



基於視覺深度學習之自走車實作

本專案中使用的內容

硬體元件

	AMD Kria KV260 視覺 AI 入門套件	× 1
	支援 ROS 的模型車	× 1
	*支援 ROS 的阿克曼轉向模型車	
	USB 網路攝影機	× 1
	Wi-Fi Dongle	× 1

軟體應用程式和嵌入式作業系統

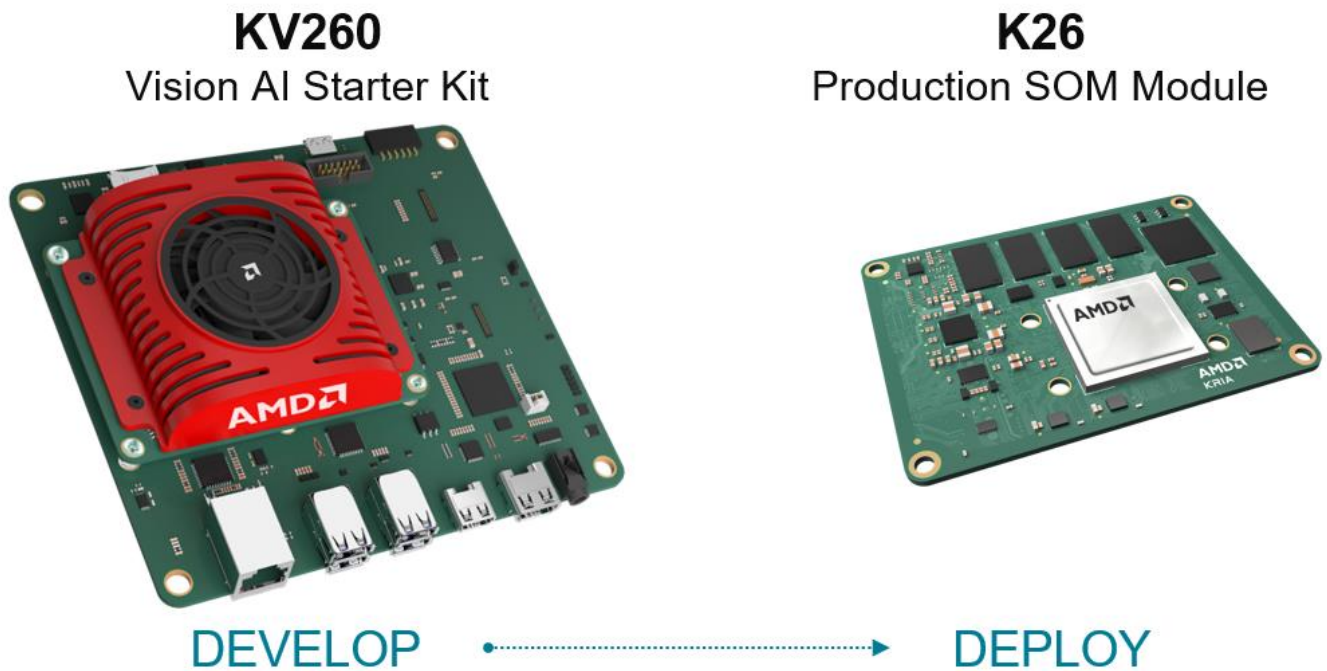
	AMD Vitis 統一軟體平臺	
	ROS 機器人作業系統	
	AMD Vitis AI	
	PYNQ - Python productivity for Zynq	
	Ubuntu 20.04 <ul style="list-style-type: none">● Install ROS Noetic and Docker.	
	Demo 影片.mp4 (命令列)	

Story

探索 AI 加速的巔峰之選：AMD 自適應性硬體加速平台與 Vitis AI 引領智能革命！

傳統上，我們大多看到用 GPU 和 CPU 開發出智能的移動車和手臂，但如何提升智能推理的效能又要在功耗散熱上取得平衡一直是工程師要面臨的挑戰和難題。

掌握智能加速的利器: AMD-Xilinx Kria™ K26 SOM/KV260 Vision Start Kit



靈活性和可編程性：AMD 第一款基於 Zynq® UltraScale+™ MPSoC 架構的 System-on-Modules (SOM)，具有豐富的接口和靈活的可編程性，您能夠根據自己的邊緣視覺應用進行配置和優化，讓您的自走車真正適應不同應用場景的需求。

高性能和能效平衡：KV260 可在性能和功耗之間取得平衡，提供高效的計算能力最高可達 1.4 TOPS。相比之下，GPU 在通用計算方面表現出色，但在靈活性和能效方面有限。

實時性能：KV260 專為實時應用設計，具備低延遲和高實時性能，可以滿足小型自走車 AMR 對快速決策和響應的需求。相比之下，GPU 和 CPU 可能在實時性能方面受到一定的限制，因為它們通常面向更廣泛的計算任務。



COMPUTE		INTERFACES	
Application Processor	64-bit Quad-Core Arm® Cortex®-A53	Camera	11 x4 Full MIPI or sub-LVDS Interfaces 1 x4 SLVS-EC Interfaces
Real-Time Processor	32-bit Dual-Core Arm Cortex-R5F	USB	4x USB 2.0 / 3.0
Graphics Processor	Arm Mali™-400MP2	Multi-Media	DisplayPort, HDMI
Programmable Logic	256K System Logic Cells	Network	1Gb up to 40Gb Ethernet (w/GigE Vision)
Deep Learning Processor	4K INT8 (upgradable to INT4)	Memory Interface	4GB 64-bit DDR4
Video Codec (H.264/H.265)	Up to 32 Streams (total resolution ≤ 4Kp60)	Transceivers	4x 12.5Gb/s, 4x 6Gb/s
Memory	26.6Mb On-Chip SRAM	Mechanical	77 x 60 x 11mm w/ dual 240-pin connectors
Security	IEC62443 Security w/HW Root-of-Trust		

