FaDAS

1) Introduction

This project is about running and accelerating vehicle detection and mapping ADAS applications in parking lot using Deep Learning Unit (DPU) on KV260 FPGA board.

1.1) Vitis-AI

The Vitis AI Library is a set of high-level libraries and APIs built for efficient AI inference with DPU cores. It is built based on the Vitis AI Runtime (VART) with unified APIs and provides easy-to-use interfaces for AI model deployment on AMD platforms.

1.2) Deep Learning Unit

DPUCZDX8G was used in this project. The DPUCZDX8G is the Deep learning Processor Unit (DPU) designed for the Zynq® UltraScale+™ MPSoC. It is a configurable computation engine optimized for convolutional neural networks. The degree of parallelism utilized in the engine is a design parameter and can be selected according to the target device and application. At a high-level, the DPU is a microcoded compute engine which has an efficient, optimized instruction set, and which can support inference of most convolutional neural networks.

1.3) Petalinux

PetaLinux is an embedded Linux Software Development Kit (SDK) targeting FPGA-based system-on-a-chip (SoC) designs or FPGA designs.

1.4) Kria KV260

The Kria KV260 FPGA board, developed by Xilinx, is designed for various applications such as artificial intelligence, image processing, embedded systems, and industrial applications. It is optimized to accelerate the prototyping and development process.

1) System Architecture

The system consists of a 2D lidar, a camera, a rotary encoder to provide the third axis to the 2D lidar and create a 3D point cloud. The rotary encoder was programmed using Arduino and communicated with the KV260 FPGA board using the I2C bus. FPGA Architecture is given in Figure 2.

The general system design has been implemented in five steps. These are:

- The Vivado project step where the hardware design is created
- The Linux project part created for the hardware obtained in the Vivado project
- Reconstructed models from the model zoo for the hardware obtained in the Vivado project
- Creating the Device Tree Overlay
- Loading and running the Linux system application code and model on the development board

The overall system architecture is given in Figure 1.

