UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Despliegue de una Red Neuronal Profunda en la Placa de desarrollo KRIA KV260 para detección e identificación de señales de tránsito

Xavier Eduardo Casanova Pabón

Ingeniería en Electrónica y Automatización

Trabajo de fin de carrera presentado como requisito
para la obtención del título de
Ingeniero en Electrónica y Automatización

Quito, 22 de diciembre de 2024

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN
DE TRABAJO DE FIN DE CARRERA

Despliegue de una Red Neuronal Profunda en la Placa de desarrollo KRIA KV260 para detección e identificación de señales de tránsito

Xavier Eduardo Casanova Pabón

Nombre del profesor, Título académico **Alberto Sánchez, PhD.**

Quito, 22 de diciembre de 2024

© DERECHOS DE AUTOR

Nombres y apellidos: Xavier Eduardo Casanova Pabón

Código: 00204102

Cédula de identidad: 1721623070

Lugar y fecha: Quito, 22 de diciembre de 2024

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

RESUMEN

Este trabajo busca ser una guía en el uso de las herramientas Xilinx para el desarrollo de una aplicación de ***Inteligencia Artificial*** (IA) de visión artificial. Se desarrolla el ***hardware*** necesario para cargar una red neuronal al ***FPGA*** de la placa de desarrollo KRIA KV-260, centrándose en el soft-IP de Xilins ***"DPU"*** (Deep-Learning Processing Unit) en ***Vivado*** y ***Vitis***. La API de ***Keras*** se utiliza para entrenar una red neuronal profunda YOLOv3 para el problema de detección e identificación de señales de tráfico de vehículos autónomos. Luego, ***Vitis-AI*** se utiliza para cuantificar y compilar la red para integrarla en el DPU instanciado en la FPGA. Finalmente, se desarrolla una aplicación a nivel de Sistema Operativo utilizando el *framework* ***"Pynq"*** (Python productivity for Zynq), para realizar la inferencia y transmitir video en tiempo real.

Palabras clave: *Inteligencia Artificial, FPGA, KV-260, Vivado, Vitis, Keras, YOLOv3, Vitis-AI, Pynq, Inferencia, Plataforma.*

ABSTRACT

This article seeks to be a guide in the use of the Xilinx tools for the development of an ***Artificial Intelligence*** (AI) application for machine vision. The ***hardware*** necessary to load a neural network to the FPGA of the KRIA KV-260 development board is developed, focusing on the soft-IP the ***DPU*** (Deep-Learning Processing Unit) in ***Vivado*** and ***Vitis***. The ***Keras*** API is used to train a YOLOv3 deep neural network focused on the autonomous vehicle road sign detection and identification problem. ***Vitis-AI*** is then used to quantize and compile the network to integrate it into the DPU instantiated on the FPGA. Finally, an application is developed at an Operating System level using the ***"Pynq"*** (Python productivity for Zynq) framework, to perform the inference and transmit video in real time.

Key words: Artificial Intelligence, FPGA, KV-260, RTL, Vivado, Vitis, Keras, YOLOv3, Vitis-AI, Pynq, Inference, Platform.

TABLA DE CONTENIDO

INTRODUCCIÓN

Un FPGA (Field Programmable Gate Array) es un dispositivo de silicio que permite al diseñador sintetizar funciones lógicas combinacionales y secuenciales, facilitando así el desarrollo de hardware fácilmente reconfigurable.

Altera, trajo el primer FPGA a la industria en 1984 (R. Wilson, 2015), en 1985, Xilinx llevó al mercado el primer FPGA accesible (P. Alfke, et al., 2011) y recientemente, estas empresas han optado por fabricar SoC's (System on Chips) que disponen de un sistema de procesamiento con CPU de varios núcleos, periféricos de entrada/salida (GPIO), dispositivos de comunicación, memoria, etc., además de un área de lógica programable basada en FPGAs.

Estos chips han ganado popularidad en diferentes áreas de la industria, donde la inferencia ML (Machine Learning) ha sido un gran foco de atención en los últimos años, debido a la flexibilidad que brindan los FPGAs para el cálculo rápido de álgebra lineal, operaciones de tensores requeridas en ML y su reconfigurabilidad.

Xilinx ha propuesto soluciones para la inferencia profunda de redes neuronales, con su soft-IP, el DPU (Deep Learning Processing Unit), que puede instanciarse en la lógica programable de sus chips (Xilinx, 2020) y ha proporcionado herramientas de inferencia como Vitis-AI, que permite cuantizar, podar, optimizar y generar la descripción del código binario de un modelo ML que luego se puede cargar en el DPU (Xilinx, 2022), de esta manera es posible reducir la latencia y aprovechar los flujos de trabajo con hardware personalizado para una amplia gama de aplicaciones.

En los últimos años, Xilinx ha lanzado dispositivos SOM al mercado. Un SOM (System on Module) se compone de una pieza mínima de hardware (una placa de circuito impreso del tamaño de una tarjeta de crédito) (Xilinx, 2023), basada en un SoC que tiene las conexiones

absolutamente necesarias, como IO y memoria, para integrar el chip en un sistema de hardware personalizado de forma fácil y segura. Uno de los SOM disponibles es el Kria K26 SOM.

Xilinx también lanzó placas de desarrollo para crear prototipos de soluciones basadas en SOM y probarlas sin tener que construir el hardware final que se implementará. La primera placa de desarrollo diseñada para soluciones SOM enfocadas en Visión Artificial fue la placa de desarrollo Kria KV260 (Xilinx, 2023).

La KV260 tiene la solución Kria SOM K26 que contiene en su núcleo un Zynq Ultrascale + MPSoC que, entre sus características más importantes, tiene soporte para múltiples cámaras (hasta 8 interfaces), 3 interfaces de sensores MIPI, cámaras USB, HDMI y Display Port. salidas, Ethernet de 1GB, USB 3.0 y 2.0, 1 puerto PMOD entre otros, lo que lo hace ideal para crear prototipos de soluciones de visión basadas en IA. (Xilinx, 2023).

El objetivo de este trabajo es documentar el flujo de desarrollo básico para la creación de la plataforma de hardware, la instanciación del DPU, el proceso de cuantización y compilación de una red neuronal previamente entrenada y la creación de una aplicación de software que integre todos estos componentes y permita la transmisión de video con inferencia en tiempo real.

Para lograr estos objetivos se utilizará el entorno Pynq compatible con Xilinx. Según Crockett, L., et al., (2019) el nombre “PYNQ” se deriva de “Python productivity for Zynq” y, como su nombre indica, utiliza el lenguaje de programación Python para simplificar el proceso de creación de aplicaciones con dispositivos Zynq”. Esto incluye la manipulación de componentes de software y hardware del IC.

Pynq proporciona acceso a varias API de Python para desarrollar aplicaciones de software y configurar PL a través de “overlays” de Python utilizando Jupyter Notebooks. De esta manera, se puede configurar el FPGA de la placa incluso en tiempo de ejecución, controlar los núcleos

IP instanciados, por tanto cargar el código de la red neuronal en el DPU, y ejecutar una aplicación de software a nivel de sistema operativo para ver los resultados de la inferencia en tiempo real.

Deep Neural Network Deployment in a KRIA KV260 development board for Road sign detection and Identification*

*A practical guide to get started in the use of DPU and Vitis AI

Casanova, Xavier
IEE Student
Universidad San Francisco de Quito USFQ
Quito, Ecuador
xecasanova@estud.usfq.edu.ec

Sánchez, Alberto
IEE Professor
Universidad San Francisco de Quito USFQ
Quito, Ecuador
asanchez@usfq.edu.ec

***Abstract*—This article seeks to be a guide in the use of the Xilinx tools for the development of an *Artificial Intelligence* (AI) application for machine vision. The hardware necessary to load a neural network to the *FPGA* of the KRIA *KV-260* development board is developed, focusing on the soft-IP the *DPU* (Deep-Learning Processing Unit) in *RTL* in *Vivado* and *Vitis*. The *Keras* API is used to train a *YOLOv3* deep neural network focused on the autonomous vehicle road sign detection and identification problem. *Vitis-AI* is then used to quantize and compile the network to integrate it into the DPU instantiated on the FPGA. Finally, an application is developed at an Operating System level using the *Pynq* (Python productivity for Zynq) framework, to perform the *inference* and transmit video in real time.**

***Index Terms*—Artificial Intelligence, FPGA, KV-260, RTL, Vivado, Vitis, Keras, YOLOv3, Vitis-AI, Pynq, inference, platform.**

I. Introduction

A FPGA (Field Programmable Gate Array) is a silicon device that allows the designer to synthesize combinational and sequential logic functions, thus facilitating the development of easily reconfigurable hardware.

Altera, brought the first FPGA to the industry in 1984 [19], in 1985, Xilinx brought the first accessible FPGA to the market [1] and recently, these companies have opted to manufacture SoC's (System on Chips) that have a processing system with CPU cores, input/output peripherals (GPIO), communication devices, memory, etc. plus a programmable logic area based on FPGA's.

These chips have gained popularity in different areas of the industry, where ML (Machine Learning) inference has been a great focus of attention in recent years, due to the flexibility FPGA's provide for reconfiguration, thus enabling quick computation of linear algebra and tensor operations required in ML.

Xilinx has proposed solutions for deep neural-network inference, with its soft-IP, the DPU (Deep-Learning Processing Unit), which can be instantiated in the programmable logic of its chips [24] and has provided inference tools such as Vitis-AI, which allows quantizing, pruning and generating the binary code description of a ML model that can then be loaded into the DPU [29], This way it is possible to reduce latency, and take advantage of workflows with customized hardware for a diverse range of applications.

In recent years, Xilinx had launched the SOM to the market. A SOM (System on Module) comprises of a minimal piece of hardware (a printed circuit board with the size of a credit card) [35], based on an SoC that has the absolute necessary connections such as IO and memory, to integrate the chip into a custom hardware system easily and safely. One of the SOM's available is the Kria K26 SOM.

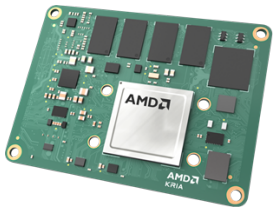

Fig. 1. Xilinx Kria SOM K26. [35]

Xilinx also released development boards to prototype solutions based on SOMs and test them without having to build the final hardware to be deployed. The first development board designed for SOM solutions focused on Machine Vision was the Kria KV260 development board [35].

A KV260 has the Kria SOM K26 solution that contains at its core a Zynq Ultrascale + MPSoC IC that among its most important features, has support for multiple cameras (up to 8 interfaces), 3 MIPI sensor interfaces, USB cameras, HDMI and Display Port outputs, 1GB Ethernet, USB 3.0 and 2.0, 1 PMOD port among others, which makes it ideal for prototyping AI based Vision solutions. [35].

The goal of this work is to document the basic development flow for the creation of the hardware platform, the instantiation of the DPU, how a pre-trained network is quantized, compiled

Fig. 2. Kria KV260 Development Board. [35]

and a software application that integrates all these components and allows the transmission of the video results in real time.

The Pynq environment supported by Xilinx will be used to achieve these goals. According to Crockett, L., et. al., the name "PYNQ" is derived from "Python productivity for Zynq"; and, as the name suggests, it uses the Python programming language to simplify the process of creating applications with Zynq devices" [6]. This includes the manipulation of software and hardware components of the IC.

Pynq provides access to various Python API's to develop software applications and configure the PL through Python overlays using Jupyter Notebooks. This way, one can configure the board's FPGA at run time, control the instantiated IP cores, and therefore load the neural network code into the DPU, and run a software application from a notebook to see the inference results in real time.

II. Preparing the Operating System

A. OS Selection

All the Xilinx Zynq Ultrascale + MPSoC based boards support a Linux OS that can be customize through the Xilinx Petalinux development tools that are based on the Yocto Project. However Xilinx has agreements with canonical to support the Ubuntu OS in some of its development boards, and one of them is the Kria KV260. Because Ubuntu is a stable operating system, has firmware support for multiple USB devices, and allows access to a desktop, this OS will be chosen.

All the steps regarding the setup of the SD card and the board's boot process can be found in the official Xilinx Website [36]. The first step is to go to the official Canonical website and download the Kria Ubuntu image by choosing the Kria K26 SOM tab. Load the image to the SD card with a program like Balena Etcher as explained on the startup guide.