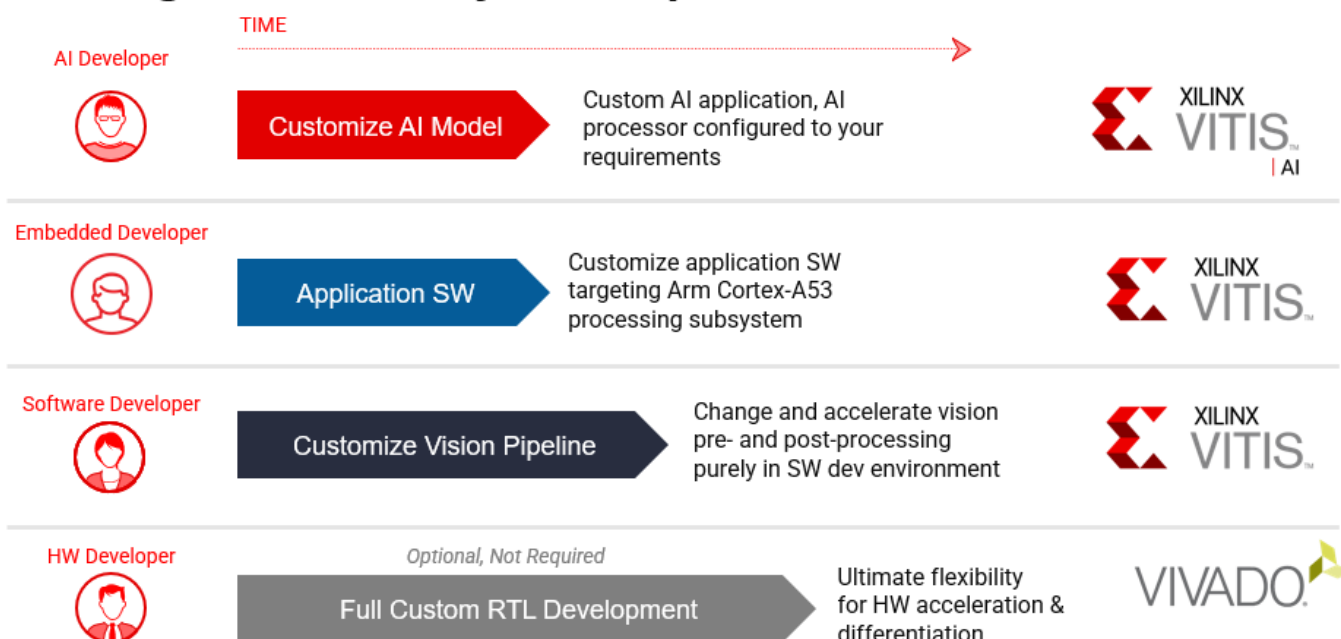
1.4 TOPs

使用 SOM 進行設計具有以下優勢：

- 通過使用經過可靠度驗證和綠色安規認證的量產核心電路降低風險。
- 將開發集中在您的軟體創新優勢上，從而加快產品上市時間。

開發工具和生態系統：除了卓越的硬件性能，AMD 還提供了全面的開發工具和庫，使開發人員能夠快速迭代、部署和優化 AI 應用。無論您是 AI 研究人員、軟/硬體工程師還是嵌入式開發者，AMD-Xilinx 都提供了豐富的資源和支持，幫助您輕鬆構建和優化自己的 AI 模型。

Design Path for Any Developer



賽靈思應用商店

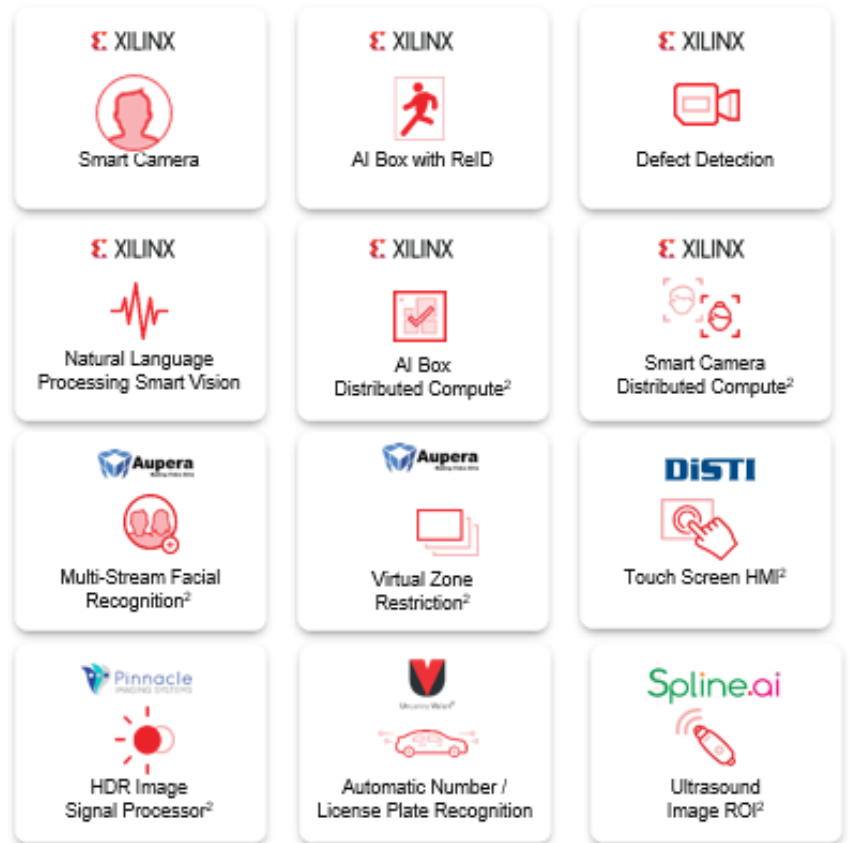
Xilinx 正在將基於 Kria™ SOM 的解決方案提供在一個集中位置，稱為 Xilinx App Store。

[Kria™ Apps on Xilinx App Store](#)

- [Smart Camera](#) by Xilinx
- [AI Box with ReID](#) by Xilinx
- [Defect Detection](#) by Xilinx

來自合作夥伴生態系統的另外兩款 Kria™ 加速應用程序

- [Facial Recognition](#) by Aupera Technologies
- [Phoenix HDR ISP Accelerated Application](#) by Xfuse



