Implement of Yolov7 on Vitis AI KV260

Zijie NING

August 27, 2023

Contents

Abstract

This tutorial will go through the steps needed to deploy a neuron network on Xilinx edge boards.

1 Introduction

The deep-learning processor unit (DPU) is a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required. It is designed to accelerate the computing workloads of deep learning inference algorithms widely adopted in various computer vision applications, such as image/video classification, semantic segmentation, and object detection/tracking. The DPU is released with the Vitis AI specialized instruction set, thus facilitating the efficient implementation of deep learning networks.

An efficient tensor-level instruction set is designed to support and accelerate various popular convolutional neural networks, such as VGG, ResNet, GoogLeNet, YOLO, SSD, and MobileNet, among others. The DPU is scalable to fit various Xilinx, Zynq UltraScale+ MPSoCs, Xilinx Kria KV260, Versal cards, and Alveo boards from Edge to Cloud to meet the requirements of many diverse applications.

We will firstly setup the environment on both host machine and edge card, and then quantize, compile the model, in order to deploy it on edge board.

The overall work flow is shown in 5. The Github repository is here.

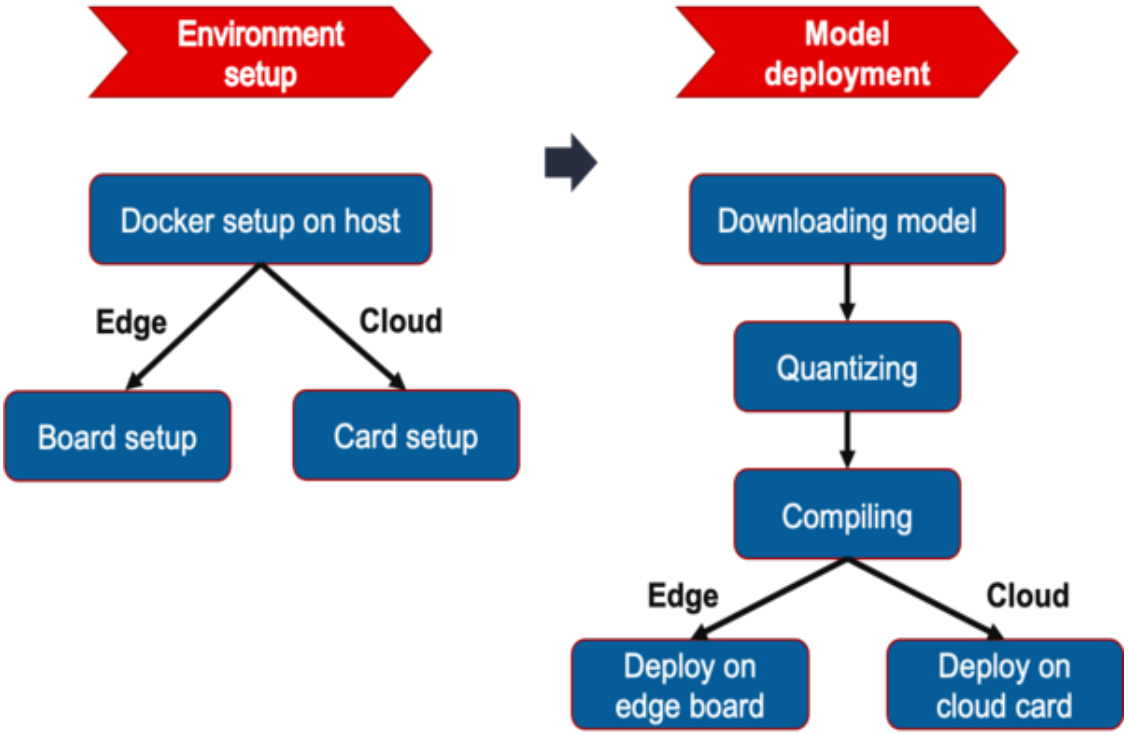

Figure 1: Work flow

2 Environment Setup

This project is completed between Dec. 2022 and Mar. 2023. The environments and versions are as follows:

- Host PC: Ubuntu 22.04
- Edge card: KV260 + PetaLinux 2022.2
- Vitis AI Library v3.0, released on Jan 13 2023
- Vitis AI User Guide (UG1414) v3.0 English
- Yolov7 offical version

2.1 Host PC (Docker)

Follow the instruction to install the docker of PyTorch version.Use this command to launch the docker and active the environment:

```
./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest
conda activate vitis-ai-pytorch
```

Move to "Yolov7" directory and use "`pip -r install requirements.txt`" to install the libraries for Yolov7.

