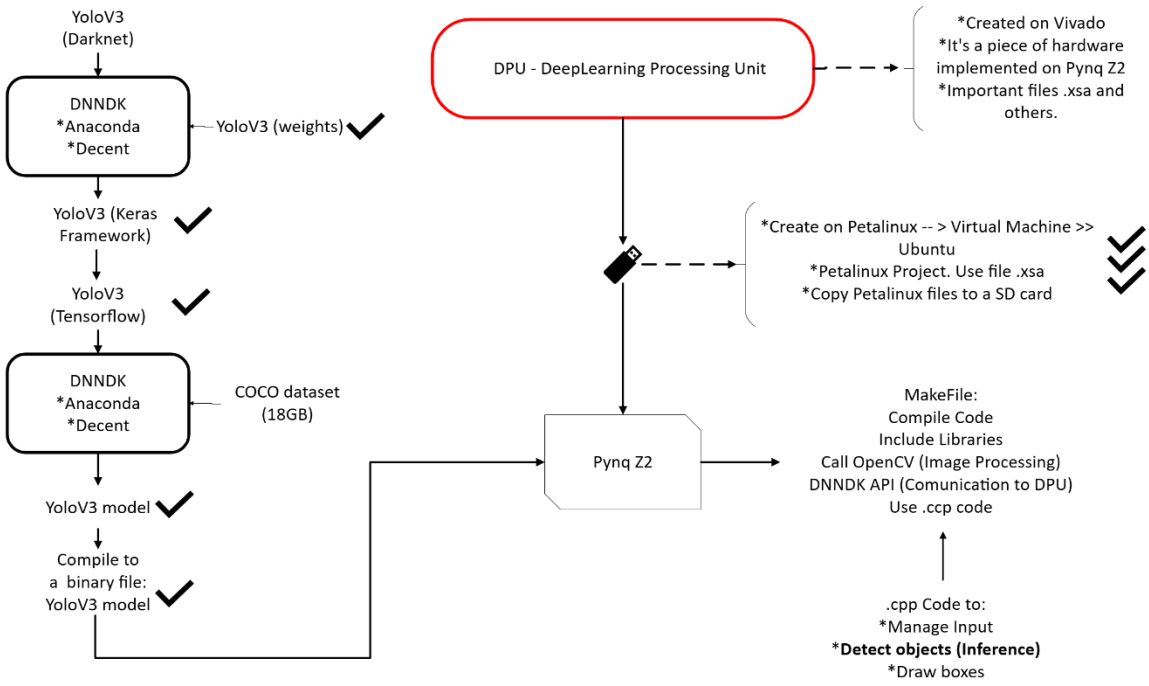


Figure 5. Implementation of the Object Detection system using a Pynq Z2 board.

Upon completion of the extensive development steps, the first milestone, which is the implementation of the object detection system, has been successfully achieved. As it can

be seen in Figure 6, the object detection system has been deployed with the previously specified characteristics.

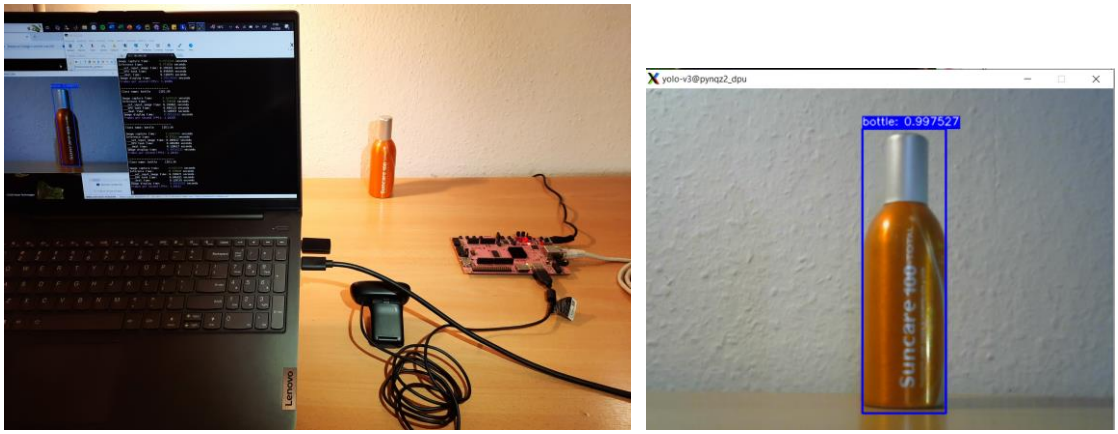


Figure 6. Implemented object detection system using Pynq Z2.

A time analysis of the system's performance is presented. The total inference time, measured under the defined hardware and operational conditions, is approximately **775 ms per frame**. Within this, the DPU task time, which corresponds solely to the inference execution on the hardware accelerator, accounts for **446 ms**. The remaining time is primarily due to data transfer overhead, which includes communication between the FPGA's processing system (PS) and programmable logic (PL) before and after inference.

Table 2 presents a summary of the results. From this point forward, Inference Time will refer to the time taken by the DPU to process a frame. FPS (Frames Per Second) represents an ideal processing rate, since in addition to the inference time, it is also necessary to consider the time spent in the Processing System (PS) of the FPGA, which handles data transfer to and from the Programmable Logic (PL). Therefore, the reported FPS (ideal) corresponds to the maximum possible frame rate, assuming an additional processing time of 0 seconds—an ideal scenario.

Table 2. Inference Time Results – Pynq Z2.

Technique	Inference Time	FPS (ideal)
Pynq Z2 + YOLOv3 (custom for Pynq Z2)	0.446 s	2.24

6.2. Kria KV260

With the goal of developing an object detection system using the KRIA KV260 board, and considering this FPGA’s capabilities for computer vision applications, the following experiments are proposed:

i. Using the Smart Camera application on the KV260 board, verify the correct operation and evaluate the FPGA’s capabilities for implementing face detection

and pedestrian detection task, which are tasks derived from general object detection.

ii. The second step is to use the Smart Cam application's architecture as a base, and quantize and compile a YOLOv3 neural network, meeting the technical requirements needed to deploy it on the KV260 board.
iii. The third step is to use only the DPU engine of the KV260 board through a C++ script, and, using the YOLOv3 neural network developed in the step ii, verify the performance.
iv. Obtain a DPU architecture using Vivado 2022.2 in order to modify different configurations and synthesis parameters, and evaluate their impact on inference time.

The following section explains in detail the process and results of steps i–iv, considering the extensive development required for each one. A detailed step-by-step tutorial is included in the Annexes.

i. Smart Camera Application on KRIA KV260

The Smart Camera application design built on the KV260 Vision AI Starter Kit provides a framework for building and customizing video platforms that consist of four pipeline stages:

- Capture pipeline
- Video processing pipeline
- Acceleration pipeline
- Output pipeline

The design has a platform and integrated accelerator functions. The platform consists of capture pipeline, output pipeline, and some video processing functions. This approach makes the design leaner and provides maximum programmable logic (PL) for the accelerator development. The platform supports capture from MIPI single sensor device, a USB webcam, and a file source. The output can be stored as files, passed forward via ethernet using the real time transport protocol (RTP) or displayed on DisplayPort/HDMI monitor. Along with video, the platform also supports audio capture and playback.
The following acceleration functions can be run on this platform using programmable deep learning processor unit (DPU). [5]

- Face Detection
  Network model: Densebox_640_360

- Cars, Bicycles, and Person Detection for ADAS
  Network model: ssd_adas_pruned_0_95

- Pedestrian Detection
  Network model: refinedet_pruned_0_96

Figure 7 below shows the results obtained from using the Smart Camera application, specifically focusing on the Face Detection Task and the Pedestrian Task. Since this application was developed for the Kria KV260, it supports various input and output data

options. All of them were tested, showing minimal lag when connecting a DP monitor directly to the FPGA, and noticeable lag when streaming frames from a camera o video via RTSP to a host computer. Annex 1 presents the initial steps for using the Kria KV260 board. Annex 2 provides the instructions for using the Smart Camera Application.