B. Hardware Setup

Once the OS image is loaded, all the board's connections can be made by first inserting the SD card, then plugging the USB/JTAG, Ethernet and HDMI cables, then connecting a USB keyboard, mouse and webcam, and finally plugging the power supply as shown in figures 3 [36] and 4. The resulting OS running on the board is shown in figure 5.

There are some example applications that Xilinx provides to download and test in their App Store [37]. The steps to

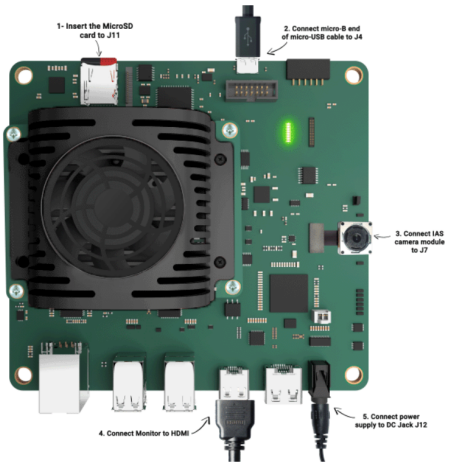

Fig. 3. Hardware Setup for the KV260 from the official Xilinx Web Page. [36]

Fig. 4. Hardware Setup for the KV260.

Fig. 5. Ubuntu OS running on the KV260.

test each individual application are available at the official Xilinx GitHub web page [39]. One example is the Smart Camera Application, and the full tutorial can be found on the official Xilinx GitHub page. The first step is to download the application firmware, by entering the following commands.

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

Install docker and add the user to the docker group to avoid

using *sudo* every time one uses docker commands.

```
$ sudo apt install docker
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
$ sudo reboot
```

After rebooting, one can try to run the hello-world docker without *sudo*.

```
$ docker run hello-world
```

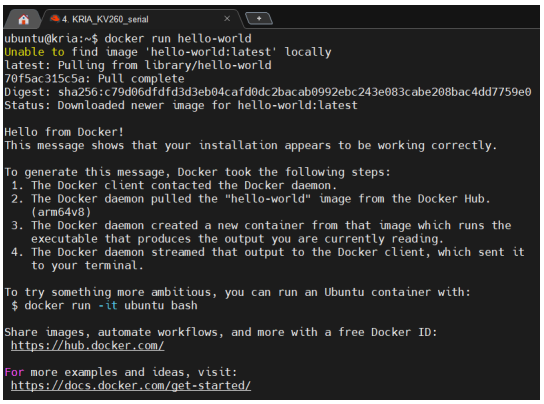

Fig. 6. Hello-World docker running on the KV260 board.

After having successfully configured docker, pull the Xilinx Smartcam docker to run the example application.

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

Because the video will be transmitted through the HDMI/Display port, the Ubuntu desktop needs to be disabled, therefore to run the app it is necessary to establish communication with the board through its UART/USB port. Start by disabling the desktop via the platform management utility *xmutil* and load the Smartcam firmware as an accelerated application.

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

Fig. 7. Loading the Smart Camera Firmware.

It is possible to choose to run the Smartcam docker with different parameters, for example the input source can be a video file, a camera plugged to the MIPI port, or a USB camera, and the output source can be the HDMI/display port, a video stream transmitted through RTSP via Ethernet or a video file saved on the OS. It is also possible to change the resolution of the input and output to various image sizes. All these parameters are explained in the official Smartcam Xilinx GitHub website [38]. In this example, the input is a LogiTech USB camera with resolution of 1280x720 at 60fps and the output is the HDMI/Display Port.

```
$ 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
root@xlnx-docker:/# smartcam --usb 0
  -W 1280 -H 720 -r 60 --target dp
```

Fig. 8. Smart Camera Application Docker.

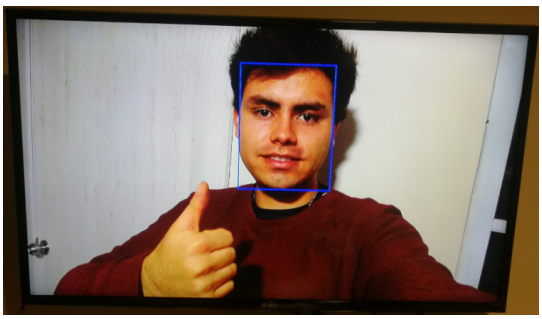

Fig. 9. Smart Camera Application Results.

There can be potential issues when running the app, like failing to recognize the DPU of the firmware. To solve these

issues, refer to the Known Issues section of the GitHub [38], where all these problems are well documented.

This example shows a final implementation of the hardware needed to run deep-learning inference on the KV260 as an accelerated application and the software application to capture and output video as a docker application. Part of the acceleration process relies on the function acceleration for the pre process of the video before passing through the DPU (ML inference) and also on the video streaming pipeline that can be seen on figure 10.

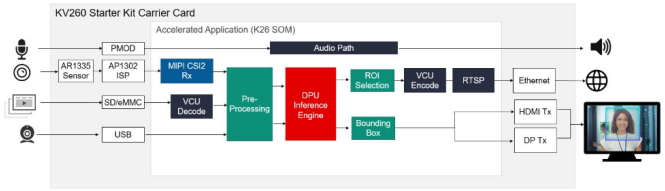

Fig. 10. Video Streaming pipeline of the Smartcam Application. [38]

As a first approach, we show the inference part of the whole pipeline, namely, the implementation of the DPU and the deployment of a deep-neural network on it, so the acceleration of the pre processing functions plus the pipeline architecture will the left out as part of a future work.

Pressing ***ctrl+c*** to stop the application, and type "exit" to exit the docker image, finally release the Smartcam firmware with the platform management utility.

```
root@xlnx-docker:/# ^C
root@xlnx-docker:/# exit
$ sudo xmutil unloadapp
$ sudo xmutil loadapp k26-starter-kits
$ sudo xmutil listapps
$ sudo xmutil desktop_enable
```

III. Installing Pynq

After having the OS running successfully on the board, and having tested some of its capabilities, the next step is to install the Pynq framework and have it running on the KV260. The full steps to install and use Pynq on the KV260 can be found on the official Kria Pynq GitHub repository [32]. There is also a wonderful blog of Whitney Knitter on hackster.io about it [12]. First, it is necessary to clone the Kria Pynq repository to the board, and it is also advisable to upgrade and update the system before it.

```
$ sudo apt update
$ sudo apt upgrade
$ git clone https://github.com/Xilinx
  /Kria-PYNQ.git
$ cd Kria-PYNQ/
Kria-PYNQ$ sudo bash install.sh -b KV260
```

After running the install.sh script, we can now open the Pynq Jupyter environment. Using a web browser of a computer connected to the same network as the KV260, one can type ***"kria:9090/"*** for a Jupyter Notebook and ***"kria:9090/lab"*** for a Jupyter Lab interface. The default password is ***"xilinx"***.

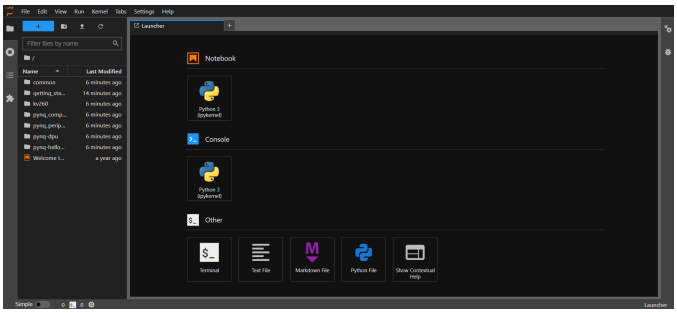

Fig. 11. Pynq Jupyter Lab environment.

The example ***dpu_yolov3*** located on the ***"pynq-dpu"*** folder shows the inference process for a custom deep-learning object detection model, the result is shown in figure 12, we will use this example as the base for the application developed later.

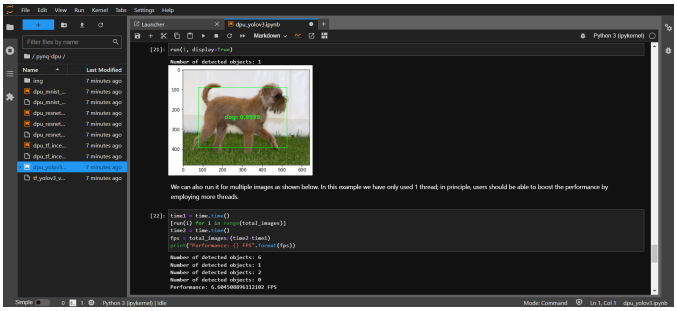

Fig. 12. Pynq DPU example for a custom YoloV3 network.

The hardware platform is configured through a Pynq DPU overlay and needs 3 files with the same name but different extensions to work:

- **dpu.bit** : The bit file is the bitstream that contains the configuration information for the FPGA of the base platform [33]. This file is obtained with Vivado.
- **dpu.hwh** : The hardware hand-off contains information about the hardware design, including the hardware interfaces, addresses, and other relevant details. It is needed for applications that require software and hardware to work together [17]. This file is obtained with Vivado.
- **dpu.xclbin** : The xclbin file is a compiled Binary File containing both bitstream and additional runtime information for use in heterogeneous computing environments [27]. This file is obtained with Vitis.

Additionally, a file "**network.xmodel**" is needed. This file contains the description of the neural network as a machine language code that the DPU can interpret. This file is obtained with Vitis-AI [29].

IV. Creating the Hardware Platform

For the rest of the work, a Linux machine will be used with Ubuntu as the host OS with the 2022.1 version of the Xilinx tools installed. Whitney Knitter has a tutorial on how to install this version of the tools on an Ubuntu machine on her hackster.io blog [13]. For this project only the Xilinx Unified Installer is needed, the Petalinux Tools will be left as optional.

The core component is the DPU. There are a variety of versions provided by Xilinx for each board. The IP compatible with Zynq Ultrascale + MPSoC is the DPUCZDX8G which also has different versions depending on the Xilinx Tools used. More information on version compatibility can be found in the Xilinx GitHub web page [40].

Zynq™ Ultrascale+™

Vitis AI Release Version	DPUCZDX8G IP Version	Software Tools Version	Linux Kernel Tested
v3.5	4.1 (not updated*)	Vivado / Vitis / PetaLinux 2023.1	6.1
v3.0	4.1	Vivado / Vitis / PetaLinux 2022.2	5.15
v2.5	4.0	Vivado / Vitis / PetaLinux 2022.1	5.15
v2.0	3.4	Vivado / Vitis / PetaLinux 2021.2	5.10
v1.4	3.3	Vivado / Vitis / PetaLinux 2021.1	5.10
v1.3	3.3	Vivado / Vitis / PetaLinux 2020.2	5.4
v1.2	3.2	Vivado / Vitis / PetaLinux 2020.1	5.4
v1.1	3.2	Vivado / Vitis / PetaLinux 2019.2	4.19
v1.0	3.1	Vivado / Vitis / PetaLinux 2019.1	4.19
N/A (DNNDK)	3.0	Vivado / Vitis / PetaLinux 2019.1	4.19
N/A (DNNDK)	2.0	Vivado / Vitis / PetaLinux 2018.2	4.14
First Release (DNNDK)	1.0	Vivado / Vitis / PetaLinux 2018.1	4.14

Fig. 13. DPUCZDX8G version compatibility with the Xilinx Tools. [40]

The compatible DPU for the Vivado/Vitis 2022.1 tools is the DPUCZDX8G 4.0 which can be seen in figure 14. This version of the IP can be downloaded here. The DPU needs 3 clocking sources, see figure 15, one to drive the register configuration ***"s_axi_clk"***, one to drive the data controller ***"m_axi_dpu_aclk"*** and another one to drive the calculation unit ***"_dpu_2c_aclk"*** that must be twice as faster as the data controller clock signal. Additionally, synchronous reset signals for each clock signal are required [28].

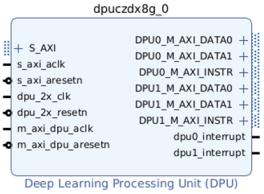

Fig. 14. Zynq Ultrascale compatible DPU, DPUCZDX8G version 4.0 [28].