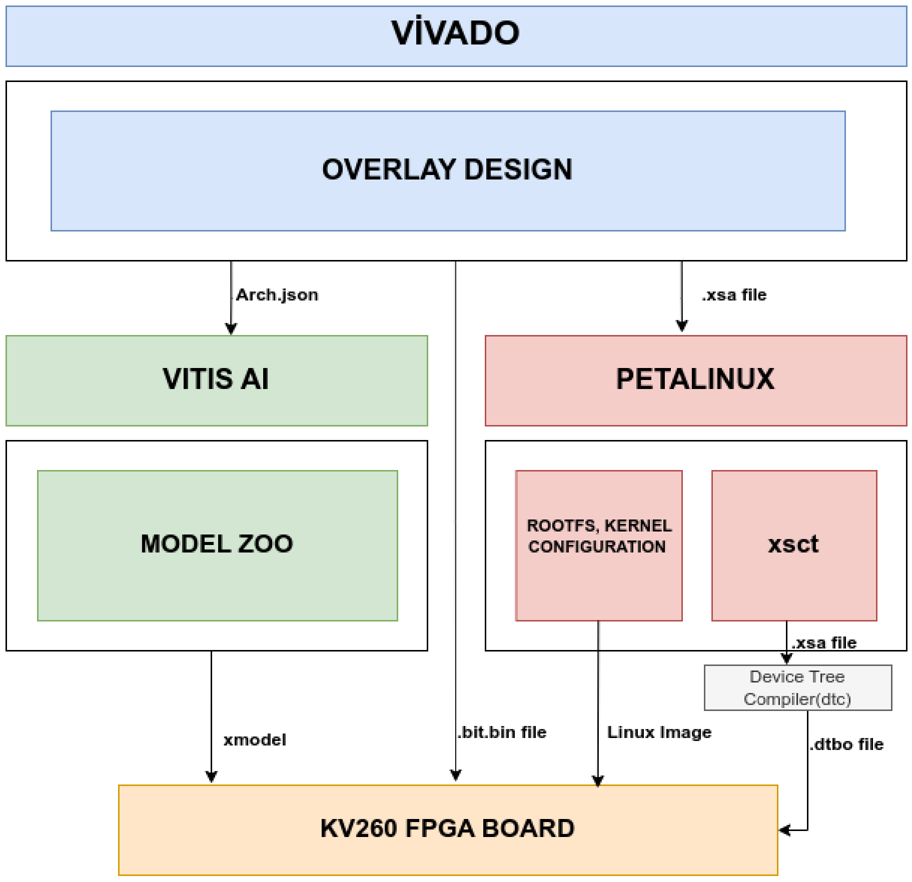


Figure 1: System Architecture

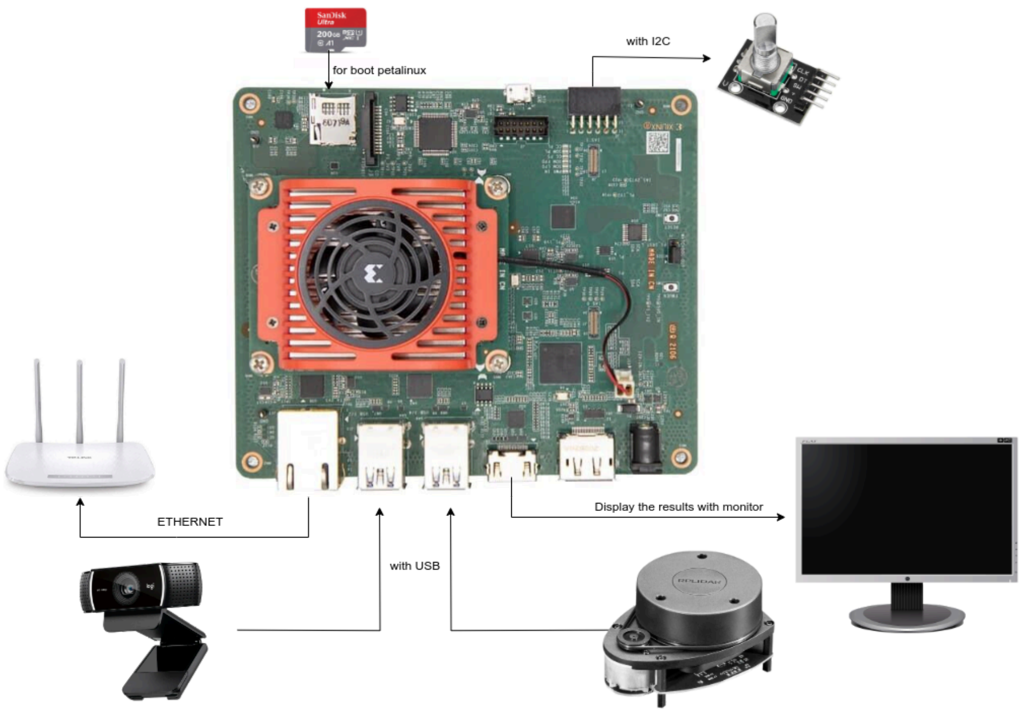


Figure 2: KV260 Connection

2.1) Hardware Design

The hardware architecture includes I2C IP, ZynqMP PS, DPU, System resets, Clock generators, Interrupt controllers, AXI connectors to control one rotary encoder. The overlay is shown in Figure2.

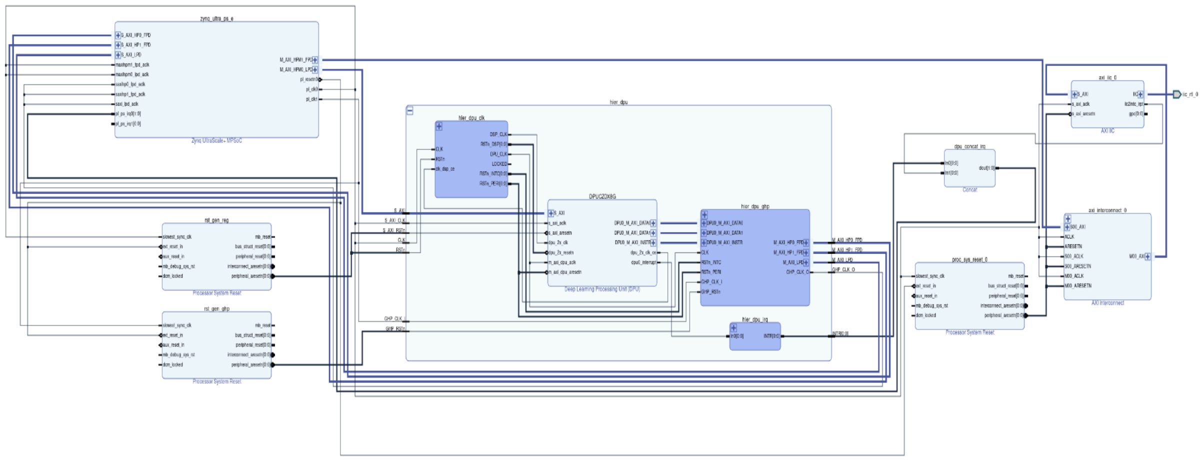


Figure 2: Overlay Design

I2C sda and scl pins must be connected to the pmods of the KV260 board. Constrait file has been updated according to the pmod pins Figure 3 and Figure 4:

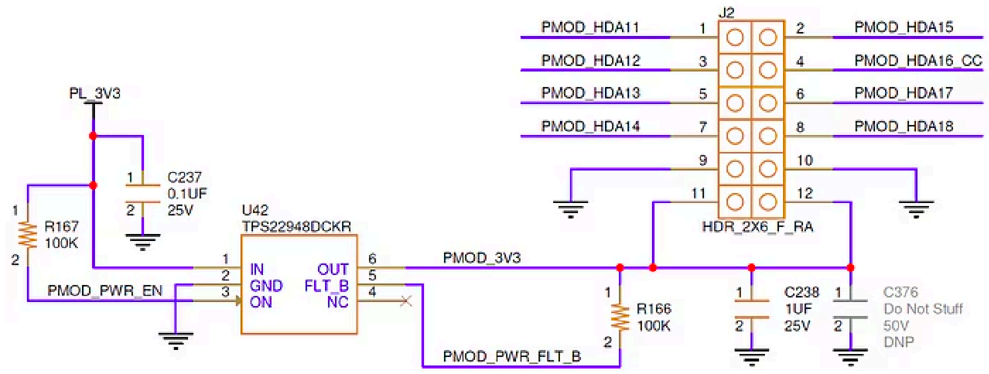
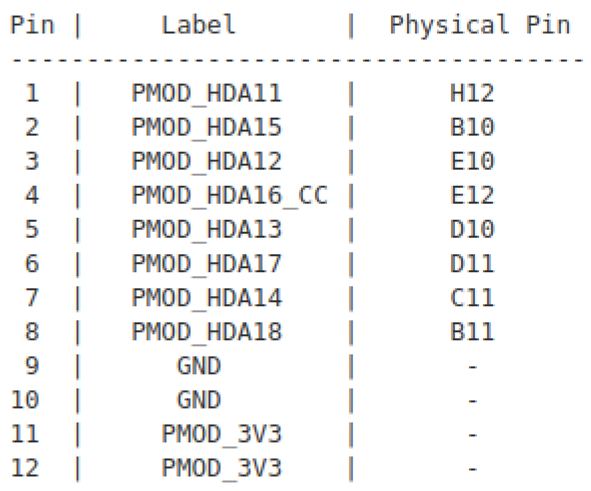


Figure 3: KV260 Pmod connection



```
Pin |     Label        |  Physical Pin
--------------------------------------
 1  |  PMOD_HDA11      |      H12
 2  |  PMOD_HDA15      |      B10
 3  |  PMOD_HDA12      |      E10
 4  |  PMOD_HDA16_CC   |      E12
 5  |  PMOD_HDA13      |      D10
 6  |  PMOD_HDA17      |      D11
 7  |  PMOD_HDA14      |      C11
 8  |  PMOD_HDA18      |      B11
 9  |     GND          |       -
10  |     GND          |       -
11  |    PMOD_3V3      |       -
12  |    PMOD_3V3      |       -
```