接下來，我們就以實際的小型自走車應用，引領大家進入 AMD-Xilinx AI 探索之旅。

Part 1: Tensorflow2 Training Step

Step 1：架設 Yolov4-tiny 的 Tensorflow2 訓練環境

● Tensorflow2 Docker

1. 安裝 docker for tensorflow-gpu：

```
docker pull tensorflow/tensorflow: 2.6.0-gpu
```

可以至[此處](#)搜尋版本

2. 安裝 docker for Cuda toolkit

```
sudo apt-get install -y nvidia-docker2  
sudo systemctl restart docker
```

```
sudo docker run --rm --gpus all nvidia/cuda: 11.4.0-base-ubuntu18.04 nvidia-smi
```

3. 啟動 tensorflow with cuda

```
docker run --gpus all -it --rm tensorflow/tensorflow: 2.6.0-gpu
```

4. 確認是否有運行到 gpu

```
python -c "import tensorflow as tf; print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

或是

```
python -c "import tensorflow as tf; print(tf.config.list_physical_devices('GPU'))"
```

5. 啟動並掛載當前工作資料夾到 Docker 中

```
docker run --gpus all --rm -v /home/norris/Tensorflow2:/home -it tensorflow:2.6.0-gpu /bin/bash
```

6. 有些 Package 在原始 Docker 中並未安裝，為防止之後 Run Code 報錯，可以先將其安裝

起來

```
pip install easydict  
pip install pillow  
pip install matplotlib  
pip install scipy  
pip install tqdm
```

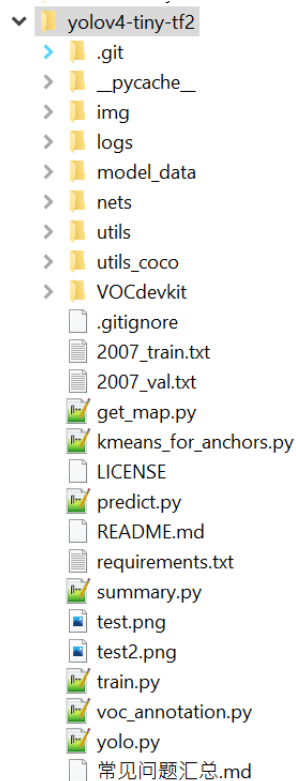
7. 若需要更新 Docker 內容避免每次開啟都需要重新安裝，可以參考以下指令

```
docker commit -m "Environment Set" containerID ImageName:Tags
```

● Yolov4-tiny model 修改

該 Project 所使用的 Yolov4-tiny Package 為[這篇 Github](#)

結構會類似於下圖



因為 Vitis AI 1.4.1 還未支援 Tensorflow split 的 operation，因此需更改模型的架構：

1. 在 `nets` ---> `CSPdarknet53_tiny.py` 中將第 78 行：

```
x = Lambda(route_group,arguments={'groups':2, 'group_id':1})(x)
```

更改為

```
x = DarknetConv2D_BN_Leaky(int(num_filters/2), (1,1), weight_decay=weight_decay)(x)
```

此步驟是將透過 1x1 Conv 取代掉 Tensorflow split

2. 在 `nets` ---> `CSPdarknet53_tiny.py` 中 `DarknetConv2D_BN_Leaky` 這個 Function 內，把

```
LeakyReLU(alpha=0.1)
```

更改為

```
ReLU(max_value=6)
```

Step 2：準備自我要訓練的資料集與標記

這一步會說明如何收集資料與標記資料

● 資料收集

我們可以透過撰寫 Python Code 來錄製 Camera 於 KV260 上的實時影像，而後再透過 FFMPEG

逐幀分析並保留要做標記的圖片

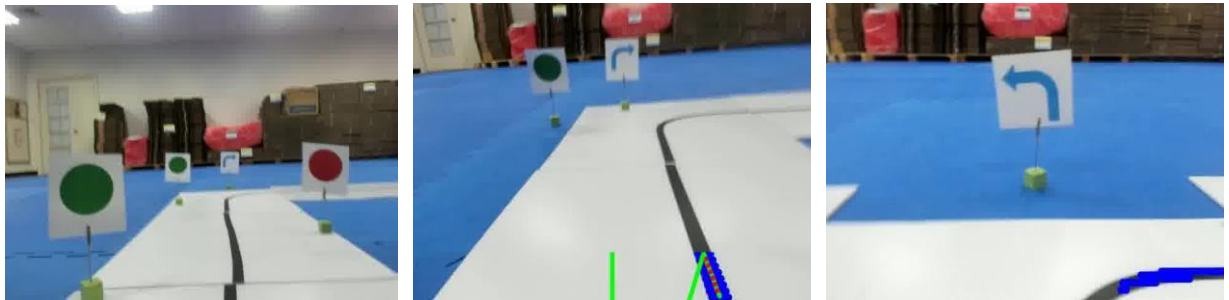
關於如何透過 Python Code 來從 KV260 端控制 STM32 傳送指令到馬達進行動作，可以參考 Part3

FFMPEG 的安裝與使用可以參考[該連結](#)

FFMPEG 裁切影片為圖片的指令可以透過以下：