There are two main ways to integrate the DPU to a hardware platform that can be found on the Xilinx pg338 product guide [28]. The first, called Vivado TRD flow, and the second, Vitis TRD flow.

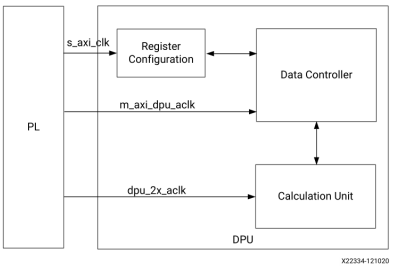

Fig. 15. DPUCZDX8G Clocking Signals retreived from the UG338. [28]

A. Vivado TRD flow

This is the standard hardware development flow based entirely on Vivado, where all the hardware components are instantiated manually to then create the bitstream and integrate the hardware to a custom Linux OS through the Petalinux tools or create a BOOT.bin file for a micro SD to configure the FPGA in a bare-metal fashion. The overview of this flow is described on the pg388 document on the Development Flow section. The example is under the downloaded DPU files for the ZCU 102 board. To adapt this example to the KV260 board, there is a tutorial from Shreyas N R on his hackster.io blog [20].

The platform created for the KV260 following this design flow with an additional AXI GPIO IP is shown in figure 16.

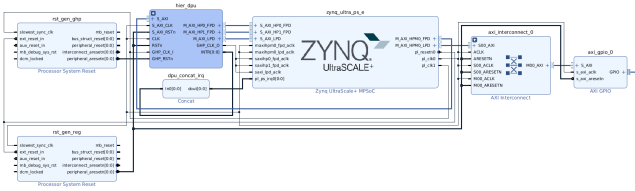

Fig. 16. Vivado TRD design Flow for the KV260 board.

B. Vitis TRD flow

The instantiation of the DPU to a base hardware platform as an accelerated application with the Vitis tools is called the Vitis TRD flow, detail explanation of this flow can be found in the pg388 document on the "Customizing and Generating the Core in the Vitis IDE" subsection. On the other hand, Pynq only supports the Vitis TRD flow to create a DPU class overlay, so in order to use Pynq, this flow must be followed.

Figure 17 shows an overview of this flow, where the customized DPU IP is passed to the Vitis v++ linker, along with a base hardware platform, as a .xo Xilinx object to be linked and compiled to a full accelerated hardware platform resulting in an xclbin hardware description file.

To do this, first create a base platform in Vivado containing an instance of the Zynq Ultrascale + MPSoC PS (Processing System), the clocking and reset signals, and any desired hardware IP and export the platform as an extensible platform with .xsa extension. After this an instance of the DPU is

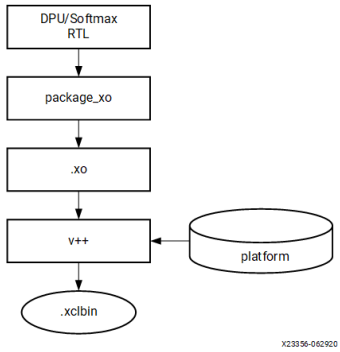

Fig. 17. DPU Vitis TRD design Flow [28].

inserted using the v++ linker. All the files needed are in the DPU downloaded folder, and there is an example for the ZCU 102 board. To create the accelerated platform for the KV260 board, the DPU configuration files plus the platform must be manually modified.

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 the Xilinx Board Store repository inside it, using the following commands.

```
$ git clone https://github.com/Xilinx
  /DPU-PYNQ.git -b dev_3.0.0
$ cd ./DPU-PYNQ/
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, figure 18 shows the same files for the KR260 board. The files are:

- **dpu_config.vh:** is a Verilog header file to customize the DPU parameters.
- **prj_config:** is the file with the project configuration information for the final accelerated application.
- **gen_platform.tcl:** this file contains the command line tcl code to create the base extensible platform project in Vivado needed for the Vitis v++ linker. This file will be changed for a .xsa platform file on the final folder.

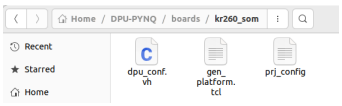

Fig. 18. Platform files of the Pynq DPU repository.

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

```
DPU-PYNQ$ cd ./boards
/boards$ mkdir /KV260_GPIO_Custom_Platform
/boards$ cp ./kv260_som/gen_platform.tcl
  ./KV260_GPIO_Custom_Platform
/boards$ cd ./KV260_GPIO_Custom_Platform
/KV260_GPIO_Custom_Platform$ source
  /tools/Xilinx/Vivado/2022.1/settings64.sh
/KV260_GPIO_Custom_Platform$ vivado
```

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

```
source gen_platform.tcl
```

This will create the base platform for the KV260 board with a block diagram as shown in figure 19.

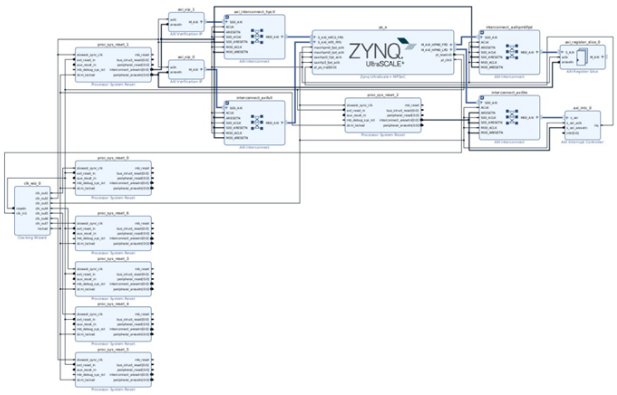

Fig. 19. Base platform for the KV260.

Using this platform, it is possible to instantiate any custom hardware module. For example, to control some LEDs, an AXI GPIO IP is instantiated and connected to the unused "AXI HPM1 FPD" Master AXI port of the PS as shown in figure 20.

To integrate the new block design and map it to the PMOD port of the board, a custom top module was created with the auto-generated code from the wrapper of the block diagram, the steps to follow are shown in the figures 21 to 24.

Finally, synthesize, run place & route, generate the bitstream and export the design as a platform in .xsa format with bitstream. Copy the ***platform.xsa*** hardware file along with the ***dpu_cong.vh*** and ***prj_conf*** files needed to create the Pynq overlay to another folder with the following commands.

```
/KV260_GPIO_Custom_Platform$ cd ..
/boards$ mkdir /KV260DPU-GPIO
/boards$ cp ./KV260_GPIO_Custom_Platform
  /platform_project/platform.xsa
  ./KV260DPU-GPIO
/boards$ cp ./kv260_som/dpu_cong.vh
  ./KV260DPU-GPIO
/boards$ cp ./kv260_som/prj_conf
  ./KV260DPU-GPIO
```

The resulting folder is shown in figure 25. To instantiate the DPU on the base platform we need the v++ vitis linker. To do this the Pynq repository provides a make-file for compilation, and the Vitis and XRT tools need to be sourced. The installation of XRT is a separate process and is only

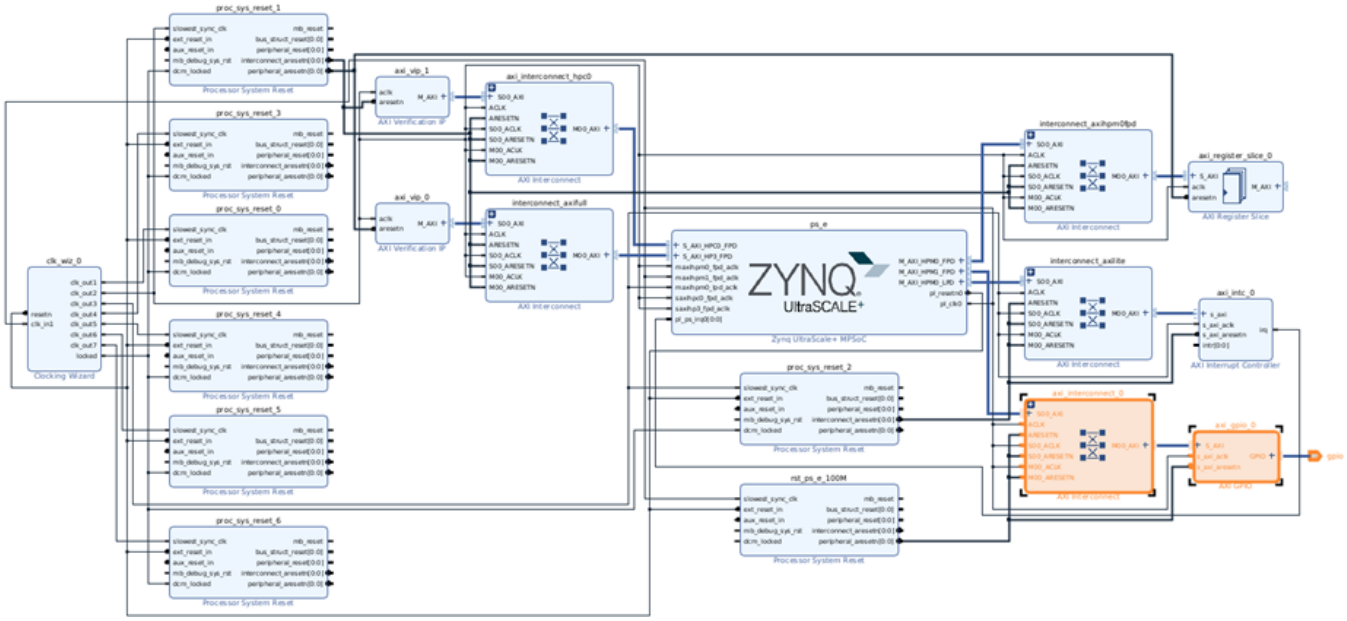

Fig. 20. Base platform for the KV260 with an AXI GPIO.

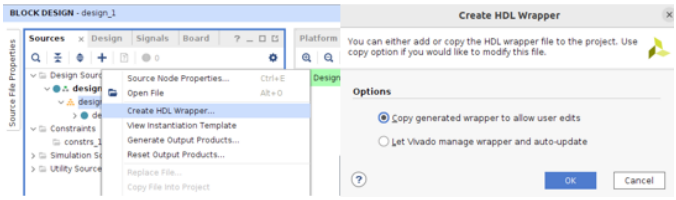

Fig. 21. Step 1: generate the wrapper, and copy the generated code.

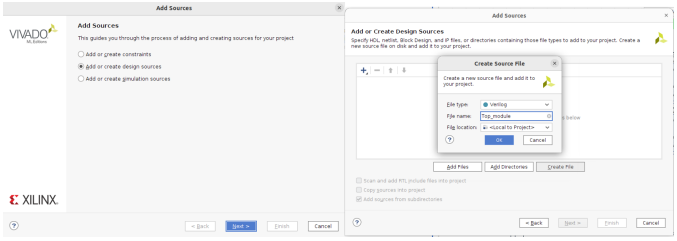

Fig. 22. Step 2: create the Top module.

used to create and test the software around an accelerated application, but in this case, all the software will be made using Python through Pynq.

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 as shown in Figure 26.

```
/boards$ gedit ./check_env.sh
```

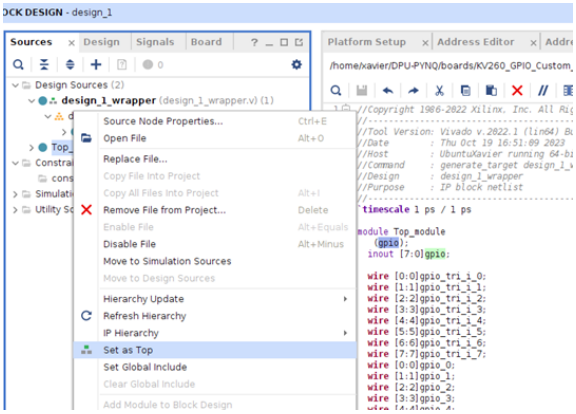

Fig. 23. Step 3: define the name of the IO ports and set the Top-Module.