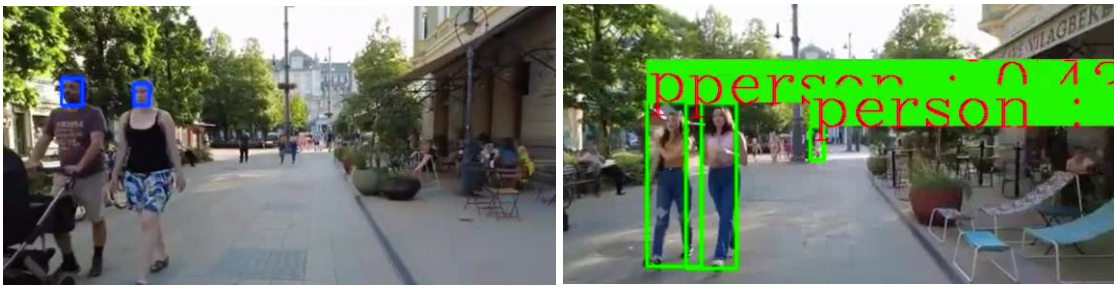


Figure 7. Results of Face Detection Task and Pedestrian Detection Task using Smart Camera Application for Kria KV260.

ii. Custom Yolo V3 for Kria Architecture

Once a functional system is established, in this case, the Smart Camera Application on the Kria board, the plan is to use its architecture but instead of deploying the .xmodel file corresponding to one of the prebuilt models of the Smart Camera Application, deploy a YOLOv3 model trained with the COCO dataset, compiled and optimized to run on the Kria board architecture. This is an extensive process detailed in the Annexes 3-5, 6. Where you can see all the steps until obtaining the .xmodel file. The results are shown below in Figure 8.

Figure 8. Object detection using Kria KV260 and custom Yolo V3.

iii. Custom YOLOv3 and DPU of Smart Camera application

In this third approach, using the trained and quantized neural network YOLOv3 through a C++ script, the goal is to use only the DPU from the Smart Camera Application, allowing inference to be done in hardware (PL). In this case, several options such as video recording or streaming via RTSP are not possible; generally, the output will be sent to a DP display/screen. To verify the effectiveness of this technique, inference time and Frames Per Second (FPS) are also printed for comparison and later analysis. The results of this approach are shown in the following figures. In Figure 9 the implemented system is presented. In Annex 6 is presented how a YOLOv3 is deploy using a C++ script.

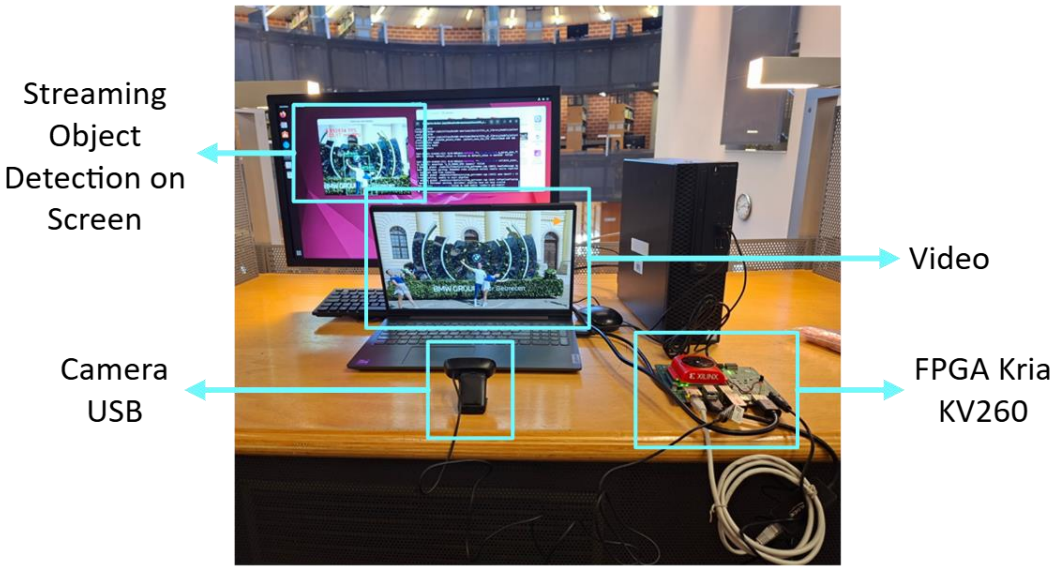


Figure 9. Implemented object detection system using Kria KV260.

In Figure 10, the object detection over the frame and the inference time as well as the FPS processed are shown.

Figure 10. Object detection over a frame with time analysis.

In Table 3, it is presented the results in the inference time over a frame, using the system shown in Figure 9, the results are as follows:

Table 3. Inference Time Results – KRIA KV260.

Technique	Inference Time	FPS (ideal)
KRIA KV260 + YOLOv3 (custom for Kria KV260) using a C++ script	0.103 s	9.67

iv. DPU architecture on Vivado.

Finally, as the last step, the goal is to obtain an architecture, or more specifically, the block design on Vivado Design Suite, of a Deep Learning Processing Unit (DPU) running on the Kria KV260. This will allow modifying the necessary parameters in the synthesis process and how the DPU operates, with the aim of optimizing its performance and achieving the most optimal inference time. This will enable real-time processing of frames from a camera, running through the YOLOv3 deep learning technique for object detection.

According to the work in [6], different approaches to obtain the accelerator architecture in Vivado are presented. In this particular case, the decision was made to use the Vitis TRD flow and, in initial experiments, to use DPU-PYNQ for the KRIA KV260. This setup allows running inference on individual images using a Jupyter notebook, thereby increasing productivity. The idea is as follows:

- Obtain a general architecture for running deep learning models on an FPGA. This architecture usually includes not only the DPU but also the management of input and output data, external devices such as a display, and communication with the Processing System (PS) of the FPGA.
- Adapt a specific DPU that runs on the KRIA KV260 and integrate it into the Block Design of the general architecture. [9]
- Synthesize and implement the design to generate the necessary files (.bit, .hwh, .xclbin)
- Using a Jupyter Notebook, and the files obtained in the previous step to verify the inference time obtained from DPUs with different modifications on the Synthesis Parameters and the DPU IP block parameters.

The results can be seen in Figure 11. This workflow has allowed us to obtain a functional inference system using the FPGA's PL, and it also provides the possibility to optimize its performance—or at least have a system where modifications and tests can be made to evaluate the impact on inference time.

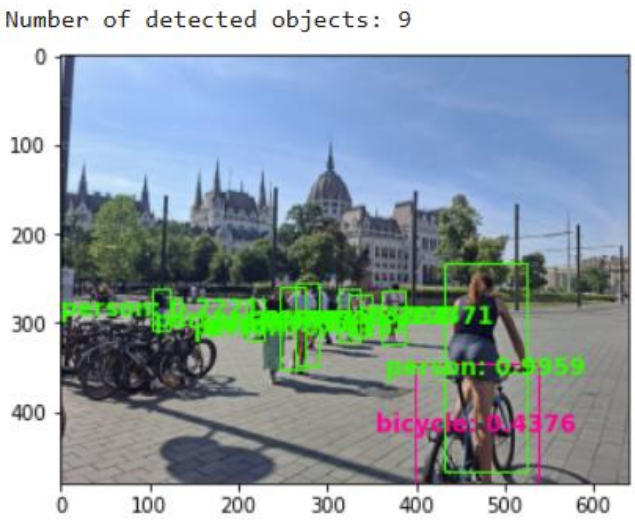


Figure 11. Object detection over an image using DPU-PYNQ and a Jupyter Notebook.

In Table 4, the applied changes and the results obtained are shown, considering the different modifications made to the synthesis parameters and DPU settings.

Table 4. Modified Parameters in DPU/Synthesis and Their Impact on Inference Time.