```
ffmpeg -i test.mp4 -vf fps=1 'test-screenshot-%04d.png'
```

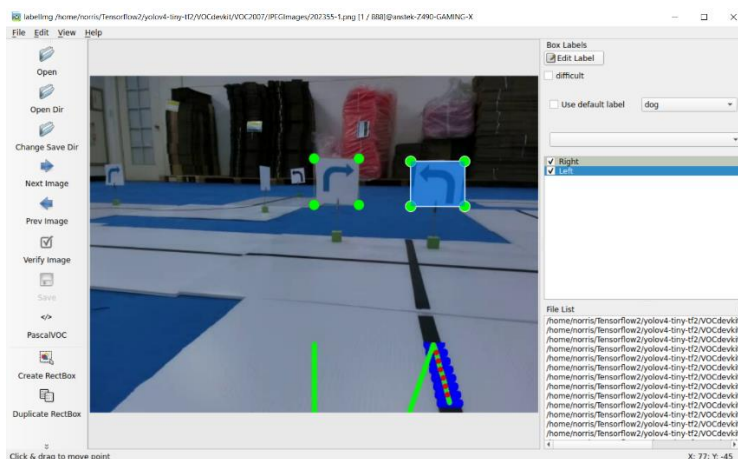
有效的圖片資料會類似下列範例



➤ Labellmg

請直接參考 [Labellmg 的 Github 教學](#)

Label 的過程會像下列畫面範例



Step 3：訓練前設定

1. 修改 voc_annotation.py 裡面的 annotation_mode=2

此舉會透過 VOCdevkit/VOC2007/JPEGImages 生成訓練用的

2007_train.txt 和 2007_val.txt

修改 `classes_path`，指向自我訓練集的 label txt 檔案，一般在 `model_data` 中

2. 訓練結果預測設定需要更改

yolo.py 中的 model_path 以及 classes_path

model_path 指向訓練好的權重，一般在 logs 中

classes_path 指向自己訓練及的 label txt 檔，一般在 model_data 中

3. 數據集的放置

本篇使用 VOC 格式進行訓練

訓練前將標籤文件放置到 VOCdevkit/VOC2007 文件夾下的 Annotation

可透過 LabelImg 設定 label 路徑

訓練前將圖片文件放置到 VOCdevkit/VOC2007 文件夾下的 JPEGImages

4. 產生數據並執行訓練

執行 voc_annotation.py 獲得訓練用的 2007_train.txt 和 2007_val.txt

執行 train.py 進行訓練，訓練中或結束的結果會保存在 logs 中

建議使用 pre-trained model weight 來提高與減少訓練的精度與時間

Part 2: Vitis AI Design Flow

Step 1: Vitis AI Docker

1. 從 Vitis AI Docker Image 抓取 Vitis AI 1.4.1

```
docker pull xilinx/vitis-ai:1.4.1.978
```

2. 從 github 抓 Vitis AI 1.4.1 包

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

3. 啟動 Vitis AI 1.4.1 Docker 環境

```
cd Vitis-AI-1.4.1/  
sudo ./docker_run.sh xilinx/vitis-ai:1.4.1.978  
conda activate vitis-ai-tensorflow2
```


Step 2 : Vitis AI Example Code

可下載官方 [Vitis AI Lab](#) 中轉換 model 的 shell 來當作參考

1. Quantize

打開 1_tf2_quantize.sh 可以看到量化 code 是用 code/com 中的 train_eval_h5.py

```
norris@ubuntu:~/Vitis-AI-2.0/vai_q_c_tf2_pt/lab$ ls -lh
total 100K
-rw-rw-r-- 1 norris norris 3.7K Aug 28 20:22 1_tf2_quantize.sh
-rw-rw-r-- 1 norris norris 3.4K Aug 28 20:22 2_tf2_eval_float_graph.sh
-rw-rw-r-- 1 norris norris 3.5K Aug 28 20:22 3_tf2_eval_quantize_graph.sh
-rw-rw-r-- 1 norris norris 3.4K Aug 28 20:22 4_tf2_compile_for_mpsoc.sh
-rw-rw-r-- 1 norris norris 3.3K Aug 28 20:22 5_pytorch_quantize.sh
-rw-rw-r-- 1 norris norris 3.4K Aug 28 20:22 6_pytorch_compile_for_mpsoc.sh
drwx----- 29 norris norris 36K Aug 28 20:22 images
drwx----- 2 norris norris 4.0K Aug 28 20:22 pt_resnet50
-rw-rw-r-- 1 norris norris 12K Aug 28 20:22 resnet50_quantize.py
drwx----- 6 norris norris 4.0K Aug 28 20:22 tf2_resnet50_imagenet_224_224_7.76G_2.0
-rw-rw-r-- 1 norris norris 17K Aug 28 20:22 val.txt
```

可參考此檔案自行撰寫一 quantize 自己 model 的 python 檔

相關 Vitis AI Python API 可參考該[官方網站](#) (UG1414) Quantizing the Model 章節說明

當中會需要 import 原本 yolov4-tiny github 中部分資料夾內文件，如以下：

```
from dataset import synth_input_fn
from dataset import input_fn, NUM_IMAGES
from dataset import get_images_infor_from_file, ImagenetSequence
from nets.yolo import yolo_body
```

nets.yolo 是原本於 yolov4-tiny github 中的 source code，這邊 import 是因為 quantize 還是需

要原始 model 的 structure，請將 yolov4-tiny github 中的 nets 資料夾全部複製過來 Vitis AI 當

前 quantize code 資料夾中

除此之外，需要注意該 Example Code 中的 dataset 主要處理 Pre-Process 的 Function

稱為 `ImagenetSequence`，需要更改的有幾段：

```
for filename in batch_x:
    # B G R format
    img = cv2.imread(filename)
    height, width = img.shape[0], img.shape[1]
    img = img.astype(float)

    # aspect_preserving_resize
    smaller_dim = np.min([height, width])

    # _RESIZE_MIN = 256
    _RESIZE_MIN = 416

    scale_ratio = _RESIZE_MIN*1.0 / (smaller_dim*1.0)
    new_height = int(height * scale_ratio)
    new_width = int(width * scale_ratio)
    resized_img = cv2.resize(img, (new_width, new_height), interpolation = cv2.INTER_I

    # central_crop

    # crop_height = 224
    # crop_width = 224

    crop_height = 416
    crop_width = 416

    # sub mean

    # _R_MEAN = 123.68
    # _G_MEAN = 116.78
    # _B_MEAN = 103.94

    _R_MEAN = 128.0
    _G_MEAN = 128.0
    _B_MEAN = 128.0

    _CHANNEL_MEANS = [_B_MEAN, _G_MEAN, _R_MEAN]
    means = np.expand_dims(np.expand_dims(_CHANNEL_MEANS, 0), 0)
    meaned_img = cropped_img - means

    meaned_img /= 255.0

    # model.predict(np.expand_dims(meaned_img, 0))
    # model.evaluate(np.expand_dims(meaned_img, 0), np.expand_dims(labels[0], 0))
    processed_imgs.append(meaned_img)
```

注意

該 Data Pre-Process 若未與 GPU Training 時的方式一樣，會造成之後 Compile 後的 Model 無法成功達到預期的 Accuracy

2. Compile

修改 `4_tf2_compile_for_mpsoc.sh` 中的設定

包含輸出路徑、開發版形式、`arch.json` 中的 DPU 形式以及輸出檔案名稱，如下圖：