```
1  # (C) Copyright 2020 - 2021 Xilinx, Inc.
2  # SPDX-License-Identifier: Apache-2.0
3
4  set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
5
6  #Digilent PMOD pins
7  set_property PACKAGE_PIN H12 [get_ports gpio[0]] ;# PMOD pin 1 - som240_1_a17
8  set_property PACKAGE_PIN B10 [get_ports gpio[1]] ;# PMOD pin 2 - som240_1_b20
9  set_property PACKAGE_PIN E10 [get_ports gpio[2]] ;# PMOD pin 3 - som240_1_d20
10 set_property PACKAGE_PIN E12 [get_ports gpio[3]] ;# PMOD pin 4 - som240_1_b21
11 set_property PACKAGE_PIN D10 [get_ports gpio[4]] ;# PMOD pin 5 - som240_1_d21
12 set_property PACKAGE_PIN D11 [get_ports gpio[5]] ;# PMOD pin 6 - som240_1_b22
13 set_property PACKAGE_PIN C11 [get_ports gpio[6]] ;# PMOD pin 7 - som240_1_d22
14 set_property PACKAGE_PIN B11 [get_ports gpio[7]] ;# PMOD pin 8 - som240_1_c22
15
16 set_property IOSTANDARD LVCMOS33 [get_ports gpio*];
17 set_property SLEW SLOW [get_ports gpio*];
18 set_property DRIVE 4 [get_ports gpio*];
```

Fig. 24. Step 4: Set the constraints for the PMOD port.

Save and Close the script. Then source the Vitis tools and run the makefile script to integrate the custom base platform with the DPU IP and produce the three files required by the Pynq DPU overlay, as shown in Figure 27.

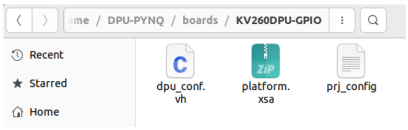

Fig. 25. Files needed to instantiate the DPU with the Vitis TRD flow.

Fig. 26. Deactivation of the XRT checking mechanism.

```
/boards$ source
  /tools/Xilinx/Vitis/2022.1/settings64.sh
/boards$ make BOARD=KV260DPU-GPIO
   VITIS_PLATFORM=
   ./KV260DPU-GPIO/platform.xsa
```

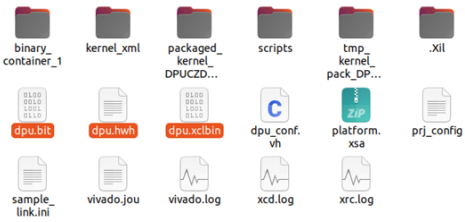

Fig. 27. Overlay Files created on the folder.

The DPU integration creates a Vivado project in the background which can be found at:

`/DPU-PYNQ/boards/KV260DPU-GPIO/binary_container_1/link/vivado/vpl/prj/prj.xpr`

This project contains a block diagram with the DPU and the AXI GPIO IP instantiated as shown in Figure 31.

Copying the three files to the Jupyter Lab environment of the KV260, allows to test if the platform is working. To develop the application, the Pynq YoloV3 example notebook will be used, so create a copy of the notebook and replace the overlay to point to the bit file created.

```
from pynq_dpu import DpuOverlay
# Configure FPGA
overlay = DpuOverlay("./dpu_gpio.bit")
overlay?
```

Figure 28 shows that the DPU and the AXI GPIO are present and working on the FPGA of the board as result of the test.

To test the GPIO, turn on some LEDs with the AxiGPIO library.

```
from pynq.lib import AxiGPIO
from functools import reduce
```

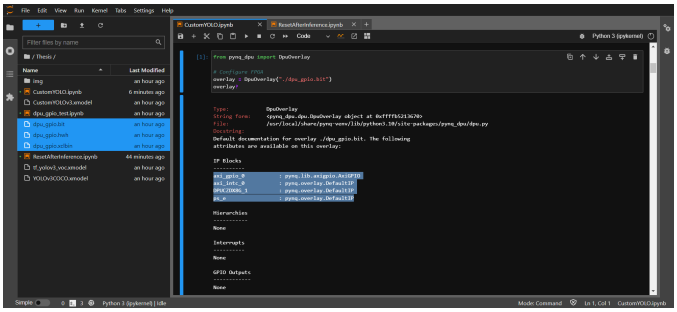

Fig. 28. Custom DPU overlay working on Pynq.

```
# Refresh the state of the GPIO
leds = overlay.axi_gpio_0[0:8]
leds.off()
led_ip = overlay.ip_dict['axi_gpio_0']
leds = AxiGPIO(led_ip).channel1
# First Test
mask = 0xff
leds.write(0xa4, mask)
```

Figures 29 and 30 show that the PMOD responds correctly to the overlay, by flashing some LEDs.

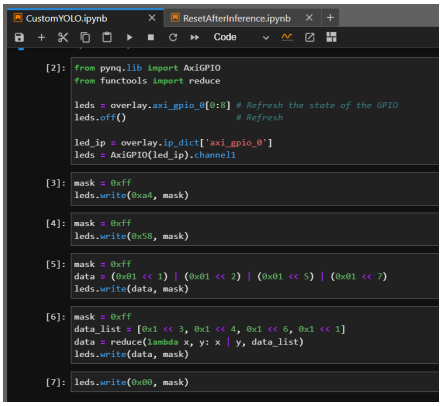

Fig. 29. Pynq code to turn on some LEDs.

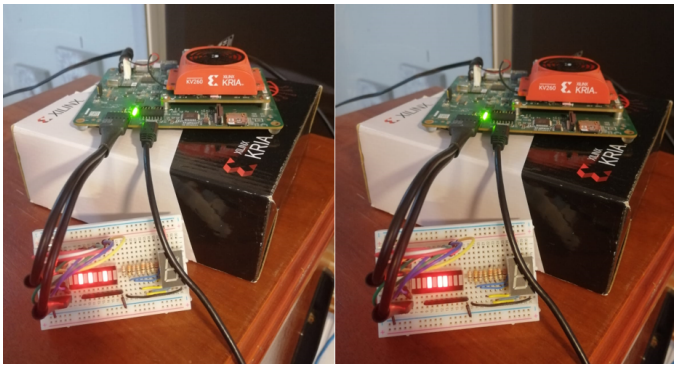

Fig. 30. Flashing LEDs from the PMOD port.

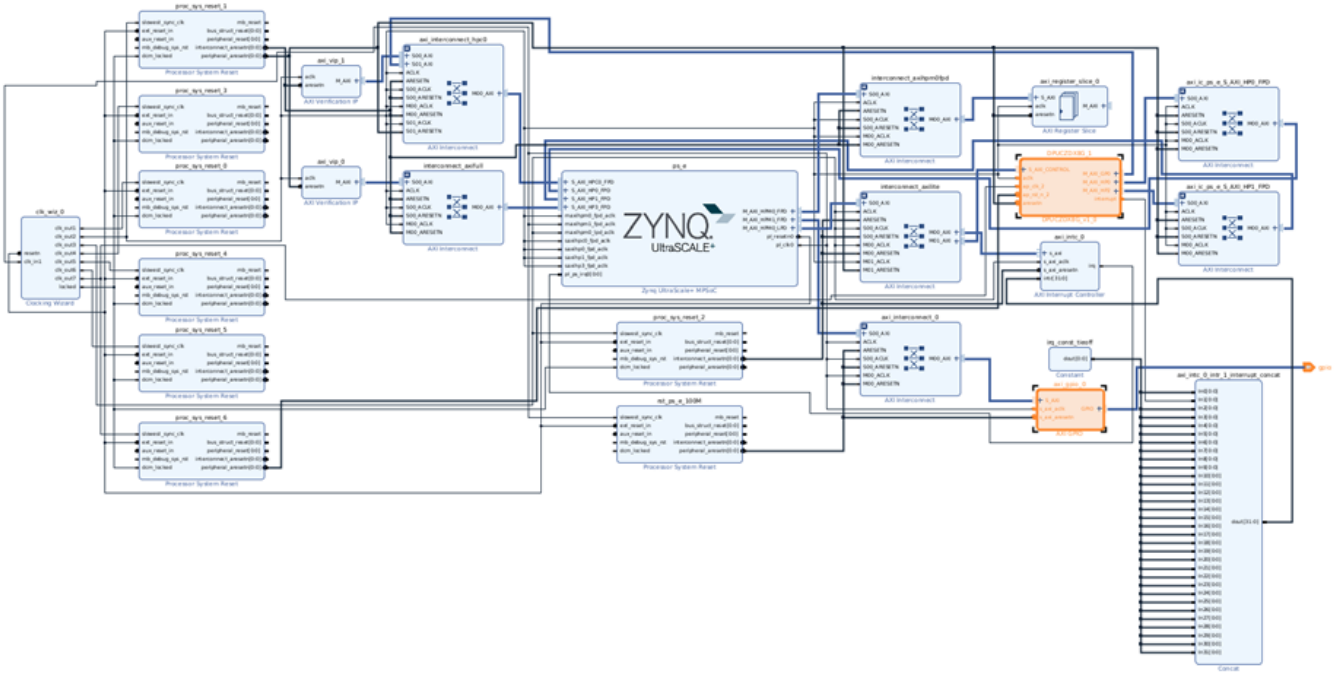

Fig. 31. Custom DPU platform.

V. Compiling a Pretrained Network

With the hardware platform working the DPU is able to implement a Neural Network (NN). Xilinx provides the Vitis AI tools to allow the developer to transform a description of the neural network from a model saved with the TensorFlow 1.x/2.x, PyTorch, Caffe or Onnx frameworks to a binary file that implements a program compatible with the instruction set of the DPU. This file has an extension of .xmodel [29].

The process of transforming the NN to an xmodel file is called compilation. But because the DPU can only do integer computations, the model to be compiled must be quantize to an INT8 (8 bit integer) format. The quantization tools come as a plug-in of the respective python APIs and due to compatibility issues this process must be performed by Vitis AI, different quantization tools other that Vitis AI are not supported for the next compilation step [29].

The compatible version of Vitis AI for the 2022.1 Xilinx tools is v2.5. In this version the support for Caffe is deprecated and the support for Onnx is not yet implemented, thus the only feasible choices are TensorFlow 1.x/2.x or PyTorch [29].

TensorFlow 1.x is no longer actively maintained by Google and "TensorFlow 1.15 is the only version of TensorFlow 1.x still supported by the tensorflow_hub library" [21]. Even though the Vitis AI User Guide ug1414 states that the TensorFlow 2.x API can handle models in a TensorFlow saved model format or Keras (.h5) models [29], the API only has documentation and examples on the use of Keras models. Finally, since there is a an example to quantize and compile a custom Keras model and none is provided for PyTorch, the TensorFlow 2.x/Keras will be chosen as a first approach.

Because the target neural network is an object detection application, some of the available architectures are Region-Based Convolutional Neural Networks (R-CNN), You-only-look-once (YOLO), Single Shot Detectors (SSD), etc. The Pynq DPU repository has an object detection example using YOLOv3, thus this architecture is chosen as a starting point. Therefore, the final choice is to compile a YOLOv3 model on the TensorFlow 2.x/Keras API.

A. Training a YOLOv3 model

The objective of this work is to have a road sign object detector running on the KV260. To do this, the Kaggle Road Sign Detection by the user LARXEL was used. This is an unbalaced data set with 4 detection classes: Traffic Light, Stop, Speed limit and Crosswalk, distributed as shown in Figure 32.

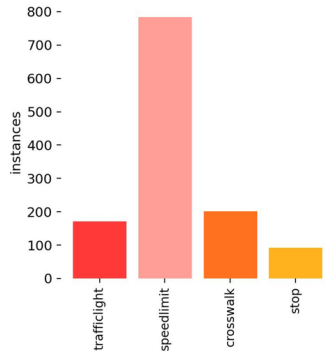

Fig. 32. Data set Distribution Histogram.

Because of the nature of the data set the architecture of the CNN might not be the best choice, however the main objective is to learn about the hardware development flow for AI on the Xilinx boards and not to train the NN.

For this work, the version implemented by qqwweee in his GitHub repository [18] and later modified and updated by the user pythonlessons in his GitHub repository has been used [15]. The latter user has a tutorial on how to create the labels for a custom Data set on his web page and how to use these annotations to train a custom Keras YOLOv3 model [16]. There is also a blog on "How to Perform Object Detection With YOLOv3 in Keras" [3] that explains how to create the architecture of a YOLOv3 NN and how to load pre-trained weights to it.

Following the guidelines of the official Python lessons web page, but modifying the calls to the Keras library functions, since they are outdated in the repository, the training was carried out by Google-Colab with an N-Vidia Tesla T4 GPU for 1 hour and 30 minutes, and the model with the final weights was exported in a .h5 format.

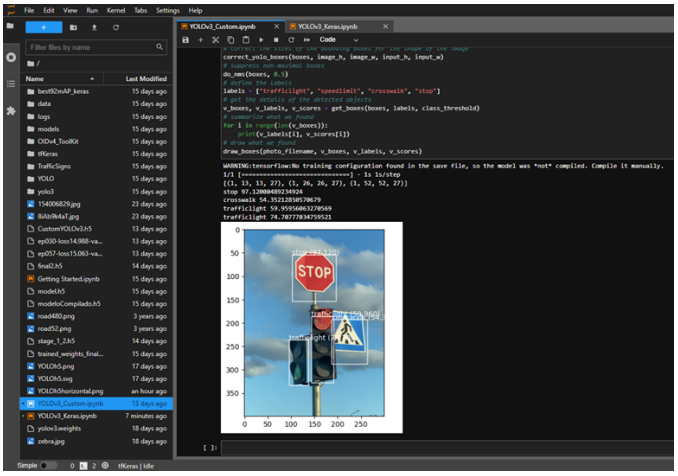

Fig. 33. Results of the trained YOLOv3 network for the custom data set.

B. Vitis AI

Now that the pre-trained neural network is available the next step is to quantize and compile it to the DPU. For this the Vitis AI tools needs to be installed on a Linux machine. Xilinx offers these tools as docker images, so docker must be installed on the Ubuntu machine the same way as shown for the KV260 OS II-B. The vitis-ai 2.5 repository needs to be cloned first and the full installation steps can be found in the Vitis AI GitHub repository.