Modification	**Result in Inference Time**	**FPS**
Original DPU-PYNQ example file	0.182 s	5.506
Arch: B4096, URAM: Enabled (5 IMG, 17 WGT, 1 BIAS), DRAM: Disabled, RAM: Low usage, Channel Aug: Enabled, ALU Parallelism: Default, CONV ReLU: LeakyReLU + ReLU6, ALU ReLU: ReLU6, DSP48: High usage, Power Mode: Normal, Target: MPSoC Synthesis strategy: Default.	0.184 s	5.429
Arch: B4096, URAM: Enabled (5 IMG, 17 WGT, 1 BIAS), DRAM: Disabled, RAM: Low usage, Channel Aug: Enabled, ALU Parallelism: Default, CONV ReLU: LeakyReLU + ReLU6, ALU ReLU: ReLU6, DSP48: High usage, Power Mode: Normal, Target: MPSoC Synthesis strategy: Flow_AreaOptimized_high	0.185 s	5.415
Arch: B4096, URAM: Enabled (5 IMG, 17 WGT, 1 BIAS), DRAM: **Enable**, RAM: Low usage, Channel Aug: Enabled, ALU Parallelism: Default, CONV ReLU: LeakyReLU + ReLU6, ALU ReLU: ReLU6, DSP48: High usage, Power Mode: Normal, Target: MPSoC Synthesis strategy: Flow_AreaOptimized_high	0.183 s	5.464

In the previous table the DPU of the block design that create the object detection system in hardware (PL) has been modified, and also the synthesis parameters (STRATEGY=Flow_AreaOptimized_high) depending of the case. But the results seems that the modification of this parameters doesnt affect the inference time. Several aditional modifications has been performed, but these approaches didnt work in the FPGA because when they were launched, the board frezed. A remarcable thing is that in this time the neural network used was the one that comes with the DPU-PYNQ, because when the custom YoloV3 was launched, any approached worked in the fpga. Annex 7 shows the instructions for deploying and testing this approach.

7. Comparative Analysis.

In the following Figure 12, a comparative analysis is presented between the technique described in Section 6.1, which corresponds to the results obtained from object detection using the YoloV3 neural network and the Pynq Z2 board. The second technique under analysis is the one described in Section 6.2, item iii. These results were obtained from object detection with YoloV3, but using the Kria KV260 board and the DPU from the Smart Camera

Application. Finally, the third approach involves object detection in a single image, as described in Section 6.2, item iv, using DPU-Pynq through a Jupyter Notebook. In this case, a predefined Yolo model available on the Pynq platform was used, along with the Kria KV260 board. The response under analysis comes from applying modifications to the DPU and the synthesis process of the block design architecture in Vivado. The results are as follows:

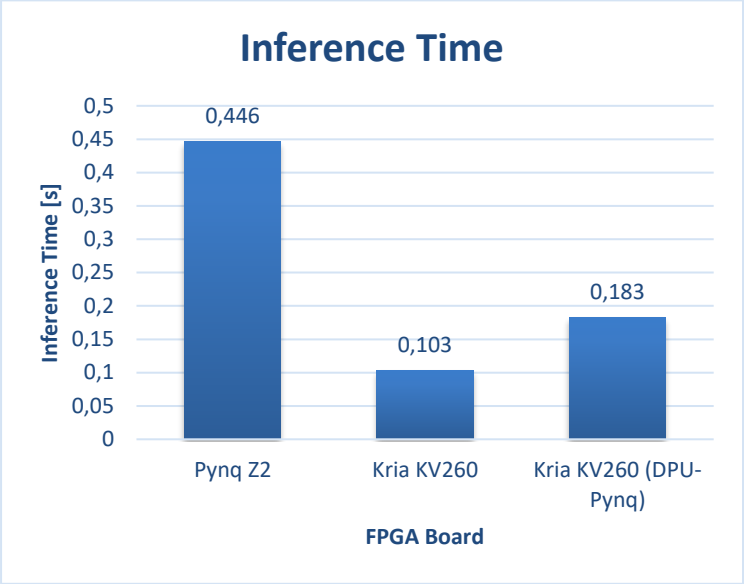


Figure 12. Comparative results of inference time using different FPGA boards.

8. Conclusions and Recomendations

- Although the main topic of this research was not initially focused on the development of a neural network for computer vision, but rather on implementing existing techniques on an FPGA and later optimizing the accelerator, it was still necessary to develop the machine learning technique, in this case for object detection, and adapt it using the manufacturer's documentation so it could run on the specific boards used: the Pynq Z2 and the Kria KV260. YOLOv3, a well-known deep learning technique, was selected and required an extensive series of steps to obtain the appropriate files for integration into the proposed system. This process was successfully completed, achieving a working implementation of YOLOv3 for both the Pynq Z2 and the Kria KV260 boards. Both versions of the neural network were tested on their respective platforms, delivering acceptable object detection results on frames coming from a camera, video input, or single images provided to the accelerator.

- The creation of the Deep Learning Processing Unit (DPU) presents its own challenges, particularly when adapting this architecture to the technical constraints of each FPGA board. The proper approach is to start by using pre-validated architectures, verify they work correctly, and then, as an additional step, optimize the hardware for a specific machine learning technique. This procedure was carried out in this work, and in addition, a native DPU from an existing application designed for a machine vision FPGA was also tested. When comparing the Pynq Z2, which uses a custom DPU developed in Vivado, with the Kria KV260 using the DPU designed for the Smart Camera application, the best inference time was achieved by the Kria

KV260. Both boards used YOLOv3 for object detection. This result can be seen in Figure 12 and is expected, considering the technical specifications in Table 1, which show that the Kria KV260 has better resources for image processing (logic cells, DSP slices, memory).

- Once both the neural network and the DPU were successfully developed and tested, and taking into account the previous results obtained using the Kria KV260, the process continued with the creation of a functional DPU architecture using a Block Design in Vivado Design Suite. The idea was to modify both internal DPU parameters as well as synthesis parameters, such as the synthesis strategy itself. Since applying these changes and generating the necessary files for hardware deployment is computationally intensive, a productive approach was chosen. For this reason, the DPU-Pynq environment was used, which allows controlling the hardware via Jupyter Notebook scripts, enabling inference to be run on an image and observing how parameter modifications affect inference processing. Once an optimal DPU configuration had been identified, the goal would have been to deploy the hardware within a Docker container and use it with camera input or video, aiming to improve upon the previously achieved 0.1s inference time. The conclusion of this phase is that, despite modifying all possible internal DPU and synthesis parameters, the expected improvements were not achieved. In most cases, the resulting inference time remained the same, and in some situations, the deployment failed due to system crashes on the FPGA board.

- As a recommendation, and considering the progress achieved so far, it is suggested to use the Vivado TRD Flow, which is a different approach from the one previously used. While the previous point followed the Vitis TRD Flow, the Vivado TRD Flow involves manually creating the DPU in Block Design, providing much more control over how the hardware behaves and allowing better adaptation to the trained neural network. However, this approach is significantly more time-consuming due to the manual nature of the hardware design process for the FPGA.

Bibliography

[1] G. A. &. J. V. T. Tausif Diwan, "Object detection using YOLO: challenges, architectural successors, datasets and applications," *Multimedia Tools and Application,* p. 9243–9275, 2023.

[2] D. E. F. L. Y. C. B. M. Peiyuan Jiang, "A Review of Yolo Algorithm Developments," *Procedia Computer Science 199,* pp. 1066-1073, 2022.

[3] "MATLAB Help Center," [Online]. Available: https://www.mathworks.com/help/vision/ug/getting-started-with-yolo-v3.html.

[4] "AMD," [Online]. Available: https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html.

[5] "Kria KV260 Vision AI Starter Kit," Xilinx, [Online]. Available: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html.

[6] A. Araujo, "Github," 2023. [Online]. Available: https://andre-araujo.gitbook.io/yolo-on-pynq-z2.

[7] "Kria Kv260," Xilinx, [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/introduction.html.

[8] X. E. C. Pabón, "Despliegue de una Red Neuronal Profunda en la Placa de desarrollo KRIA KV260 para detección e identificación de señales de tránsito," Quito, Ecuador, 2024.

[9] "AMD," [Online]. Available: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai/Integrating-the-DPU-into-Custom-Platforms.

Date: July 8th, 2025

.....