Figure 4: KV260 Pmod pins

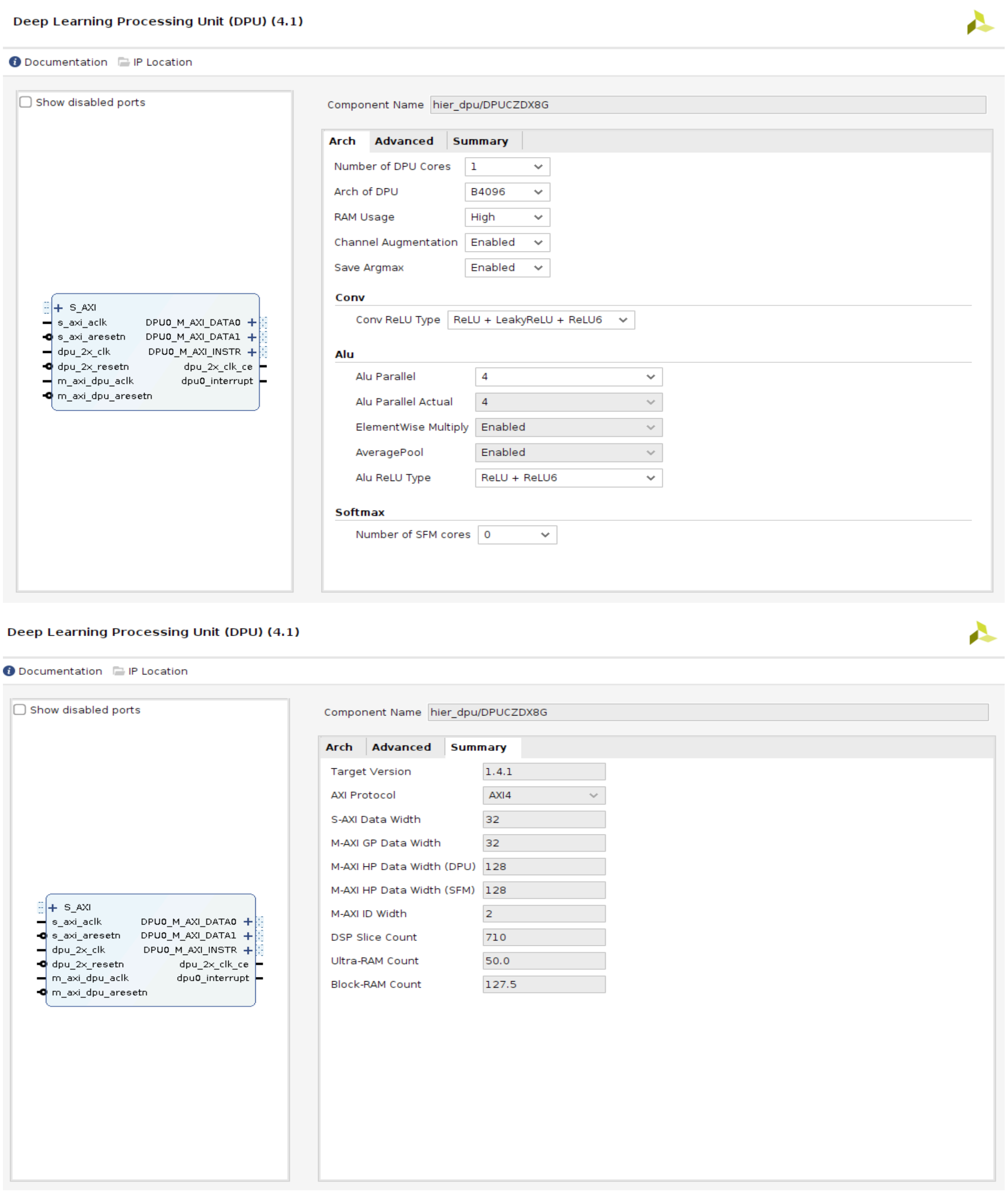
The Constrait file will be as follows.

```
set_property PACKAGE_PIN H12 [get_ports iic_rtl_scl_io] ;
set_property PACKAGE_PIN B10 [get_ports iic_rtl_sda_io] ;

set_property PULLUP true [get_ports iic_rtl_scl_io]
set_property PULLUP true [get_ports iic_rtl_sda_io]

set_property IOSTANDARD LVCMOS33 [get_ports {iic_rtl_scl_io}]
set_property IOSTANDARD LVCMOS33 [get_ports {iic_rtl_sda_io}]
```

The SCL (clock) and SDA (data) lines used in I2C communication usually have open collector or open drain outputs and these lines need pull-up resistors to operate correctly. By enabling the internal pull-up resistors in the FPGA, these lines can be kept at a certain voltage level at all times, ensuring correct operation.

The set_property PULLUP true commands in the given constraint file enable the internal pull-up resistors for the pins connected to the iic_rtl_0_scl_io and iic_rtl_0_sda_io ports. This ensures correct operation of the I2C line.

DPU IP configuration is shown in Figure 5.



Component Name: hier_dpu/DPUCZDX8G

Number of DPU Cores	1
Arch of DPU	B4096
RAM Usage	High
Channel Augmentation	Enabled
Save Argmax	Enabled
Conv	
Conv ReLU Type	ReLU + LeakyReLU + ReLU6
Alu	
Alu Parallel	4
Alu Parallel Actual	4
ElementWise Multiply	Enabled
AveragePool	Enabled
Alu ReLU Type	ReLU + ReLU6
Softmax	
Number of SFM cores	0

Component Name: hier_dpu/DPUCZDX8G

Target Version	1.4.1
AXI Protocol	AXI4
S-AXI Data Width	32
M-AXI GP Data Width	32
M-AXI HP Data Width (DPU)	128
M-AXI HP Data Width (SFM)	128
M-AXI ID Width	2
DSP Slice Count	710
Ultra-RAM Count	50.0
Block-RAM Count	127.5

Figure 5: DPU IP Configuration

Figure 6 shows the ZynqMP PS Interface and the peripherals selected in this interface. For the input/output settings, USB was used to receive images from the camera and Lidar, SD card to load the operating system, UART channels and Ethernet to communicate with the development board and receive outputs.