```
$ git clone https://github.com/
   Xilinx/Vitis-AI.git -b 2.5
$ cd ./Vitis-AI
```

The docker_run.sh script is used to activate the environment or download it for the first time. The TensorFlow 2.x can be activated in a conda environment. Finally, the quantization and compilation processes can be carried out in a Jupyter Notebook.

```
/Vitis-AI$ ./docker_run.sh xilinx/
   vitis-ai-cpu:2.5
/workspace > conda activate
   vitis-ai-tensorflow2
/workspace > jupyter notebook
```

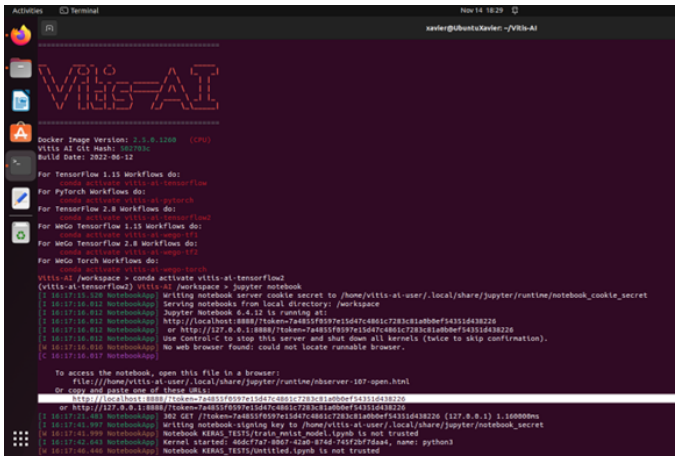

Fig. 34. Vitis AI tensorflow2 environment.

Loading the model to the Vitis AI environment proved that the model was working as expected as shown in Figures 35 and 36. For this, the h5 model must be placed on the same folder as the jupyter file, the pre and post processing functions are taken from Jason Brownlee [3].

```
from keras.models import load_model
model = load_model('CustomYOLOv3Keras.h5')
```

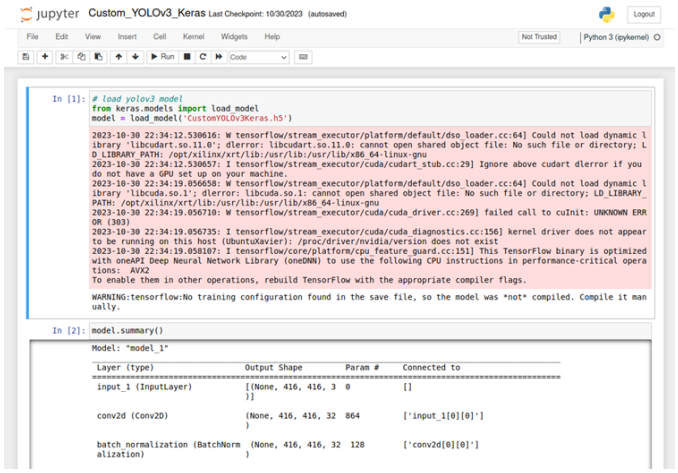

Fig. 35. Custom YOLOv3 model loaded on the Vitis AI environment.

Using the vitis_inspect tool, as in Figure 37, allows to verify that the model is compatible with the DPU architecture of the KV-260 board and that all its operations will be executed on it as shown in Figure 38, the python code is.

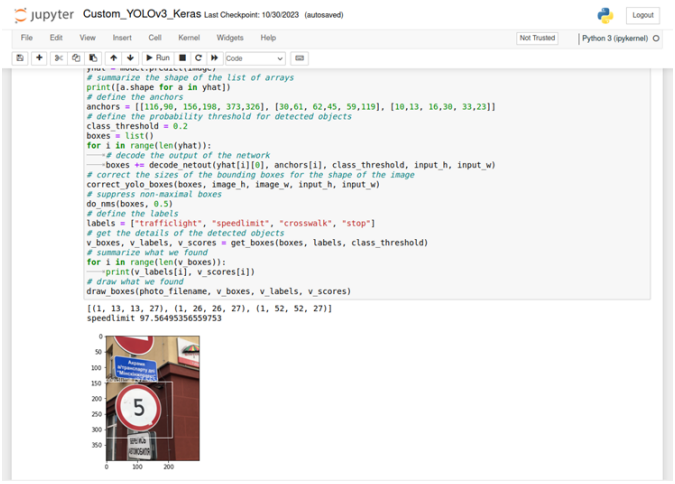

Fig. 36. Custom YOLOv3 model running inference on the Vitis AI environment.

```
from tensorflow_model_optimization.
 quantization.keras import vitis_inspect
inspector = vitis_inspect.VitisInspector
 (target="/opt/vitis_ai/compiler/arch/
 DPUCZDX8G/KV260/arch.json")
inspector.inspect_model(model,
    plot=True,
    plot_file="model.svg",
    dump_results=True,
    dump_results_file="inspect_results.txt"
     ,verbose=1)
```

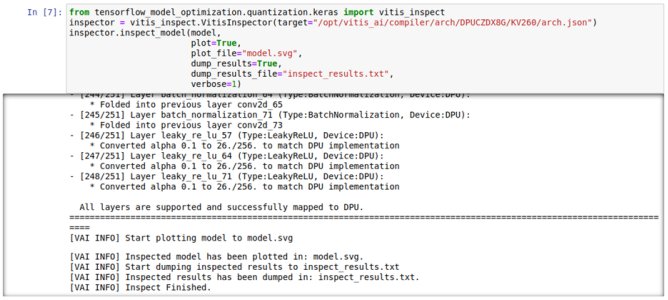

Fig. 37. Vitis_Inspect tool.

The results indicate that all layers can be correctly mapped to the DPU. The tool also saves an image in SVG format with the network structure mapped to the DPU and the names and types of each layer as shown in Figure 38.

According to Xilinx [29] in the Vitis-AI ug1414 User Guide, to use the quantization tool, the Keras model and a calibration set of 100-1000 images are needed as shown in Figure 39. In this case, 500 images will be stored in an array of that depth in a 416x416 (RGB) format pre-processed for YOLOv3 see Figure 40 for the implementation. The python code is.

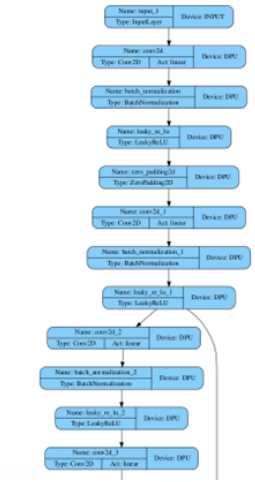

Fig. 38. Inspected model graph.

```
# load and prepare an image
def load_image_pixels_quant(filename, shape):
  # load the image with the required size
  image = load_img(filename, target_size=shape)
  # convert to numpy array
  image = img_to_array(image)
  # scale pixel values to [0, 1]
  image = image.astype('float32')
  image /= 255.0
  # add a dimension so that we have one sample
  image = expand_dims(image, 0)
  return image
import os
# load and prepare image
calib_dataset = 0
calib_dataset = load_image_pixels_quant(
 'test1.jpeg', (input_w, input_h)
)
i = 0
for photo_filename in os.listdir(data_dir):
  image = load_image_pixels_quant(
   os.path.join(data_dir, photo_filename),
   (input_w, input_h)
  )
  calib_dataset = np.append(
   calib_dataset, image, axis=0
  )
  i = i+1
  if (i == 500-1): break
```

No.	Name	Description
1	float model	Floating-point TensorFlow 2 models, either in h5 format or saved model format.
2	calibration dataset	A subset of the training dataset or validation dataset to represent the input data distribution, usually 100 to 1000 images are enough.

Fig. 39. Vitis Quantize parameters. [29]

```
In [9]: # load and prepare an image
def load_image_pixels_quant(filename, shape):
    # load the image with the required size
    image = load_img(filename, target_size=shape)
    # convert to numpy array
    image = img_to_array(image)
    # scale pixel values to [0, 1]
    image = image.astype('float32')
    image /= 255.0
    # add a dimension so that we have one sample
    image = expand_dims(image, 0)
    return image

In [12]: # load and prepare image
calib_dataset = 0
calib_dataset = load_image_pixels_quant('test1.jpeg', (input_w, input_h))
i = 0
for photo_filename in os.listdir(data_dir):
    image = load_image_pixels_quant(os.path.join(data_dir, photo_filename), (input_w, input_h))
    calib_dataset = np.append(calib_dataset, image, axis=0)
    i = i+1
    if (i == 500-1): break

print(type(calib_dataset))
print(calib_dataset.size)
print(calib_dataset.shape)

<class 'numpy.ndarray'>
259584000
(500, 416, 416, 3)
```

Fig. 40. Obtaining the calibration data set.

Quantize the model with the vitis_quantize tool and save it in h5 format as shown in Figure 41. Then compile the quantized model with the vai_c_tensorflow2 tool with the DPU architecture specified with the fingerprint arch.json as shown in Figure 42. The Jupyter Notebook code to do this is:

```
from tensorflow_model_optimization.
 quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(
    model
)
# Perform Quantization
quantized_model = quantizer.quantize_model(
    calib_dataset = calib_dataset,
    weight_bit=8,
    activation_bit=8
)
# Save the Quantized Model
quantized_model.save('YOLOv3_quantized.h5')
# Compile the Quantized Model
!vai_c_tensorflow2 \
    --model ./YOLOv3_quantized.h5 \
    --arch /opt/vitis_ai/compiler/
      arch/DPUCZDX8G/KV260/arch.json \
    --output_dir . \
    --net_name CustomYOLOv3
```

The compilation tool creates the binary model of the network in .xmodel format as shown in Figure 43 that can run on the DPU.

```
In [13]: from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset = calib_dataset, weight_bit=8, activation_bit=8)

[VAI INFO] Update activation_bit: 8
[VAI INFO] Update weight_bit: 8
[VAI INFO] Start CrossLayerEqualization...
10/10 [==============================] - 19s 2s/step
[VAI INFO] CrossLayerEqualization Done.
[VAI INFO] Start Quantize Calibration...
16/16 [==============================] - 653s 36s/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quant Model Refinement...
[VAI INFO] Start Quantize Position Ajustment...
[VAI INFO] Quantize Position Ajustment Done.
[VAI INFO] Post-Quant Model Refinement Done.
[VAI INFO] Start Model Finalization...
[VAI INFO] Model Finalization Done.
[VAI INFO] Quantization Finished.

In [14]: quantized_model.save('YOLOv3_quantized.h5')

WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics`
will be empty until you train or evaluate the model.
```

Fig. 41. Quantization with Vitis AI.