Signature of topic leader
Dr. Oniga István

Annex 1

KRIA KV260 - First Steps

The next step is deploying the trained, quantized and compile model .xmodel on an FPGA, in this case on the board KRIA KV260. For this, the first step is to flash and set the Smartcamera container to run our model on it. For detail info about it, read the next documents:

https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/linux_boot.html

Boot Kria Starter Kit Linux on KV260

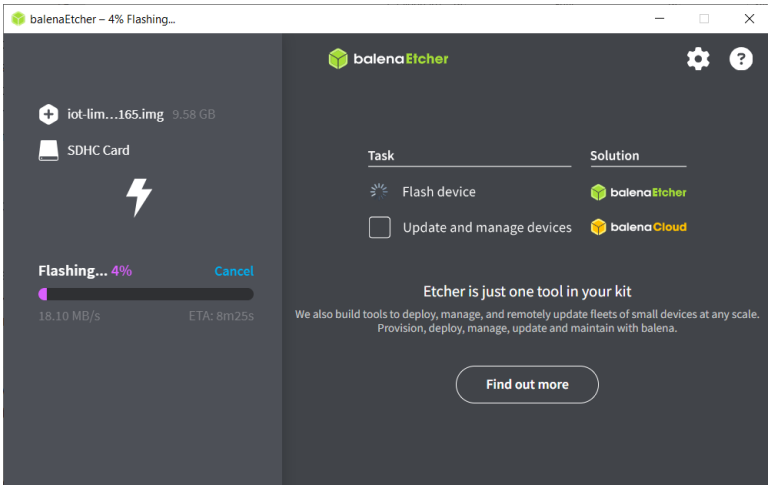
For new users evaluating the AMD® Kria Starter Kit, AMD recommends starting with the latest version of Ubuntu (Ubuntu 24.04), or Ubuntu 22.04 if you want to run example applications.

For users configuring their Kria Starter Kit for a specific application, refer to the table below to find your application and select the appropriate version of Linux.

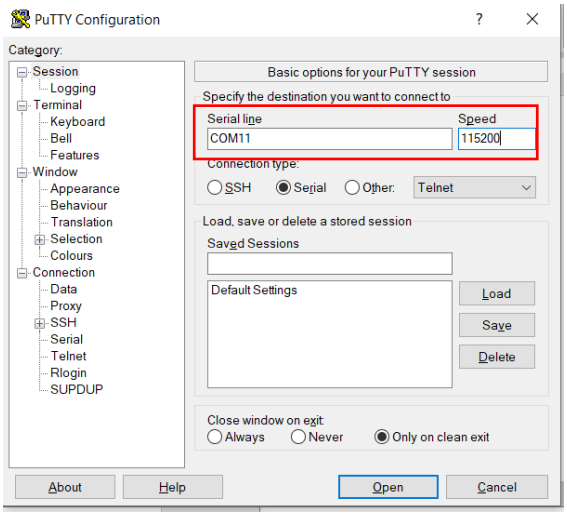
Starter Kit Linux Version	KV260 Instructions	Out of Box Validated Apps
Kria Ubuntu 22.04	Boot Linux Instructions	• Smart Camera • AI Box ReID • Defect Detect • NLP SmartVision • AI Box Distributed ReID • Built-In Self Test (BIST) • Dynamic Function eXchange (DFX)
Kria Ubuntu 24.04	Boot Linux Instructions	• As of now, there are no example applications for Ubuntu 24.04.

The process to flash is:

1. Download the Kria Ubuntu 22.04 LTS Image
2. Flash it on an SD card, in this case using balenaEtcher

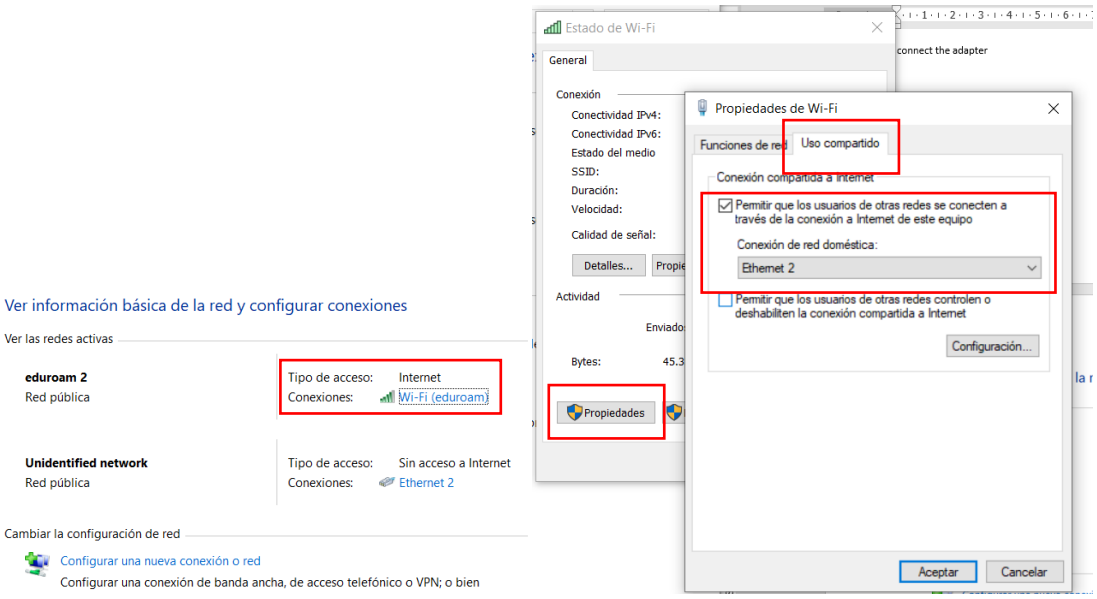


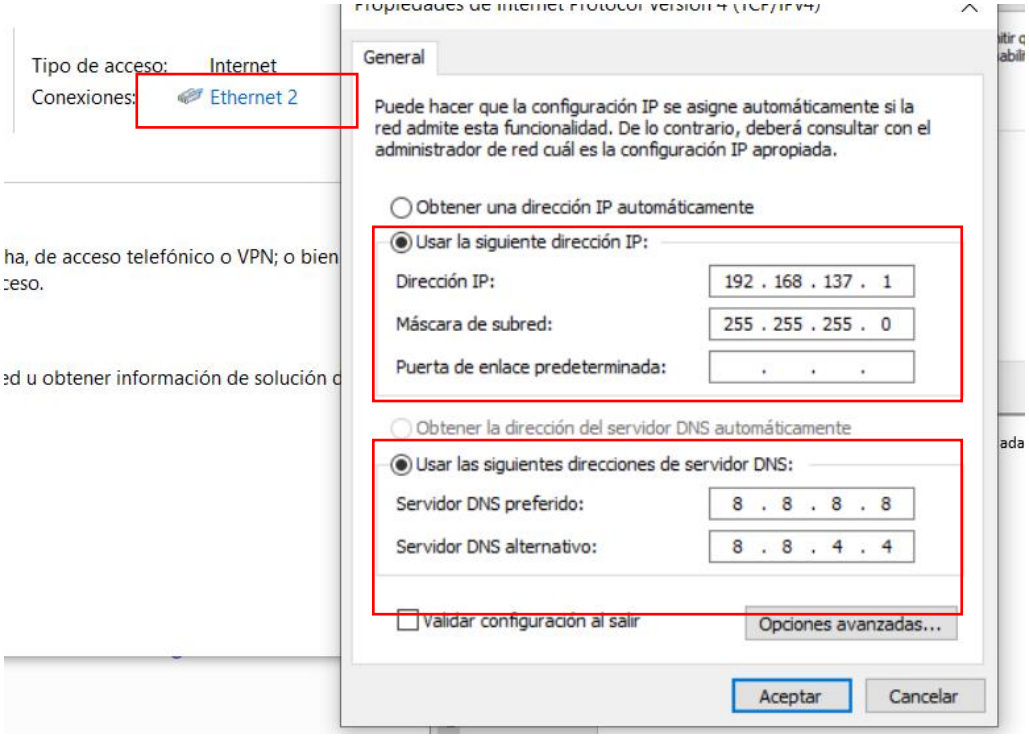
3. Insert the card on the board, connect the micro USB cable and Ethernet cable to a Host computer
4. Connect the DC power supply to the board
5. Log in over the USB-UART serial port to access the command line interface. Use putty according to your port. After that kria login: ubuntu, original password: ubuntu.