2.2 Edge Device (KV260)

Follow the Getting Start tutorial to connect the KV260. When preparing the microSD card, we use the image of Petalinux for KV260 rather than that of Ubuntu.

Then follow the User Guide to install Vitis AI Runtime.

Use "`dnf install <package>`" to install packages if needed.

Remember to set time using "`sudo date <mmddhhmmyyyy>`" to avoid verification and git errors.

3 Quantification

The process of inference is computation intensive and requires a high memory bandwidth to satisfy the low-latency and high-throughput requirement of Edge applications.

Generally, 32-bit floating-point weights and activation values are used when training neural networks. By converting the 32-bit floating-point weights and activations to 8-bit integer (INT8) format, the Vitis AI quantizer can reduce computing complexity with little degradation in accuracy. The fixed-point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model.

The Vitis AI quantizer supports common layers in neural networks, including, but not limited to, convolution, pooling, fully connected, and batchnorm. **Unfortunately, there are still some layers of Yolov7 are not supported.**

The quantizer offered by Xilinx is called "`Vai_q_pytorch`". There are 3 steps in the flow of quantification, corresponding to the 3 modes of quantizer: **inspect**, **calib** and **test**. We can refer to resnet18_quant.py to understand how to edit model script and apply the quantizer.

We should prepare the following files for `Vai_q_pytorch`.

Table 1: Input Files for `Vai_q_pytorch`

No.	Name	Description
1	best.pt	Pre-trained PyTorch model, generally pt file.
2	test.py	A Python script including float model definition.
3	calibration dataset	A subset of the training dataset containing 100 to 1000 images.

3.1 Inspect

`Vai_q_pytorch` provides a function called inspector to help diagnose neural network (NN) models under different device architectures. The inspector can predict target device assignments based on hardware constraints. The generated inspection report can be used to guide users to modify or optimize the NN model, greatly reducing the difficulty and time of deployment. It is recommended to inspect float models before quantization.

We should add an instance of quantizer in the code of "`test.py`" as follows:

```
# Load model
# weights is a trained yolov7 model
model = attempt_load(weights, map_location=device)  # load FP32 model
# Generate a fake input
input = torch.randn([batch_size, 3, 640, 640])

quant_model = model # inspect the model in float version
if not target: # target = DPUCZDX8G_ISA1_B4096 for KV260
    raise RuntimeError("A target should be specified for inspector.")

from pytorch_nndct.apis import Inspector

# create inspector
inspector = Inspector(target)  # by name
# start to inspect
inspector.inspect(quant_model, (input,), device=device,
    image_format="svg",verbose_level = 0, output_dir="./quantize_result".)
sys.exit()
```

"`target`" is the architecture name of DPU (i.e. "DPUCZDX8G_ISA1_B4096" for KV260) or its fingerprint (i.e. "0x101000056010407" for our system). Use command line "`xdputil query`" on the edge system to check its DPU arch and fingerprint.

Notice that when calling a dataset or a file from within docker, the path is not the same as when we call it from outside. The path should be like "`/workspace/...`". We can ignore the warnings during execution.

If the inspector runs successfully, three important files are usually generated under the output directory "./quantize_result":

- `inspect_{target}.txt`: Target information and all the details of operations in float model
- `inspect_{target}.svg`: If `image_format` is not None. A visualization of inspection result is generated
- `inspect_{target}.gv`: If `image_format` is not None. Dot source code of inspection result is generated

We firstly try to quantize a pre-trained yolov7-tiny model.

After inspecting, in the output txt file, we notice that some nodes are not deployed on DPU. Some of these nodes are "LeakyReLU", the others are from "`Model::Model/IDetect[model]/IDetect[77]`". which indicates that they are from the "IDetect" module. These nodes are shown with one of the following explanations:

- "Try to assign {pattern name} to DPU failed.": The compiler refuses to deploy this pattern on DPU.
- "Convert nndct graph to XIR failed.": If you encounter this problem, please contact the developer.
- "{op type} can't be converted to XIR.": The operator cannot be represented by XIR.
- "{op type} can't be assigned to DPU.": Although the operator can be converted to XIR, it cannot be deployed on the DPU.