```
export TF2_NETWORK_PATH='tf2_resnet50_imagenet_224_224_7.76G_1.4'

vai_c_tensorflow2 -m ${TF2_NETWORK_PATH}/code/com/car_quantized_yolov4-tiny-relu-0508.h5 \
-a /opt/vitis_ai/compiler/arch/DPUCZDX8G/KV260/arch.json \
-o ${TF2_NETWORK_PATH}/vai_c_output \
-n car-yolov4-tiny-0508 \
```

相關參數設定一樣可以參考[官方網站](#) (UG1414) Compiling the Model 章節

該步驟時常會發生幾個問題，以下會列出常見的狀況提供參考：

➤ [UNILog][FATAL][XIR_MULTI_DEFINED_OP][Multiple definition of OP!] quant_max_pooling2d_1

```
[INFO] Namespace(batchsize=1, inputs_shape=None, layout='NHWC', model_files=['tf2_resnet50_imagenet_224_224_7.76G_1.4/code/com/car_quantized_yolov4-tiny-relu-0508.h5'], model_type='tensorflow2', named_inputs_shape=None, out_filename='/tmp/car-yolov4-tiny-0508_org.xmodel', proto=None)
[INFO] tensorflow2 model: /workspace/vai_q_tf2_pt/lab/tf2_resnet50_imagenet_224_224_7.76G_1.4/code/com/car_quantized_yolov4-tiny-relu-0508.h5
[INFO] keras version: 2.4.0
[INFO] Tensorflow Keras model type: functional
[INFO] parse raw model :100%| 61/61 [00:00<00:00, 14734.65it/s]
[INFO] infer shape (NHWC) :100%| 96/96 [00:00<00:00, 199.63it/s]
[INFO] perform level-0 opt :100%| 2/2 [00:00<00:00, 294.71it/s]
[INFO] perform level-1 opt :100%| 2/2 [00:00<00:00, 1183.16it/s]
[INFO] generate xmodel :100%| 96/96 [00:00<00:00, 542.03it/s]
[INFO] dump xmodel: /tmp/car-yolov4-tiny-0508_org.xmodel
[UNILog][INFO] Target architecture: DPUCZDX8G_ISA0_B4096_MAX_BG2
[UNILog][INFO] Compile mode: dpv
[UNILog][INFO] Debug mode: function
[UNILog][INFO] Target architecture: DPUCZDX8G_ISA0_B4096_MAX_BG2
[UNILog][INFO] Graph name: functional_1, with op num: 192
[UNILog][INFO] Begin to compile...
[UNILog][FATAL][XIR_MULTI_DEFINED_OP][Multiple definition of OP!] quant_max_pooling2d_1
*** Check failure stack trace: ***
Aborted (core dumped)
```

該問題是因為 Vitis AI 在轉換模型中與原先定義的模型 Layer 名稱衝突，因此需要更改轉換後的

xmodel 並重新採用 Pytorch 的 compile 方式進行

✓ Solution

A. 將/tmp/car-yolov4-tiny-0508_org.xmodel 複製出來，並以 B 步驟的 code 更改 Layer 名稱

B. 轉換 Layer Name Code 如下：

```
import xir
g = xir.Graph.deserialize("car-yolov4-tiny-0508_org.xmodel")
ops = g.toposort()
for op in ops:
    if op.get_name() == "quant_max_pooling2d":
        replace_pool = g.create_op("quant_max_pooling2d_0", op.get_type(), op.get_attrs(), op.get_input_ops())
        [succ.replace_input_ops(op, replace_pool) for succ in op.get_fanout_ops()]
        g.remove_op(op)
g.serialize("renamed_quantize_model.xmodel")
```

C. 先將原先 Tensorflow 環境退出，再開啟 Pytorch 環境

```
conda deactivate
conda activate vitis-ai-pytorch
```

D. 執行 Pytorch Compile 轉換 Shell

```
#!/bin/sh
TARGET=MPSOC
NET_NAME=yolov4-tiny
DEPLOY_MODEL_PATH=vai_q_output
```

```
ARCH=/opt/vitis_ai/compiler/arch/DPUCZDX8G/KV260/arch.json
```

```
vai_c_xir -x tf2_resnet50_imagenet_224_224_7.76G_1.4/code/com/renamed_quantize_model.xmodel \  
-o pt_yolov4-tiny/vai_c_output/ \  
-n car_pt_yolov4-tiny-0508 \  
-a ${ARCH}
```

E. 最後的 xmodel 就可以丟到 FPGA 上驗證了

➤ **Total device subgraph number 49, DPU subgraph number 24**

這問題是 model 內有些 operation 不被當前 Vitis AI DPU 所支持，因此在 compile 的時候，一部分的 operation 被分到 CPU 去做執行，可以看到 DPU graph 被分成多個 subgraph，如果直接丟到 FPGA 上執行，會發生錯誤，因單一 DPU 只吃一個 graph，加上多個 subgraph 並沒有被整合成單一 graph 而導致

✓ **Solution**

A. 更改原先 GPU model operation 並且重新 training

B. 參考官方 [Graph Runner 流程與範例](#)

Part 3: KV260 Ubuntu Environment

Step 1: Install Ubuntu 20.04 on KV260

1. Download Ubuntu 20.04 and put it in SDcard through balenaetcher
2. Boot Ubuntu 20.04 from KV260
3. Update Ubuntu package
4. `sudo apt update`
5. `sudo apt upgrade`
6. Install Xilinx system management snap package ---> `sudo snap install xlnx-config --classic --channel=1.x`

Step 2: Install Wi-Fi Driver