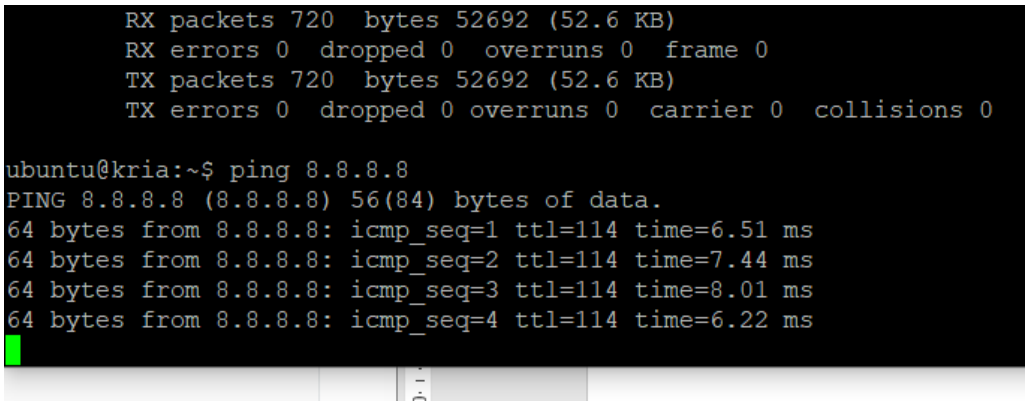
6. Because my computer doesn't have an ethernet port, I have to use an adapter, and for connect to the internet, first I will set an IP to the board. In this case 192.168.137.100. As a gateway the IP will be 192.168.137.1, this is because when I configure in the Host computer to share internet with the usb adapter, the host assigned automatically to it 192.168.137.1 /24.

```
sudo nmcli con add type ethernet ifname eth0 con-name kria-static ip4 192.168.137.100/24 gw4 192.168.137.1

sudo nmcli con modify kria-static ipv4.dns "8.8.8.8 8.8.4.4"

sudo nmcli con up kria-static
```





If everything went well, from the board to ping 8.8.8.8 show the next result



7. Firmware update following this steps: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/linux_boot/ubuntu_22_04/build/html/docs/fwupdate.html From https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/3020685316/Kria+SOM+Boot+Firmware+Update#K26-Boot-Firmware-Updates For compatibility, download and install this version:

2022.1 Boot FW Update	Unified FW for KV260 and KR260 Starter Kits. Addresses KR260 USB2.0 interfaces on U46 connector stack. Addresses KR260 PS Ethernet functionality on J10C physical interface. Fixes Linux WOL functionality for KR260 SGMII interface. Workaround for vai-lab chipset based USB hub.	KV260, KR260	Xilinx download - 2022.1_update3_BOOT.BIN	PetaLinux-generated, legacy versioning

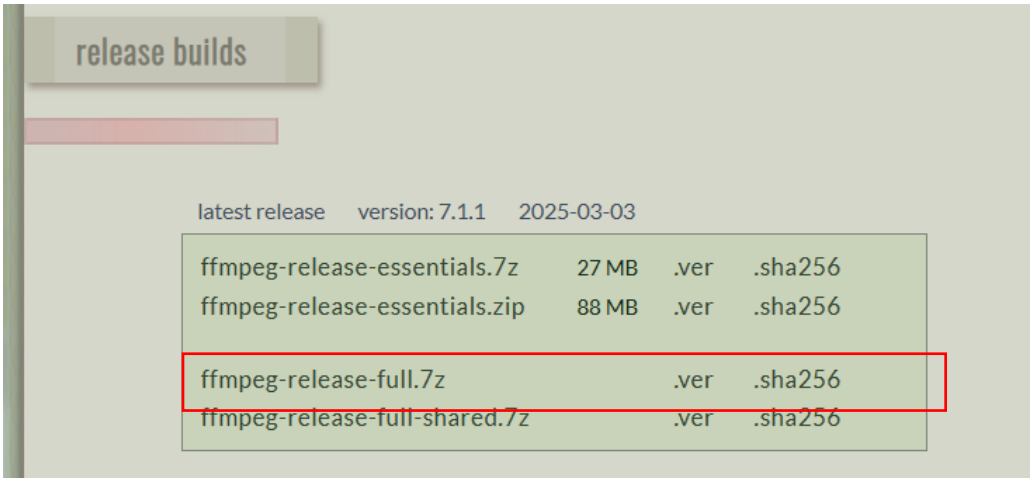
8. Reebot system.

```
sudo reboot
```

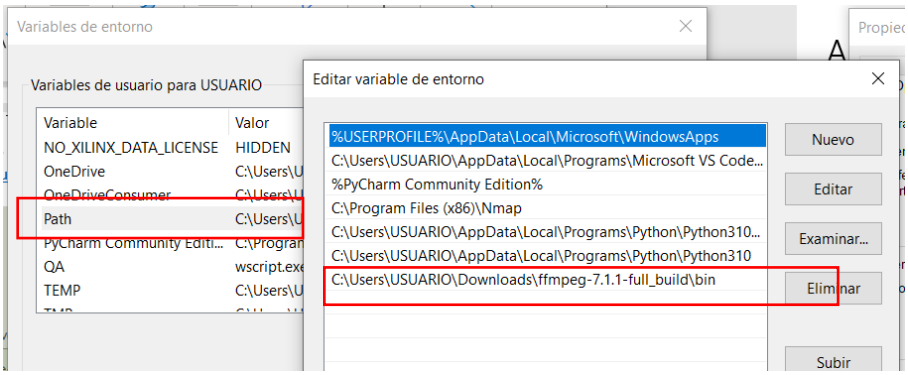
Annex 2
KRIA KV260 - Smartcamera Docker

This part is based on: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/app_deployment.html

6. For display video on screen of the host pc, it is useful to install ffmpeg. For windows go to : https://www.gyan.dev/ffmpeg/builds/



Decompress and go to folder “bin” and find the file “ffplay”. Then only add “bin” to the ”PATH” on Windows. Go to Panel control > system and security > system > Advanced configuration of the system then click on Environment variables button. On “system variables” edit “Path” and add the plat of the folder “bin”



Open cmd and check it

```
ffplay -version
```

7. Search for the package feed for packages compatible with the KV260 on the Kria board.

```
sudo apt search xlnx-firmware-kv260
```

8. Install the firmware binaries.

```
sudo apt install xlnx-firmware-kv260-smartcam
```

9. Pull the 2022.1 Docker image for smartcam using the following command:

```
docker pull xilinx/smartcam:2022.1
```

10. Check the docker image

```
docker images
```

Run pre-install AI tasks on Smartcam Docker Container on FPGA

1. Disable the desktop environment, and switch to the kv260-smartcam application.

```
sudo xmutil      desktop_disable
sudo xmutil unloadapp
sudo xmutil loadapp kv260-smartcam
```

2. Launch the Docker using the follwowing command:

```
docker run \
--env="DISPLAY" \
-h "xlnx-docker" \
--env="XDG_SESSION_TYPE" \
--net=host \
--privileged \
--volume="$HOME/.Xauthority:/root/.Xauthority:rw" \
-v /tmp:/tmp \
-v /dev:/dev \
-v /sys:/sys \
-v /etc/vart.conf:/etc/vart.conf \
-v /lib/firmware/xilinx:/lib/firmware/xilinx \
-v /run:/run \
-it xilinx/smartcam:2022.1 bash
```

3. You will have several combinations depending on the type of input (MIPI cam, USB cam or file) and output (DP screen, RTPS or file) Check on: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/app_deployment.html

- USB cam | RSTP stream

```
smartcam --usb 0 -W 640 -H 480 -r 30 --target rtsp
```

(You could modify values of usb camera port, size and FPS)

To open the stream, open a cmd and type the next:

```
ffplay rtsp://192.168.137.100:554/test
```

- MIPI cam | File

First attache a camera MIPI to port J7 on the FPGA. Then create a folder on the root directory of the FPGA. In my case the folder is called “BS_shared”, then start the docker container with the next command:

```
docker run \
--env="DISPLAY" \
-h "xlnx-docker" \
--env="XDG_SESSION_TYPE" \
--net=host \
--privileged \
--volume="$HOME/.Xauthority:/root/.Xauthority:rw" \
-v /tmp:/tmp \
-v /dev:/dev \
-v /sys:/sys \
-v /etc/vart.conf:/etc/vart.conf \
-v /lib/firmware/xilinx:/lib/firmware/xilinx \
-v /run:/run \
-v /home/ubuntu/BS_shared:/mnt/compartido:rw \
-it xilinx/smartcam:2022.1 bash
```

The command create a folder inside of it placed on “/mnt/compartido” this file is connected with the folder “BS_shared” outside of the container. Work with WinSCP, pointing to the BS_shared folder to easily upload and download files.