```
In [15]: !vai_c_tensorflow2 \
    --model ./YOLOv3_quantized.h5 \
    --arch /opt/vitis_ai/compiler/arch/DPUCZDX8G/KV260/arch.json \
    --output_dir . \
    --net_name CustomYOLOv3

**************************************************
* VITIS_AI Compilation - Xilinx Inc.
**************************************************
[INFO] Namespace(batchsize=1, inputs_shape=None, layout='NHWC', model_files=['./YOLOv3_quantized.h5'], model_type='t
ensorflow2', named_inputs_shape=None, out_filename='/tmp/CustomYOLOv3_DPUCZDX8G_ISA1_B4096_org.xmodel', proto=None)
[INFO] tensorflow2 model: /workspace/KERAS_TESTS/CustomYolo/YOLOv3_quantized.h5
[INFO] keras version: 2.8.0
[INFO] Tensorflow Keras model type: functional
[INFO] parse raw model     :100%|| 181/181 [00:00<00:00, 18486.11it/s]
[INFO] infer shape (NHWC)  :100%|| 283/283 [00:00<00:00, 540.34it/s]
[INFO] perform level-0 opt :100%|| 2/2 [00:00<00:00, 38.21it/s]
[INFO] perform level-1 opt :100%|| 2/2 [00:00<00:00, 151.57it/s]
[INFO] generate xmodel     :100%|| 283/283 [00:00<00:00, 284.27it/s]
[INFO] dump xmodel: /tmp/CustomYOLOv3_DPUCZDX8G_ISA1_B4096_org.xmodel
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: function
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISA1_B4096
[UNILOG][INFO] Graph name: model_1, with op num: 583
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 5, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/KERAS_TESTS/CustomYolo/./meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/KERAS_TESTS/CustomYolo/./CustomYOLOv3.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is 5c29094fc9faf85880854eb6ea4edfff, and has been saved to "/workspace/K
ERAS_TESTS/CustomYolo/./md5sum.txt"
```

Fig. 42. Compilation of the YOLOv3 CNN.

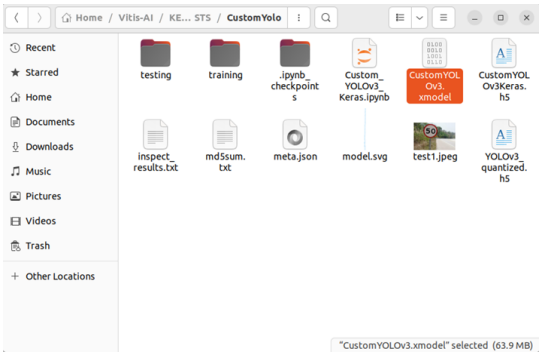

Fig. 43. Resulting YOLOv3 xmodel file.

VI. PYNQ SOFTWARE APPLICATION

The software application configures the FPGA with the custom overlay (this has been done in section IV and can be seen on Figure 28), loads the xmodel to the DPU, pre process the input images, post process the output tensors of the YOLOv3 CNN, and streams real time inference video.

A. Loading the xmodel

The model can be integrated into the custom platform using the YOLOv3 Pynq example that uses the same network trained for the VOC data set. The xmodel file is first uploaded to the DPU by the load_model() method as shown in Figure 51.

```
overlay.load_model("CustomYOLOv3.xmodel")
```

The VART api (Vitis AI Runtime Library) allows to inspect the input and output tensors of the model. The inspection of the model can be seen in Figure 52 and the tensor shapes and sizes in Figure 53.

```
dpu = overlay.runner
dpu?
inputTensors = dpu.get_input_tensors()
for inputTensor in inputTensors:
  print('Input tensor :',
     inputTensor.name,
     inputTensor.dtype,
     inputTensor.dims)
outputTensors = dpu.get_output_tensors()
```

```
for outputTensor in outputTensors:
  print('Output tensor :',
    outputTensor.name,
    outputTensor.dtype,
    outputTensor.dims)
```

B. Pre and Post Process Functions

Because this application is using the YOLOv3 DPU example, the pre and post processing functions are the same, and by changing the classes accordingly it can display inference results of some example images.

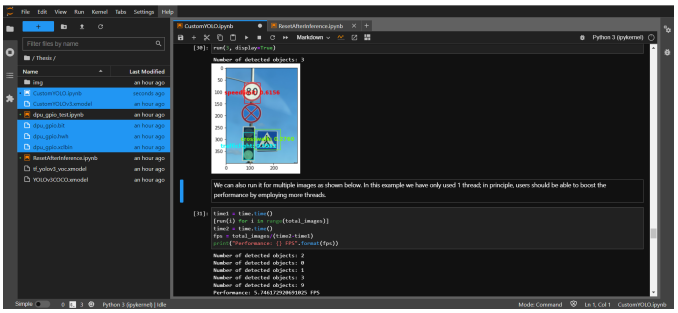

Fig. 44. Result of inference running on the FPGA for an image.

The results shown in Figure 44 were generated with a custom YOLOv3 network trained for the detection of 4 classes of road signs that runs on the Kria KV260 board. The next step is to create an application that allows real-time inference. To do this, modify the run function and use the Display Port API that is integrated into Pynq.

C. Real Time Inference

Because Pynq is running in a remote Jupyter Lab, it is not possible to stream in a software window, however, Pynq provides a couple of examples on how to use the Display/HDMI port while doing some image processing, therefore, mimicking the Smart Camera Application we will stream real-time inference video through HDMI.

The user feiticeir0 tried to do something similar on his Kria KV260 Product Review on element14.com and pointed out that it is possible to source Pynq as a virtual environment and work on the Ubuntu desktop directly, in which case, it is possible to stream video on a software window [4]. To access the Pynq virtual environment, source the ***pynq_venv.sh*** script from a terminal window on the Ubuntu desktop.

```
$ source /etc/profile.d/pynq_venv.sh
```

If the display port is used, there is no point in using the desktop. Following a similar trajectory as feiticeir0, the "***DrawBoxes***" and "***run***" functions were modified from the YOLOv3 Pynq DPU example. To make use of the AXI GPIO IP present in the custom platform, the new "***DrawBoxes***" function also turns on a LED if the respective class is detected. The new "***run***" function, internally measures the latency of the inference process in the DPU. Figure 54 shows the new "***DrawBoxes***" function, and the code is:

```
# New draw boxes function
mask = 0xff
def draw_boxes_display(
 image, boxes, scores, classes
):
  image_h, image_w, _ = image.shape
  data_list = np.zeros(20, dtype = int)
  for i, bbox in enumerate(boxes):
    [top, left, bottom, right] = bbox
    top = int(top)
    left = int (left)
    bottom = int (bottom)
    right = int (right)
    score, class_index = scores[i], classes[i]
    # save the clases recognized
    data_list[i] = 0x02 << class_index
    label = '{}: {:.4f}'.format(
      class_names[class_index], score
    )
    color = tuple([color for color in colors[
      class_index
    ]])
    # show frame
    cv2.rectangle(image, (left,top),
      (right,bottom), color, 2)
    # show class
    cv2.putText(image, label, (left,top-10),
      font, 1, color, 2, cv2.LINE_AA)

    # Print the LEDs
    data = int(reduce(lambda x, y:
      x | y, data_list))
    leds.write(data, mask)
    return image
```

Figure 54 shows the new "***run***" function and the code is:

```
# New run function
def run_display(frame):
  # Pre-processing
  image_size = frame.shape[:2]
  image_data = np.array(
    pre_process(frame, (416, 416)),
    dtype=np.float32
  )
  # Fetch data to DPU and trigger it
  image[0,...] = image_data.reshape(
    shapeIn[1:])
  # To calculate the inference fps -------
  prev_frame = time.time()
  job_id = dpu.execute_async(
    input_data,
    output_data
  )
  dpu.wait(job_id)
  new_frame = time.time()
  # ------------------------------------
```

```
  fps = 1 / (new_frame - prev_frame)
  # Final fps
  fps = int(fps)
  # Retrieve output data
  conv_out0 = np.reshape(output_data[0],
    shapeOut0)
  conv_out1 = np.reshape(output_data[1],
    shapeOut1)
  conv_out2 = np.reshape(output_data[2],
    shapeOut2)
  yolo_outputs = [conv_out0, conv_out1,
    conv_out2]
  # Decode output from YOLOv3
  boxes, scores, classes = evaluate(
    yolo_outputs,
    image_size,
    class_names,
    anchors
  )
  #new_image = draw_boxes2(
    frame, boxes, scores, classes
  )
  draw_boxes_display(
    frame, boxes, scores, classes
  )
  return fps
```

Finally, the Display port and an OpenCV2 object must be configured to capture and transmit video from a USB camera. This is shown in Figure 56. Finally, the infinite loop of video streaming calls the new run function to make the inference. This is shown in Figure 57.

```
from pynq.lib.video import *
# Configure the display port
displayport = DisplayPort()
displayport.configure(VideoMode(640,480,24),
  PIXEL_RGB)

# Create a Video Capture object
cap = cv2.VideoCapture(0)

# Test if errors
if not cap.isOpened():
    print ("cannot open camera")

# Define the output width and height
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

# Optimize the open_cv video buffer
cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)
buf_size = cap.get(cv2.CAP_PROP_BUFFERSIZE)
prev_frame = 0
new_frame = 0
font = cv2.FONT_HERSHEY_SIMPLEX

# Infinite Loop
while True:
  try:
    frame = displayport.newframe()
    cap.read(frame)

    new_frame = time.time()
    fps = 1 / (new_frame - prev_frame)
    prev_frame = new_frame
    fps = int(fps)

    #display fps
    inference_fps = run_display(frame)
    cv2.putText(
      frame,
      str(inference_fps) + 'fps inference',
      (15, 50),
      font,
      1,
      (255, 0, 0),
      2,
      cv2.LINE_AA
    )
    cv2.putText(
      frame, str(fps) + 'fps overall',
      (15, 80),
      font,
      1,
      (255, 0, 0),
      2,
      cv2.LINE_AA
    )

    #cv2.imshow("output", frame)
    displayport.writeframe(frame)

  # To exit
  except KeyboardInterrupt:
    !sudo xmutil unloadapp
    !sudo xmutil loadapp k26-starter-kits
    !sudo xmutil listapps
    break