```
sudo apt install -y dkms git build-essential  
mkdir repos; cd repos  
git clone https://github.com/morrownr/88x2bu-20210702  
cd 88x2bu-20210702  
sudo ARCH=arm64 ./install-driver.sh
```

測試 Wi-Fi Driver

```
iwconfig
lo          no wireless extensions.

eth0        no wireless extensions.

sit0        no wireless extensions.

wlx1cbfcee7b IEEE 802.11  ESSID:off/any
            Mode:Managed  Access Point: Not-Associated   Tx-Power=-100 dBm
            Retry short limit:7   RTS thr:off   Fragment thr:off
            Power Management:on
```

notice: wlx1cbfcee7b 為偵測到的 Wi-Fi 裝置，每個人不一定一樣

接著打開 Wi-Fi 並搜尋目前能找到的熱點資訊

```
sudo apt install net-tools
sudo ifconfig wlx1cbfcee7b up
sudo iwlist wlx1cbfcee7b scan | grep ESSID
```

有搜尋到的話會像以下

```
ubuntu@kria:~$ sudo iwlist wlx1cbfcee7b scan | grep ESSID
            ESSID:"BELL113"
            ESSID:"ramduq"
            ESSID:"BELL709"
            ESSID:"BELL113"
            ESSID:"ramduq"
            ESSID:""
            ESSID:"BELL709"
            ESSID:"HP-Print-BB-Officejet Pro X476dw"
```

連接 Wi-Fi，將 Wi-Fi 設定寫入到 config 內

```
wpa_passphrase your-ESSID your-passphrase | sudo tee /etc/wpa_supplicant.conf
eg. wpa_passphrase "BELL709" 12345678 | sudo tee /etc/wpa_supplicant.conf
```

連接

```
sudo wpa_supplicant -c /etc/wpa_supplicant.conf -i wlx1cbfcee7b -B
```

連接成功之後給它一個 IP

```
sudo dhclient wlx1cbfcee7b
```

測試網路是否有通

```
ping 8.8.8.8
```

成功的話會像下圖

```
ubuntu@kria:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=8.19 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=7.45 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=116 time=8.51 ms
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 7.453/8.052/8.511/0.443 ms
```

Step 3: Install ROS1 in Ubuntu 20.04

直接下載以下檔案

```
git clone https://piobot_stm32:piobot-zeus@code.aliyun.com/surewin1102/piobot_ros.git
```

下載之後 cd 到 piobot_ros

```
cd piobot_ros
```

更改 piobot_install_ros.sh 130 行後的內容為以下：

```
# read -s -n1 -p "install ros gui tools?(y/N)"
# if [ "$REPLY" = "y" -o "$REPLY" = "Y" ]; then
sudo apt-get -y --allow-unauthenticated install ros-${ROS_DISTRO}-rviz ros-${ROS_DISTRO}-rqt-reconfigure ros-${ROS_DISTRO}-rqt-tf-tree \
ros-${ROS_DISTRO}-image-view

if [ "$ROS_DISTRO" = "noetic" ]; then
    echo "please run ros_package/make_cartographer.sh to compile cartographer"
else
    sudo apt-get -y --allow-unauthenticated install ros-${ROS_DISTRO}-cartographer-rviz
fi
# fi
```

接著執行 sh piobot_install_ros.sh 進行安裝

```
sh piobot_install_ros.sh
```

安裝結束後在 piobot_ros 資料夾內進行 ./piobot_init_env.sh 以進行 piobot_ros 的環境設定

```
./pibot_init_env.sh
```

會要求輸入數字，依序為

```
0 0 2 3 0
```

端看硬體搭配 ROS 系統的實際設計

輸入結束後要記得 source 環境變數：

```
source ~/.bashrc
```

再來要 make pibot_ros 形成動態連結檔

```
cd pibot_ros/ros_ws  
catkin_make
```

再來等 make 結束後再 source ~/.bashrc 一次就可以了

```
source ~/.bashrc
```

Step 4: Update Ubuntu Kernel

按照 Xilinx 官方 Wiki 去做 custom Ubuntu function modified：

[Rebuilding the Certified Ubuntu for Xilinx Devices 20.04 LTS Kernel from Source - Xilinx Wiki - Confluence \(atlassian.net\)](https://www.xilinx.com/support/answers/69029.html)

需要注意的是你需要創建一個虛擬機（Ubuntu 20.04）或是一台 Ubuntu 20.04 的主機來去 build

你所要放到 KV260 Ubuntu 20.04 上的安裝檔案

當中執行到 fakeroot debian/rules editconfigs 這個步驟時，要去開啟 cp210x driver 的功能，主要

是開啟以下兩者：

- Device Drivers -> USB support -> USB Serial Converter support
- Device Drivers -> USB support -> USB Serial Converter support -> USB CP210x family of UART Bridge Controllers

上述連結內都是在另外一台 Ubuntu 20.04 上作執行，到最後產生出來的五個檔案如下：


```
linux-buildinfo-5.4.0-1017-xilinx-zynqmp_5.4.0-1017.20_arm64.deb
linux-headers-5.4.0-1017-xilinx-zynqmp_5.4.0-1017.20_arm64.deb
linux-image-5.4.0-1017-xilinx-zynqmp_5.4.0-1017.20_arm64.deb
linux-modules-5.4.0-1017-xilinx-zynqmp_5.4.0-1017.20_arm64.deb
linux-xilinx-zynqmp-headers-5.4.0-1017_5.4.0-1017.20_all.deb
```

需要將這些檔案放到 KV260 Ubuntu 20.04 做安裝，安裝的指令為：