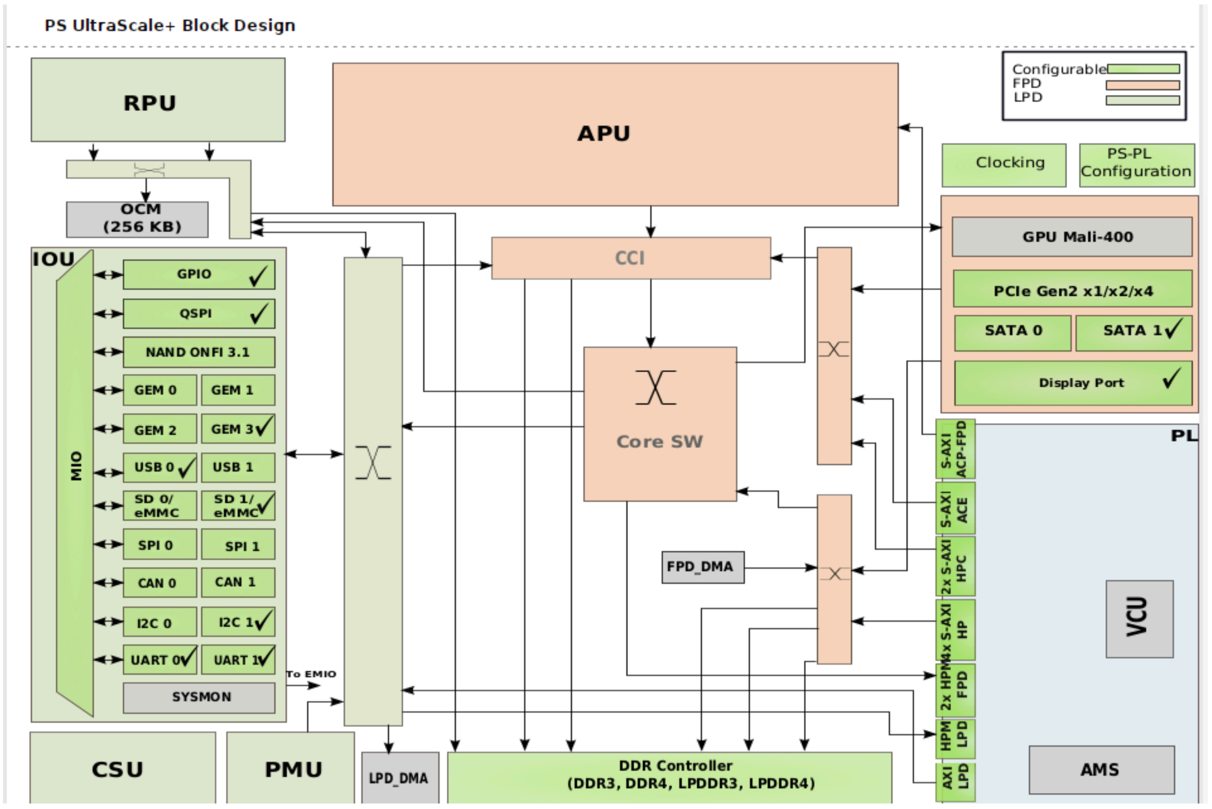


Figure 6: ZynqMP PS hardware block diagram

The bitstream is generated. After successful bitstream generation export the hardware, File -> Export -> Hardware, -> Finish.

2.2) Petalinux Configuration and Compilation

First of all, the Board Support Package (bsp) file should be downloaded as petalinux 2.2 for vitis ai 3.0 and dpu 4.1.[1]

Then Petalinux Tool should be installed.[2]

Xilinx provides the environment for building the software platform from the hardware platform and developing the application code using the embedded CPU. In this work, the petalinux configuration consists of three phases. These are;

- Project Creation
- Configuration and Compilation
- Packaging and Booting

2.2.1) Project Creation

The following command line was run to create a new petalinux project.

```
$ petalinux-create -t project -s
/home/murattokez/Xilinx/Vitis-AI/DPU_TRD/xilinx-kv260-starterkit-v2022.2-05230256.bsp
--name dpuOS
```