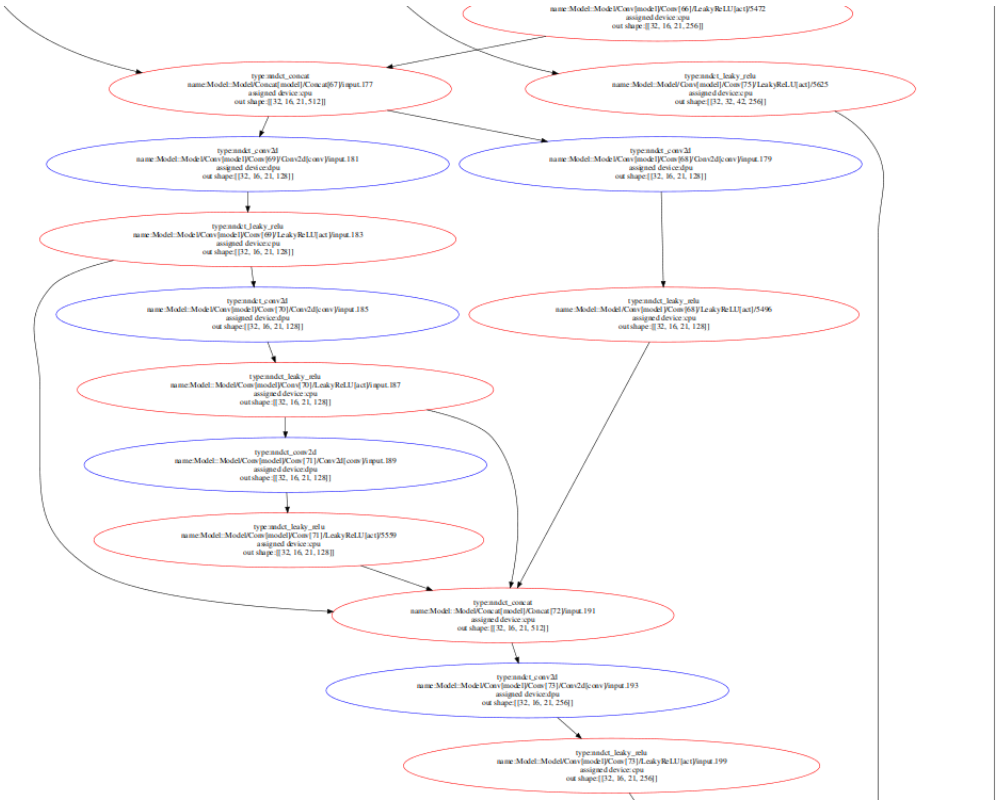

Figure 2: Part of inspect_DPUCZDX8G_ISA1_B4096.svg with LeakyReLU(0.1)

These nodes are shown in red in the `svg` file, while the others are shown in blue.

According to this Github issue, **we change the alpha of LeakyReLU to 26/256 (0.1015625)**, to make them deployable on DPU.

For the "IDetect" module, it is instantiated in "`cfg/training/yolov7-tiny.yaml`":

```
[[74,75,76], 1, IDetect, [nc, anchors]],   # Detect(P3, P4, P5)
```

The "IDetect" module should be removed and added back later to get a fully DPU-deployable model.

```
# Load model
model = attempt_load(weights, map_location=device)  # load FP32 model
# IDetect module
model_detect = copy.deepcopy(model)
model_detect.model = nn.Sequential(model.model[-1])
# Rest of the model
model.model = nn.Sequential(*list(model.model.children())[:-1])
# Model to be quantized
quant_model = model
```

Our initial idea was to replace the "IDetect" module by "Detect" module. Though, in fact, this is not necessary since we finally decided to deploy this layer on CPU. *TODO: Re-do the whole flow with "IDetect".*

With debug tools, we notice that the "Detect" module needs 3 tensors from previous layers. We modify the output of "`forward_once`" method of class "`Model`" in "`model/yolo.py`" to expose these tensors.

```
# Before
# return x
# After
return x,y[-3:]
```

We rerun the inspect process and find from both `txt` and `svg` file that all remained modules can be deployed on DPU.