Run MIPI camera with the next command:

```
smartcam --mipi -W 1920 -H 1080 --target file
```

when you want to stop the record, press cntrl + c, and the move the file .h264 to the folder “compartido” with the next command.

```
mv out.h264 /mnt/compartido/
```

Using WinSCP move the file to the desktop of the Host computer, open a cmd and run the next to transform the file from .h264 to .mp4

```
cd C:\Users\USUARIO\Desktop

ffmpeg -i out.h264 -c:v copy out.mp4
```

4. Finally, you are able to combinane inputs and outputs, and also with different AI tasks, because the default task is the facedetection, but there are also 2 aditional tasks called SSD and refinedet. This may be selected adding to the command the next [facedetect|ssd|refinedet]

```
smartcam --usb 0 -W 640 -H 480 -r 30 --target rtsp --aitask ssd
```

Annex 3
Vitis AI- Installation

For the purpose of customizing AI models, it is necessary the installation of Vitis AI 2.5.0. This installation must be in a host computer, preferably with ubuntu 22.04, similar to the one used in the KRIA board, despite other versions are useful as well. For the right installation, the next web is used as reference: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai/Vitis-AI-Overview

1. Install Docker container on the fpga. Go to https://docs.docker.com/engine/install/ubuntu/ and install it following the step for "Install using the apt repository"

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}") stable"
| \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin

sudo docker run hello-world
```

2. Clone the Vitis AI repository to obtain the examples, reference code, and scripts:

```
git clone --recurse-submodules https://github.com/Xilinx/Vitis-AI
```

3. Pull the container of Vitis AI

```
docker pull xilinx/vitis-ai-cpu:2.5.0
```

4. Add the container to the group

```
sudo usermod -aG docker $USER
```

5. To run the cointaner type this command

```
cd Vitis-AI
sudo ./docker_run.sh xilinx/vitis-ai-cpu:2.5.0
```

For transferring files from the host root to the container, is just copy the files or folder to the Vitis-AI folder, and after the container is run, the files could be seen inside of it.

Annex 4
Yolo V3

- Download from https://pjreddie.com/darknet/yolo/ YOLOv3-416 files .cfg and .weights
- Install the repository https://github.com/SundanceMultiprocessorTechnology/keras-YOLOv3-model-set
- Install Miniconda3, to used the repository above, there is a requirement.txt file in order to get all the necessary files. Some of them works only on python 3.7. Name “mi_entorno” could be different, according to you.

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
source ~/.bashrc
conda create -n mi_entorno python=3.7
conda activate mi_entorno
```

- In the requirements.txt file, change “tensorflow-gpu” by “tensorflow==2.10”, then:

```
pip install -r /home/bsarabia/keras-YOLOv3-model-set/requirements.txt
```

- With the .cfg, .weights copied in the directory “keras-YOLOv3-model-set”, run the next code to convert from Darknet to Keras the pretrained model:

```
:~/keras-YOLOv3-model-set$
python3 tools/model_converter/convert.py yolov3.cfg yolov3.weights yolov3.h5
```

- For activate and des active the miconda environment use:

```
conda deactivate
conda activate mi_entorno
```

- Additionally, if it is needed to convert from keras to tensorflow to obtain .pb file use the next. But Vitis AI accept the .h5 file to quantize and compile the model.

```
python3 tools/model_converter/keras_to_tensorflow.py --input_model="yolov3.h5"
--output_model="yolov3.pb"
```

- The result is:


yolov3.h5

yolov3.pb

Annex 5
Yolo V3 – Quantization, Compilation

This part is based on: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai/TensorFlow-2.x-Version-vai_q_tensorflow2

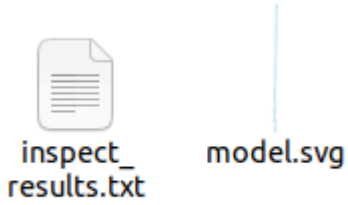
- It is recommended for this step to use a Virtual Machine with Ubuntu 22.04, and install Vitis AI docker container. After running a container, activate the Conda environment vitis-ai-tensorflow2 (Go to installation of Vitis-AI).

  ```
  cd Vitis-AI

  sudo ./docker_run.sh xilinx/vitis-ai-cpu:2.5.0

  conda activate vitis-ai-tensorflow2
  ```

- In directory Vitis-AI, create a folder, in my case “quantizeBS”, and then copy the .h5 file, to be reflected the file in the Vitis-AI container. For inspect the model, according to AMD on “Inspecting the Float Model” it is useful to use a piece of code to inspect it. It was developed a code based on this information (AI generated), the python script for inspecting is “insp.py”. Run that code:

  ```
  cd quantizeBS

  python3 insp.py yolov3.h5
  ```

  Two files are created “inspect_results.txt” and “model.svg”. This step is optional but useful to check the model of the DPU (DPUCZDX8G_ISA1_B3136=>0x101000016010406, for KRIA KV260).

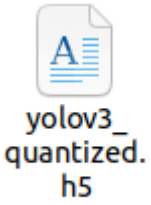



```
511 ================================================================================================================================
512 [SUMMARY INFO]:
513 - [Target Name]: DPUCZDX8G_ISA1_B3136
514 - [Target Type]: DPUCZDX8G
515 - [Total Layers]: 252
516 - [Layer Types]: InputLayer(1) Conv2D<linear>(75) BatchNormalization(72) LeakyReLU(72) ZeroPadding2D(5) Add(23) UpSampling2D(2)
    Concatenate(2)
517 - [Partition Results]: INPUT(1) DPU(251)
518
519   All layers are supported and successfully mapped to DPU.
520 ================================================================================================================================
521 [NOTES INFO]:
522
523   All layers are supported and successfully mapped to DPU.
524 ================================================================================================================================
```

- The next step is quantized the model. Which is the process to change the model to float point to int8 for deploy on the fpga board. This is based on “Running vai_q_tensorflow2”. The code was AI generated and cured by me, and the name of it is “quant.py”. It is necessary from 100 to 1000 coco images for this step. Run the next command for quantizing.

python3 quant.py

At the end the file “yolov3_quantized.h5” will be generated.



- Copy the arch.json file, which is explain later but have essentially the fingerprint of the DPU, to the folder quantizeBS. and run the command for compiling the model and obtain .xmodel file:

```
vai_c_tensorflow2   -m yolov3_quantized.h5   -a arch.json   -o compiled_model
-n yolov3_coco_416_tf2
```

- A compiled model .xmodel is created and store in the folder “compiled_model”. Copy “yolov3_coco_416_tf2.xmodel” for deploy in the FPGA.

Annex 6
KRIA KV260 - Deploying YoloV3

There are 2 approaches for deploying the YoloV3 model:

- Using the smart camera container, for this option is necessary to copy specific files on folders and run using commands, the main advantage for this is the option of record and export files.
- Using the smart camera container but with a custom file that run the inference, the advantage is less process and reduce latency. C++ files must be compiled previous the use on the workflow.