2.2.2) Configuration and Compilation

Configure the Project

Before starting the configuration settings in the created project, first navigate to the Linux terminal where the project folder is created. Then, using the command line shown in below, the .xsa file previously created in Vivado is added to the petalinux project for hardware configuration.

```
$ petalinux-config --get-hw-description= $PATH_XSA_FILE
```

Here, make the following changes:

- **Enable FPGA Manager**: This is often under Subsystem AUTO Hardware Settings.
- **Disable TFTPboot copy**: Usually found under Image Packaging Configuration.
- **Select Root filesystem type as EXT4**

Configure the Kernel

To configure the kernel to include the DPU driver:

```
$ petalinux-config -c kernel
```

Navigate to:

Device Drivers → Misc devices --> Enable the Xilinx Deep Learning Processing Unit (DPU) Driver.

Also Enable the USB-to-Serial Converter Driver for the connection of the 2d RPLIDAR:

In the menu, enable the CP210x driver by following the path below:

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

Copy Necessary Recipes

Copy the required recipes from your Vivado project into your PetaLinux project directory:

This step is not a must but it makes it easier to find and select all required packages in next step. If this step is skipped, please enable the required packages one by one in next step.
Add vitis-ai-library to rootfs

- Copy the lastest recipes-vitis-ai to your <your_petalinux_project_dir>/project-spec/meta-user

```
$ cp -r src/vai_petalinux_recipes/recipes-vitis-ai <petalinux project>/project-spec/meta-user/
```

NOTE: recipes-vitis-ai is used for Vitis flow by default. In vivado flow, please comment out the following line in recipes-vitis-ai/vart/vart_3.0.bb

```
#PACKAGECONFIG:append = " vitis"
```

Modify user-rootfsconfig

Add the following lines to project-spec/meta-user/conf/user-rootfsconfig:

```
CONFIG_vitis-ai-library
CONFIG_vitis-ai-library-dev
CONFIG_vitis-ai-library-dbg
CONFIG_xrt
CONFIG_xrt-dev
CONFIG_zocl
CONFIG_packagegroup-petalinux-self-hosted
CONFIG_cmake
CONFIG_opencl-clhpp-dev
CONFIG_opencl-headers-dev
CONFIG_packagegroup-petalinux-opencv
CONFIG_packagegroup-petalinux-opencv-dev
```

The following settings should be made for I2c.

Under Filesystem Packages > misc > python3, enable
Under Filesystem Packages > base > i2c-tools, enable
i2c-tools
i2c-tools-dev

Configure the Root File System

In Linux and similar operating systems, the filesystem where all files reside is called the root filesystem, rootfs. This is where the libraries, files, applications, etc. that you want to have on the kernel are located in the rootfs partition.

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

Select User Packages we added in Modify user-rootfsconfig

Build the project.

After saving these configuration settings, the system needs to be compiled. The command called for compilation is shown below. After the compilation process is complete, an images folder is created in the project folder. Within this folder is the linux folder. The files in this folder will be used in the next step, packaging and preloading.

```
$ petalinux-build
```

2.2.3) Packaging and Booting

Since the SD card was selected as the Petalinux boot vehicle in the configuration phase, it is necessary to package the SD card in a format suitable for the SD card in the packaging and preloading phase. First, the package that will do the preloading process is prepared.

```
$ cd images/linux
$ petalinux-package --boot --fsbl zynqmp_fsbl.elf --u-boot u-boot.elf --pmufw pmufw.elf --fpga
system.bit --force
```

Generate WIC Image for SD Card

```
$ petalinux-package --wic --bootfile "BOOT.BIN boot.scr Image system.dtb" --wic-extra-args
"-c gzip"
```

Then the created Petalinux image was flashed to the sd card using gzip with the following command.

```
$ gzip -dkc petalinux-sdimage.wic.gz | sudo dd of=/dev/mmcblk0 bs=4M status=progress
```

2.3 Creating the Device Tree Overlay

Device Tree Overlay (DTO) is a dynamically loadable data structure that specifies the hardware configuration. It is important for hardware flexibility and configurability, especially in FPGA and embedded systems.

To generate the device tree overlay follow the steps below:

1. Navigate to the XSCT Binary Directory

First, you need to navigate to the directory where the XSCT binaries are located. Open a terminal and enter the following commands:

```
$ cd ~/Xilinx/PetaLinux/tools/xsct/bin
$ ./xsct
```

2. Create the Device Tree Domain and Generate the Device Tree

Once you are in the XSCT environment, execute the following command to create a device tree domain and generate the device tree:

```
$ createdts -hw $PATH_XSA_FILEkv260_i2c.xsa -zocl -platform-name KV260 -git-branch
xlnx_rel_v2022.2 -overlay -compile -out
/home/murattokez/Xilinx/Vitis-AI/DPU_TRD/xsct_out/kv260_dt
```

This command will generate the necessary device tree files based on your hardware description.