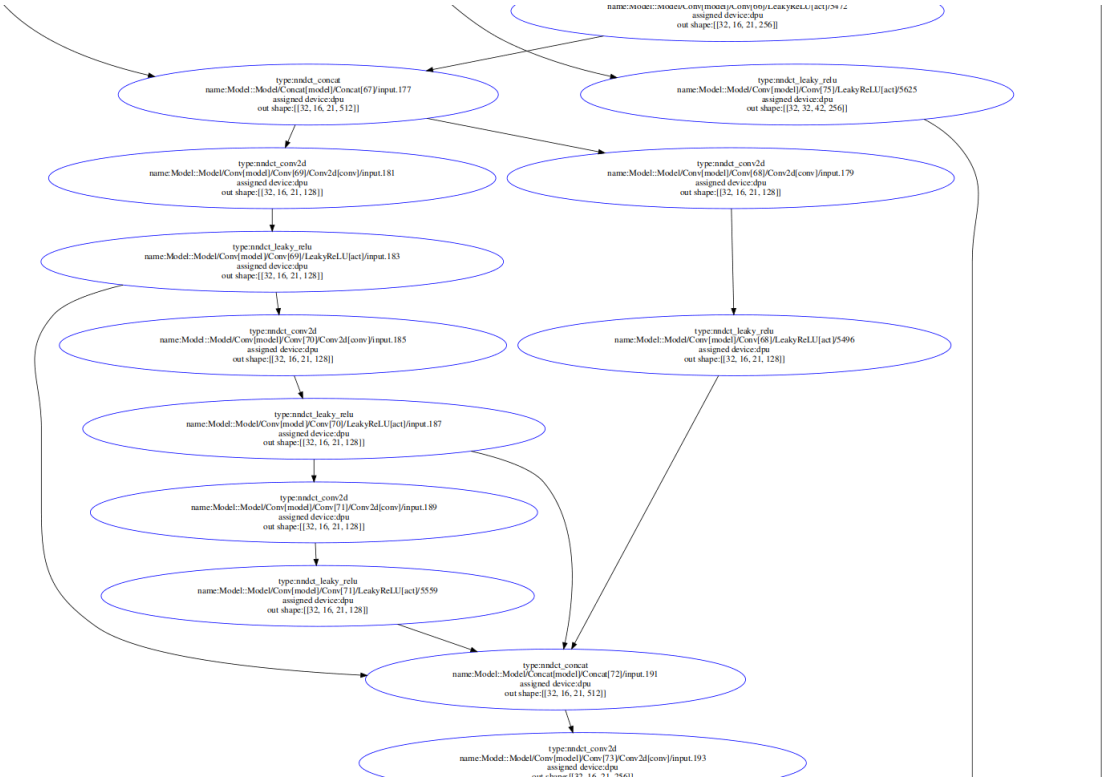

Figure 3: Same part of inspect_DPUCZDX8G_ISA1_B4096.svg with LeakyReLU(26/256)

Now we can start to quantize this model.

3.2 Calib

Quantize calibration determines quantization steps of tensors in evaluation process if flag "`quant_mode`" is set to "calib". We use the inference script of Yolov7 "`test.py`" and replace the float model by the quantized model:

```
# Define quantizer
quantizer = torch_quantizer(
    quant_mode=calib, model, (input), device=device, quant_config_file=config_file,
    target=target, output_dir=output_dir
)
quant_model = quantizer.quant_model
# Forward once
...
out = quant_model(img, augment=augment)  # inference and training outputs
...
# Handle quantization result
quantizer.export_quant_config()
sys.exit(0)
```

If this step runs successfully, two important files are generated in the output directory "`./quantize_result`":

- "`Model.py`": Converted `Vai_q_pytorch` format model.
- "`Quant_info.json`": Quantization steps of tensors. Retain this file for evaluating quantized models.

3.3 Test

After calibration, evaluate the quantized model by setting "`quant_mode`" to "test" and the batch size to 1.

```
# Define quantizer
quantizer = torch_quantizer(
    quant_mode=test, model, (input), device=device, quant_config_file=config_file,
    target=target, output_dir=output_dir
)
quant_model = quantizer.quant_model
# Forward once
...
out = quant_model(img, augment=augment)  # inference and training outputs
...
# Handle quantization result
quantizer.export_torch_script(output_dir=output_dir)
quantizer.export_onnx_model(output_dir=output_dir)
quantizer.export_xmodel(deploy_check=False, output_dir=output_dir)
sys.exit(0)
```

If this step runs successfully, three files are generated in the output directory "`./quantize_result`". The xmodel file is further used for the Vitis AI compiler to deploy to the FPGA.