# Release overlay and objects
cap.release()
displayport.stop()
displayport.close()
del overlay
del dpu
print("Succesfully released everything!")
```

The application was tested with 4 Traffic Sign Images, one for each class. Figures 45 to 47 show the GPIO results of the inference for all 4 classes. Figures 48 to 50 show the GPIO results of the inference for a subset consisting only on the "speed limit" and "stop" signs.

Fig. 45. Inference Setup for the 4 classes.

Fig. 46. Inference Result for the 4 classes.

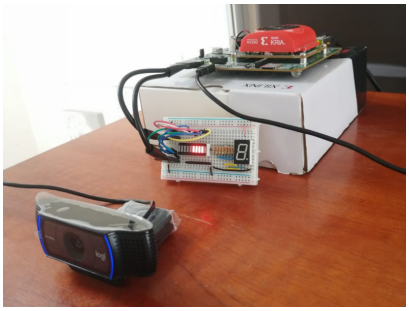

Fig. 47. GPIO Result for the 4 classes.

Fig. 48. Inference Setup for the speed limit and stop signs.

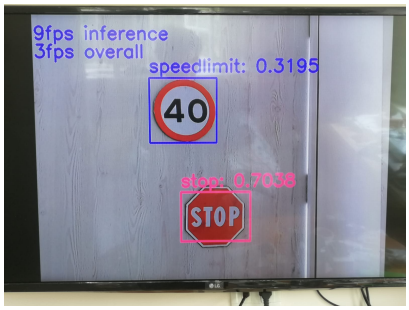

Fig. 49. Inference Result for the speed limit and stop signs.

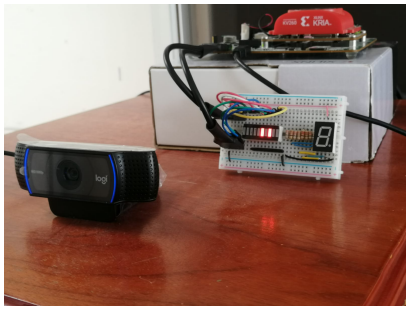

Fig. 50. GPIO Result for the speed limit and stop signs.

VII. Conclusions

As can be seen, the application runs from start to finish at 3fps, which is extremely low for practical applications, and the DPU has a refresh rate of 9fps. This is because the Python (a very high-level interpreted language) on the Ubuntu operating system is used, and this adds considerable latency to the communication and flow of information.

Another problem encountered is the complexity of computations that are needed both for the pre processing of the image and for the post processing of the inference results. These algorithms are compute intensive tasks and can be offloaded into the FPGA to speed up the application.

The application can be improved to have a faster inference and be more robust in terms of precision. To improve accuracy, it can be trained for longer, with a larger data set using augmentation. However, the YOLOv3 network, being relatively old, has its limitations, and it might be interesting to test other architectures, such as R-CNN, SSD, or the most current YOLO models such as v5, v7, v8, etc.

To improve the fps of the application, the pre and post processing algorithms can be accelerated by offloading them

to the FPGA. This can be achieved with the Vitis libraries for high level synthesis (HLS) using the xilinx Vitis-HLS tool.

The DMA (Direct Memory Access) flow can also be used to skip the transmission of information through Pynq, and allow the FPGA components to have direct access to the card's memory. To achieve this, there is the Xilinx AXI DMA IP that allows developers to create video pipelines in hardware.

Acknowledgment

I first thank God for giving me the opportunity to study, the strength and wisdom to face all adversity and the graces to sustain me in my worst moments. To my mother and father for always being by my side giving me their unconditional support. To my mentor Mauricio Barros for giving me his friendship and knowledge. To my sister for always making me see the positive side of things. To my friend Ismael Ramos for his advises and his unconditional friendship. And to all the teachers and colleagues who have pushed me to be better every day.

References

[1] P. Alfke, I. Bolsens, B. Carter, M. Santarini, and S. Trimberger, "It's an FPGA!" *IEEE Solid-State Circuits Magazine*, vol. 3, no. 4, pp. 15-20, 2011. DOI: https://doi.org/10.1109/MSSC.2011.942449

[2] N. Buduma, "Fundamentals of Deep Learning." O'Reilly Media, 2017.

[3] Brownlee, J. (2019). "How to Perform Object Detection With YOLOv3 in Keras." [Online]. Available: https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/

[4] feiticeir0. (2023). "Product Roadtest Review of AMD Xilinx Kria KV260 Vision Starter Kit." [Online]. Available: https://community.element14.com/products/roadtest/rv/roadtest_reviews/1681/step_learning_curve

[5] F. Chollet, "Deep learning with Python." Manning Publications, 2017.

[6] L. Crockett, D. Northcote, C. Ramsay, F. Robinson, and R. Stewart, "Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications," 2019.

[7] DataScientest. (2023). "Keras: La API de Deep Learning." Retrieved from https://datascientest.com/es/keras-la-api-de-deep-learning

[8] W. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115-133, 1943.

[9] OpenAI (2018). "Ai and Compute." Available at: https://openai.com/research/ai-and-compute (Accessed: 05 September 2023).

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016. DOI: https://doi.org/10.1109/CVPR.2016.91

[11] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386-408, 1958.

[12] Knitter, W. (2022). "Update Boot Binary and Install PYNQ on Kria KV260." [Online]. Available: https://www.hackster.io/whitney-knitter/update-boot-binary-and-install-pynq-on-kria-kv260-03b7e9

[13] Knitter, W. (2022). "Vitis+PetaLinux 2022.1 & KRS 1.0 Install on Ubuntu 22.04." [Online]. Available: https://www.hackster.io/whitney-knitter/vitis-petalinux-2022-1-krs-1-0-install-on-ubuntu-22-04-145c1b

[14] G. Shmueli, P. C. Bruce, P. Gedeck, and N. R. Patel, "Data mining for business analytics: Concepts, techniques, and applications in Python." Wiley, 2020.

[15] PyLessons. (2019). "Training custom YOLOv3 object detection model." [Online]. Available: https://pylessons.com/YOLOv3-custom-training

[16] PyLessons. (2019). "Preparing YOLO v3 Custom Data." [Online]. Available: https://pylessons.com/YOLOv3-custom-data

[17] PYNQ. (2021). "Overlay Design." [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/overlay_design_methodology/overlay_design.html#overlay-hwh-file

[18] qqwweee. (2018). "keras-yolo3." [Online]. Available: https://github.com/qqwweee/keras-yolo3

[19] R. Wilson. (2015, April 21). "In the Beginning." altera.com. Available at: https://web.archive.org/web/20150421045728/https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html

[20] Shreyas N R. (2023). "KV260 DPU-TRD Petalinux 2022.1 Vivado Flow." [Online]. Available: https://www.hackster.io/shreyasnr/kv260-dpu-trd-petalinux-2022-1-vivado-flow-000c0b

[21] TensorFlow. (2023). "Installation." [Online]. Available: https://www.tensorflow.org/hub/installation

[22] Tesla, Inc. (2023). "Autopilot." Tesla. Available at: https://www.tesla.com/es_ES/autopilot

[23] Wood, Luke and Tan, Zhenyu and Stenbit, Ian and Bischof, Jonathan and Zhu, Scott and Chollet, et. al., (2022), "KerasCV". Retrieved from https://github.com/keras-team/keras-cv

[24] Xilinx. (2020). "PG338 : Zynq DPU v3.2." Xilinx Corporation.

[25] Xilinx. (2021). "Versal ACAP Hardware, IP, and Platform Development Methodology User Guide UG1387." [Online]. Available: https://docs.xilinx.com/r/2020.2-English/ug1387-acap-hardware-ip-platform-dev-methodology/Exporting-the-Hardware-Handoffs

[26] Xilinx. (no date). "System-on-Modules (SOMs): How and Why to Use Them." Available at: https://www.xilinx.com/products/som/what-is-a-som.html

[27] Xilinx. (2022). "XRT Binary Formats." [Online]. Available: https://xilinx.github.io/XRT/master/html/formats.html

[28] Xilinx. (2022). "DPUCZDX8G for Zynq UltraScale+ MPSoCs, (pg338 V4.0)." Available at: https://docs.xilinx.com/r/4.0-English/pg338-dpu/Introduction

[29] Xilinx. (2022). "Vitis AI User Guide, (ug1414 V2.5)." Available at: https://docs.xilinx.com/r/2.5-English/ug1414-vitis-ai

[30] Xilinx. (2022). "Vitis AI Library User Guide, (ug1354 V2.5)." Available at: https://docs.xilinx.com/r/2.5-English/ug1354-xilinx-ai-sdk/Vitis-AI-Library-v2.5-Release-Notes

[31] Xilinx. (2022). "Kria KV260 Vision AI Starter Kit Data Sheet (Report No. DS986, v1.1)." Available at: https://docs.xilinx.com/r/en-US/ds986-kv260-starter-kit/Summary

[32] Xilinx. (2022). "DPU-PYNQ." GitHub. Available at: https://github.com/Xilinx/DPU-PYNQ

[33] Xilinx. (2022). "Vivado Design Suite User Guide UG908." [Online]. Available: https://docs.xilinx.com/r/2022.1-English/ug908-vivado-programming-debugging/Vivado-Lab-Edition

[34] Xilinx. (2023). "UG1085: Zynq UltraScale+ Device." Available at: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm

[35] Xilinx. (2023). "Kria Products." [Online]. Available: https://www.xilinx.com/products/som/kria.html

[36] Xilinx. (2023). "Getting Started with Kria KV260 Vision AI Starter Kit." [Online]. Available: https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit/kv260-getting-started/getting-started.html

[37] Xilinx. (2023). "Kria App Store." [Online]. Available: https://www.xilinx.com/products/app-store/kria.html

[38] Xilinx. (2023). "Smart Camera - Setting up the Board and Application Deployment." [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/app_deployment.html

[39] Xilinx. (2023). "Kria KV260 Vision AI Starter Kit Applications." [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/index.html

[40] Xilinx. (2023). "Vitis AI - IP and Tool Version Compatibility." [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/version_compatibility.html#version-compatibility

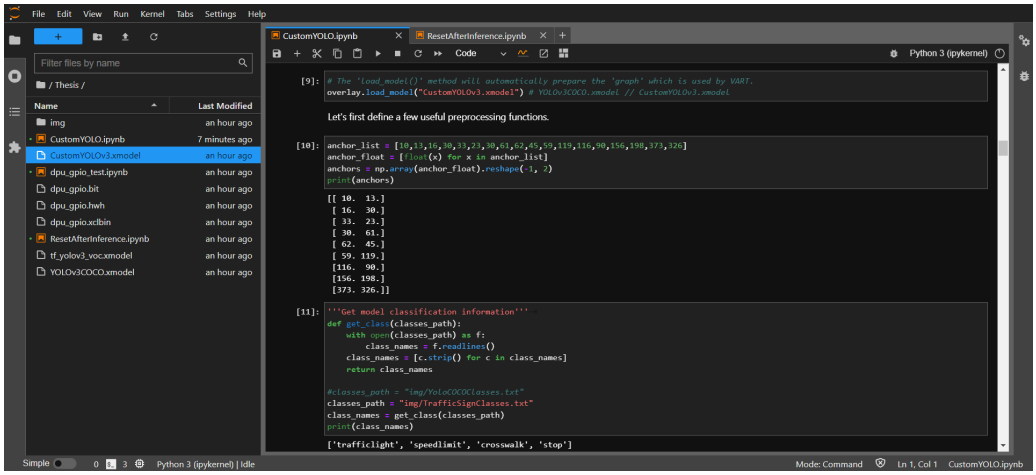

Fig. 51. YOLOv3 model loaded to the DPU overlax.

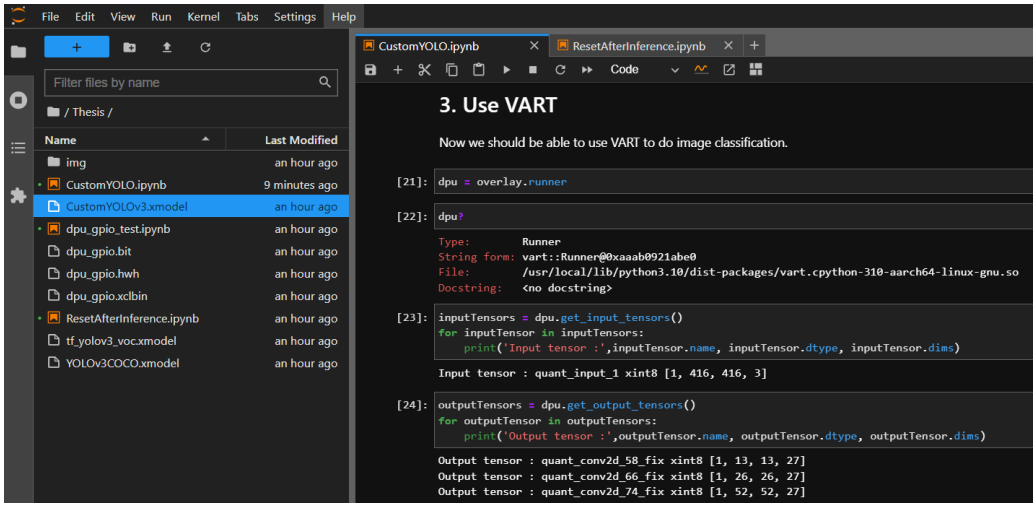

Fig. 52. Using VART to inspect the uploaded model.

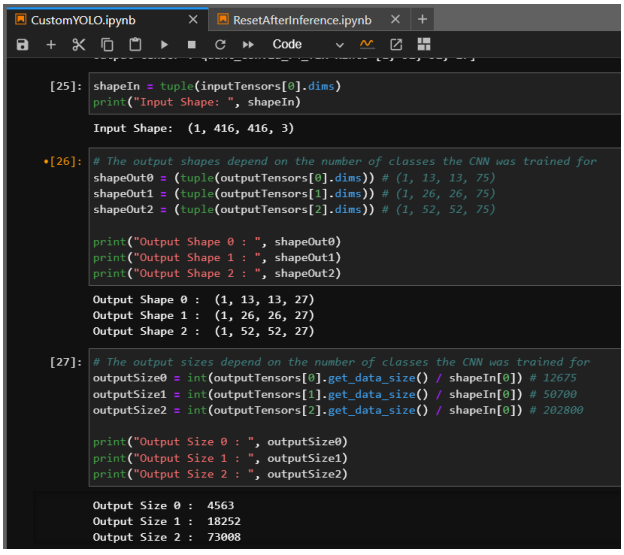

Fig. 53. Shapes and Sizes of the model.