3. Compile the Device Tree

Next, compile the generated device tree using the Device Tree Compiler (DTC). Execute the following command:

```
$ dtc -@ -O dtb -o ./kv260_i2c_dt/myApp-i2c/kv260_.dtbo
./kv260_dt/KV260/psu_cortexa53_0/device_tree_domain/bsp/pl.dtsi
```

4. Create shell.json

To define the shell configuration, create a shell.json file with the following content:

```
$ echo '{ "shell_type" : "XRT_FLAT", "num_slots": "1" }' > shell.json
```

5. Prepare the Bitstream File

Make a copy of the top_wrapper.bit file from its current directory to a different directory and rename it to kv260.bit.bin. This file is your bitstream file that will be used to program the FPGA.

6. Summary of Required Files

At this point, you should have the following files ready for use:

- **kv260.bit.bin: The bitstream file.**
- **kv260.dtbo: The compiled device tree overlay file.**
- **shell.json: The shell configuration file.**

Now the KV260 FPGA board is ready to boot with sd card.

Creating an Accelerated application.

1. Make a directory in your user space

```
xilinx-kv260-starterkit-2022:~$ mkdir myApp
```

2. Copy kv260.bit.bin, kv260.dtbo, shell.json to myApp directory with scp command.

4. Move the myApp directory.

```
xilinx-kv260-starterkit-20221:~$ sudo mv myApp/ /lib/firmware/xilinx/
```

5. Load the myApp

```
xilinx-kv260-starterkit-2022:~$  sudo xmutil unloadapp
xilinx-kv260-starterkit-2022:~$  sudo xmutil loadapp myApp

xilinx-kv260-starterkit-2022:~$ chmod a+rw /dev/i2c-3
xilinx-kv260-starterkit-2022:~$ chmod a+rw /dev/dpu
```

The i2cdetect -l command will show that i2c has been successfully added and the show_dpu command will show that dpu has been successfully added.

3) Lidar and Encoder Integration

This system enables the real-time mapping of parking areas and the detection of occupied parking spaces. Data obtained from the Lidar can be used to map with high accuracy in both indoor and outdoor environments. With camera integration, the status of the parking areas can also be detected instantly. Position information is obtained from the encoder, and a camera facing perpendicular to the direction of movement is used to detect vehicles. As soon as the Lidar detects objects, the camera is activated, which enhances the system's efficiency. When a vehicle is detected, the system marks the location on the map in real-time using the position information, accurately indicating the occupied space.

3.1) Rotary Encoder

The rotary encoder was programmed using an Arduino script to capture the rotation and convert it into distance data. The rotary encoder's readings are then sent to the KV260 FPGA board over I2C.

File: fadas_encoder.ino

- The Wire.h library is used for I2C communication.
- encoderPinA and encoderPinB are defined as encoder connection pins.
- encoderValue and lastEncoded store the encoder position and the last encoder value.
- wheelDiameter is the wheel diameter, and distancePerRevolution calculates the distance traveled in one wheel revolution.
- The setup function initializes serial communication, sets encoder pins as inputs, defines interrupt functions, starts I2C communication, and defines the data sending function upon request.
- The loop function is left as an empty loop, waiting for data requests via I2C.
- The updateEncoder function updates the encoder values.
- The requestEvent function sends the distance via I2C.

3.2) Lidar and Rotary Encoder Data Processing

A Python script reads data from the 2D Lidar and rotary encoder, processes this data to generate 3D coordinates, and sends it to the KV260 FPGA board. Serial connections for the Lidar and encoder, reads the distance and angle to calculate X-axis and Y-axis positions from the Lidar, and computes the Z-axis position using the encoder data.

File: 3d_lidar.py

• The serial and math libraries are used.

• ser and ser_enc define the serial connections for the Lidar and encoder.

• In an infinite loop, Lidar data is read for angle and distance to calculate x and y positions, encoder data is read for the z position, and x, y, and z coordinates are calculated and printed.

Z-axis Calculation:
The encoder measures the rotation angle of the wheel and converts this data into distance. This distance is combined with the X and Y measurements from the Lidar to create 3D coordinates. The Z-axis calculation extends the 2D Lidar data into a 3D space, providing a comprehensive map of the parking area around the vehicle.