- "`Model_int.xmodel`": Deployed XIR format model.
- "`Model_int.onnx`": Deployed onnx format model.
- "`Model_int.pt`": Deployed torch script format model.

4 Compilation

The Vitis™ AI compiler (`VAI_C`) is the unified interface to a compiler family targeting the optimization of neural-network computations to a family of DPUs. Each compiler maps a network model to a highly optimized DPU instruction sequence.

The simplified description of `VAI_C` framework is shown in the following figure. After parsing the topology of optimized and quantized input model, `VAI_C` constructs an internal computation graph as intermediate representation (IR). Therefore, a corresponding control flow and a data flow representation. It then performs multiple optimizations, for example, computation nodes fusion such as when batch norm is fused into a presiding convolution, efficient instruction scheduling by exploit inherent parallelism, or exploiting data reuse.

For PyTorch, the quantizer NNDCT takes the quantized model and outputs the quantized model in the XIR format directly (Fig. 4). Use `vai_c_xir` to compile it:

```
vai_c_xir -x /PATH/TO/quantized.xmodel \
    -a /PATH/TO/arch.json \
    -o /OUTPUTPATH \
    -n netname
```

The `quantized.xmodel` is the file generated in the quantization step. The `arch.json` can be found in docker at path `/opt/vitis_ai/compiler/arch/`. It includes the name of DPU, for example:

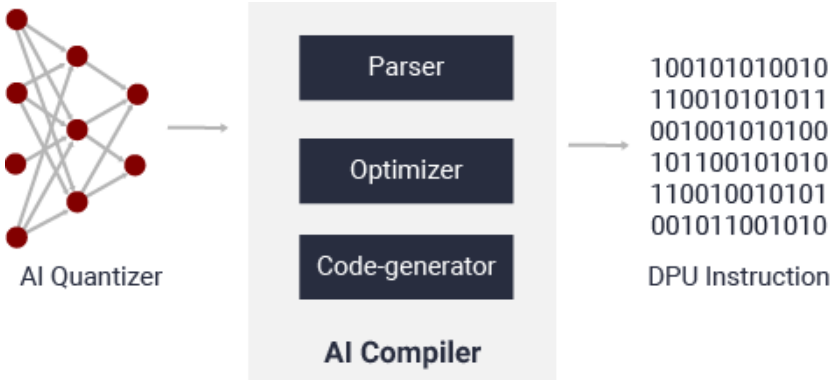

Figure 4: Vitis AI Compiler Framework

```
{
    "target": "DPUCZDX8G_ISA1_B4096"
}
```

After execution, we should get a "`deploy.xmodel`" under `output_path/Model_quant`. This is the file to be executed on the DPU of edge devices. Notice that before and after the compilation, we have a `xmodel` file respectively. Be careful not to get confused, otherwise, no subgraph will be found during execution.

5 Implementation

In this section, we will deploy the executable file on the edge device (KV260). Notice that only a part of Yolo network can be deployed on DPU, while the other part still runs on CPU or GPU. The main work of this section is to connect these two parts and make necessary data conversions. The overall flow is as shown in Fig. 5.

5.1 Input image

Since the model is trained with image size 640x640, the input image of quantized model should also have size of 640x640.Refer to this line of code in `utile/dataset.py`:

```
shape = self.batch_shapes[self.batch[index]] if self.rect else self.img_size
```

We set "`rect=False`" to active image resize to get exactly the same shape as the desired input tensors.

```
# Dataloader (set rect=False)
dataloader = create_dataloader(
    data[task], imgsz, batch_size, gs, opt, pad=0.5, rect=False,
    prefix=colorstr(f"{task}: "))[0]
```

These images will be used later with bounding boxes to show the output. For the input of DPU, we still need to do permutation and scaling.

The shape of PyTorch image tensors is `(batch size, channels, height, width)`, but that of Vitis AI is `(batch size, height, width, channels)`. We use `permute` method to change the order of tensors.

Also, the tensors of PyTorch are represented by float number data, while that of Vitis AI are represented by fix point data. We should get the location of fix point and make the scaling.