```
[32]: # New draw boxes function
mask = 0xff
def draw_boxes_display(image, boxes, scores, classes):
    image_h, image_w, _ = image.shape

    data_list = np.zeros(20, dtype = int) # Initialize an array of zeroes to write to the LEDs
    for i, bbox in enumerate(boxes):
        [top, left, bottom, right] = bbox
        top = int(top)
        left = int (left)
        bottom = int (bottom)
        right = int (right)
        score, class_index = scores[i], classes[i]

        # save the clases recognized
        data_list[i] = 0x02 << class_index

        label = '{}: {:.4f}'.format(class_names[class_index], score)
        color = tuple([color for color in colors[class_index]])
        # show frame
        cv2.rectangle(image, (left,top), (right,bottom), color, 2)
        # show class
        cv2.putText(image, label, (left,top-10), font, 1, color, 2, cv2.LINE_AA)

    # Print the LEDs
    data = int(reduce(lambda x, y: x | y, data_list))
    leds.write(data, mask)
    return image
```

Fig. 54. Modified DrawBoxes function.

```
# New run function
def run_display(frame):
    ''' This function accepts a frame as input, and draws the results of the inference to it. '''
    # Pre-processing
    image_size = frame.shape[:2]
    image_data = np.array(pre_process(frame, (416, 416)), dtype=np.float32)

    # Fetch data to DPU and trigger it
    image[0,...] = image_data.reshape(shapeIn[1:])
    prev_frame = time.time()                                    # To calculate the inference fps
    job_id = dpu.execute_async(input_data, output_data)
    dpu.wait(job_id)
    new_frame = time.time()                                     # To calculate the inference fps
    fps = 1 / (new_frame - prev_frame)
    fps = int(fps)                                              # Final fps

    # Retrieve output data
    conv_out0 = np.reshape(output_data[0], shapeOut0)
    conv_out1 = np.reshape(output_data[1], shapeOut1)
    conv_out2 = np.reshape(output_data[2], shapeOut2)
    yolo_outputs = [conv_out0, conv_out1, conv_out2]

    # Decode output from YOLOv3
    boxes, scores, classes = evaluate(yolo_outputs, image_size, class_names, anchors)

    #new_image = draw_boxes2(frame, boxes, scores, classes)
    draw_boxes_display(frame, boxes, scores, classes)

    return fps
```

Fig. 55. Modified Run function.

Set the display port and prepare a Video Capture object from open_cv2

```
[33]: from pynq.lib.video import *
      # Configure the display port
      displayport = DisplayPort()
      displayport.configure(VideoMode(640,480,24), PIXEL_RGB)

      # Create a Video Capture object
      cap = cv2.VideoCapture(0)

      # Test if errors
      if not cap.isOpened():
          print ("cannot open camera")

      # Define the output width and height
      cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
      cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

      # Optimize the open_cv video buffer
      cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)
      buf_size = cap.get(cv2.CAP_PROP_BUFFERSIZE)
      prev_frame = 0
      new_frame = 0
      font = cv2.FONT_HERSHEY_SIMPLEX
```

Fig. 56. Configuration of the Video Objects.

```
while True:
    try:
        frame = displayport.newframe()
        cap.read(frame)

        new_frame = time.time()
        fps = 1 / (new_frame - prev_frame)
        prev_frame = new_frame
        fps = int(fps)

        #display fps
        inference_fps = run_display(frame)
        cv2.putText(frame, str(inference_fps) + 'fps inference', (15, 50), font, 1, (255, 0, 0), 2, cv2.LINE_AA)
        cv2.putText(frame, str(fps) + 'fps overall', (15, 80), font, 1, (255, 0, 0), 2, cv2.LINE_AA)

        #cv2.imshow("output", frame)
        displayport.writeframe(frame)
    except KeyboardInterrupt:
        !sudo xmutil unloadapp
        !sudo xmutil loadapp k26-starter-kits
        !sudo xmutil listapps
        break
```

Allways clean the video objects and overlays

```
cap.release()
displayport.stop()
displayport.close()
del overlay
del dpu
print("Succesfully released everything!")
```

Fig. 57. Infinite Video Streaming Loop.

CONCLUSIONES

Como puede verse, la aplicación se ejecuta de principio a fin a 3 fps, lo cual es extremadamente bajo para aplicaciones prácticas, y la DPU tiene una frecuencia de actualización de 9 fps. Esto se debe a que se utiliza Python (un lenguaje interpretado de muy alto nivel) en el sistema operativo Ubuntu, y esto añade una latencia considerable a la comunicación y al flujo de información.

Otro problema encontrado es la complejidad de los cálculos que se necesitan tanto para el procesamiento previo de la imagen como para el procesamiento posterior de los resultados de la inferencia. Estos algoritmos son tareas informáticas intensivas y se pueden descargar en la FPGA para acelerar la aplicación.

La aplicación se puede mejorar para tener una inferencia más rápida y ser más robusta en términos de precisión. Para mejorar la precisión, se puede entrenar durante más tiempo, con un conjunto de datos más grande mediante aumento. Sin embargo, la red YOLOv3, al ser relativamente antigua, tiene sus limitaciones, y podría ser interesante probar otras arquitecturas, como R-CNN, SSD, o los modelos YOLO más actuales como v5, v7, v8, etc.

Para mejorar los fps de la aplicación, los algoritmos de pre y post procesamiento se pueden acelerar descargándolos a la FPGA. Esto se puede lograr con las bibliotecas de Vitis para síntesis de alto nivel (HLS) utilizando la herramienta xilinx Vitis-HLS.

El flujo DMA (Direct Memory Access) también se puede utilizar para omitir la transmisión de información a través de Pynq y permitir que los componentes FPGA tengan acceso directo a la memoria de la tarjeta. Para lograr esto, existe Xilinx AXI DMA IP que permite a los desarrolladores crear canales de video en hardware.

REFERENCIAS BIBLIOGRÁFICAS

Alfke, P., Bolsens, I., Carter, B., Santarini, M., & Trimberger, S. (2011). It's an FPGA! IEEE Solid-State Circuits Magazine, 3(4), 15-20. https://doi.org/10.1109/MSSC.2011.942449

Buduma, N. (2017). Fundamentals of Deep Learning. O'Reilly Media.

Brownlee, J. (2019). How to Perform Object Detection With YOLOv3 in Keras. [Online]. Available: https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/

DataScientest. (2023). Keras: La API de Deep Learning. Retrieved from https://datascientest.com/es/keras-la-api-de-deep-learning

feiticeir0. (2023). Product Roadtest Review of AMD Xilinx Kria KV260 Vision Starter Kit. [Online]. Available: https://community.element14.com/products/roadtest/rv/roadtest_reviews/1681/step_learning_curve

Chollet, F. (2017). Deep learning with Python. Manning Publications.

Crockett, L., Northcote, D., Ramsay, C., Robinson, F., & Stewart, R. (2019). Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications.

DataScientest. (2023). Keras: La API de Deep Learning. Retrieved from https://datascientest.com/es/keras-la-api-de-deep-learning

McCulloch, W., & Pitts, W. (1943). A Logical Calculus of Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics, 5(4), 115-133.

OpenAI. (2018). Ai and Compute. Available at: https://openai.com/research/ai-and-compute (Accessed: 05 September 2023).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 779–788). https://doi.org/10.1109/CVPR.2016.91

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386-408.

Knitter, W. (2022). Update Boot Binary and Install PYNQ on Kria KV260. [Online]. Available: https://www.hackster.io/whitney-knitter/update-boot-binary-and-install-pynq-on-kria-kv260-03b7e9

Knitter, W. (2022). Vitis+PetaLinux 2022.1 & KRS 1.0 Install on Ubuntu 22.04. [Online]. Available: https://www.hackster.io/whitney-knitter/vitis-petalinux-2022-1-krs-1-0-install-on-ubuntu-22-04-145c1b

Shmueli, G., Bruce, P. C., Gedeck, P., & Patel, N. R. (2020). Data mining for business analytics: Concepts, techniques, and applications in Python. Wiley.

PyLessons. (2019). Training custom YOLOv3 object detection model. [Online]. Available: https://pylessons.com/YOLOv3-custom-training

PyLessons. (2019). Preparing YOLO v3 Custom Data. [Online]. Available: https://pylessons.com/YOLOv3-custom-data

PYNQ. (2021). Overlay Design. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/overlay_design_methodology/overlay_design.html#overlay-hwh-file

qqwweee. (2018). keras-yolo3. [Online]. Available: https://github.com/qqwweee/keras-yolo3

Wilson, R. (2015, April 21). In the Beginning. altera.com. Available at: https://web.archive.org/web/20150421045728/https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html

Shreyas N R. (2023). KV260 DPU-TRD Petalinux 2022.1 Vivado Flow. [Online]. Available: https://www.hackster.io/shreyasnr/kv260-dpu-trd-petalinux-2022-1-vivado-flow-000c0b

TensorFlow. (2023). Installation. [Online]. Available: https://www.tensorflow.org/hub/installation

Tesla, Inc. (2023). Autopilot. Tesla. Available at: https://www.tesla.com/es_ES/autopilot

Wood, L., Tan, Z., Stenbit, I., Bischof, J., Zhu, S., Chollet, et al. (2022). KerasCV. Retrieved from https://github.com/keras-team/keras-cv

Xilinx. (2020). PG338 : Zynq DPU v3.2. Xilinx Corporation.

Xilinx. (2021). Versal ACAP Hardware, IP, and Platform Development Methodology User Guide UG1387. [Online]. Available: https://docs.xilinx.com/r/2020.2-English/ug1387-acap-hardware-ip-platform-dev-methodology/Exporting-the-Hardware-Handoffs

Xilinx. (no date). System-on-Modules (SOMs): How and Why to Use Them. Available at: https://www.xilinx.com/products/som/what-is-a-som.html

Xilinx. (2022). XRT Binary Formats. [Online]. Available: https://xilinx.github.io/XRT/master/html/formats.html

Xilinx. (2022). DPUCZDX8G for Zynq UltraScale+ MPSoCs, (pg338 V4.0). Available at: https://docs.xilinx.com/r/4.0-English/pg338-dpu/Introduction

Xilinx. (2022). Vitis AI User Guide, (ug1414 V2.5). Available at: https://docs.xilinx.com/r/2.5-English/ug1414-vitis-ai

Xilinx. (2022). Vitis AI Library User Guide, (ug1354 V2.5). Available at: https://docs.xilinx.com/r/2.5-English/ug1354-xilinx-ai-sdk/Vitis-AI-Library-v2.5-Release-Notes

Xilinx. (2022). Kria KV260 Vision AI Starter Kit Data Sheet (Report No. DS986, v1.1). Available at: https://docs.xilinx.com/r/en-US/ds986-kv260-starter-kit/Summary

Xilinx. (2022). DPU-PYNQ. GitHub. Available at: https://github.com/Xilinx/DPU-PYNQ

Xilinx. (2022). Vivado Design Suite User Guide UG908. [Online]. Available: https://docs.xilinx.com/r/2022.1-English/ug908-vivado-programming-debugging/Vivado-Lab-Edition

Xilinx. (2023). UG1085: Zynq UltraScale+ Device. Available at: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm

Xilinx. (2023). Kria Products. [Online]. Available: https://www.xilinx.com/products/som/kria.html

Xilinx. (2023). Getting Started with Kria KV260 Vision AI Starter Kit. [Online]. Available: https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit/kv260-getting-started/getting-started.html

Xilinx. (2023). Kria App Store. [Online]. Available: https://www.xilinx.com/products/app-store/kria.html

Xilinx. (2023). Smart Camera - Setting up the Board and Application Deployment. [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/smartcamera/docs/app_deployment.html

Xilinx. (2023). Kria KV260 Vision AI Starter Kit Applications. [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/index.html

Xilinx. (2023). Vitis AI - IP and Tool Version Compatibility. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/version_compatibility.html#version-compatibility