• The encoder reads the number of pulses generated as the wheel turns.

• Each pulse corresponds to a specific angle.

• Knowing the wheel diameter, the distance traveled per revolution is calculated.

• The total distance traveled (Z-axis) is determined by dividing the number of pulses by the number of pulses per revolution and multiplying by the distance traveled per revolution.

This integration of the rotary encoder and Lidar provides high accuracy for 3D mapping of the parking area.

3.3) 3D Point Cloud Generation

Another Python script runs on the host machine, receiving data from the FPGA over a socket connection, and generates a 3D point cloud using the Open3D library. The script sets up a server to receive point cloud data, processes the received data, and visualizes it as a 3D point cloud.

File: 3d_pointcloud_gnrt_in_HOST.py

• The socket, numpy, and open3d libraries are used.

• A server socket is created and set to listen on port 8080.

• Incoming connections are accepted.

• In an infinite loop, incoming data is read to create a 3D point cloud, which is then visualized.

• The connection is closed and the socket is closed.

`3d_lidar.py` is transferred to the KV260 FPGA board via Ethernet and run as follows:

```
$ scp petalinux@IP_ADRRS
$ python3 3d_lidar.py
```

Then, to obtain the results on the host computer, the following command is entered:

```
$ python3 3d_point_cloud.py
```

4) RUN VITIS-AI APPLICATION

In this project, Vitis-AI Library Run Time (VART) was used for Adas application. Yolov3 was used as the model.

Installing the AI Library Package

Download the vitis-ai-runtime-3.0.0.tar.gz[3] . Untar it and copy the following files to the target using the scp command.

```
$ tar -xzvf vitis-ai-runtime-3.0.0.tar.gz
$ scp -r vitis-ai-runtime-3.0.0/2022.2/aarch64/centos root@IP_OF_BOARD:~/
```

For Zynq UltraScale+ MPSoCs, run the zynqmp_dpu_optimize.sh script on the board.

```
$ cd ~/dpu_sw_optimize/zynqmp/
$ ./zynqmp_dpu_optimize.sh
```

Install the Vitis AI Library.

```
$ cd ~/centos
 $ bash setup.sh
```

After the installation is complete, the directories are as follows:

- The library files are stored in the /usr/lib location.
- The header files are stored in the /usr/include/vitis/ai location.

INSTALLING THE XMODEL

For each model, there is a .yaml file that describes all the details about the model. The .yaml file contains download links for various Xilinx target boards. Select your model and the platform you want and download it. we can find the yaml file in the modelzoo directory. The following commands were used on **KV260** for yolov3 used in this study.

```
$ wget
https://www.xilinx.com/bin/public/openDownload?filename=yolov3_adas_pruned_0_9-zcu102_zcu104_kv260-r3.0.0.tar.gz -O yolov3_adas_pruned_0_9-zcu102_zcu104_kv260-r3.0.0.tar.gz

$ mkdir -p /usr/share/vitis_ai_library/models

$ tar -xzvf yolov3_adas_pruned_0_9-zcu102_zcu104_kv260-r3.0.0.tar.gz

$ cp yolov3_adas_pruned_0_9 /usr/share/vitis_ai_library/models -r
```

We are now ready to run the ADAS application.

```
$ cd ~/examples/vai_runtime
$ cd adas_detection
$ bash -x build.sh

$ ./adas_detection /dev/video0
/usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel
```

5) Conclusion

This project outlines the process of integrating and accelerating Advanced Driver Assistance Systems (ADAS) applications using the Deep Learning Unit (DPU) on the Kria KV260 FPGA board. By leveraging the Vitis AI Library, Petalinux, and various hardware components such as 2D LiDAR, camera, and rotary encoder, the system aims to provide real-time vehicle detection and mapping in parking lots. The system's architecture, hardware design, Petalinux configuration, and software integration were carefully implemented to ensure proper communication between all components. The integration of a rotary encoder and LiDAR provides high-precision 3D mapping capabilities, crucial for accurately detecting and mapping parking spaces.

A significant aspect of this project is the use of the YOLOv3 model from the Vitis-AI Model Zoo. The Model Zoo provides a collection of pre-trained models optimized for Xilinx hardware, which simplifies the deployment of AI applications on FPGA platforms. The YOLOv3 model was selected for its capability in object detection, crucial for identifying vehicles in a parking lot.

The project's implementation of the YOLOv3 model for object detection showcases the application of AI models on edge devices. The successful execution of this model demonstrates the practicality of using pre-trained models from the Vitis-AI Model Zoo for real-world applications.