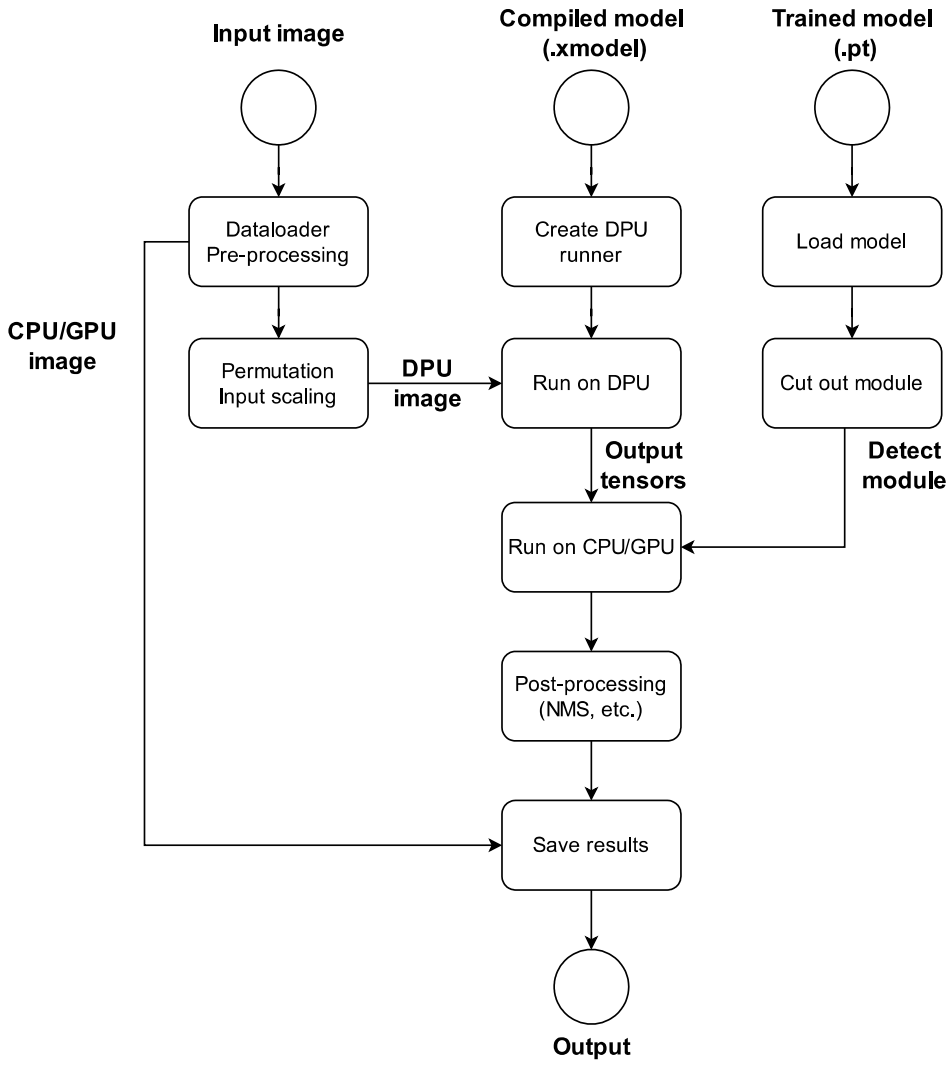

Figure 5: Flow chart of implementation

```
input_fixpos = dpu_runner.get_input_tensors()[0].get_attr("fix_point")
input_scale = 2**input_fixpos
# Inference loop
for batch_i, (img, targets, paths, shapes) in enumerate(tqdm(dataloader, desc=s)):
    nb, _, height, width = img.shape  # batch size, channels, height, width
    # Input scaling
    # img_DPU.shape = batch size, height, width, channels
    img_DPU  = img.permute(0, 2, 3, 1).float().numpy() / 255 * input_scale
    img_DPU = img_DPU.astype(np.int8)
    img_DPU = torch.from_numpy(img_DPU)
```

5.2 DPU runner

We can refer to the exemple file of MNIST app_mt.py to understand how to execute a `xmodel` file on DPU.

The function "`get_child_subgraph_dpu`" is used to get subgraphs from quantized model. In most of the cases, there is only 1 subgraph since we only have 1 part to be deployed on DPU. If more than

1 part of model to be deployed on DPU, the work of connecting modules will be much more complexe.

```python
from typing import List
import xir
import vart

def get_child_subgraph_dpu(graph: "Graph") -> List["Subgraph"]:
    assert graph is not None, "'graph' should not be None."
    root_subgraph = graph.get_root_subgraph()
    assert root_subgraph is not None,
        "Failed to get root subgraph of input Graph object."
    if root_subgraph.is_leaf:
        return []
    child_subgraphs = root_subgraph.toposort_child_subgraph()
    assert child_subgraphs is not None and len(child_subgraphs) > 0
    return [cs for cs in child_subgraphs
        if cs.has_attr("device") and cs.get_attr("device").upper() == "DPU"]

# Create DPU runner
g = xir.Graph.deserialize(xmodel)
subgraphs = get_child_subgraph_dpu(g)
dpu_runner = vart.Runner.create_runner(subgraphs[0], "run")
```

The function "`get_child_subgraph_dpu`" is used to prepare input and output buffer, launch program on DPU and do the output scaling:

```python
def runDPU(dpu, img):
    """get tensor"""
    # TODO Multi-thread
    inputTensors = dpu.get_input_tensors()
    outputTensors = dpu.get_output_tensors()
    input_ndim = tuple(inputTensors[0].dims)
    output_ndim_0 = tuple(outputTensors[0].dims)
    output_ndim_1 = tuple(outputTensors[1].dims)
    output_ndim_2 = tuple(outputTensors[2].dims)

    # start = 0
    batchSize = img.shape[0]

    outputData = [
        np.empty(output_ndim_0, dtype=np.float32, order="C"),
        np.empty(output_ndim_1, dtype=np.float32, order="C"),
        np.empty(output_ndim_2, dtype=np.float32, order="C"),
    ]

    """prepare batch input/output """
    inputData = []
    inputData = [np.empty(input_ndim, dtype=np.int8, order="C")]

    """init input image to input buffer """
    for i in range(batchSize):
        imageRun = inputData[0]
        count = count+1

    """run """
    job_id = dpu.execute_async(imageRun, outputData)

```

```
    t = time_synchronized()
    dpu.wait(job_id)
    t1 = time_synchronized() - t
    # print(colorstr("green", t1))
    # output scaling

    outputData[0] = torch.from_numpy(outputData[0] / (2 **
        outputTensors[0].get_attr("fix_point"))).permute(0, 3, 1, 2)
    outputData[1] = torch.from_numpy(outputData[1] / (2 **
        outputTensors[1].get_attr("fix_point"))).permute(0, 3, 1, 2)
    outputData[2] = torch.from_numpy(outputData[2] / (2 **
        outputTensors[2].get_attr("fix_point"))).permute(0, 3, 1, 2)
    return outputData
```

Notice that we can use threads to inference several images at one time. This feature is not developed yet but we can refer to the exemple file of MNIST app_mt.py.

We run DPU in each inference loop:

```
# Inference loop
for batch_i, (img, targets, paths, shapes) in enumerate(tqdm(dataloader, desc=s)):
    # Load data
    # ...

    # Run model
    t = time_synchronized()
    out_DPU = runDPU(dpu_runner, img_DPU)
    # Pass output tensors to detect module
    out, train_out = forward_detect(model_detect, out_DPU)
    t0 += time_synchronized() - t
```

5.3 Detect module

We cut out the detect module from the trained model (.pt), which is not going to be run on DPU:

```
import torch.nn as nn
import copy

# Load model
model = attempt_load(weights, map_location=device)  # load FP32 model
model_detect = copy.deepcopy(model)
model_detect.model = nn.Sequential(model.model[-1]) # Last layer
model_detect.eval()
```

This module takes the output tensors from DPU as input. It can be seen as a simple neural network and be run with code:

```
def forward_detect(model_detect, x):
    m = model_detect.model[0]
    x = m(x)  # run
    return x
```

5.4 Main function

We add necessary arguments in main function:

```
def test(
    # ...
    config_file=None,
    threads=1,
    xmodel="quantize_result/compiled/yolov7.xmodel",
)
if __name__ == "__main__":
    # DPU execution
    parser.add_argument("--config_file", default=None,
        help="quantization configuration file")
    parser.add_argument("--threads", type=int, default=1,
        help="Number of threads. Default is 1")
    parser.add_argument("--xmodel", type=str,
        default="quantize_result/compiled/yolov7.xmodel", help="Path of xmodel.")
```

We should find saved results in output directory.