```
sudo dpkg -i *.deb
```

記得安裝完後 reboot，就可以認到 STM32F4 的 cp210x serial 了

因為接下來的步驟會使用由 AMD 開發的開源框架 – PYNQ，因此這邊也概略簡介甚麼是 PYNQ。

PYNQ 旨在將 Python 編程語言與 Xilinx Zynq SoCs 結合使用，提供更高的

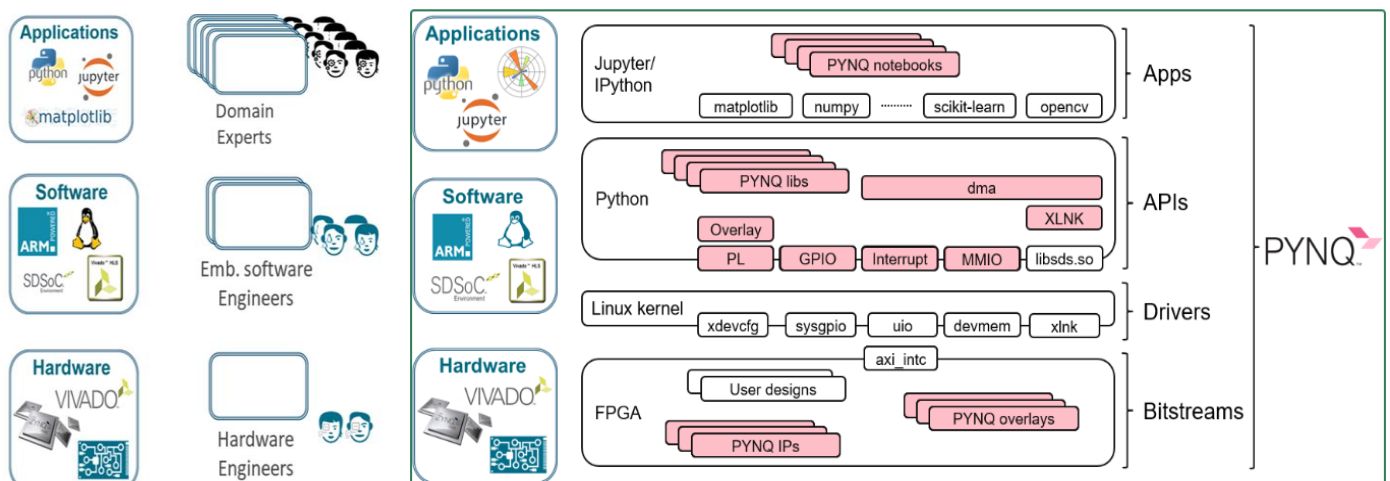
開發效率和便捷性。開發環境以 Jupyter Notebook 為基礎，可解決

FPGA 編程能力的問題，即便是 FPGA 經驗較少的開發人員，也可快速實作能充分發揮 FPGA 效能

優勢的設計，加速運算 AI 應用的開發。可支持基於 Zynq 的開發板（Zynq、Zynq Ultrascale+、Zynq

RFSoc）、Kria SOM、Xilinx Alveo 加速器板和 AWS-F1。[Downloadable PYNQ images](#)

下圖為 PYNQ 開發框架，軟體連接硬體層、驅動層和應用層之間的接口設計：



其中 PYNQ 覆蓋層（overlays）是用 PYNQ 框架加載到 Zynq SoC 可編程邏輯部分的硬件設計（通常稱

為“位流”）。覆蓋層指的是定制的位流，旨在為 Python 編程環境提供特定的功能和接口。這些覆蓋層

通常由可編程邏輯電路組成，例如自定義加速器、IP 核或預設計的硬件模塊，可以通過運行在 Zynq

SoC 的 ARM 處理器上的 Python 腳本動態加載和控制。PYNQ 框架提供了一組 API 和庫，使 Python 程式員可以直接從其 Python 代碼中與覆蓋層交互並訪問其功能。這使開發人員能夠快速原型化和部署利用 Zynq SoC 的可編程邏輯和軟件功能的複雜系統。總體而言，PYNQ Overlays 是 PYNQ 框架的關鍵組成部分，其中 KV260 便可以透過安裝 PYNQ-DPU overlays 來實現 Python 軟體層面的開發，並簡單的透過 API 驅動 FPGA 韌體，以下開始則為安裝流程以及使用的範例。

Step 5: Install PYNQ-DPU Overlay

1. Download the full package from Xilinx DPU-PYNQ github

```
ubuntu@kria:~$ git clone https://github.com/Xilinx/Kria-PYNQ.git
Cloning into 'Kria-PYNQ'...
remote: Enumerating objects: 93, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (37/37), done.
remote: Total 93 (delta 27), reused 14 (delta 10), pack-reused 46
Unpacking objects: 100% (93/93), 1.26 MiB | 2.38 MiB/s, done.
```

2. Install PYNQ package.

```
ubuntu@kria:~/Kria-PYNQ$ sudo bash install.sh -b KV260

This version of Kria-PYNQ is not compatible with Ubuntu 20.04 please checkout tag v1.0 with the command