Smart Camera Container

1. Connect FPGA KV260 using miniUSB port for serial connection, ethernet port for communication with WinSCP, and a USB/MIPI camera.
2. Run the next code to disable the desktop environment in the FPGA

```
sudo xmutil      desktop_disable
sudo xmutil unloadapp
sudo xmutil loadapp kv260-smartcam
```

3. Run docker, the shared folder "BS_shared" on the host should contain the next files: aiinference.json, drawresult.json, preprocess.json, yolov3_coco_416_tf2.xmodel, yolov3_coco_416_tf2.prototxt, label.json. The places where every file should be move in are according the next tutorial: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/customize_ai_models.html

```
mkdir -p /opt/xilinx/kv260-smartcam/share/vvas/yolov3_coco_416_tf2

cp /mnt/compartido/aiinference.json /mnt/compartido/drawresult.json
/mnt/compartido/preprocess.json/opt/xilinx/kv260-
smartcam/share/vvas/yolov3_coco_416_tf2/

mkdir -p
/opt/xilinx/kv260smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/

cp /mnt/compartido/yolov3_coco_416_tf2.xmodel
/mnt/compartido/yolov3_coco_416_tf2.prototxt /mnt/compartido/label.json
/opt/xilinx/kv260-smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/
```

4. After this the custom model YoloV3 is ready to be used using command lines as in the "KRIA KV260 - Smartcamera Docker" section.

- USB cam | RSTP stream

```
smartcam --usb 0 -W 640 -H 480 -r 5 --target rtsp --aitask yolov3_coco_416_tf2
```

CMD host computer (Windows 10)

```
ffplay rtsp://192.168.137.100:554/test
```

- USB cam | File

  ```
  smartcam --usb 0 -W 640 -H 480 -r 5 --target file --aitask yolov3_coco_416_tf2
  ```

  CMD host computer (Windows 10)

  ```
  mv out.h264 /mnt/compartido/
  ```

  Move file to Windows desktop

  ```
  cd C:\Users\USUARIO\Desktop
  ffmpeg -i out.h264 -c:v copy out.mp4
  ```

- File | File (Input file in h.264 format)
  File .mp4 on windows desktop, then move to the shared folder in the FPGA

  ```
  ffmpeg -i "debrec.mp4" -c:v libx264 -preset slow -crf 23 -an "debrec.h264"
  ```

  On FPGA

  ```
  cd /mnt/compartido
  smartcam --file ./debrec.h264 -i h264 -W 640 -H 360 -r 25 --target file
  ```

  Move file to Windows desktop

  ```
  cd C:\Users\USUARIO\Desktop
  ffmpeg -i out.h264 -c:v copy out.mp4
  ```

Smart Camera Container with custom .cpp

This part is a compilation of the next works: https://www.sundance.com/yolov3-on-the-xilinx-kria-kv260/ https://www.paltek.co.jp/techblog/techinfo/230215_01 and the build.sh was found on: https://github.com/Xilinx/Vitis-AI/tree/2.5/examples/Vitis-AI-Library/samples/yolov3

.cpp for Images

1. The file for image on .cpp should be place in the shared folder, and also the build.sh, a test image (debrecen.jpg), yolov3_coco_416_tf2.prototxt, label.json, and yolov3_coco_416_tf2.xmodel. Then run the container as previous but, after that it is necessary some libraries for compile .cpp file: Do it whit the next commands:

   ```
   docker run \
   --env="DISPLAY" \
   -h "xlnx-docker" \
   --env="XDG_SESSION_TYPE" \
   --net=host \
   --privileged \
   --volume="$HOME/.Xauthority:/root/.Xauthority:rw" \
   -v /tmp:/tmp \
   ```

```
-v /dev:/dev \
-v /sys:/sys \
-v /etc/vart.conf:/etc/vart.conf \
-v /lib/firmware/xilinx:/lib/firmware/xilinx \
-v /run:/run \
-v /home/ubuntu/BS_shared:/mnt/compartido:rw \
-it xilinx/smartcam:2022.1 bash
```

2. Install the next libraries:

```
apt update
apt install -y libopencv-dev pkg-config
apt install -y build-essential
apt install -y libgoogle-glog-dev
apt install -y libprotobuf-dev protobuf-compiler
apt install -y libjson-c-dev
apt-get install -y libgtk-3-dev
```

3. After that in the docker, go to the /mnt/compartido directory and then run builder, this is needed only once, if the .cpp do not change, then you could skip this step:

```
cd /mnt/compartido
bash build.sh
```

4. Create “yolov3_coco_416_tf2” and move the files as next:

```
mkdir -p /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2

cp /mnt/compartido/yolov3_coco_416_tf2.xmodel \
   /mnt/compartido/yolov3_coco_416_tf2.prototxt \
   /mnt/compartido/label.json \
   /mnt/compartido/custom_yolov3_image \
   /mnt/compartido/test.png \
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/

mv /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2.
* \
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/label.json \
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2/
```

5. Run the inference for the image:

```
cd /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/
```

```
./custom_yolov3_image ./yolov3_coco_416_tf2 debrecen.jpg
```

6. Move the generated image with the detected objects and then you could see on your host computer:

```
cp debrecen.jpg_detected.jpg /mnt/compartido/
```

.cpp for Video/Camera

1. For display video or camera, it is needed to use of a screen through DP port. After that, send this command:

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

2. Copy and compile the .cpp for processing video. This is needed only once (Follow steps 1, 2 of the previous section).
3. Start the Smart camera container with the next code, in this case the display is activated:

```
export DISPLAY=:1
xhost +local:root

docker run \
--env="DISPLAY=$DISPLAY" \
-h "xlnx-docker" \
--env="XDG_SESSION_TYPE=$XDG_SESSION_TYPE" \
--net=host \
--privileged \
--volume="$HOME/.Xauthority:/root/.Xauthority:rw" \
-v /tmp/.X11-unix:/tmp/.X11-unix:rw \
-v /dev:/dev \
-v /sys:/sys \
-v /etc/vart.conf:/etc/vart.conf \
-v /lib/firmware/xilinx:/lib/firmware/xilinx \
-v /run:/run \
-v /home/ubuntu/BS_shared:/mnt/compartido:rw \
-it xilinx/smartcam:2022.1 bash
```

4. Create “yolov3_coco_416_tf2” and move the files as next, it is important to put in the shared folder a video called debrec.h264:

```
mkdir -p /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2

cp /mnt/compartido/yolov3_coco_416_tf2.xmodel \
   /mnt/compartido/yolov3_coco_416_tf2.prototxt \
   /mnt/compartido/label.json \
   /mnt/compartido/custom_yolov3_video \
   /mnt/compartido/debrec.h264 \
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/

mv /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2.
* \
```

```
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/label.json \
   /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/yolov3_coco_416_tf2/
```

5. Run the inference using a video in format .h264:

```
cd /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/

./custom_yolov3_video ./yolov3_coco_416_tf2 debrec.h264 640 360
```

6. Run the inference using a camera:

```
cd /opt/xilinx/kv260-
smartcam/share/vitis_ai_library/models/yolov3_coco_416_tf2/

./custom_yolov3_video ./yolov3_coco_416_tf2 /dev/video0 640 480
```

```
sudo shutdown -h now
```

Annex 7
DPU – Vivado

Based on: https://repositorio.usfq.edu.ec/bitstream/23000/13162/1/204102.pdf

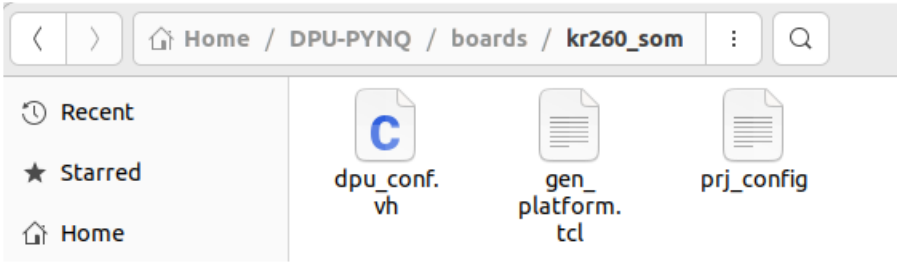
There are two main ways to integrate the DPU to a hardware platform that can be found on the Xilinx: The first, called Vivado TRD flow, and the second, Vitis TRD flow.