git checkout tags/v1.0
```

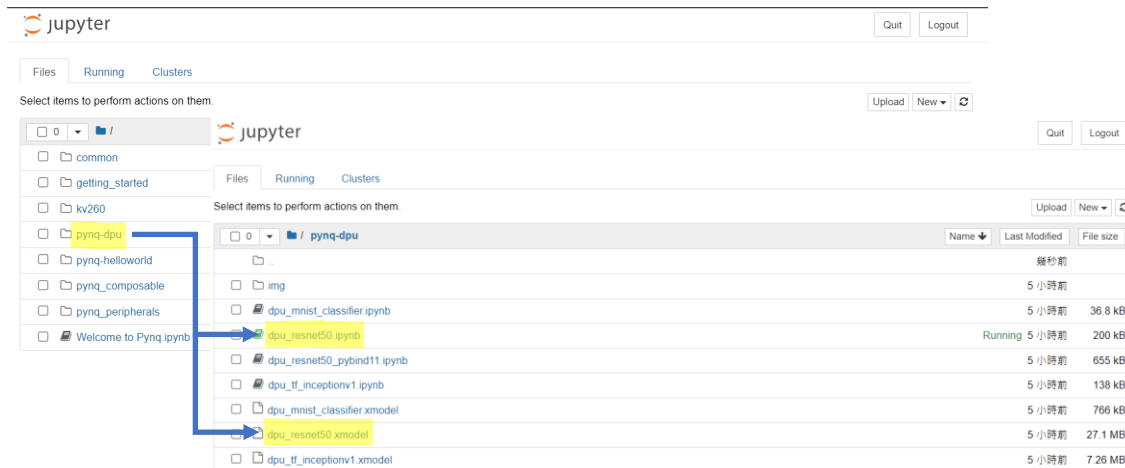
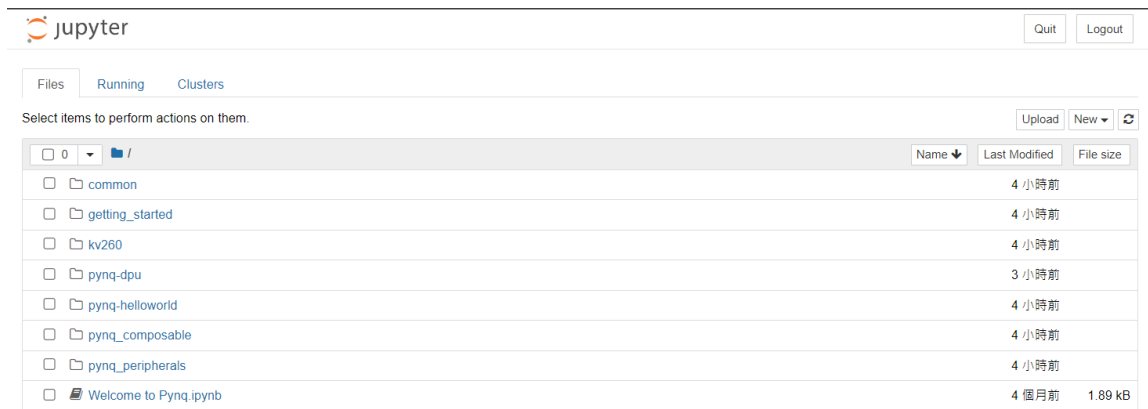
3. PYNQ-DPU has been installed successfully.

```
You should consider upgrading via the '/usr/local/share/pynq-venv/bin/python3 -m pip install --upgrade pip' command.
/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq/pl_server/xrt_device.py:59: UserWarning: xbutil failed to run - unable to determine XRT version
  warnings.warn("xbutil failed to run - unable to determine XRT version")
The following notebooks modules will be delivered:
- pynq_peripherals (source: pynq_peripherals)
- pynq-helloworld (source: pynq_helloworld)
- pynq-dpu (source: pynq_dpu)
- pynq_composable (source: pynq_composable)
- kv260 (source: kv260)
Do you want to proceed? [Y/n] Delivering notebooks '/home/root/jupyter_notebooks/pynq_peripherals'...
Delivering notebooks '/home/root/jupyter_notebooks/pynq-helloworld'...
Downloading file 'resizer.hwh'. This may take a while...
Downloading file 'resizer.bit'. This may take a while...
Delivering notebooks '/home/root/jupyter_notebooks/pynq-dpu'...
Delivering notebooks '/home/root/jupyter_notebooks/pynq_composable'...
Delivering notebooks '/home/root/jupyter_notebooks/kv260'...
PYNQ Installation completed.

To continue with the PYNQ experience, connect to JupyterLab via a web browser using this url: 192.168.1.52:9090/lab or kria:9090/lab - The password is xilinx

ubuntu@kria:~/Kria-PYNQ$
```

4. Open Jupyter notebook on web browser



5. Reference the DPU Example Code

可以透過 Jupyter notebook 提供的 `dpu_resnet50.ipynb` 來參考如果調用 DPU 與原先的 model data pre-process 進行配合，來達到 AI cooperate with FPGA 的結果

Part 4: Run Project

此處分為兩個部分

一個是在 KV260 上執行的主程式，負責由 Camera 接收影像到 FPGA 內部，由 DPU 讀取 AI Model 對接收到的影像進行 Inference，並與 OpenCV 循環算法搭配，來去根據當前分析的結果，透過 ROS，傳送動作指令給 STM32 控制馬達

[KV260 Source Code](#)

第二個是於 PC 端上的副程式，主要用於接收 KV260 上處理後的影像，來提供使用者觀察目前檢測的結果是否有如預期

[PC Source Code](#)

以下分為兩部分，分別敘述如何執行 KV260 與 PC 上的 Code

Step 1: Set and Run KV260 Source Code

1. 設定 ROS 溝通環境

```
pibot_init_env  
0 0 2 3 0  
source ~/.bashrc
```

2. 啟動 ROS 與 STM32 馬達溝通

```
pibot_bringup
```

若遇到以下問題，則需要重新更新 Ubuntu Kernel 並 reboot

```
sudo dpkg -i *.deb  
sudo reboot
```

```
setting /run_id to 62a8af3c-0668-11ee-b748-c13585b74e3f  
process[rosout-1]: started with pid [2414]  
started core service [/rosout]  
process[pibot_driver-2]: started with pid [2417]  
[ INFO] [1686275593.003159022]: port:/dev/pibot baudrate:921600  
[ INFO] [1686275593.015067157]: out_pid_debug_enable:0  
[ INFO] [1686275593.020492896]: BaseDriver startup ...  
[ INFO] [1686275593.020826085]: open /dev/pibot 921600  
[ERROR] [1686275593.020972480]: open /dev/pibot err  
[ERROR] [1686275593.021061208]: oops!!! can't connect to main board, please check the usb connection or baudrate!  
[ INFO] [1686275593.171365957]: read err -1  
[ INFO] [1686275593.321706756]: read err -1  
[ INFO] [1686275593.472031455]: read err -1  
[ INFO] [1686275593.622316935]: read err -1  
[ INFO] [1686275593.772635365]: read err -1  
[ INFO] [1686275593.923007082]: read err -1  
[ INFO] [1686275594.073360171]: read err -1
```

3. 執行 KV260 Source Code

```
sudo su -  
python tiny-yolov4-camera-ros-node.py
```

Step 2: Set and Run PC Source Code

1. 設定 ROS 溝通環境

```
pibot_init_env  
0 0 2 3 1  
  
輸入 KV260 上的 IP 位址  
  
source ~/.bashrc
```

2. 執行 PC Source Code

```
python camera_test.py
```