Vitis TDR flow: https://docs.amd.com/r/4.0-English/pg338-dpu/Customizing-and-Generating-the-Core-in-the-Vitis-IDE

1. The DPU Pynq GitHub repository provides tcl scripts to create Vivado base platforms for a variety of Xilinx boards and the necessary configuration files to instantiate the DPU using the Vitis TRD flow. The first step is to clone the Pynq DPU repository and clone. This step is made on a VM with Ubuntu and Vitis 2022.1 installed.

   ```
   git clone https://github.com/Xilinx/DPU-PYNQ.git -bdev_3.0.0
   cd ./DPU-PYNQ/
   git clone https://github.com/Xilinx/XilinxBoardStore -b 2022.1
   ```

   The boards folder contains all the supported platforms. In particular a folder kv260_som contains the 3 files needed to create the KV260's platform

2. The following commands create a folder, copy the gen platform.tcl file, source the Vivado tools and open the Vivado GUI:

   ```
   cd ./boards
   mkdir KV260_Custom_Platform

   cp ./kv260_som/gen_platform.tcl ./KV260_Custom_Platform
   cd ./KV260_Custom_Platform
   source /tools/Xilinx/Vivado/2022.1/settings64.sh
   vivado
   ```

3. To create the platform, source the gen platform file in the Vivado Tcl console.

   ```
   source gen_platform.tcl
   ```

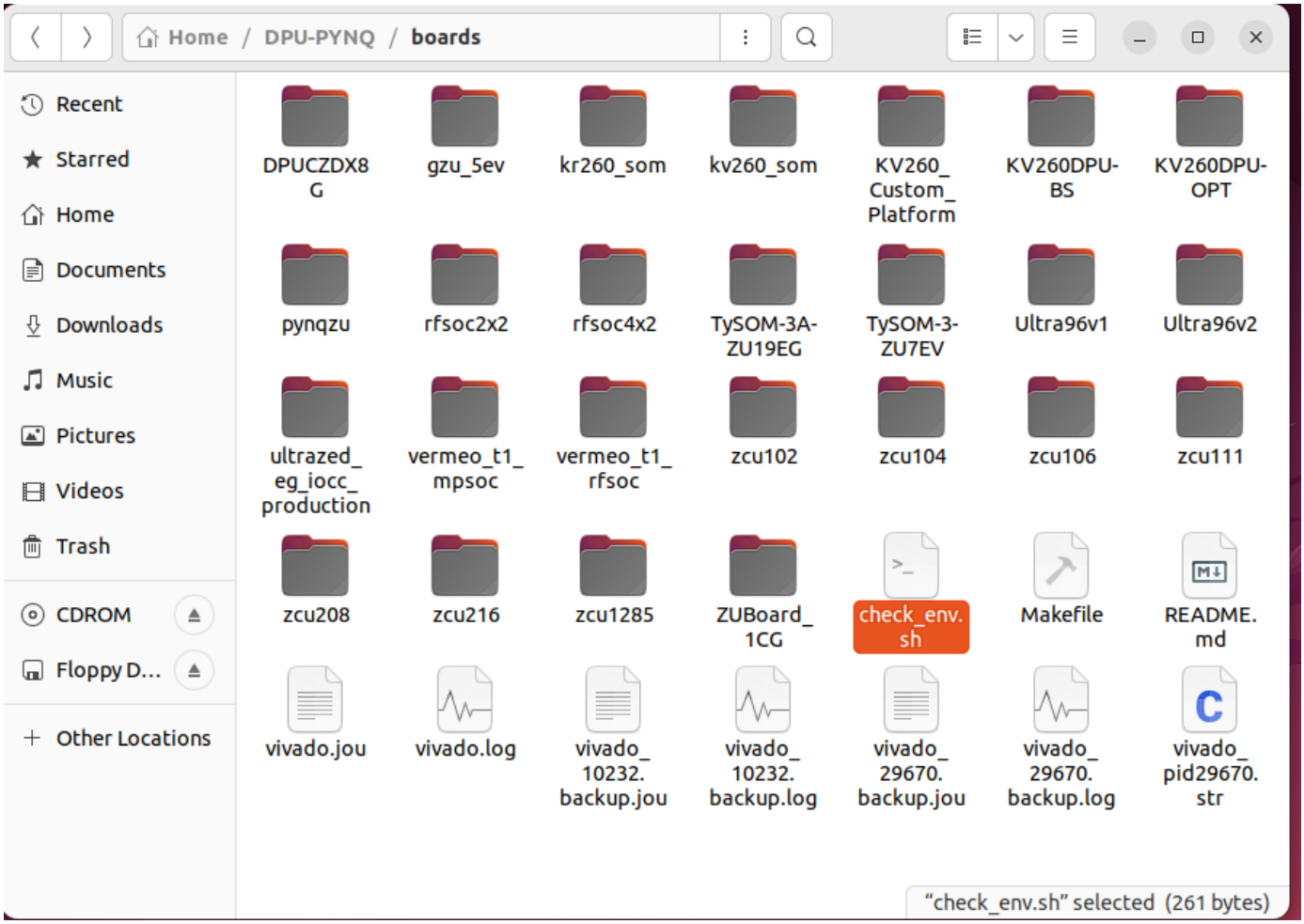
4. In the folder after the creation of the platform. Move the next files to a new folder to integrate the DPU. Copy the platform.xsa hardware file along with the dpu_conf.vh and prj_config files neede do create the Pynq overlay to another folder with the following commands.

```
cd ..
mkdir KV260DPU-BS
cp ./KV260_Custom_Platform/platform.xsa ./KV260DPU-BS
cp ./kv260_som/dpu_conf.vh ./KV260DPU-BS
cp ./kv260_som/prj_config ./KV260DPU-BS
```

5. In order to test different configuration of the dpu, is recommended to modify dpu_conf.vh, for this reason the next 2 configuration were implemented, the first one was native of the file that comes from “folder kv260_som”, the second configuration was implemented as follows:

```
42 `define B3136
53 `define URAM_ENABLE
54
55 //config URAM
56 `ifdef URAM_ENABLE
57     `define def_UBANK_IMG_N          16
58     `define def_UBANK_WGT_N          64
59     `define def_UBANK_BIAS           1
60 `elsif URAM_DISABLE
61     `define def_UBANK_IMG_N          0
62     `define def_UBANK_WGT_N          0
63     `define def_UBANK_BIAS           0
64 `endif
75 `define DRAM_ENABLE
76
77 //config DRAM
78 `ifdef DRAM_ENABLE
79     `define def_DBANK_IMG_N          8
80     `define def_DBANK_WGT_N          16
81     `define def_DBANK_BIAS           1
82 `elsif DRAM_DISABLE
83     `define def_DBANK_IMG_N          0
84     `define def_DBANK_WGT_N          0
85     `define def_DBANK_BIAS           0
86 `endif
97 `define RAM_USAGE_LOW
125 `define ALU_PARALLEL_8
136 `define CONV_RELU_LEAKYRELU_RELU6
147 `define ALU_RELU_LEAKYRELU_RELU6
159 `define DSP48_USAGE_HIGH
170 `define LOWPOWER_DISABLE
```

6. To avoid the XRT installation, deactivate the code that checks for the sourcing of these tools, by opening the *check_env.sh* script, and commenting lines 10 to 13.




```
#!/bin/bash

set -e

if [[ -z $(vitis -version | fgrep 2022.1) ]]; then
        echo "Error: Please source Vitis 2022.1 settings."
        exit 1
fi

#if [[ -z ${XILINX_XRT} ]]; then
#        echo "Error: Please source XRT 2021.1 settings."
#        exit 1
#fi
```

7. After that in the cmd in boards folder directory, type the next command to integrate the DPU in the platform.

```
source /tools/Xilinx/Vitis/2022.1/settings64.sh
```

For the regular DPU

```
make BOARD=KV260DPU-BS VITIS_PLATFORM=/home/bsarabia/DPU-PYNQ/boards/KV260DPU-BS/platform.xsa
```

For the modified parameters DPU after create folder KV260DPU-OPT and copy required files

```
make BOARD=KV260DPU-OPT VITIS_PLATFORM=/home/bsarabia/DPU-PYNQ/boards/KV260DPU-OPT/platform.xsa
```

8. At the end you will have a folder with the next files: