Vitis AI User Guide

UG1414 (v3.0) February 24, 2023

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

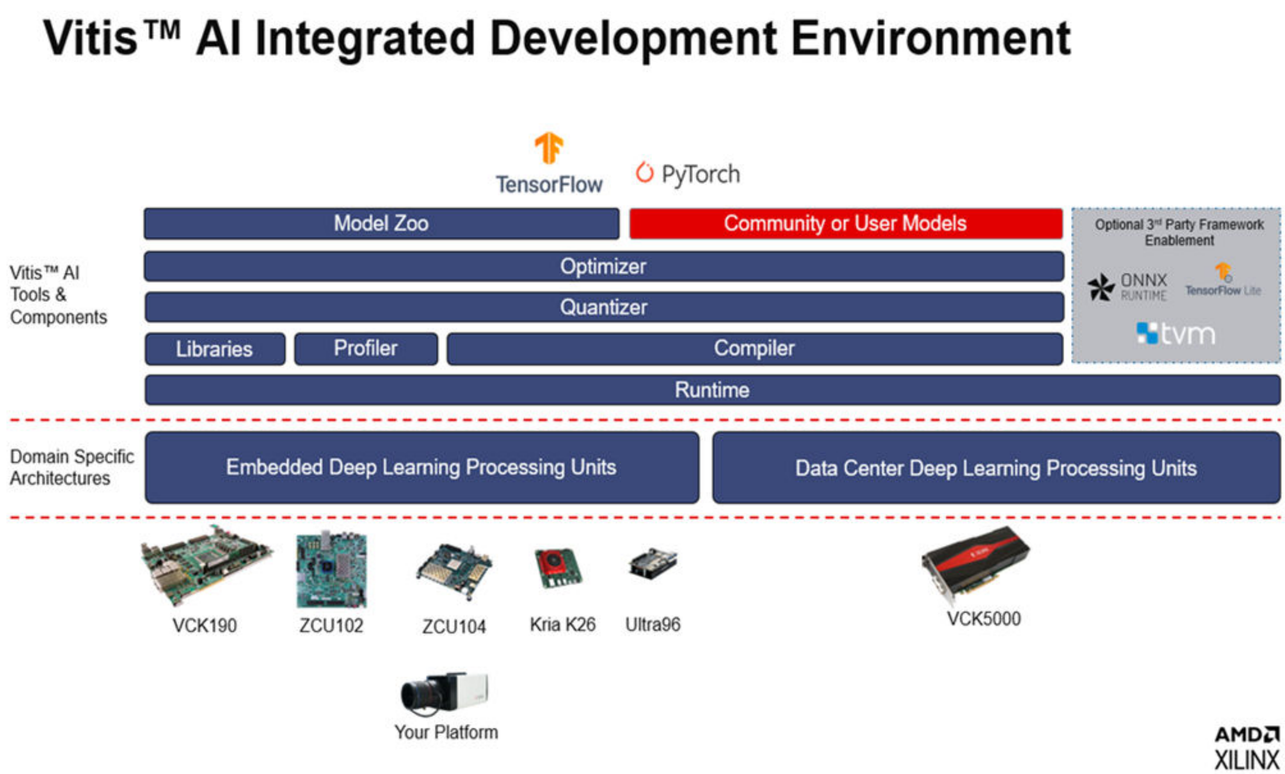
Table of Contents

Chapter 1

Vitis AI Overview

The Vitis™ AI development environment accelerates AI inference on Xilinx® hardware platforms, including both edge devices and Alveo™ accelerator cards. It consists of optimized IP cores, tools, libraries, models, and example designs. It is designed with high efficiency and ease of use in mind to unleash the full potential of AI acceleration on Xilinx SoCs and on adaptive compute acceleration platforms (ACAPs). The Vitis AI development environment makes it easy for users without FPGA knowledge to develop deep-learning inference applications by abstracting the intricacies of the underlying programmable logic.

Figure 1: **Vitis AI Integrated Development Environment**



***Note*:** Caffe support has been deprecated for releases ≥ 2.5. For Caffe support, see the Vitis AI 2.0 User Guide.

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

- **Machine Learning and Data Science:** Importing a machine learning model from a PyTorch, TensorFlow, or other popular framework onto Vitis™ AI, and then optimizing and evaluating its effectiveness. Topics in this document that apply to this design process include:
  - Chapter 2: Getting Started
  - Chapter 3: Quantizing the Model
  - Chapter 4: Compiling the Model
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
  - Chapter 8: Integrating the DPU into Custom Platforms
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Chapter 8: Integrating the DPU into Custom Platforms
- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure. Topics in this document that apply to this design process include:
  - Chapter 6: Profiling the Model

Features

Vitis AI includes the following features:

- Supports mainstream frameworks and the latest models capable of diverse deep learning tasks.
- Provides a comprehensive set of pre-optimized models that are ready to deploy on Xilinx devices.
- Provides a powerful quantizer that supports model quantization, calibration, and fine tuning. For advanced users, Xilinx also offers an optional AI optimizer that can prune a model by up to 90% with a tolerable accuracy loss.
- Provides layer-by-layer analysis to help with bottlenecks.
- Offers unified high-level C++ and Python APIs for maximum portability from Edge to Cloud.
- Customizes efficient and scalable IP cores to meet your needs for many different applications from a throughput, latency, and power perspective.

Vitis AI Tools Overview

Deep-Learning Processor Unit

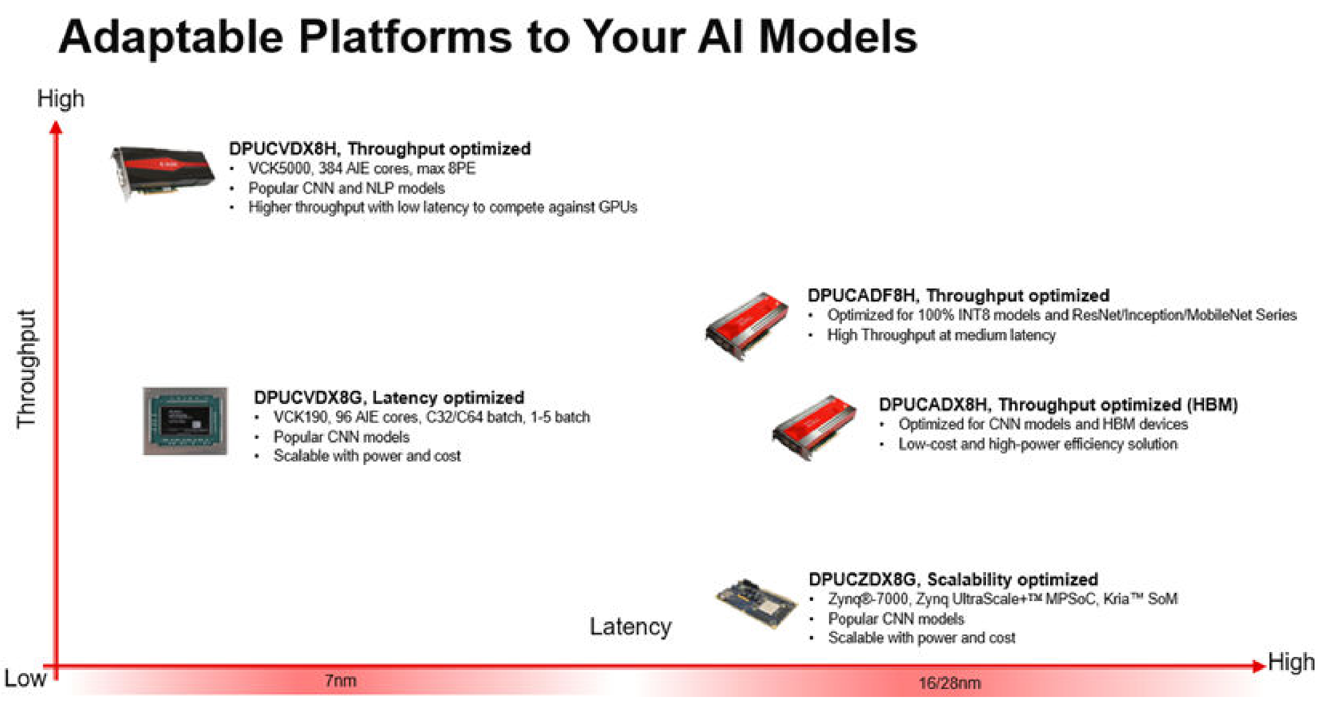
The deep-learning processor unit (DPU) is a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required. It is designed to accelerate the computing workloads of deep learning inference algorithms widely 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.

A configuration file, `arch.json`, is generated during the Vitis flow. The `arch.json` file is used by the Vitis AI compiler for model compilation. Once the configuration of the DPU is modified, a new `arch.json` must be generated. The models must be regenerated using the new `arch.json` file. For example, in the ZCU102 TRD of DPUCZDX8G in Vitis flow, the `arch.json` file is located at `$TRD_HOME/prj/Vitis/binary_container_1/link/vivado/vpl/prj/prj.gen/sources_1/bd/xilinx_zcu102_base/ip/xilinx_zcu102_base_DPUCZDX8G_1_0/arch.json`.

Vitis AI offers a series of different DPUs for both embedded devices such as Xilinx , Zynq® UltraScale+™ MPSoC, Kria KV260, Versal cards and Alveo cards such as U50LV, U200, U250, and U55C enabling unique differentiation and flexibility in terms of throughput, latency, scalability, and power.

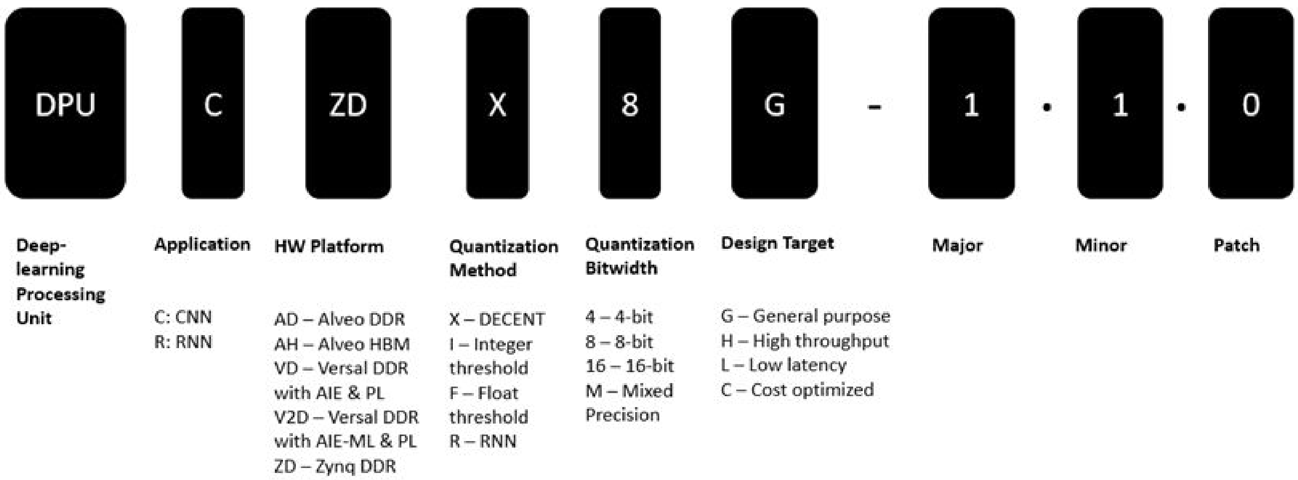
Figure 2: **DPU Options**



DPU Naming

Different fields of DPU name is used to indicate different features or purposes, and the naming scheme is shown in the following figure:

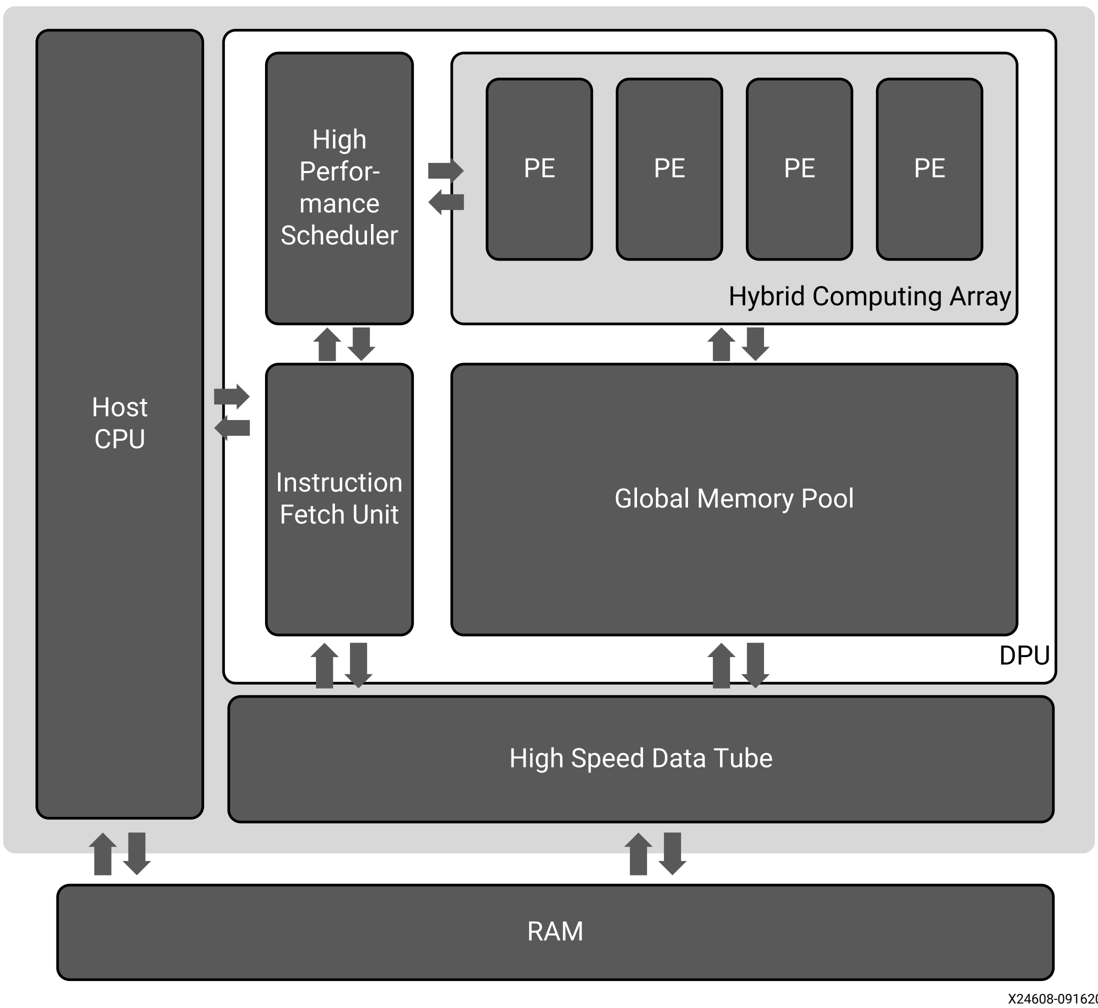
Figure 3: **DPU Nomenclature**



Zynq UltraScale+ MPSoC: DPUCZDX8G

The DPUCZDX8G IP has been optimized for Zynq UltraScale+ MPSoC. You can integrate this IP as a block in the programmable logic (PL) of the selected Zynq UltraScale+ MPSoCs with direct connections to the processing system (PS). The DPU is user-configurable and exposes several parameters which can be specified to optimize PL resources or customize enabled features. If you want to integrate the DPU in the customized AI projects or products, see https://www.xilinx.com/bin/public/openDownload?filename=DPUCZDX8G.tar.gz.

Figure 4: **DPUCZDX8G Architecture**

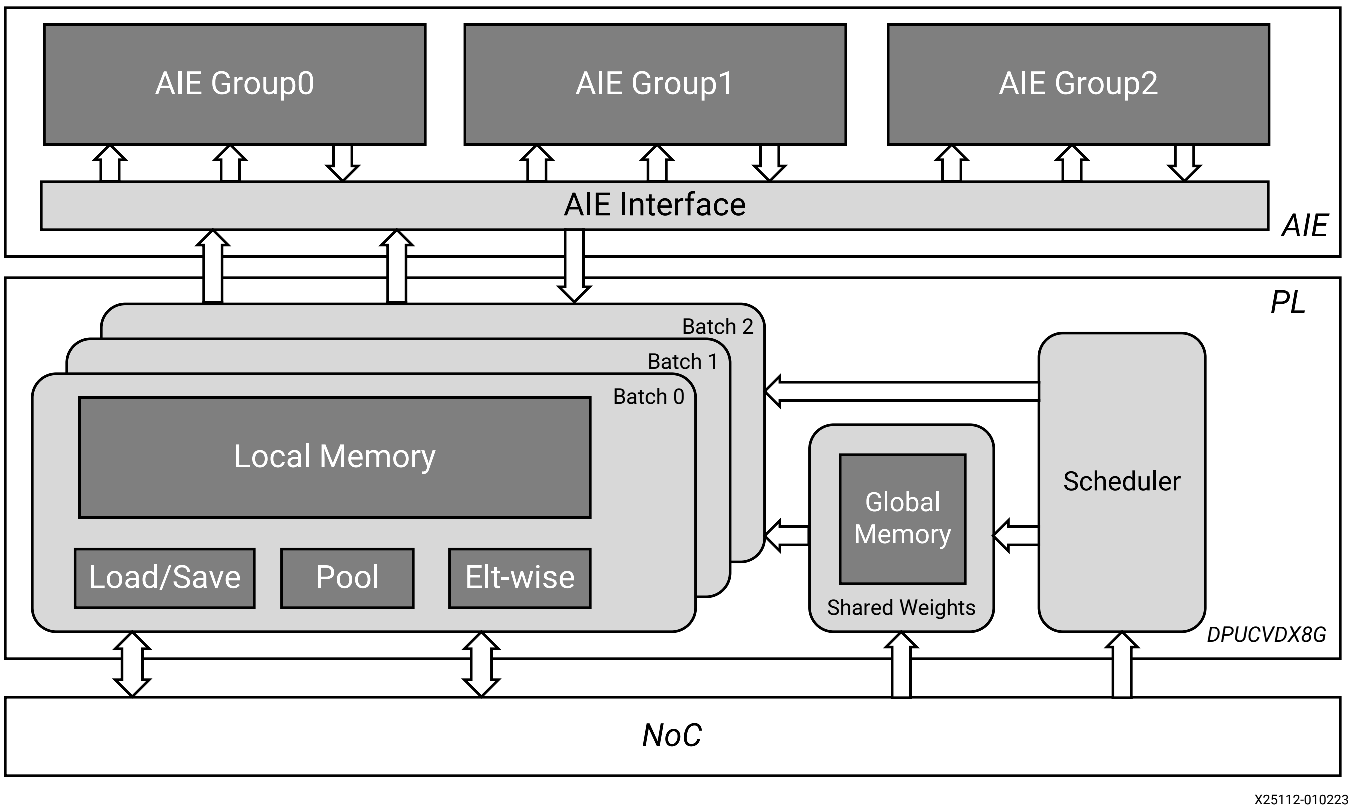


Versal AI Core Series: DPUCVDX8G

The DPUCVDX8G is a high-performance general CNN processing engine optimized for the Versal AI Core Series. The Versal devices can provide superior performance/watt over conventional FPGAs, CPUs, and GPUs. The DPUCVDX8G is composed of AI Engines and PL circuits. This IP is user-configurable and exposes several parameters which can be specified to optimize AI Engines and PL resources or customize features.

The top-level block diagram of DPUCVDX8G is shown in the following figure.

Figure 5: **DPUCVDX8G Architecture**

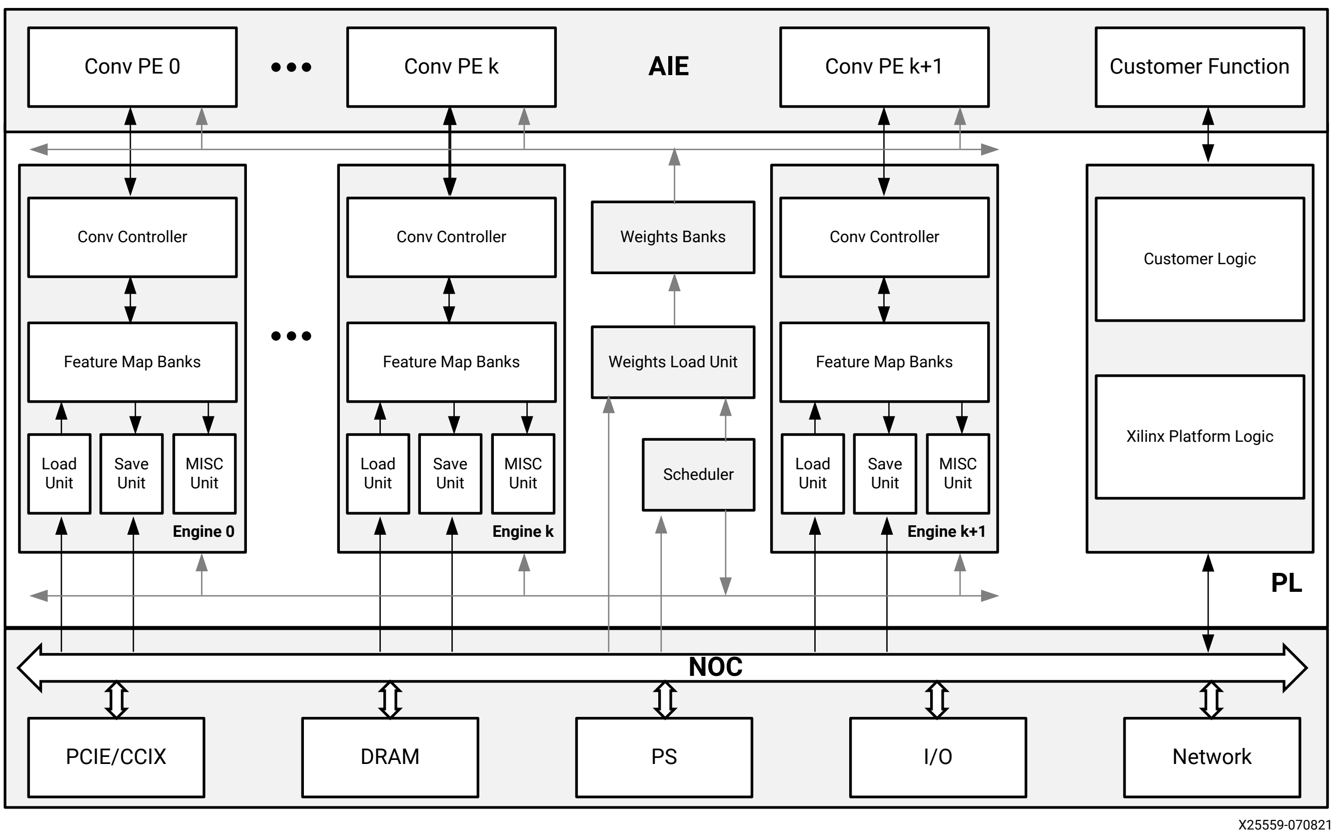


Versal AI Core Series: DPUCVDX8H

The DPUCVDX8H is a high-performance and high-throughput general CNN processing engine optimized for the Versal AI Core series. Besides traditional program logic, Versal devices integrate high performance AI engine arrays, high bandwidth NoCs, DDR/LPDDR controllers, and other high-speed interfaces that can provide superior performance/watt over conventional FPGAs, CPUs, and GPUs. The DPUCVDX8H is implemented on Versal devices to leverage these benefits. You can configure the parameters to meet your data center application requirements.

The top-level block diagram of the DPUCVDX8H is shown in the following figure.

Figure 6: **DPUCVDX8H Block Diagram**

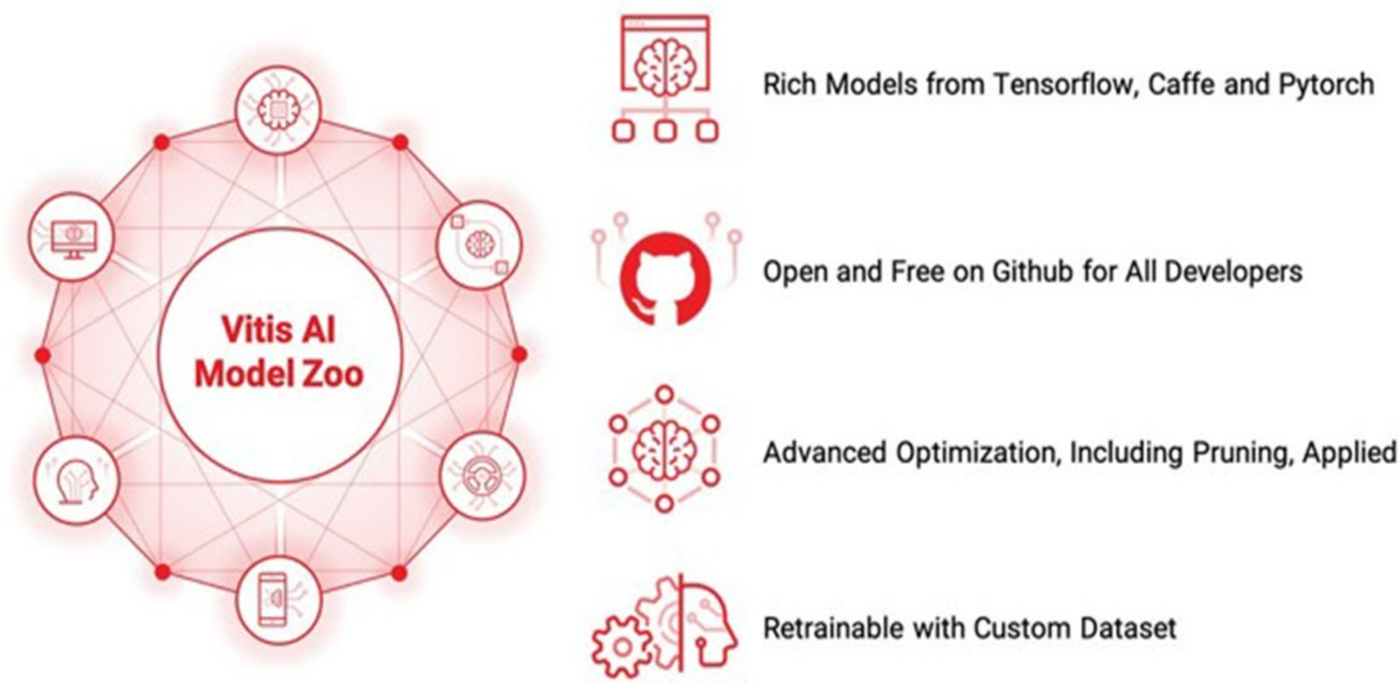


Vitis AI Model Zoo

The Vitis AI Model Zoo includes optimized deep learning models to speed up the deployment of deep learning inference on Xilinx platforms. These models cover different applications, including ADAS/AD, video surveillance, robotics, and data center. You can get started with these pre-trained models to enjoy the benefits of deep learning acceleration.

For more information, see Vitis AI Model Zoo on GitHub.

Figure 7: **Vitis AI Model Zoo**



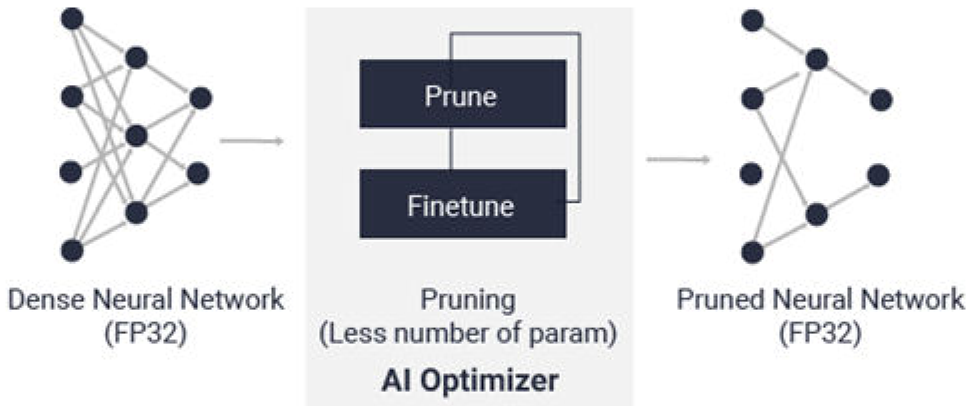
Note: Caffe has been deprecated from Vitis™ AI 2.5. For information on Caffe, see Vitis AI 2.0 User Guide.

Vitis AI Optimizer

With world-leading model compression technology, you can reduce model complexity by 5x to 50x with minimal accuracy degradation. See *Vitis AI Optimizer User Guide* (UG1333) for information on the Vitis AI Optimizer.

The Vitis AI optimizer requires a commercial license to run. Contact your Xilinx sales representative for more information.

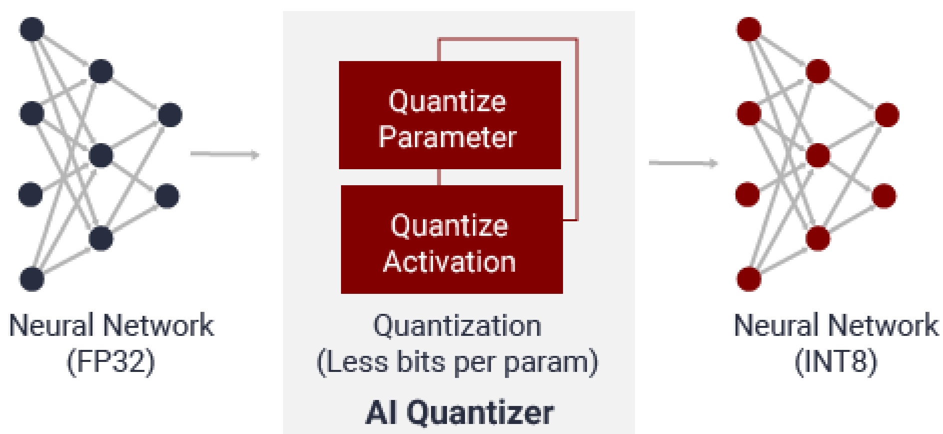
Figure 8: **Vitis AI Optimizer**



Vitis AI Quantizer

By converting the 32-bit floating-point weights and activations to fixed-point like INT8, the Vitis AI quantizer can reduce the computing complexity without losing prediction accuracy. The fixed-point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model.

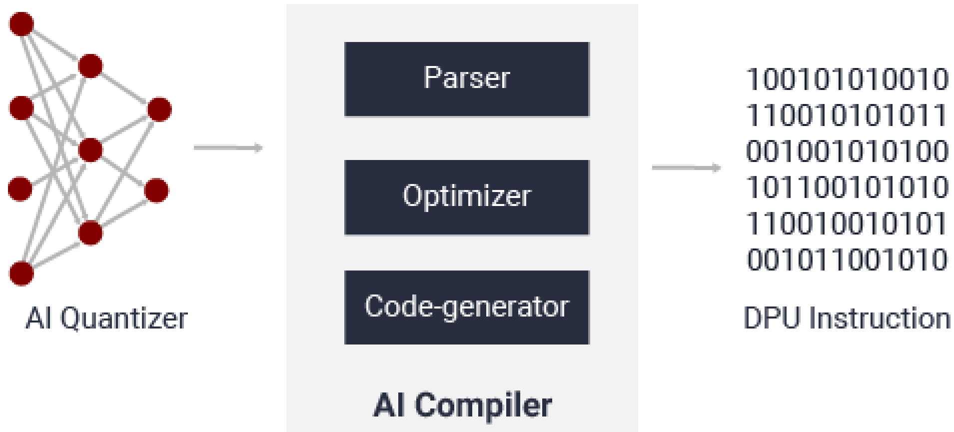
Figure 9: **Vitis AI Quantizer**



Vitis AI Compiler

The Vitis AI compiler maps the AI model to a highly-efficient instruction set and dataflow model. It also performs sophisticated optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible.

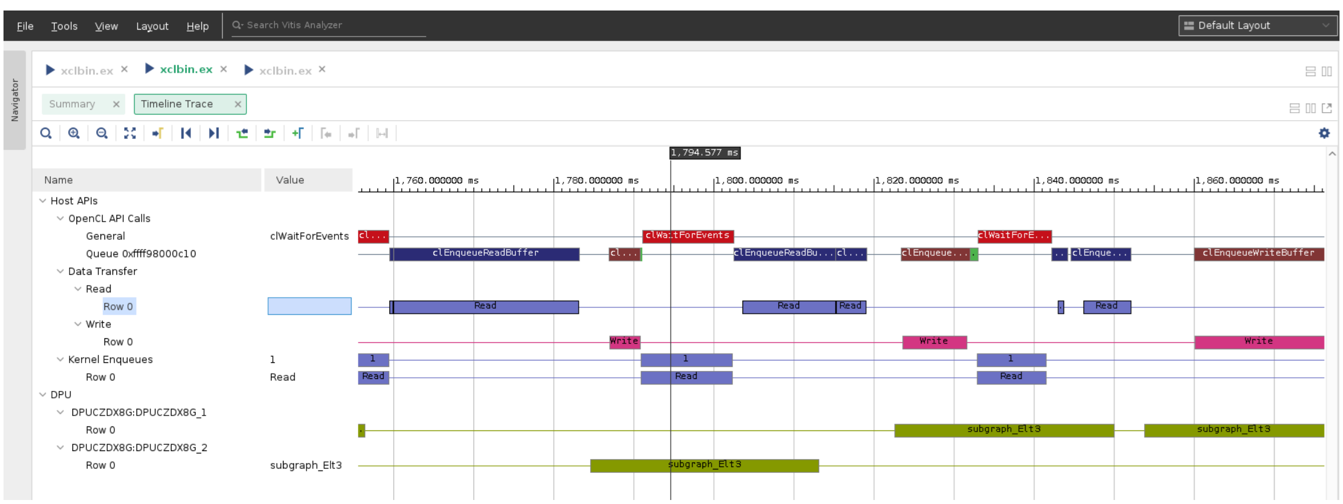
Figure 10: **Vitis AI Complier**



Vitis AI Profiler

The Vitis AI profiler profiles and visualizes AI applications to find bottlenecks and allocates computing resources among different devices. It is easy to use and requires no code changes. It can trace function calls and run time, and also collect hardware information, including CPU, DPU, and memory utilization.

Figure 11: **Vitis AI Profiler**

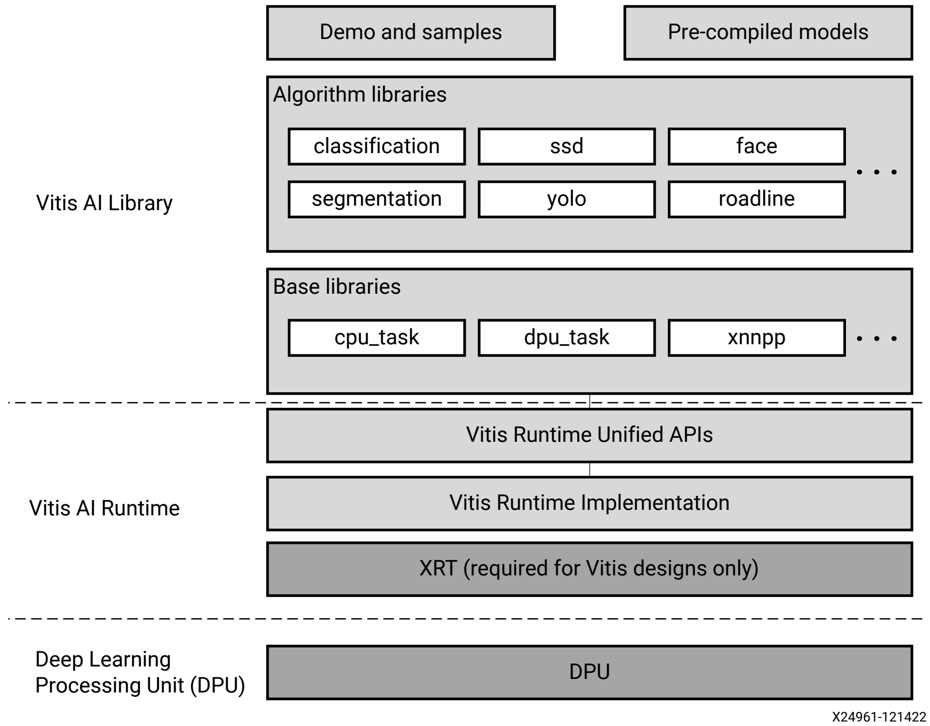


Vitis AI Library

The Vitis AI Library is a set of high-level libraries and APIs built for efficient AI inference with DPUs. It fully supports the XRT and is built on Vitis AI runtime with Vitis runtime unified APIs.

The Vitis AI Library provides an easy-to-use and unified interface by encapsulating many efficient and high-quality neural networks. This simplifies the use of deep-learning neural networks, even for users without knowledge of deep-learning or FPGAs. The Vitis AI Library allows you to focus more on developing your applications rather than the underlying hardware.

Figure 12: **Vitis AI Library**



Vitis AI Runtime

The Vitis AI runtime enables applications to use the unified high-level runtime API for both Cloud and Edge making Cloud-to-Edge deployments seamless and efficient.

Following are the features for the AI runtime API:

- Asynchronous submission of jobs to the accelerator
- Asynchronous collection of jobs from the accelerator
- C++ and Python implementations
- Support for multi-threading and multi-process execution

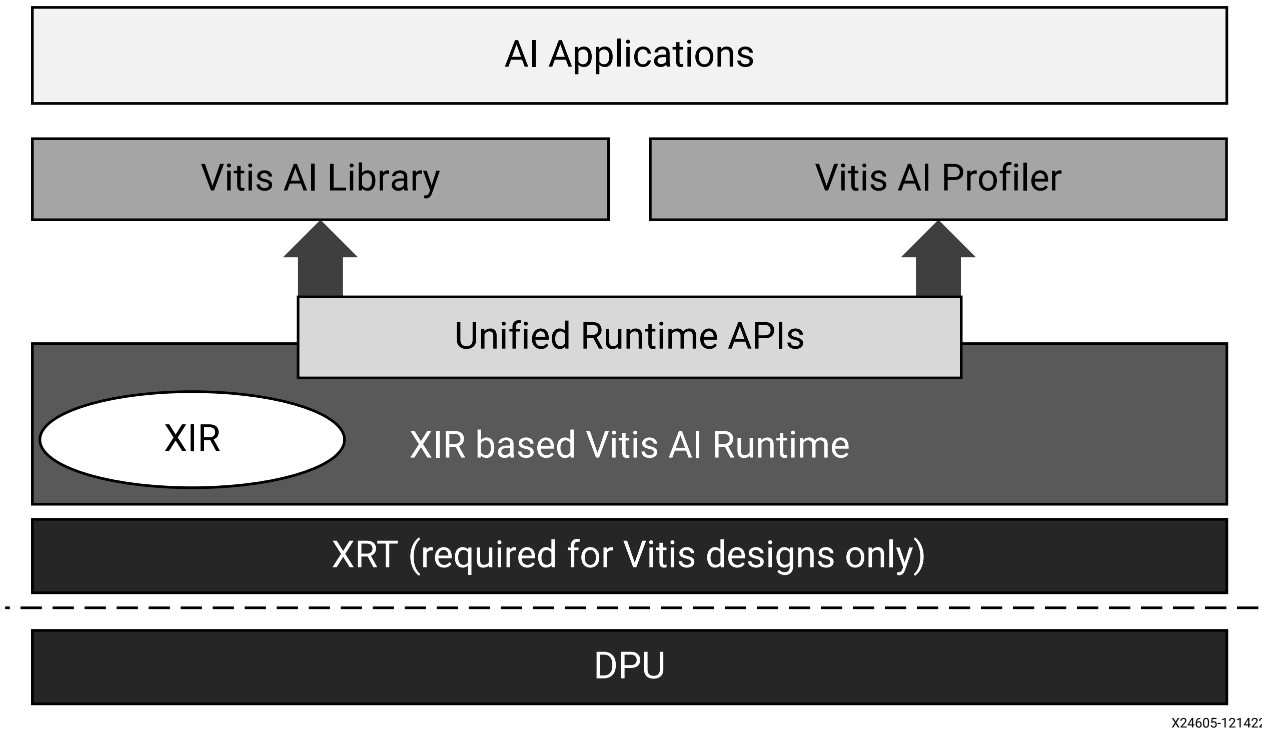
The Vitis AI Runtime (VART) is the next generation runtime suitable for devices based on DPUCZDX8G, DPUCVDX8G, and DPUCVDX8H.

- DPUCZDX8G is used for Edge devices, such as the ZCU102 and the ZCU104 evaluation boards, and the KV260 starter kit.

- DPUCVDX8G is used for the Versal evaluation boards, such as the VCK190 board.
- DPUCVDX8H is used for the Versal ACAP VCK5000 board.

The framework of VART is shown in the following figure. For this Vitis AI release, VART is based on XRT. XIR is the Xilinx Intermediate Representation.

Figure 13: **VART Stack**



Vitis AI Containers

The Vitis AI 3.0 release uses containers to distribute the AI software. The release consists of the following components.

- Tools container
- Runtime package for Zynq UltraScale+ MPSoC and VCK190
- Public GitHub for examples (https://github.com/Xilinx/Vitis-AI)
- Vitis AI Model Zoo (https://github.com/Xilinx/Vitis-AI/tree/v3.0/model_zoo)

Tools Container

The tools container consists of the following:

- Containers distributed through Docker Hub: https://hub.docker.com/r/xilinx/vitis-ai/tags
- Unified compiler flow includes:
  - Compiler flow for DPUCZDX8G (Edge)
  - Compiler flow for DPUCVDX8G (Edge)
  - Compiler flow for DPUCVDX8H (Data Center)
- Pre-built conda environment to run frameworks:
  - `conda activate vitis-ai-tensorflow` for TensorFlow-based flows
  - `conda activate vitis-ai-tensorflow2` for TensorFlow2-based flows
  - `conda activate vitis-ai-pytorch` for PyTorch-based flows

    Note: For WeGO workflow in PyTorch, please activate following conda environment:`conda activate vitis-ai-wego-torch`
- Versal Runtime tools

Note: Caffe has been deprecated from Vitis™ AI 2.5. For information on Caffe, see Vitis AI 2.0 User Guide.

Runtime Packages for Zynq UltraScale+ MPSoC and VCK190

For MPSoC, the runtime packages are located at https://github.com/Xilinx/Vitis-AI/tree/v3.0/board_setup/mpsoc. For VCK190, the runtime packages are located at https://github.com/Xilinx/Vitis-AI/tree/v3.0/board_setup/vck190. It contains the following items:

- PetaLinux SDK and Cross compiler tool chain
- Vitis AI board packages based on the 2022.2 release, including the Vitis AI new generation runtime VART.

Models and overlaybins are located at https://github.com/Xilinx/Vitis-AI. You can also find the following items here:

- All public pre-trained models
- Overlays for Zynq UltraScale+ MPSoCs and Versal accelerator cards
- Scripts to automate the downloading and installation processes for models and overlays.

Minimum System Requirements

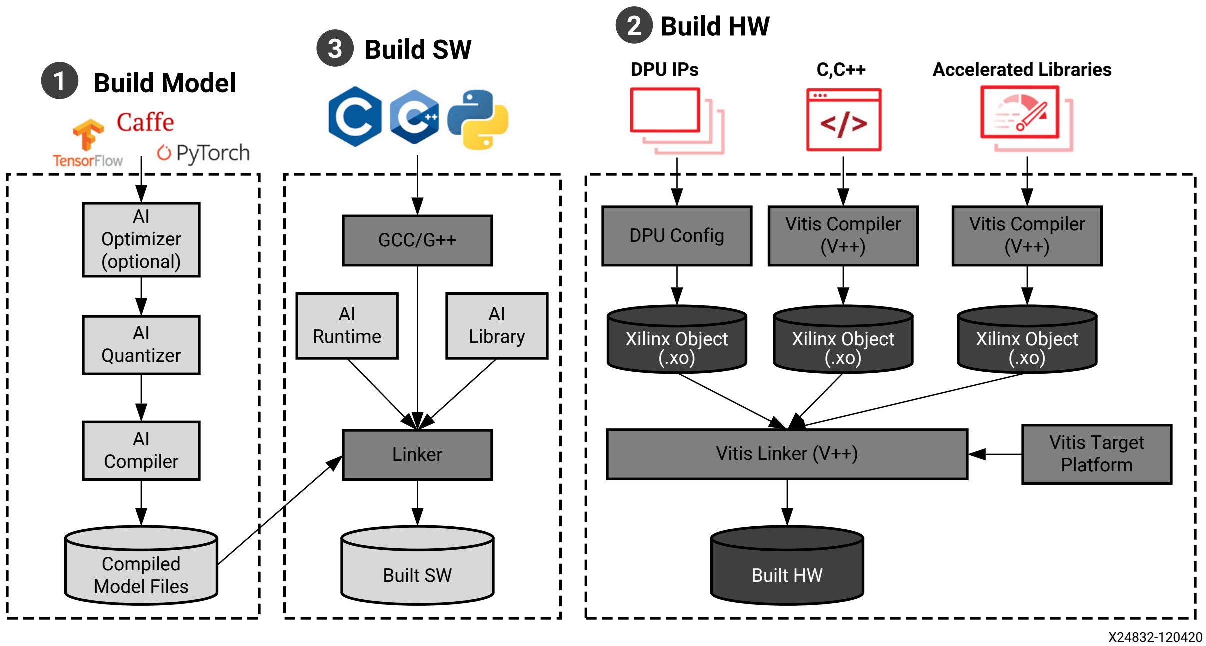
The following URLS contains the system requirements for running containers as well as Versal boards:

https://xilinx.github.io/Vitis-AI/docs/reference/system_requirements.html

Development Flow Overview

The recommended development flow for Vitis™ AI is illustrated as the following figure. Vitis AI and Vitis IDE are needed for this flow which has three basic steps:

Figure 14: **Vitis AI Flow**



1. A custom hardware platform is built using the Vitis software platform based on the Vitis Target Platform. The generated hardware includes the DPU IP and other kernels. In the Vitis AI release package, pre-built SD card images (for ZCU102/104, KV260 and VCK190) and Versal shells are included for quick start and application development. You can also use the Vivado® Design Suite to integrate the DPU and build the custom hardware to suit your need. For more information, see Chapter 8: Integrating the DPU into Custom Platforms.

2. The Vitis AI toolchain in the host machine is used to build the model. It takes the pre-trained floating models as the input and runs them through the AI Optimizer (optional).

3. You can build executable software which runs on the built hardware. You can write your applications with C++ or Python which calls the Vitis AI Runtime and Vitis AI Library to load and run the compiled model files.

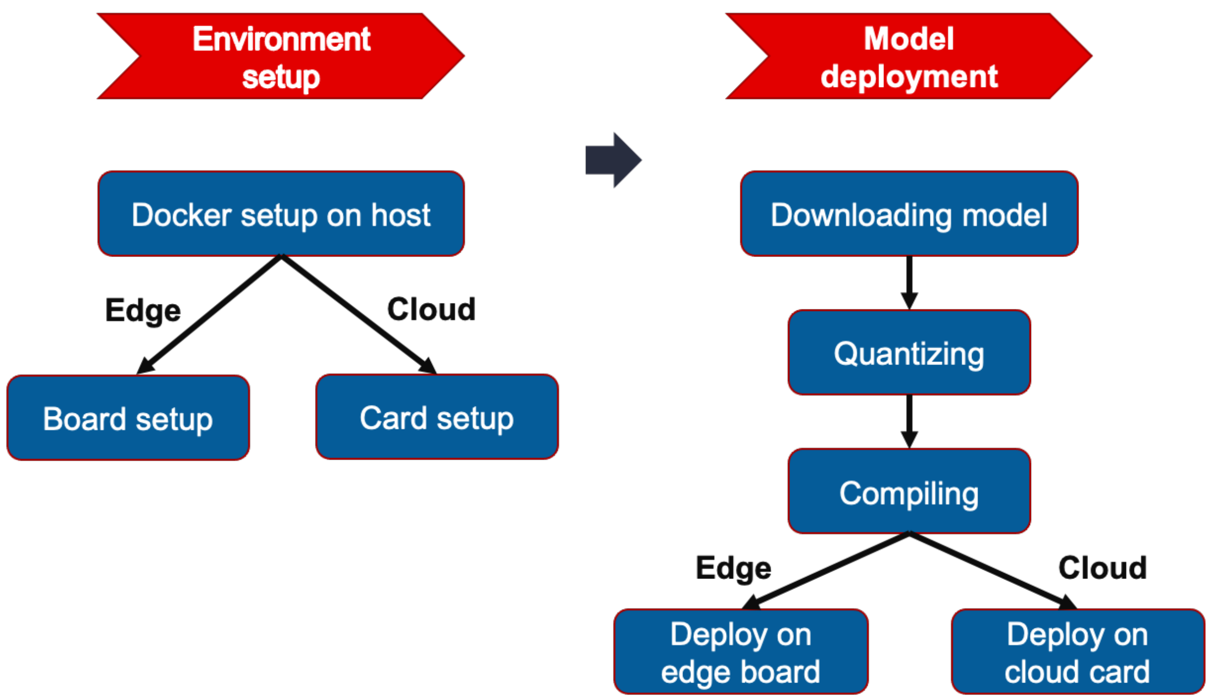
Chapter 2

Getting Started

Quick Start

This is a quick start section for Vitis™ AI. You can refer to it to quickly run the tensorflow `resnet50` model on the edge or cloud platform. For edge, use the `ZCU102` example, and for cloud, use `VERSAL` DPUCVDX8H-8PE.

Figure 15: **Quick Start of Vitis AI**



Environment Setup

Docker Setup on the Host

To set up the Docker on the host, follow these steps:

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

```
[Host]$ git clone https://github.com/Xilinx/Vitis-AI
[Host]$ cd Vitis-AI
```

2. Install Docker. If Docker is not installed on your machine yet, see the official Docker documentation.
3. Ensure your Linux user is in the group docker.
4. Download the latest Vitis AI Docker.

```
[Host]$ docker pull xilinx/vitis-ai-<pytorch/tensorflow/tensorflow2>-
cpu:latest
```

For more information about docker, see System Requirements and Docker installation.

Board Setup (Edge)

1. Run the following command to install the cross-compilation system environment.

```
[Host]$ cd Vitis-AI/board_setup/mpsoc
[Host]$ ./host_cross_compiler_setup.sh
```

2. Download and set up the board Image.
   - Download the SD card system image files from the Xilinx website.

     Use ZCU102 for an example (Registration is required for downloading this system image file from this public link).
   - Use the Etcher software to burn the image file onto the SD card.
   - Insert the SD card with the image into the destination board.
   - Plug in the power and boot the board using the serial port to operate on the system.
   - Set up the IP information of the board using the serial port.

For more information on boards, see MPSoC Setup and VCK190 Setup.

Card Setup (Cloud)

To explain the card set up, this topic assumes that you are using CentOS 7.9 on the host and have inserted the Versal VCK5000 card into the PCIe slot.

1. Execute the following command to install Versal card target platform, XRT, and XRM.

```
[Host]$ cd Vitis-AI/board_setup/vck5000
[Host]$ source ./install.sh
```

2. After the script is executed successfully, manually reboot the host server once.

```
[Host]$ sudo reboot
```

3. Use the `/opt/xilinx/xrt/bin/xbutil examine` to check that the installation was successful. The results should appear as follows:

```
System Configuration
  OS Name              : Linux
  Release              : 3.10.0-1127.el7.x86_64
  Version              : #1 SMP Tue Mar 31 23:36:51 UTC 2020
  Machine              : x86_64
  CPU Cores            : 12
  Memory               : 46556 MB
  Distribution         : CentOS Linux 7 (Core)
  GLIBC                : 2.17
  Model                : PowerEdge R740

XRT
  Version              : 2.14.354
  Branch               : 2022.2
  Hash                 : 43926231f7183688add2dccfd391b36a1f000bea
  Hash Date            : 2022-10-08 16:50:02
  XOCL                 : 2.14.354, 43926231f7183688add2dccfd391b36a1f000bea
  XCLMGMT              : 2.14.354, 43926231f7183688add2dccfd391b36a1f000bea

Devices present
BDF             :  Shell                          Platform UUID                          Device ID       Device Ready*
-------------------------------------------------------------------------------------------------------------------------
[0000:d8:00.1]  :  xilinx_vck5000_gen4x8_qdma_base_2  D44E2200-9FE8-5F2A-2B10-3AB6A5063AAB  user(inst=128)  Yes


* Devices that are not ready will have reduced functionality when using XRT tools
```

For more information on boards, see Setting up a Versal Accelerator Card.

Model Deployment

Downloading float model from model zoo

Take `tensorflow resnet50` for example.

```
[Host]$ cd Vitis-AI
[Host]$ wget https://www.xilinx.com/bin/public/openDownload?
filename=tf_resnetv1_50_imagenet_224_224_6.97G_3.0.zip -O
tf_resnetv1_50_imagenet_224_224_6.97G_.0.zip
[Host]$ unzip tf_resnetv1_50_imagenet_224_224_6.97G_3.0.zip
```

Find information on more models, see AI-Model-Zoo

Quantizing the model

- Prepare dataset. For `tf_resnet50` model, download the calibration images from Imagenet_calib.tar.gz, and copy into Vitis-AI folder (Full dataset is from ImageNet).
- Launch the docker image.

  ```
  [Host]$ ./docker_run.sh xilinx/vitis-ai-tensorflow-cpu:latest
  ```

- Quantize the model.

Set `CALIB_BATCH_SIZE` in the `tf_resnetv1_50_imagenet_224_224_6.97G_3.0/code/quantize/config.ini` to 5. Then run

```
[Docker]$ conda activate vitis-ai-tensorflow
[Docker]$ tar -xzvf Imagenet_calib.tar.gz -C
tf_resnetv1_50_imagenet_224_224_6.97G_3.0/data
[Docker]$ cd tf_resnetv1_50_imagenet_224_224_6.97G_3.0/code/quantize
[Docker]$ bash quantize.sh
```

After running `quantize.sh`, the quantized model are available in `tf_resnetv1_50_imagenet_224_224_6.97G_3.0/quantized`

Compiling the model

- For edge, take `ZCU102` target as an example.

```
[Docker]$ cd ../..
[Docker]$ vai_c_tensorflow -f ./quantized/quantize_eval_model.pb -a /opt/
vitis_ai/compiler/arch/DPUCZDX8G/ZCU102/arch.json -o ./compiled -n
resnet50_tf
```

- For data center, take DPUCVDX8H-8PE target as an example.

```
[Docker]$ cd ../..
[Docker]$ vai_c_tensorflow -f ./quantized/quantize_eval_model.pb -a /opt/
vitis_ai/compiler/arch/DPUCVDX8H/VCK50008PE/arch.json -o ./compiled -n
resnet50_tf
```

Deployment on Edge boards

- Copy the compiled model to the board.

```
[Host]$ scp tf_resnetv1_50_imagenet_224_224_6.97G_3.0/compiled/
resnet50_tf.xmodel root@[BOARD_IP]:~
```

- Download the vitis_ai_runtime_r3.0.0_image_video.tar.gz test image and unzip the `vitis_ai_runtime_r3.0.0_image_video.tar.gz` package on the target.

```
[Target]# cd ~
[Target]# tar -xzvf vitis_ai_runtime_r*3.0*_image_video.tar.gz -C Vitis-
AI/examples/vai_runtime
```

- Run the resnet50 example.

```
[Target]# cd ~/Vitis-AI/examples/vai_runtime/resnet50
[Target]# ./resnet50 ~/resnet50_tf.xmodel
```

The result is shown below.

```
root@xilinx-zcu102-20221:~/Vitis-AI/examples/VART/resnet50# ./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
WARNING: Logging before InitGoogleLogging() is written to STDERR
I0516 06:06:17.544564  8335 main.cc:292] create running for subgraph: subgraph_conv1

Image : 001.jpg
top[0] prob = 0.982662  name = brain coral
top[1] prob = 0.008502  name = coral reef
top[2] prob = 0.006621  name = jackfruit, jak, jack
top[3] prob = 0.000543  name = puffer, pufferfish, blowfish, globefish
top[4] prob = 0.000330  name = eel
```

***Note*:** To improve the user experience, the Vitis AI Runtime packages, VART samples, Vitis AI Library samples and models have been built into the board image. Find more information refer to Quick Start For Edge

Deployment on Data Center cards

- Copy the compiled model to the work place

```
[Docker]$ cp compiled/resnet50_tf.xmodel ~
```

- Download the vitis_ai_runtime_r3.0.0_image_video.tar.gz package and unzip it in the docker container.

```
[Docker]# cd /workspace/examples
[Docker]# wget https://www.xilinx.com/bin/public/openDownload?
filename=vitis_ai_runtime_r3.0.0_image_video.tar.gz -O
vitis_ai_runtime_r3.0.0_image_video.tar.gz
[Docker]# tar -xzvf vitis_ai_runtime_r3.0.0_image_video.tar.gz -C
vai_runtime
```

- Environment variable setup in docker container.

```
source /workspace/board_setup/vck5000/setup.sh DPUCVDX8H_8pe_normal
```

- Build and Run the resnet50 example.

```
[Docker] cd /workspace/examples/vai_runtime/resnet50
[Docker] bash -x build.sh
[Docker] ./resnet50 ~/resnet50_tf.xmodel
```

The result is shown below.

```
(vitis-ai-tensorflow) Vitis-AI /workspace/demo/VART/resnet50 > ./resnet50 ~/resnet50_tf.xmodel
WARNING: Logging before InitGoogleLogging() is written to STDERR
I0125 02:18:10.667263   431 main.cc:292] create running for subgraph: subgraph_resnet_v1_50/block1/unit_1/bottleneck_v1/add

Image : 001.jpg
top[0] prob = 0.990261  name = brain coral
top[1] prob = 0.005196  name = coral reef
top[2] prob = 0.001159  name = puffer, pufferfish, blowfish, globefish
top[3] prob = 0.000903  name = eel
top[4] prob = 0.000427  name = rock beauty, Holocanthus tricolor
```

***Note*:** To improve the user experience, the pre-built models are already available in Model Zoo. For more information, see Quick Start For Cloud.

Installation and Setup

Downloading Vitis AI Development Kit

The Vitis™ AI software is made available through Docker Hub. Vitis AI consists of the following two packages:

- Vitis AI tools docker `xilinx/vitis-ai-<pytorch/tensorflow/tensorflow2>-cpu:latest`
- Vitis AI runtime package for Edge

The tools container contains the Vitis AI quantizer, AI compiler, and AI runtime for Data Center DPUs. The Vitis AI runtime package for edge is for edge DPU development, which includes Vitis AI runtime installation package for Xilinx® evaluation boards and Arm® GCC cross-compilation toolchain.

Xilinx FPGA devices and evaluation boards supported by the Vitis AI development kit v3.0 release are:

- Data Center
    - Versal ACAP VCK5000 board
- Edge
    - Zynq® UltraScale+™ MPSoC ZCU102 and ZCU104 evaluation boards
    - Kria™ KV260 Vision AI starter kit
    - Versal ACAP VCK190 board
    - Versal ACAP VEK280 board

Setting Up the Host

The following two options are available for installing the containers with the Vitis AI tools and resources.

1. Pre-built containers on Docker Hub: xilinx/vitis-ai
2. Build containers locally with Docker recipes: Docker Recipes

Installing the Tools

Use the following steps for installation:

1. Install Docker, if Docker is not installed on your machine.

2. Follow the Post-installation steps for Linux to ensure that your Linux user is in the group Docker.
3. Clone the Vitis AI repository to obtain the examples, reference code, and scripts.

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

4. Run Docker Container

   Refer to https://xilinx.github.io/Vitis-AI/docs/reference/docker_image_versions.html for the dedicated docker version number.

   a. Run the CPU image from Docker Hub:

   ```
   docker pull xilinx/vitis-ai:<x.y.z>
   ./docker_run.sh xilinx/vitis-ai
   ```

   b. Build the CPU image locally and run it:

   ```
   cd setup/docker
   ./docker_build_cpu.sh

   # After build finished
   cd ..
   ./docker_run.sh xilinx/vitis-ai-cpu:<x.y.z>
   ```

   c. Build the GPU image locally and run it:

   ```
   cd setup/docker
   ./docker_build_gpu.sh

   # After build finished
   cd ..
   ./docker_run.sh xilinx/vitis-ai-gpu:<x.y.z>
   ```

Setting Up the Host (Using VART)

For Edge

Use the following steps to set up the host for Edge:

1. Install the cross-compilation system environment.

```
cd Vitis-AI/board_setup/mpsoc
./host_cross_compiler_setup.sh
```

   Note: The `~/petalinux_sdk_2022.2` path is recommended for the installation. Regardless of the path you choose for the installation, make sure the path has read-write permissions. In this section, it is installed in `~/petalinux_sdk_2022.2`

2. When the installation is complete, follow the prompts and enter the following command.

```
source ~/petalinux_sdk_2022.2/environment-setup-cortexa72-cortexa53-
xilinx-linux
```

Note: If you close the current terminal, you need to re-execute the above instructions in the new terminal to set up the environment.

3. Cross compile the sample taking `resnet50` as an example.

```
cd Vitis-AI/examples/vai_runtime/resnet50
bash -x build.sh
```

If the compilation process does not report any error and the executable file `resnet50` is generated, then the host environment is installed correctly.

For Data Center

Use the following steps to set up the host. These steps apply to the Versal ACAP VCK5000 board.

1. Start the Docker container. After the Docker image is loaded and running, the Vitis AI runtime is automatically installed in the docker system.
2. Follow the instructions to set up the Versal ACAP VCK5000 board here to install the XRT/XRM platform and the DPU xclbin file.

Note: If there is more than one card installed on the server and you want to specify which cards will be used for inference, set `XLNX_ENABLE_DEVICES`. It takes the following options:

- To use device 0 for the DPU, set `export XLNX_ENABLE_DEVICES=0`.
- To use device 0, device 1, and device 2 for the DPU, set `export XLNX_ENABLE_DEVICES=0,1,2`.
- By default, all available devices are used for the DPU if you do not set this environment variable.

For more information, see Answer Record 75975.

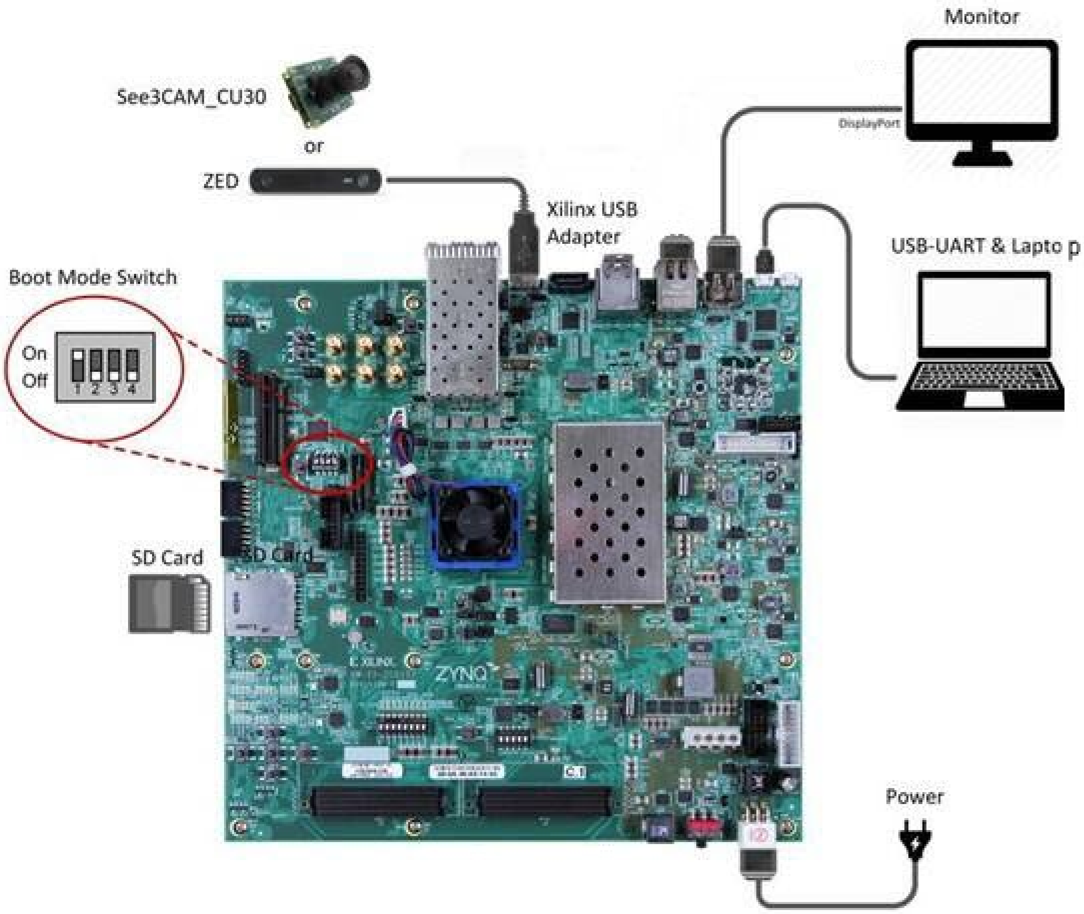
Setting Up the Evaluation Board

Setting Up the ZCU102/ZCU104/KV260/VCK190 Evaluation Board

The Xilinx ZCU102 evaluation board uses the mid-range ZU9 Zynq® UltraScale+™ MPSoC to enable you to jumpstart your machine learning applications. For more information on the ZCU102 board, see the ZCU102 product page on the Xilinx website: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

The Vitis AI pre-built board images support the ZCU102 interfaces shown in the following figure:

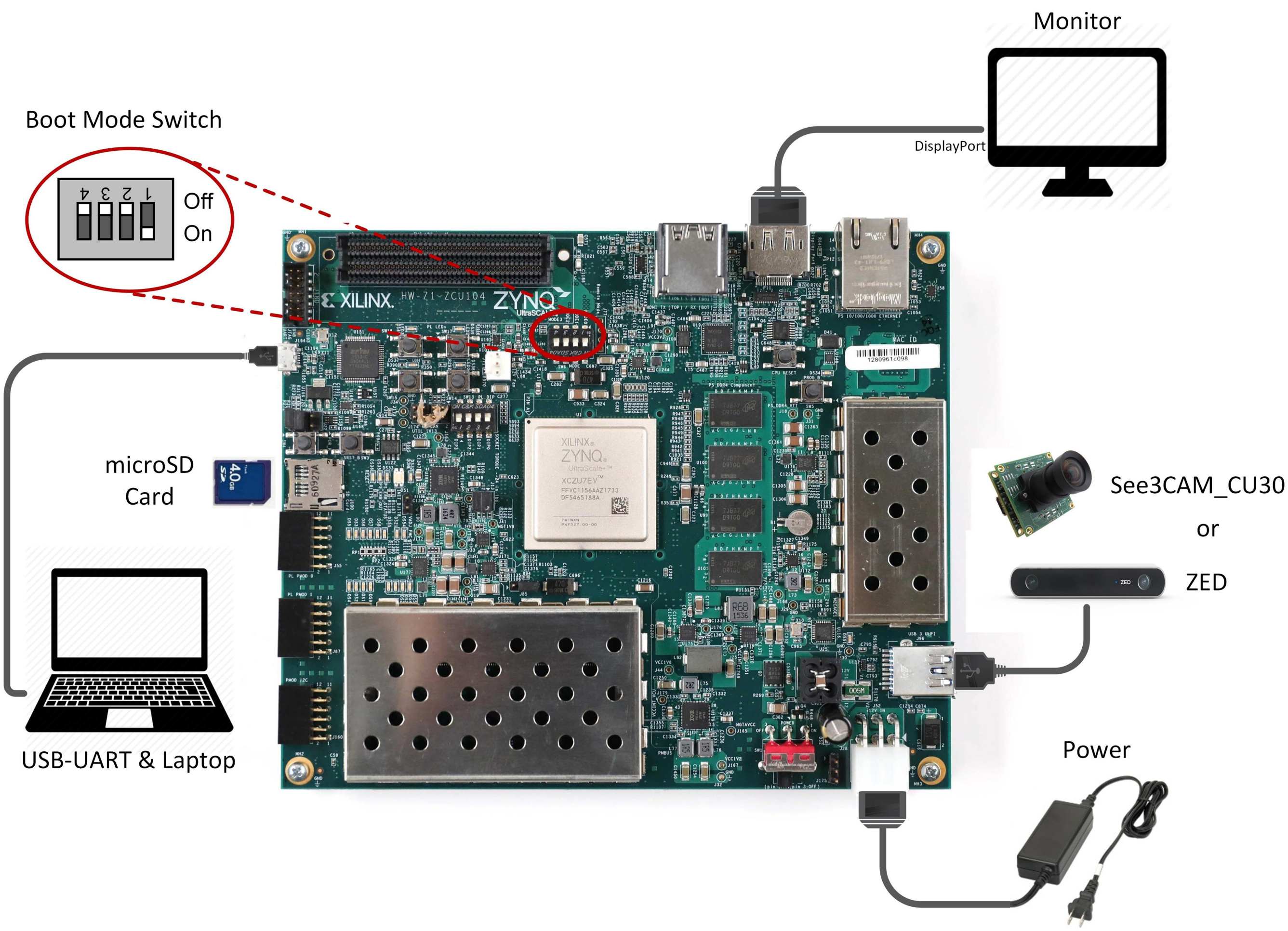
Figure 16: **Xilinx ZCU102 Evaluation Board and Peripheral Connections**



The Xilinx ZCU104 evaluation board uses the mid-range ZU7 Zynq UltraScale+ device to enable you to jumpstart your machine learning applications. For more information on the ZCU104 board, see the Xilinx website: https://www.xilinx.com/products/boards-and-kits/zcu104.html.

The Vitis AI pre-built board images support the ZCU104 interfaces shown in the following figure:

Figure 17: **Xilinx ZCU104 Evaluation Board and Peripheral Connections**



The KV260 Starter Kit is fully featured and optimized for the K26 SOM. Designed for Vision AI applications, the KV260 is the fastest way to develop unique solutions for production volume deployment with the K26 SOM. For more information on the KV260 Starter Kit, see the KV260 product page on the Xilinx website: https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit.html.

The VCK190 kit is the first Versal AI Core series evaluation kit, enabling designers to develop solutions using AI and DSP engines capable of delivering over 100X greater compute performance than today's server-class CPUs.

With a range of connectivity options and standardized development flows, the VCK190 kit features the Versal AI Core series VC1902 device, providing the portfolio's highest AI inference and signal processing throughput.

For more information on the VCK190 board, see the VCK190 product page on the Xilinx website: https://www.xilinx.com/products/boards-and-kits/vck190.html.

IMPORTANT! *In the following sections, ZCU102 is used as an example.*

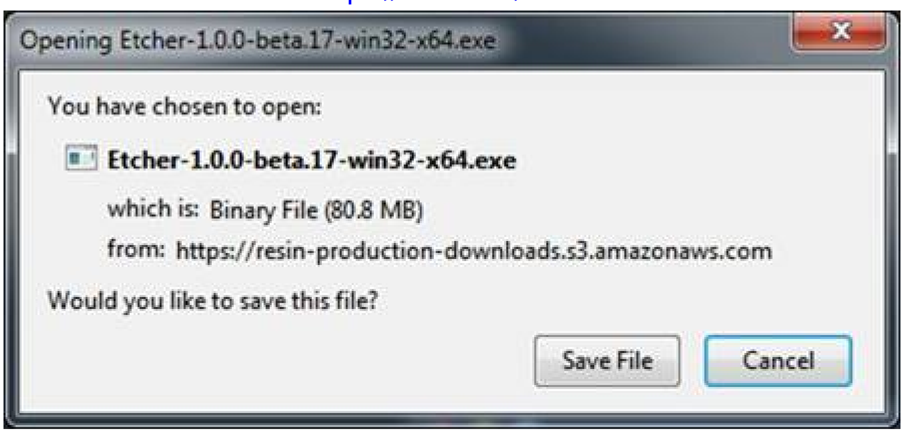
Flashing the OS Image to the SD Card

Download the pre-built Vitis AI board images from the following links:

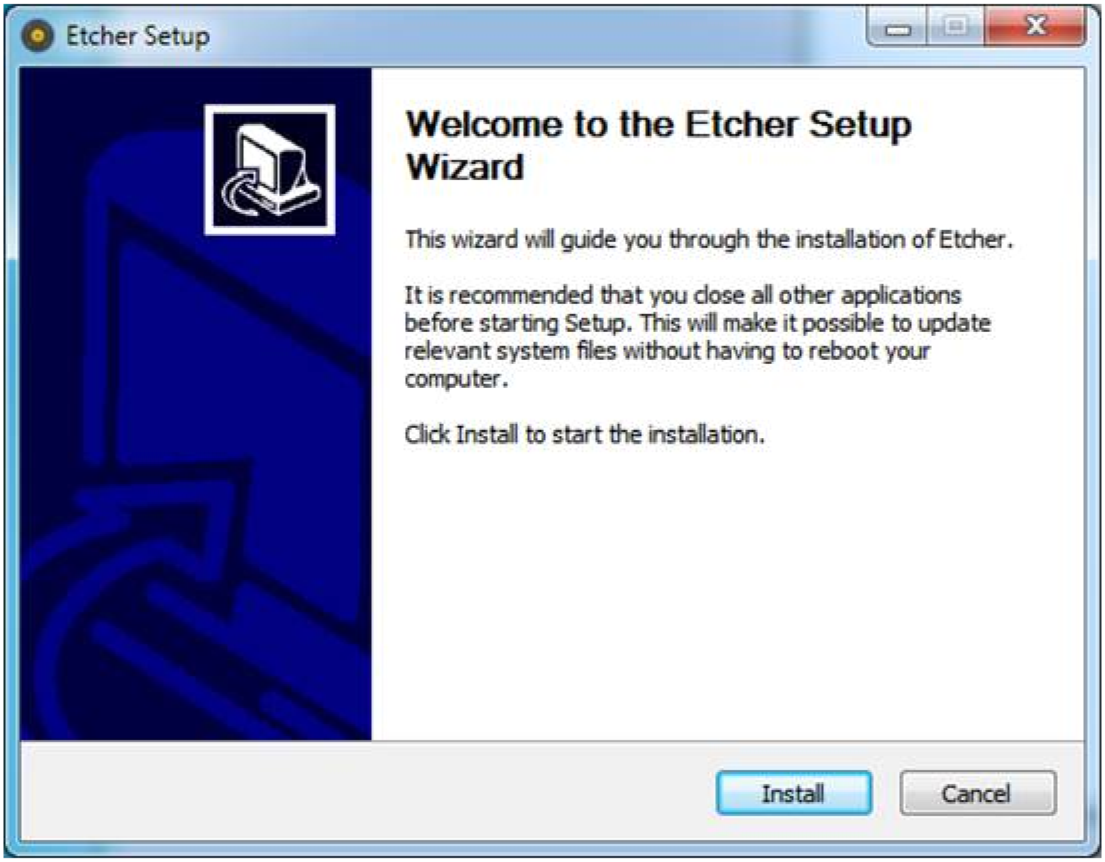
- For ZCU102, download from here
- For ZCU104, download from here
- For KV260, download from here
- For VCK190 production board, download from here

RECOMMENDED: *For flashing the SD card, use Etcher. It is a cross-platform tool for flashing OS images to SD cards, available for Windows, Linux, and Mac systems. The following example uses Windows.*

1. Download Etcher from: https://etcher.io/ and save the file as shown in the following figure.



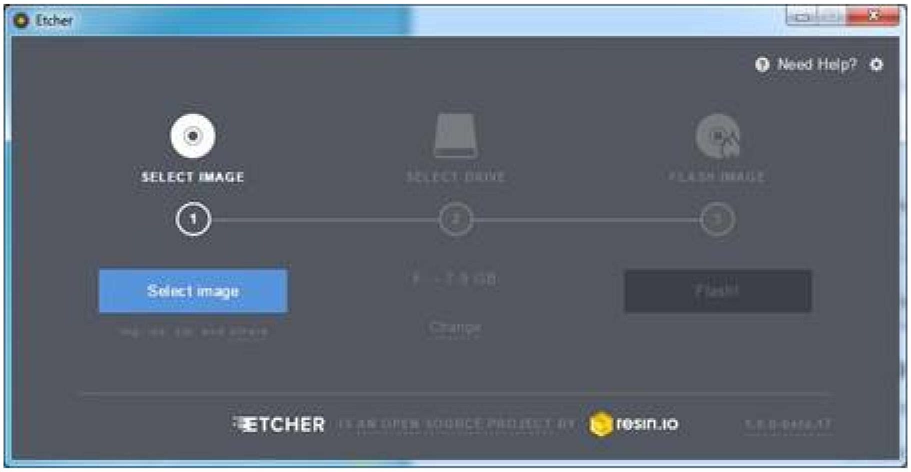
2. Install Etcher, as shown in the following figure.



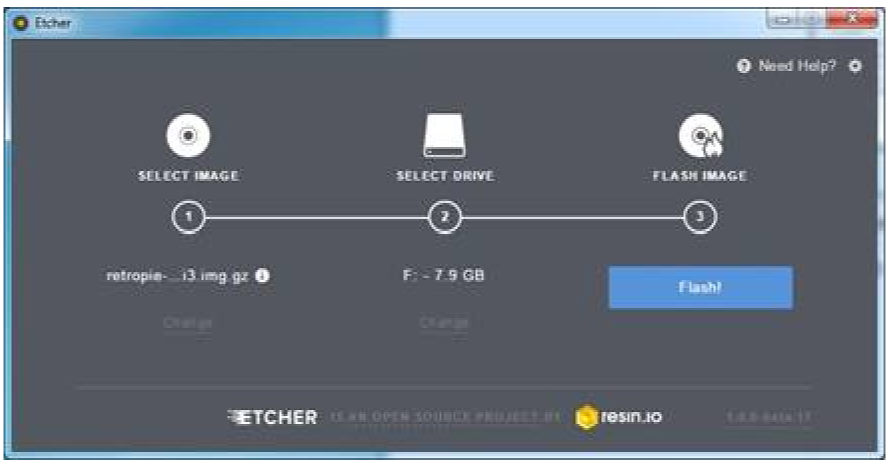
3. Eject any external storage devices such as USB flash drives and backup hard disks. This makes it easier to identify the SD card. Then, insert the SD card into the slot on your computer, or into the reader.
4. Run the Etcher program by double clicking on the Etcher icon shown in the following figure, or select it from the Start menu.

   Etcher launches, as shown in the following figure.



5. Select the image file by clicking **Select Image**. You can select a **.zip** or **.gz** compressed file.
6. Etcher tries to detect the SD drive. Verify the drive designation and the image size.
7. Click **Flash!**.



Booting the Evaluation Board

This example uses a ZCU102 board to illustrate how to boot a Vitis AI evaluation board. Follow these steps to boot the evaluation board.

1. Connect the power supply (12V ~ 5A).
2. Connect the USB-UART interface to the host, and connect other peripherals as required.
3. Turn on the power and wait for the system to boot.
4. Log in to the system.

> **IMPORTANT!** *The system executes a flash file system reconfiguration during the initial boot.*

5. The user must reboot the board for these configurations to take effect.

Accessing the Evaluation Board

There are three ways to access the ZCU102 board:

- USB-UART port
- Ethernet connection
- Standalone

USB-UART Port

Apart from monitoring the boot process and checking Linux kernel messages for debugging, you can log in to the board through the UART. The configuration parameters of the UART are shown in the following example. Log in to the system with username “root” and password “root.”

- baud rate: 115200 bps
- data bit: 8
- stop bit: 1
- no parity

Note: On a Linux system, you can use Minicom to connect to the target board directly; for a Windows system, a USB to UART driver is needed before connecting to the board through a serial port.

Using the Ethernet Interface

The ZCU102 board has an Ethernet interface, and SSH service is enabled by default. You can log into the system using an SSH client after the board has booted.

Use the `ifconfig` command via the UART interface to set the IP address of the board, then use the SSH client to log into the system.

Using the Board as a Standalone Embedded System

The ZCU102 board allows a keyboard, mouse, and monitor to be connected. After a successful boot, a Linux GUI desktop is displayed. You can then access the board as a standalone embedded system.

Installing Vitis AI Runtime on the Evaluation Board

To improve the user experience, the Vitis AI Runtime packages, VART samples, Vitis-AI-Library samples and models have been built into the board image. The examples are pre-compiled. Therefore, you do not need to install Vitis AI Runtime packages and model package on the board separately. However, you can still install the model or Vitis AI Runtime on your own image or on the official image by following these steps.

Establish an Ethernet connection and copy the Vitis™ AI runtime (VART) package from GitHub to the evaluation board. Then, set up a Vitis AI running environment for the ZCU102 board.

1. Download the `vitis-ai-runtime-3.0.x.tar.gz` from here. Untar it and copy the following files to the board using scp.

```
tar -xzvf vitis-ai-runtime-3.0.x.tar.gz
scp -r vitis-ai-runtime-3.0.x/aarch64/centos root@IP_OF_BOARD:~/
```

   ***Note*:** You can take the rpm package as a normal archive, and extract the contents on the host side, if you only need some of the libraries. Only model libraries can be independent, while the others are common libraries. The operation command is as follows.

```
rpm2cpio libvart-3.0.x-r<x>.aarch64.rpm | cpio -idmv
```

2. Log in to the board using ssh. You can also use the serial port to log in.
3. Install the Vitis AI runtime. Execute the following commands in order.

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

   You can also execute the following command to install the library one by one.

```
cd ~/centos
rpm -ivh --force libunilog-3.0.0-r<x>.aarch64.rpm
rpm -ivh --force libxir-3.0.0-r<x>.aarch64.rpm
rpm -ivh --force libtarget-factory-3.0.0-r<x>.aarch64.rpm
rpm -ivh --force libvart-3.0.0-r<x>.aarch64.rpm
rpm -ivh --force libvitis_ai_library-3.0.0-r<x>.aarch64.rpm
```

   After the installation is complete, the Vitis AI Runtime library will be installed under `/usr/lib`.

Setting Up the Custom Board

Vitis AI supports the official ZCU102/ZCU104 as well as user-defined boards.

If you want to run Vitis AI on your custom board, follow these steps. Ensure that you complete a step before proceeding to the next step.

1. Create the platform system of your custom board. For more information, see *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) and https://github.com/Xilinx/Vitis_Embedded_Platform_Source/tree/master/Xilinx_Official_Platforms.
2. Integrate the DPU IP. See https://github.com/Xilinx/Vitis-AI/tree/v3.0/dpu for more information.

   ***Note*:** After this step is completed, an `sd_card` directory and an `sd_card.img` image with DPU are created.

3. Install the Vitis AI libraries.

   There are two ways to install the Vitis AI libraries. One is to rebuild the system by configuring PetaLinux and the other is to install the Vitis AI libraries online. After you install the `Vitis-AI` libraries, the `Vitis-AI` dependent libraries are installed.

a. To rebuild the system by configuring PetaLinux:

If users want to install Vitis AI 3.0 into rootfs when generating system image by PetaLinux, users need to get the Vitis AI 3.0 recipes. The Vitis AI 3.0 recipes folder is located in `Vitis-AI/src/vai_petalinux_recipes/recipes-vitis-ai`.

1. Copy Vitis AI recipes folder into `<petalinux project>/project-spec/meta-user/`.

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

2. Edit `<petalinux project>/project-spec/meta-user/conf/user-rootfsconfig`, appending the lines:

```
CONFIG_vitis-ai-library
CONFIG_vitis-ai-library-dev
CONFIG_vitis-ai-library-dbg
```

3. `Source` petalinux tool and run `petalinux-config -c rootfs` command.

4. Select `user packages --->`.

5. Select `[*] vitis-ai-library` , save it and exit.

6. Run `petalinux-build`.

Note: If you want to compile the example on the target, select the `vitis-ai-library-dev` and `packagegroup-petalinux-self-hosted` and recompile the system.

Note: If you encounter errors like `cpio: cannot seek on output: Invalid argument` in step 6, enter `Image Packaging Configuration --->` and remove `cpio.gz` and `cpio.gz.u-boot` in `Root filesystem formats`, save them and exit. Run `petalinux-build` command again to recompile the system.

b. To install the Vitis AI libraries online, execute `dnf install vitis-ai-library` command on board directly.

Note: Before the release of Petalinux VAI3.0 update, the previous version of Vitis AI will be installed. Usually, Petalinux VAI3.0 update will be released approximately 1 month after Vitis 3.0 release.

Note: If you use this method, ensure that the board is connected to the Internet.

4. Flash the image to the SD card.

   See Flashing the OS Image to the SD Card to flash the new image to the SD card.

5. Update the Vitis AI Runtime libraries to the latest, if needed. To upgrade to the latest Vitis AI, update the following library packages.

   - libunilog
   - libxir
   - libtarget-factory

- libvart
- libvitis_ai_library

See Installing Vitis AI Runtime on the Evaluation Board to install the Vitis AI Runtime libraries.

After you install the Vitis AI Runtime, a `vart.conf` file is generated in the `/etc` directory. This contains the location of the `dpu.xclbin` file, as shown below. The Vitis AI examples fetch the `dpu.xclbin` file by reading the `vart.conf` file. If the `dpu.xclbin` file on your board is not in the same location as the default, change the `dpu.xclbin` path in the `vart.conf` file.

```
root@xilinx-zcu102-20222:~# cat /etc/vart.conf
firmware: /run/media/mmcblk0p1/dpu.xclbin
```

***Note*:** This step generates a system that can run the Vitis AI examples.

6. Run the Vitis AI examples. See Running Examples to run the Vitis AI examples.

Running Examples

For the Vitis AI development kit v3.0 release, VART-based examples demonstrate the use of the Vitis AI unified C++/Python APIs (which are available across Cloud-to-Edge).

These samples can be found at https://github.com/Xilinx/Vitis-AI/tree/v3.0/examples/vai_runtime. If you are using Xilinx ZCU102 and ZCU104 boards to run samples, make sure to enable X11 forwarding with the "ssh -X" option, or the command export DISPLAY=192.168.0.10:0.0 (assuming the IP address of host machine is 192.168.0.10), when logging in to the board using an SSH terminal, as all the examples require X11 to work properly.

***Note*:** The examples will not work through a UART connection due to the lack of X11 support. Alternatively, you can connect boards with a monitor directly instead of using the Ethernet.

Vitis AI Examples

Vitis AI provides several C++ and Python examples to demonstrate the use of the unified cloud-edge runtime programming APIs.

***Note*:** The sample code helps you get started with the new runtime (VART). They are not meant for performance benchmarking.

To familiarize yourself with the unified APIs, use the VART examples. These examples are only to understand the APIs and do not provide high performance. These APIs are compatible between the edge and cloud, though cloud boards may have different software optimizations such as batching and on the edge would require multi-threading to achieve higher performance. If you desire higher performance, see the Vitis AI Library samples and demo software.

If you want to do optimizations to achieve high performance, here are some suggestions:

- Rearrange the thread pipeline structure so that every DPU thread has its own "DPU" runner object.
- Optimize display thread so that when DPU FPS is higher than display rate, skipping some frames. 200 FPS is too high for video display.
- Pre-decoding. The video file might be H.264 encoded. The decoder is slower than the DPU and consumes a lot of CPU resources. The video file has to be first decoded and transformed into raw format.
- The batch mode on Versal boards needs special consideration as it may cause video frame jittering. ZCU102 has no batch mode support.
- OpenCV `cv::imshow` is slow, so you need to use `libdrm.so`. This is only for local display, not through X server.

The following table below describes these Vitis AI examples.

Table 1: **Vitis AI Examples**

ID	Example Name	Models	Framework	Notes
1	resnet50	ResNet-50	Caffe	Image classification with Vitis AI unified C++ APIs.
2	resnet50_pt	ResNet-50	PyTorch	Image classification with Vitis AI unified extension C++ APIs.
3	resnet50_ext	ResNet-50	Caffe	Image classification with Vitis AI unified extension C++ APIs.
4	resnet50_mt_py	ResNet-50	Caffe	Multi-threading image classification with Vitis AI unified Python APIs.
5	inception_v1_mt_py	Inception-v1	TensorFlow	Multi-threading image classification with Vitis AI unified Python APIs.
6	pose_detection	SSD, Pose detection	Caffe	Pose detection with Vitis AI unified C++ APIs.
7	video_analysis	SSD	Caffe	Traffic detection with Vitis AI unified C++ APIs.
8	adas_detection	YOLOv3	Caffe	ADAS detection with Vitis AI unified C++ APIs.
9	segmentation	FPN	Caffe	Semantic segmentation with Vitis AI unified C++ APIs.
10	squeezenet_pytorch	Squeezenet	PyTorch	Image classification with Vitis AI unified C++ APIs.

The typical code snippet to deploy models with Vitis AI unified C++ high-level APIs is as follows:

```
// get dpu subgraph by parsing model file
auto runner = vart::Runner::create_runner(subgraph, "run");
// get input scale and output scale,
// they are used for fixed-floating point conversion
auto outputTensors = runner->get_output_tensors();
auto inputTensors = runner->get_input_tensors();
auto input_scale = get_input_scale(inputTensors[0]);
auto output_scale = get_output_scale(outputTensors[0]);
```

```
// do the image pre-process, convert float data to fixed point data
// populate input/output tensors
auto job_id = runner->execute_async(inputsPtr, outputsPtr);
runner->wait(job_id.first, -1);
// process outputs, convert fixed point data to float data
```

The typical code snippet to deploy models with Vitis AI unified extension C++ high-level APIs is as follows:

```
// get dpu subgraph by parsing model file
std::unique_ptr<vart::RunnerExt> runner =
          vart::RunnerExt::create_runner(subgraph, attrs.get());
// get input & output tensor buffers
auto input_tensor_buffers = runner->get_inputs();
auto output_tensor_buffers = runner->get_outputs();
// get input scale and output scale,
// they are used for fixed-floating point conversion
auto input_tensor = input_tensor_buffers[0]->get_tensor();
auto output_tensor = output_tensor_buffers[0]->get_tensor();
auto input_scale = get_input_scale(input_tensor);
auto output_scale = get_output_scale(output_tensor);
// do the image pre-process, convert float data to fixed point data
setImageBGR(images[batch_idx], (void*)data_in, input_scale);
// sync data for input
input->sync_for_write(0, input->get_tensor()->get_data_size() /
                         input->get_tensor()->get_shape()[0]);
// populate input/output tensors
auto v = runner->execute_async(input_tensor_buffers, output_tensor_buffers);
auto status = runner->wait((int)v.first, -1);
// sync data for output
output->sync_for_read(0, output->get_tensor()->get_data_size() /
                         output->get_tensor()->get_shape()[0]);
// process outputs, conver fixed point data to float data
```

The typical code snippet to deploy models with Vitis AI unified Python high-level APIs is shown below:

```
dpu_runner = runner.Runner(subgraph, "run")
# populate input/output tensors
jid = dpu_runner.execute_async(fpgaInput, fpgaOutput)
dpu_runner.wait(jid)
# process fpgaOutput
```

***Note*:**

- For VART, the input data format supported are `fp32` and `int8`.
- The input and output of DPU is `NHWC` format.
- For the Softmax IP on the MPSOC, the input is `int8` and the output is `float32`.

***Note*:** DPU processes only work with the input and output of fixed-point data. For improved performance and more efficient memory usage, use int8 data as input and run the float to fixed-point conversion along with preprocessing. If the input data is float, the VART converts the float data to fixed-point data which consumes more time.

Note: Since the default rounding mode of quantizer is "HALF_UP", users need to use the same rounding mode when convert inputs and outputs to/from INT8. This ensures that the pre- and post-processing parts run the same on the board as they do on the server. But to balance the performance and accuracy, “cut off” is used to convert inputs and outputs to/from INT8.

Running Vitis AI Examples

Before running Vitis™ AI examples on Edge or on Cloud, download the `vitis_ai_runtime_r3.0.0_image_video.tar.gz` from here. The images and videos used in the following example can be found in the package.

To improve the user experience, the Vitis AI Runtime packages, VART samples, Vitis-AI-Library samples and models have been built into the board image, and the examples are pre-compiled. You can directly run the example program on the target.

For Edge (DPUCZDX8G/DPUCVDX8G)

1. Download the `vitis_ai_runtime_r3.0.0_image_video.tar.gz` from host to the target using `scp` with the following command.

   ```
   scp vitis_ai_runtime_r3.0.0_image_video.tar.gz root@[IP_OF_BOARD]:~/
   ```

2. Unzip the `vitis_ai_runtime_r3.0.0_image_video.tar.gz` package.

   ```
   tar -xzvf vitis_ai_runtime_r3.0.0_image_video.tar.gz -C ~/Vitis-AI/demo/
   VART
   ```

3. Download the model. The download link of the model is described in the YAML file of the model. You can find the YAML file in `Vitis-AI/model_zoo` and download the model of the corresponding platform. Take `resnet50` as an example:

   ```
   wget https://www.xilinx.com/bin/public/openDownload?filename=resnet50-
   zcu102_zcu104_kv260-r3.0.0.tar.gz -O resnet50-zcu102_zcu104_kv260-
   r3.0.0.tar.gz

   scp resnet50-zcu102_zcu104_kv260-r3.0.0.tar.gz root@[IP_OF_BOARD]:~/
   ```

4. Untar the model on the target and install it.

   Note: If the `/usr/share/vitis_ai_library/models` folder does not exist, create it first.

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

   To install the model package, run the following command:

   ```
   tar -xzvf resnet50-zcu102_zcu104_kv260-r3.0.0.tar.gz
   cp resnet50 /usr/share/vitis_ai_library/models -r
   ```

5. Enter the directory of samples in the target board. Take `resnet50` as an example.

   ```
   cd ~/Vitis-AI/examples/vai_runtime/resnet50
   ```

6. Run the example.

```
./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
```

***Note*:** If the above executable program does not exist, cross-compile it on the host first.

***Note*:** Applications can also be compiled natively on the target. Run the following command on the target.

```
sh -x build.sh
```

***Note*:** For examples with video input, only `webm` and `raw` format are supported by default with the official system image. If you want to support video data in other formats, install the relevant packages on the system.

The following table shows the run commands for all the Vitis AI samples.

Table 2: **Launching Commands for Vitis AI Samples on ZCU102/ZCU104**

ID	Example Name	Command
1	resnet50	`./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
2	resnet50_pt	`./resnet50_pt /usr/share/vitis_ai_library/models/resnet50_pt/resnet50_pt.xmodel ../images/001.jpg`
3	resnet50_ext	`./resnet50_ext /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel ../images/001.jpg`
4	resnet50_mt_py	`python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
5	inception_v1_mt_py	`python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel`
6	pose_detection	`./pose_detection video/pose.webm /usr/share/vitis_ai_library/models/sp_net/sp_net.xmodel /usr/share/vitis_ai_library/models/ssd_pedestrian_pruned_0_97/ssd_pedestrian_pruned_0_97.xmodel`
7	video_analysis	`./video_analysis video/structure.webm /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel`
8	adas_detection	`./adas_detection video/adas.webm /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel`
9	segmentation	`./segmentation video/traffic.webm /usr/share/vitis_ai_library/models/fpn/fpn.xmodel`
10	squeezenet_pytorch	`./squeezenet_pytorch /usr/share/vitis_ai_library/models/squeezenet_pt/squeezenet_pt.xmodel`

For Cloud(DPUCVDX8H)

Before running the samples on the Cloud, ensure that either the Versal VCK5000 evaluation board is installed on the server, and the docker system is loaded and running.

If you have downloaded `Vitis-AI`, entered `Vitis-AI` directory, and then started Docker.

Thus, VART examples are located in the path of `/workspace/examples/vai_runtime/` in the docker system.

1. Download the `vitis_ai_runtime_r3.0.0_image_video.tar.gz` package and unzip it.

   ```
   tar -xzvf vitis_ai_runtime_r3.0.0_image_video.tar.gz -C /workspace/
   examples/vai_runtime
   ```

2. Compile the sample. Take `resnet50` as an example.

   ```
   cd /workspace/examples/vai_runtime/resnet50
   bash -x build.sh
   ```

   When the compilation is complete, the executable `resnet50` is generated in the current directory.

3. Download the model. The download link of the model is described in the YAML file of the model. You can find the YAML file in `Vitis-AI/model_zoo/model-list`. Take `resnet50` on Versal VCK5000 Card DPUCVDX8H_8pe DPU IP as an example:

   ```
   wget https://www.xilinx.com/bin/public/openDownload?filename=resnet50-
   vck5000-DPUCVDX8H-8pe-r3.0.0.tar.gz -O resnet50-vck5000-DPUCVDX8H-8pe-
   r3.0.0.tar.gz
   ```

4. Untar the model on the target and install it.

   Note: If the `/usr/share/vitis_ai_library/models` folder does not exist, create it.

   ```
   sudo mkdir -p /usr/share/vitis_ai_library/models
   ```

   Then install the model package.

   ```
   tar -xzvf resnet50-vck5000-DPUCVDX8H-8pe-r3.0.0.tar.gz
   sudo cp resnet50 /usr/share/vitis_ai_library/models -r
   ```

5. Run the sample.

   ```
   ./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
   ```

The following table shows the run commands for all the Vitis AI samples in the cloud.

Table 3: **Launching Commands for Vitis AI Samples for Cloud DPUs**

ID	Example Name	Command
1	resnet50	`./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
2	resnet50_pt	`./resnet50_pt /usr/share/vitis_ai_library/models/resnet50_pt/resnet50_pt.xmodel ../images/001.jpg`
3	resnet50_ext	`./resnet50_ext /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel ../images/001.jpg`
4	resnet50_mt_py	`/usr/bin/python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
5	inception_v1_mt_py	`/usr/bin/python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel`
6	pose_detection	`./pose_detection video/pose.mp4 /usr/share/vitis_ai_library/models/sp_net/sp_net.xmodel /usr/share/vitis_ai_library/models/ssd_pedestrian_pruned_0_97/ssd_pedestrian_pruned_0_97.xmodel`

Table 3: **Launching Commands for Vitis AI Samples for Cloud DPUs** *(cont'd)*

ID	Example Name	Command
7	video_analysis	`./video_analysis video/structure.mp4 /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel`
8	adas_detection	`./adas_detection video/adas.avi /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel`
9	segmentation	`./segmentation video/traffic.mp4 /usr/share/vitis_ai_library/models/fpn/fpn.xmodel`
10	squeezenet_pytorch	`./squeezenet_pytorch /usr/share/vitis_ai_library/models/squeezenet_pt/squeezenet_pt.xmodel`

Support

You can visit the Vitis AI Library forum on the Xilinx website for topic discussions, knowledge sharing, FAQs, and requests for technical support.

Chapter 3

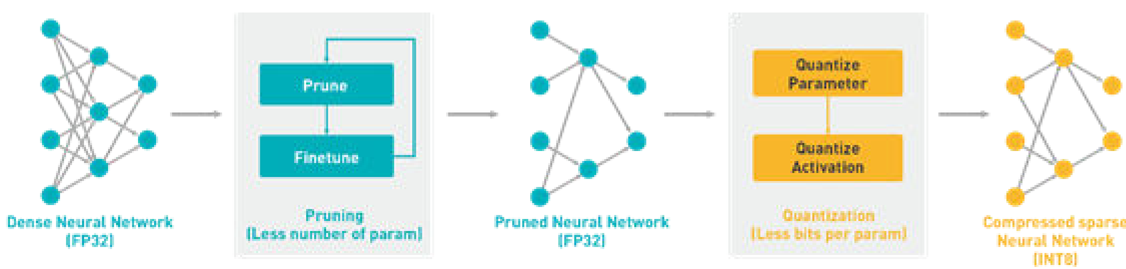
Quantizing the Model

Overview

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.

Quantization and channel pruning techniques are employed to address these issues while achieving high performance and high energy efficiency with little degradation in accuracy. Quantization makes it possible to use integer computing units and to represent weights and activations by lower bits, while pruning reduces the overall required operations. In the Vitis™ AI quantizer, only the quantization tool is included. The pruning tool is packaged in the Vitis AI optimizer. Contact the support team for the Vitis AI development kit if you require the pruning tool.

Figure 18: **Pruning and Quantization Flow**



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 without losing prediction 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.

The Vitis AI quantizer now supports TensorFlow (both 1.x and 2.x), and PyTorch. . The quantizer names are vai_q_tensorflow and vai_q_pytorch, respectively. Quantizer for Caffe has been deprecated in Vitis AI 2.5. If you want to use Vitis AI quantizer for Caffe, please refer to Vitis AI 2.0. In Vitis AI 2.5 and previous versions, for TensorFlow 1.x, the Vitis AI quantizer is based on TensorFlow 1.15 and released with Tensorflow 1.15 package. Starting from Vitis AI 3.0, the Vitis AI quantizer is a standalone Python package with several quantization APIs for both Tensorflow1.x and Tensorflow2.x. You can import this package, and the Vitis AI quantizer works like a plugin for TensorFlow.

Table 4: **Vitis AI Quantizer Supported Frameworks and Features**

Model	Versions	Features			
		Post Training Quantization (PTQ)	Quantization Aware Training (QAT)	Fast Finetuning (Advanced Calibration)	Inspector
TensorFlow 1.x	Supports 1.15	Yes	Yes	No	No
TensorFlow 2.x	Supports 2.3 - 2.10	Yes	Yes	Yes	Yes
PyTorch	Supports 1.2 - 1.12	Yes	Yes	Yes	Yes

Post training quantization (PTQ) requires only a small set of unlabeled images to analyze the distribution of activations. The running time of quantize calibration varies from a few seconds to several minutes, depending on the size of the neural network. Generally, there is some drop in accuracy after quantization. However, for some networks such as Mobilenet, the accuracy loss might be large. In this situation, quantization aware training (QAT) can be used to further improve the accuracy of the quantized models. QAT requires the original training dataset. Several epochs of finetuning are needed and the finetune time varies from several minutes to several hours. It is recommended to use small learning rates when performing QAT.

Note: From Vitis AI 1.4 onwards, the term "quantize calibration" is replaced with "post training quantization" and "quantize finetuning" is replaced with "quantization aware training."

Note: Vitis AI only performs *signed* quantization. It is strongly recommended that standardization (i.e., scale the input pixel values to have zero mean and unit variance) be performed such that the DPU effectively sees values in the range [-1.0, +1.0). Note that scaled unsigned inputs, e.g., dividing the raw input by 255.0 to obtain an input range of [0.0, 1.0] will effectively "lose" a bit since the sign bit must always be zero to denote a positive value. Tensorflow 2.x and Pytorch quantizers provide configurations to preform unsigned quantization for experiments purposes, the results are not deployable for DPU now.

Note: When viewing a model with a tool like Netron, there will be a fix_point parameter for some layers indicating the quantization parameters used for that layer. The fix_point parameter refers to the number of fractional bits used. For example, for 8-bit signed quantization with fix_point= 7, the Q-format representation will be Q0.7, i.e., 1 sign bit, 0 integer bits, and 7 fractional bits. To convert an integer value in Q-format to floating-point, multiply the integer value by 2^-fixed_point.

For PTQ, the cross layer equalization [1] algorithm is implemented. Cross layer equalization can improve the calibration performance, especially for networks including depth-wise convolution.

With a small set of unlabeled data, the AdaQuant algorithm [2] not only calibrates the activations but also finetunes the weights. AdaQuant uses a small set of unlabeled data similar to calibration but it changes the model, which is like finetuning. Vitis AI quantizer implements this algorithm and call it "fast finetuning" or "advanced calibration." Fast finetuning can achieve better performance than quantize calibration but it is slightly slower. One thing worth noting is that for fast finetuning, each run will get a different result. This is similar to finetuning.

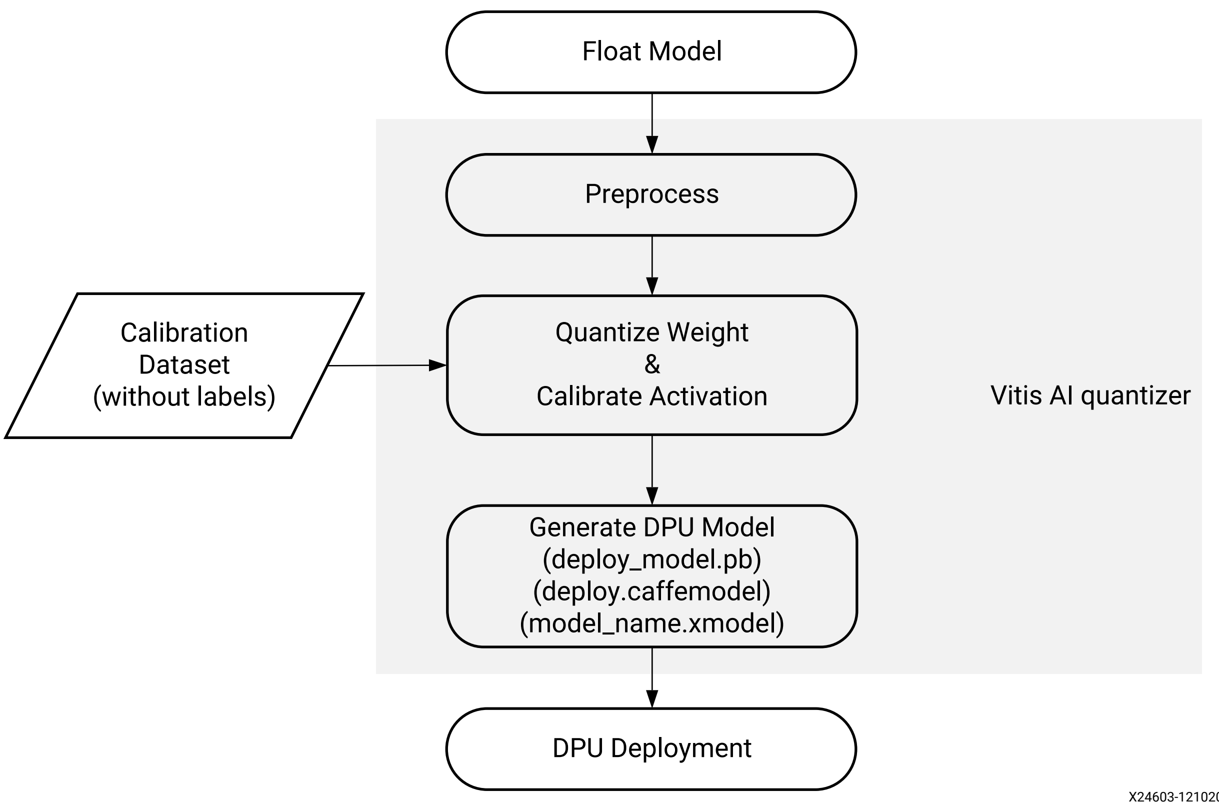
***Note*:**

1. Markus Nagel et al., Data-Free Quantization through Weight Equalization and Bias Correction, arXiv:1906.04721, 2019.
2. Itay Hubara et.al., Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming, arXiv:2006.10518, 2020.

Vitis AI Quantizer Flow

The overall model quantization flow is detailed in the following figure.

Figure 19: **VAI Quantizer Workflow**



***Note*:** Caffe has been deprecated since Vitis AI 2.5. For information on Caffe, see Vitis AI 2.0 user guide.

The Vitis AI quantizer takes a floating-point model as input and performs pre-processing (folds batchnorms and removes nodes not required for inference), and then quantizes the weights/biases and activations to the given bit width.

Before quantizing the float model, there is an optional step called "inspector". It is used to inspect the model before quantizing it. Inspector will output the partition information, indicating which operators will run on which device (DPU/CPU). In general, DPU is faster than CPU. The idea is to run as many operators as possible on DPU devices. The partition results also include messages on why this operator cannot be run on DPU. This will help users to better understand DPU's ability and can further help users fit their model to DPU.

To capture activation statistics and improve the accuracy of quantized models, the Vitis AI quantizer must run several iterations of inference to calibrate the activations. A calibration image dataset input is, therefore, required. Generally, the quantizer works well with 100–1000 calibration images. Because there is no need for back propagation, an unlabeled dataset is sufficient.

After calibration, the quantized model is transformed into a DPU deployable model (named `deploy_model.pb` for vai_q_tensorflow, `model_name.xmodel` for vai_q_pytorch), which follows the data format of a DPU. This model can then be compiled by the Vitis AI compiler and deployed to the DPU. The quantized model cannot be taken in by the standard version of TensorFlow or PyTorch framework.

TensorFlow 1.x Version (vai_q_tensorflow)

Installing vai_q_tensorflow

There are three ways to install the vai_q_tensorflow:

Install Using Docker Containers

Vitis AI provides a Docker container for quantization tools, including vai_q_tensorflow. After running a container, activate the Conda environment "vitis-ai-tensorflow".

```
conda activate vitis-ai-tensorflow
```

If there is a patch package, install the vitis-ai-tensorflow patch package inside the Docker container.

```
# [optional]
$ sudo env CONDA_PREFIX=/opt/vitis_ai/conda/envs/vitis-ai-tensorflow/
PATH=/opt/vitis_ai/conda/bin:$PATH conda install patch_package.tar.bz2
```

Install Using Source Code

vai_q_tensorflow is a Xilinx maintained plug-in tool for tensorflow 1.15. It is open source in Vitis_AI_Quantizer. To build vai_q_tensorflow, run the following command:

```
sh build.sh
```

Running vai_q_tensorflow

Preparing the Float Model and Related Input Files

Before running vai_q_tensorflow, prepare the frozen inference TensorFlow model in floating-point format and calibration set, including the files listed in the following table.

Table 5: **Input Files for vai_q_tensorflow**

No.	Name	Description
1	frozen_graph.pb	Floating-point frozen inference graph. Ensure that the graph is the inference graph rather than the training graph.
2	calibration dataset	A subset of the training dataset containing 100 to 1000 images.
3	input_fn	An input function to convert the calibration dataset to the input data of the frozen_graph during quantize calibration. Usually performs data pre-processing and augmentation.

Generating the Frozen Inference Graph

Training a model with TensorFlow 1.x creates a folder containing a GraphDef file (usually ending with a `.pb` or `.pbtxt` extension) and a set of checkpoint files. What you need for mobile or embedded deployment is a single GraphDef file that has been "frozen," or had its variables converted into inline constants, so everything is in one file. To handle the conversion, TensorFlow provides `freeze_graph.py`, which is automatically installed with the vai_q_tensorflow quantizer.

An example of command-line usage is as follows:

```
$ freeze_graph \
    --input_graph  /tmp/inception_v1_inf_graph.pb \
    --input_checkpoint  /tmp/checkpoints/model.ckpt-1000 \
    --input_binary  true \
    --output_graph  /tmp/frozen_graph.pb \
    --output_node_names  InceptionV1/Predictions/Reshape_1
```

The `-input_graph` should be an inference graph other than the training graph. Because the operations of data preprocessing and loss functions are not needed for inference and deployment, the `frozen_graph.pb` should only include the main part of the model. In particular, the data preprocessing operations should be taken in the `Input_fn` to generate correct input data for quantize calibration.

Note: Some operations, such as dropout and batchnorm, behave differently in the training and inference phases. Ensure that they are in the inference phase when freezing the graph. For examples, you can set the flag `is_training=false` when using `tf.layers.dropout/tf.layers.batch_normalization`. For models using `tf.keras`, call `tf.keras.backend.set_learning_phase(0)` before building the graph.

> **TIP:** *Type* `freeze_graph --help` *for more options.*

The input and output node names vary depending on the model, but you can inspect and estimate them with the vai_q_tensorflow quantizer. See the following code snippet for an example:

```
$ vai_q_tensorflow inspect --input_frozen_graph=/tmp/
inception_v1_inf_graph.pb
```

The estimated input and output nodes cannot be used for quantization if the graph has in-graph pre- and post-processing. This is because some operations are not quantizable and can cause errors when compiled by the Vitis AI compiler, if you deploy the quantized model to the DPU.

Another way to get the input and output name of the graph is by visualizing the graph. Both TensorBoard and Netron can do this. See the following example, which uses Netron:

```
$ pip install netron
$ netron /tmp/inception_v3_inf_graph.pb
```

Preparing the Calibration Dataset and Input Function

The calibration set is usually a subset of the training/validation dataset or actual application images (at least 100 images for performance). The input function is a Python importable function to load the calibration dataset and perform data preprocessing. The vai_q_tensorflow quantizer can accept an input_fn to do the preprocessing, which is not saved in the graph. If the pre-processing subgraph is saved into the frozen graph, the input_fn only needs to read the images from dataset and return a feed_dict.

The format of input function is `module_name.input_fn_name`, (for example, `my_input_fn.calib_input`). The input_fn takes an int object as input, indicating the calibration step number, and returns a dict("`placeholder_name, numpy.Array`") object for each call, which is fed into the placeholder nodes of the model when running inference. The `placeholder_name` is always the input node of frozen graph, that is to say, the node receiving input data. Please note that `placeholder_name` should be replaced with the actual name of the input node receiving the input images. For example, if the input placeholder node is named "the_input_node", then `placeholder_name` should be replaced with "the_input_node". The `input_nodes`, in the vai_q_tensorflow options, indicates where quantization starts in the frozen graph. Note that the `placeholder_names` and the `input_nodes` option are sometimes different. For example, when the frozen graph includes in-graph preprocessing, the placeholder_name is the input of the graph though it is recommended that `input_nodes` be set to the last node of preprocessing. The shape of `numpy.array` must be consistent with the placeholders. See the following pseudo code example:

```
$ “my_input_fn.py”
def calib_input(iter):
"""
A function that provides input data for the calibration
  Args:
    iter: A `int` object, indicating the calibration step number
  Returns:
    dict( placeholder_name, numpy.array): a `dict` object, which will be
fed into the model
"""
  image = load_image(iter)
  preprocessed_image = do_preprocess(image)
  return {"placeholder_name": preprocessed_images}
```

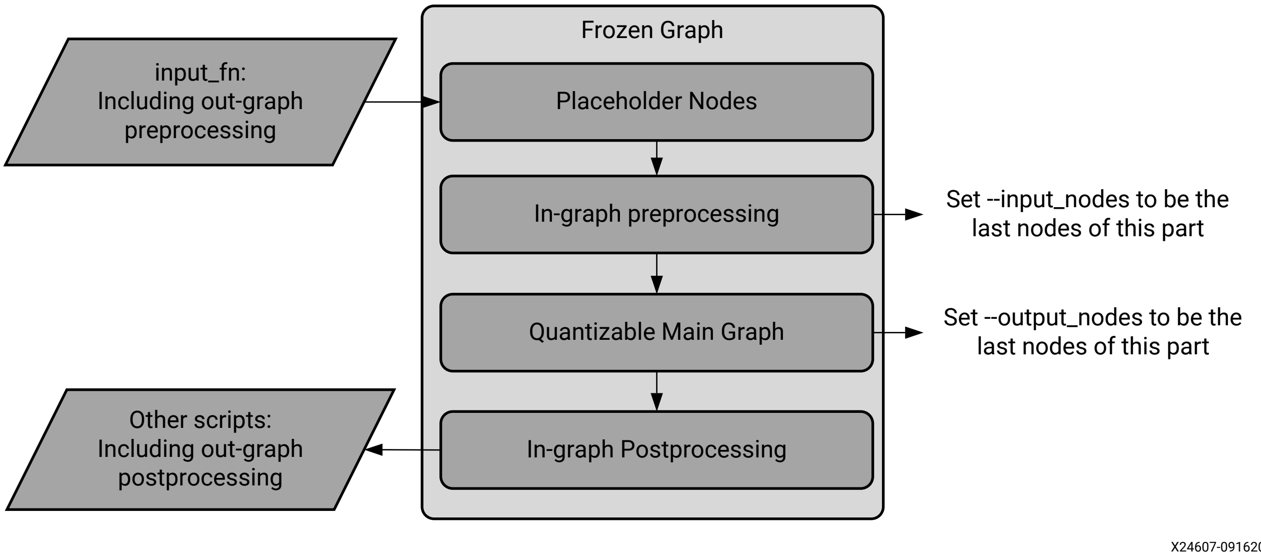
Quantizing the Model Using vai_q_tensorflow

Run the following commands to quantize the model:

```
$vai_q_tensorflow quantize \
                --input_frozen_graph  frozen_graph.pb \
                --input_nodes    ${input_nodes} \
                --input_shapes   ${input_shapes} \
                --output_nodes   ${output_nodes} \
                --input_fn  input_fn \
                [options]
```

The input_nodes and output_nodes arguments are the name list of input nodes of the quantize graph. They are the start and end points of quantization. The main graph between them is quantized if it is quantizable, as shown in the following figure.

Figure 20: **Quantization Flow for TensorFlow**



It is recommended to set `-input_nodes` to be the last nodes of the preprocessing part and to set `-output_nodes` to be the last nodes of the main graph part because some operations in the pre- and postprocessing parts are not quantizable and might cause errors when compiled by the Vitis AI quantizer if you need to deploy the quantized model to the DPU.

The input nodes might not be the same as the placeholder nodes of the graph. If no in-graph preprocessing part is present in the frozen graph, the placeholder nodes should be set to input nodes.

The `input_fn` should be consistent with the placeholder nodes.

[options] stands for optional parameters. The most commonly used options are as follows:

- **weight_bit:** Bit width for quantized weight and bias (default is 8).
- **activation_bit:** Bit width for quantized activation (default is 8).
- **method:** Quantization methods, including 0 for non-overflow, 1 for min-diffs, and 2 for min-diffs with normalization. The non-overflow method ensures that no values are saturated during quantization. The results can be affected by outliers. The min-diffs method allows saturation for quantization to achieve a lower quantization difference. It is more robust to outliers and usually results in a narrower range than the non-overflow method.

Generating the Quantized Model

After the successful execution of the `vai_q_tensorflow` command, one output file is generated in the `${output_dir}` location:

- `quantize_eval_model.pb` is used to evaluate the CPU/GPUs, and can be used to simulate the results on hardware.

Table 6: **vai_q_tensorflow Output Files**

No.	Name	Description
1	deploy_model.pb	Quantized model for the Vitis AI compiler (extended TensorFlow format) for targeting DPUCZDX8G implementations.
2	quantize_eval_model.pb	Quantized model for evaluation (also, the Vitis AI compiler input for most DPU architectures, like DPUCAHX8H, and DPUCADF8H).

(Optional) Exporting the Quantized Model to ONNX

The quantized model is tensorflow protobuf format by default. If you want to get a ONNX format model, just add `output_format` argument to the `vai_q_tensorflow` command.

```
$vai_q_tensorflow quantize \
--input_frozen_graph frozen_graph.pb \
--input_nodes ${input_nodes} \
--input_shapes ${input_shapes} \
--output_nodes ${output_nodes} \
--input_fn input_fn \
--output_format onnx \
[options]
```

- **output_format:** Indicates what format to save the quantized model, 'pb' for saving tensorflow frozen pb, 'onnx' for saving onnx model. The default value is 'pb'.

(Optional) Evaluating the Quantized Model

If you have scripts to evaluate floating point models, like the models in Vitis AI Model Zoo, apply the following two changes to evaluate the quantized model:

- Prepend the float evaluation script with `import vai_q_tensorflow`.
- Replace the float model path in the scripts to quantization output model `"quantize_results/quantize_eval_model.pb"`.
- Run the modified script to evaluate the quantized model.

(Optional) Dumping the Simulation Results

vai_q_tensorflow dumps the simulation results with the `quantize_eval_model.pb` generated by the quantizer. This allows you to compare the simulation results on the CPU/GPU with the output values on the DPU.

To dump the quantize simulation results, run the following commands:

```
$vai_q_tensorflow dump \
    --input_frozen_graph  quantize_results/quantize_eval_model.pb \
    --input_fn  dump_input_fn \
    --max_dump_batches 1 \
    --dump_float 0 \
    --output_dir quantize_results
```

The input_fn for dumping is similar to the input_fn for quantize calibration, but the batch size is often set to 1 to be consistent with the DPU results.

If the command executes successfully, dump results are generated in `${output_dir}`. There are folders in `${output_dir}`, and each folder contains the dump results for a batch of input data. Results for each node are saved separately. For each quantized node, results are saved in `*_int8.bin` and `*_int8.txt` format. If dump_float is set to 1, the results for unquantized nodes are dumped. The / symbol is replaced by _ for simplicity. Examples for dump results are shown in the following table.

Table 7: **Examples for Dump Results**

Batch No.	Quant	Node Name	Saved files
1	Yes	resnet_v1_50/conv1/biases/ wquant	{output_dir}/dump_results_1/ resnet_v1_50_conv1_biases_wquant_int8.bin {output_dir}/dump_results_1/ resnet_v1_50_conv1_biases_wquant_int8.txt
2	No	resnet_v1_50/conv1/biases	{output_dir}/dump_results_2/resnet_v1_50_conv1_biases.bin {output_dir}/dump_results_2/resnet_v1_50_conv1_biases.txt

vai_q_tensorflow Quantization Aware Training

Quantization aware training (QAT) is similar to float model training/finetuning, but in QAT, the vai_q_tensorflow APIs are used to rewrite the float graph to convert it to a quantized graph before the training starts. The typical workflow is as follows:

1. Preparation: Before QAT, prepare the following files:

Table 8: **Input Files for vai_q_tensorflow QAT**

No.	Name	Description
1	Checkpoint files	Floating-point checkpoint files to start from. Omit this if you training the model from scratch.
2	Dataset	The training dataset with labels.

Table 8: **Input Files for vai_q_tensorflow QAT** *(cont'd)*

No.	Name	Description
3	Train Scripts	The Python scripts to run float train/finetuning of the model.

2. Evaluate the float model (optional): Evaluate the float checkpoint files first before doing quantize finetuning to check the correctness of the scripts and dataset. The accuracy and loss values of the float checkpoint can also be a baseline for QAT.

3. Modify the training scripts: To create the quantize training graph, modify the training scripts to call the function after the float graph is built. The following is an example:

```
# train.py

# ...

# Create the float training graph
model = model_fn(is_training=True)

# *Set the quantize configurations
import vai_q_tensorflow
q_config = vai_q_tensorflow.QuantizeConfig(input_nodes=['net_in'],
                output_nodes=['net_out'],
                input_shapes=[[-1, 224, 224, 3]])
# *Call Vai_q_tensorflow api to create the quantize training graph

vai_q_tensorflow.CreateQuantizeTrainingGraph(config=q_config)

# Create the optimizer
optimizer = tf.train.GradientDescentOptimizer()

# start the training/finetuning, you can use sess.run(), tf.train,
tf.estimator, tf.slim and so on
# ...
```

***Note*:** One can use `import vai_q_tensorflow as decent_q` for compatibility with codes of older versions vai_q_tensorflow which was `import tensorflow.contrib.decent_q`

The `QuantizeConfig` contains the configurations for quantization.

Some basic configurations like `input_nodes`, `output_nodes`, `input_shapes` need to be set according to your model structure.

Other configurations like `weight_bit`, `activation_bit`, `method` have default values and can be modified as needed. See vai_q_tensorflow Usage for detailed information of all the configurations.

- `input_nodes`/`output_nodes`: They are used together to determine the subgraph range you want to quantize. The pre-processing and post-processing components are usually not quantizable and should be out of this range. The input_nodes and output_nodes should be the same for the float training graph and the float evaluation graph to match the quantization operations between them.

  ***Note*:** Operations with multiple output tensors (such as FIFO) are currently not supported. You can add a tf.identity node to make an alias for the input_tensor to make a single output input node.

- `input_shapes`: The shape list of input_nodes must be a 4-dimension shape for each node. The information is comma separated, for example, [[1,224,224,3] [1, 128, 128, 1]]; support unknown size for batch_size, for example, [[-1,224,224,3]].

4. Evaluate the quantized model and generate the frozen model: After QAT, generate the frozen model after evaluating the quantized graph with a checkpoint file. This can be done by calling the following function after building the float evaluation graph. As the freeze process depends on the quantize evaluation graph, they are often called together.

   Note: Function `vai_q_tensorflow.CreateQuantizeTrainingGraph` and `vai_q_tensorflow.CreateQuantizeEvaluationGraph` will modify the default graph in Tensorflow. Please not that they need to be called on different graph phases. `vai_q_tensorflow.CreateQuantizeTrainingGraph` need to be called on the float training graph while `vai_q_tensorflow.CreateQuantizeEvaluationGraph` need to be called on the float evaluation graph. `vai_q_tensorflow.CreateQuantizeEvaluationGraph` can not be called right after calling function `vai_q_tensorflow.CreateQuantizeTrainingGraph`, because the default graph has been converted to a quantize training graph. The correct way is to call it right after the float model creation function.

```
# eval.py

# ...

# Create the float evaluation graph
model = model_fn(is_training=False)

# *Set the quantize configurations
import vai_q_tensorflow
q_config = vai_q_tensorflow.QuantizeConfig(input_nodes=['net_in'],
                output_nodes=['net_out'],
                input_shapes=[[-1, 224, 224, 3]])
# *Call Vai_q_tensorflow api to create the quantize evaluation graph

vai_q_tensorflow.CreateQuantizeEvaluationGraph(config=q_config)
# *Call Vai_q_tensorflow api to freeze the model and generate the deploy
model

vai_q_tensorflow.CreateQuantizeDeployGraph(checkpoint="path to
checkpoint folder", config=q_config)

# start the evaluation, users can use sess.run, tf.train, tf.estimator,
tf.slim and so on
# ...
```

Generated Files

After you have performed the previous steps, the following files are generated in the `${output_dir}` location:

Table 9: **Generated File Information**

Name	TensorFlow Compatible	Usage	Description
quantize_train_graph.pb	Yes	Train	The quantize train graph.
quantize_eval_graph_{suffix}.pb	Yes	Evaluation with checkpoint	The quantize evaluation graph with quantize information frozen inside. There are weights in this file and it should be used together with the checkpoint file in evaluation.
quantize_eval_model_{suffix}.pb	Yes	1: Evaluation 2: Dump 3: Input to VAI compiler (DPUCAHX8H)	The frozen quantize evaluation graph, weights in the checkpoint, and quantize information are frozen inside. It can be used to evaluate the quantized model on the host or to dump the outputs of each layer for cross check with DPU outputs. The XIR compiler uses it as an input.

The suffix contains the iteration information from the checkpoint file and the date information. For example, if the checkpoint file is "model.ckpt-2000.*" and the date is 20200611, then the suffix is "2000_20200611000000."

QAT APIs for TensorFlow 1.x

There are three APIs for QAT in the `vai_q_tensorflow` Python package.

vai_q_tensorflow.CreateQuantizeTrainingGraph(config)

Convert the float training graph to a quantize training graph by in-place rewriting on the default graph.

Arguments

- `config`: A `vai_q_tensorflow.QuantizeConfig` object, containing the configurations for quantization.

vai_q_tensorflow.CreateQuantizeEvaluationGraph(config)

Convert the float evaluation graph to quantize evaluation graph, this is done by in-place rewriting on the default graph.

Arguments

- `config`: A `vai_q_tensorflow.QuantizeConfig` object, containing the configurations for quantization.

vai_q_tensorflow.CreateQuantizeDeployGraph(checkpoint, config)

Freeze the checkpoint into the quantize evaluation graph.

Arguments

- `checkpoint`: A `string` object that specifies the path to the checkpoint folder or file.
- `config`: A `vai_q_tensorflow.QuantizeConfig` object that contains the configurations required for quantization.

Tips for QAT

The following are some tips for QAT.

- **Keras Model:**

  For Keras models, please set `backend.set_learning_phase(1)` before creating the float train graph, and set `backend.set_learning_phase(0)` before creating the float evaluation graph. Moreover, `backend.set_learning_phase()` should be called after `backend.clear_session()`. Tensorflow1.x QAT APIs are designed for Tensorflow native training APIs. Using Keras `model.fit()` APIs in QAT may lead to some "nodes not executed" issues. It is recommended to use QAT APIs in Tensorflow2 quantization tool with Keras APIs.

- **Dropout:** Experiments shows that QAT works better without dropout ops. This tool does not support finetuning with dropouts at the moment and they should be removed or disabled before running QAT. This can be done by setting `is_training=false` when using tf.layers or call `tf.keras.backend.set_learning_phase(0)` when using tf.keras.layers.
- **Hyper-parameter:** QAT is like float model training/finetuning, so the techniques for float model training/finetuning are also needed. The optimizer type and the learning rate curve are some important parameters to tune.

Converting to Float16 or BFloat16

The vai_q_tensorflow supports data type conversions for float models, including Float16, BFloat16, Float, and Double. To achieve this, you can add `convert_datatype` argument to the vai_q_tensorflow command.

```
$vai_q_tensorflow quantize \
--input_frozen_graph frozen_graph.pb \
--input_nodes ${input_nodes} \
--input_shapes ${input_shapes} \
--output_nodes ${output_nodes} \
--input_fn input_fn \
--convert_datatype 1 \
[options]
```

- **convert_datatype:** Int, specifies the target datatype to convert to. Options are: 1 for Float16, 2 for Double, 3 for BFloat16 and 4 for Float. The default value is 0.

***Note*:** In order to implement the conversion, BatchNorm operation will be folded in advance.

vai_q_tensorflow Supported Operations and APIs

The following table lists the supported operations and APIs for vai_q_tensorflow.

Table 10: **Supported Operations and APIs for vai_q_tensorflow**

Type	Operation Type	tf.nn	tf.layers	tf.keras.layers
Convolution	Conv2D DepthwiseConv2dNative	atrous_conv2d conv2d conv2d_transpose depthwise_conv2d_native separable_conv2d	Conv2D Conv2DTranspose SeparableConv2D	Conv2D Conv2DTranspose DepthwiseConv2D SeparaleConv2D
Fully Connected	MatMul	/	Dense	Dense
BiasAdd	BiasAdd Add	bias_add	/	/
Pooling	AvgPool Mean MaxPool	avg_pool max_pool	AveragePooling2D MaxPooling2D	AveragePooling2D MaxPool2D
Activation	ReLU ReLU6 Sigmoid Swish Hard-sigmoid Hard-swish	relu relu6 leaky_relu swish	/	ReLU Leaky ReLU

Table 10: **Supported Operations and APIs for vai_q_tensorflow** *(cont'd)*

Type	Operation Type	tf.nn	tf.layers	tf.keras.layers
BatchNorm[#1]	FusedBatchNorm	batch_normalization batch_norm_with_global_normalization fused_batch_norm	BatchNormalization	BatchNormalization
Upsampling	ResizeBilinear ResizeNearestNeighbor	/	/	UpSampling2D
Concat	Concat ConcatV2	/	/	Concatenate
Others	Placeholder Const Pad Squeeze Reshape ExpandDims Max Transpose	dropout[#2] softmax[#3] depth_to_space	Dropout[#2] Flatten	Input Flatten Reshape Zeropadding2D Softmax

Notes:

1. Only supports Conv2D/DepthwiseConv2D/Dense+BN. BN is folded to speed up inference.
2. Dropout is deleted to speed up inference.
3. vai_q_tensorflow does not quantize the softmax output.

vai_q_tensorflow Usage

The options supported by `vai_q_tensorflow` are shown in the following tables.

Table 11: **vai_q_tensorflow Options**

Name	Type	Description
Common Configuration		
--input_frozen_graph	String	TensorFlow frozen inference GraphDef file for the floating-point model. It is used for quantize calibration.
--input_nodes	String	Specifies the name list of input nodes of the quantize graph, used together with `-output_nodes`, comma separated. Input nodes and output nodes are the start and end points of quantization. The subgraph between them is quantized if it is quantizable. **RECOMMENDED:** *Set `-input_nodes` as the last nodes for pre-processing and `-output_nodes` as the last nodes for post-processing because some of the operations required for pre- and post-processing are not quantizable and might cause errors when compiled by the Vitis AI compiler if you need to deploy the quantized model to the DPU. The input nodes might not be the same as the placeholder nodes of the graph.*
--output_nodes	String	Specifies the name list of output nodes of the quantize graph, used together with `-input_nodes`, comma separated. Input nodes and output nodes are the start and end points of quantization. The subgraph between them is quantized if it is quantizable. **RECOMMENDED:** *Set `-input_nodes` as the last nodes for pre-processing and `-output_nodes` as the last nodes for post-processing because some of the operations required for pre- and post-processing are not quantizable and might cause errors when compiled by the Vitis AI compiler if you need to deploy the quantized model to the DPU.*
--input_shapes	String	Specifies the shape list of input nodes. Must be a 4-dimension shape for each node, comma separated, for example 1,224,224,3; support unknown size for batch_size, for example ?,224,224,3. In case of multiple input nodes, assign the shape list of each node separated by :, for example, ?,224,224,3:?,300,300,1.

Table 11: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description

--input_fn	String	Provides the input data for the graph used with the calibration dataset. The function format is `module_name.input_fn_name` (for example, `my_input_fn.input_fn`). The `-input_fn` should take an `int` object as input which indicates the calibration step, and should return a dict`(placeholder_node_name, numpy.Array)` object for each call, which is then fed into the placeholder operations of the model.
		For example, assign `-input_fn` to `my_input_fn.calib_input`, and write `calib_input` function in `my_input_fn.py` as:
		`def calib_input_fn:`
		`# read image and do some preprocessing`
		`    return {“placeholder_1”: input_1_nparray,`
		`“placeholder_2”: input_2_nparray}`
		Note: You do not need to do in-graph pre-processing again in input_fn because the subgraph before `-input_nodes` remains during quantization. Remove the pre-defined input functions (including default and random) because they are not commonly used. The pre-processing part which is not in the graph file should be handled in the input_fn.

Quantize Configuration		
--weight_bit	Int32	Specifies the bit width for quantized weight and bias. Default: 8
--activation_bit	Int32	Specifies the bit width for quantized activation. Default: 8
--nodes_bit	String	Specifies the bit width of nodes. Node names and bit widths form a pair of parameters joined by a colon; the parameters are comma separated. When specifying the conv op name, only `vai_q_tensorflow` will quantize the weights of conv op using the specified bit width. For example, 'conv1/Relu:16,conv1/weights:8,conv1:16'.
--method	Int32	Specifies the method for quantization. 0: Non-overflow method in which no values are saturated during quantization. Sensitive to outliers. 1: Min-diffs method that allows saturation for quantization to get a lower quantization difference. Higher tolerance to outliers. Usually ends with narrower ranges than the non-overflow method. 2: Min-diffs method with strategy for depthwise. It allows saturation for large values during quantization to get smaller quantization errors. A special strategy is applied for depthwise weights. It is slower than method 0 but has higher endurance to outliers. Default value: 1
--nodes_method	String	Specifies the method of nodes. Node names and method form a pair of parameters joined by a colon; the parameter pairs are comma separated. When specifying the conv op name, only `vai_q_tensorflow` will quantize weights of conv op using the specified method, for example, 'conv1/Relu:1,depthwise_conv1/weights:2,conv1:1'.
--calib_iter	Int32	Specifies the iterations of calibration. Total number of images for calibration = calib_iter * batch_size. Default value: 100

Table 11: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--ignore_nodes	String	Specifies the name list of nodes to be ignored during quantization. Ignored nodes are left unquantized during quantization.
--skip_check	Int32	If set to 1, the check for float model is skipped. Useful when only part of the input model is quantized. Range: [0, 1] Default value: 0
--align_concat	Int32	Specifies the strategy for the alignment of the input quantizeposition for concat nodes. 0: Aligns all the concat nodes 1: Aligns the output concat nodes 2: Disables alignment Default value: 0
--align_pool	Int32	Specifies the strategy for the alignment of the input quantize position for maxpool/avgpool nodes. 0: Aligns all the maxpool/avgpool nodes 1: Aligns the output maxpool/avgpool nodes 2: Disables alignment Default value: 0
--simulate_dpu	Int32	Set to 1 to enable the simulation of the DPU. The behavior of DPU for some operations is different from TensorFlow. For example, the dividing in LeakyRelu and AvgPooling are replaced by bit-shifting, so there might be a slight difference between DPU outputs and CPU/GPU outputs. The vai_q_tensorflow quantizer simulates the behavior of these operations if this flag is set to 1. Range: [0, 1] Default value: 1
--adjust_shift_bias	Int32	Specifies the strategy for shift bias check and adjustment for DPU compiler. 0: Disables shift bias check and adjustment 1: Enables with static constraints 2: Enables with dynamic constraints Default value: 1
--adjust_shift_cut	Int32	Specifies the strategy for shift cut check and adjustment for DPU compiler. 0: Disables shift cut check and adjustment 1: Enables with static constraints Default value: 1
--arch_type	String	Specifies the arch type for fix neuron. 'DEFAULT' means quantization range of both weights and activations are [-128, 127]. 'DPUCADF8H' means weights quantization range is [-128, 127] while activation is [-127, 127]
--output_dir	String	Specifies the directory in which to save the quantization results. Default value: “./quantize_results”
--max_dump_batches	Int32	Specifies the maximum number of batches for dumping. Default value: 1

Table 11: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--dump_float	Int32	If set to 1, the float weights and activations are dumped. Range: [0, 1] Default value: 0
--dump_input_tensors	String	Specifies the input tensor name of Graph when graph entrance is not a placeholder. Add a placeholder to the `dump_input_tensor`, so that input_fn can feed data.
--scale_all_avgpool	Int32	Set to 1 to enable scale output of AvgPooling op to simulate DPU. Only kernel_size <= 64 will be scaled. This operation does not affect the special case such as kernel_size=3,5,6,7,14 Default value: 1
--do_cle	Int32	1: Enables implement cross layer equalization to adjust the weights distribution 0: Skips cross layer equalization operation Default value: 0
--replace_relu6	Int32	Available only for do_cle=1 1: Allows you to ReLU6 with ReLU 0: Skips replacement. Default value: 1
--replace_sigmoid	Int32	1: Enable replace sigmoid with hard-sigmoid 0: Skips replacement. Default value: 0
--replace_softmax	Int32	1: Enable replace softmax with hard-softmax 0: Skips replacement. Default value: 0
--convert_datatype	Int32	4: Do fold bn and convert to data type fp32 3: Do fold bn and convert to data type bfloat16 2: Do fold bn and convert to data type double 1: Do fold bn and convert to data type fp16 0: Skips conversion. Default value: 0
--output_format	String	Indicates what format to save the quantized model, 'pb' for saving tensorflow frozen pb, 'onnx' for saving onnx model. Default value: 'pb'
Session Configurations		
--gpu	String	Specifies the IDs of the GPU device used for quantization separated by commas.
--gpu_memory_fraction	Float	Specifies the GPU memory fraction used for quantization, between 0-1. Default value: 0.5
Others		
--help		Shows all available options of vai_q_tensorflow.

Table 11: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--version		Shows the version information for vai_q_tensorflow .

Examples

```
show help: vai_q_tensorflow --help
quantize:
vai_q_tensorflow quantize --input_frozen_graph frozen_graph.pb \
                          --input_nodes inputs \
                          --output_nodes predictions \
                          --input_shapes ?,224,224,3 \
                          --input_fn my_input_fn.calib_input
dump quantized model:
vai_q_tensorflow dump --input_frozen_graph quantize_results/
quantize_eval_model.pb \
                      --input_fn my_input_fn.dump_input
```

Refer to Xilinx Model Zoo for more TensorFlow model quantization examples.

vai_q_tensorflow Error Codes

Table 12: **vai_q_tensorflow Error Codes**

Error Code	Cause	Solution
Quantize_TF1_Invalid_Input	The specified `input_frozen_graph` file is not found	Check whether `input_frozen_graph` is correct and the file exists or not
Quantize_TF1_Invalid_Bitwidth	The specified `nodes_bit` value is illegal, such as less than 1	Check whether the content of `nodes_bit` is correct
Quantize_TF1_Invalid_Method	The specified `method` value is invalid and is not within the range of [0,1,2]	Check whether the value of `method` is correct
Quantize_TF1_Length_Mismatch	The specified `input_shapes` is illegal, such as mismatch with input_nodes, or is not 4-dimensional, or contains a element that is not an integer	Check whether the value of `input_shapes` is correct and matches `input_nodes`
Quantize_TF1_Invalid_Input_Fn	The specified `input_fn` module import failed	Check whether `input_fn` is correct and make sure the function is implemented correctly
Quantize_TF1_Invalid_Target_Dtype	The specified `convert_datatype` value is invalid and is not within the range of [0,1,2,3,4]	Check whether the value of `convert_datatype` is correct
Quantize_TF1_Unsupported_Op	An unsupported op, such as FusedBatchNorm, is encountered when converting datatype	Replace the unsupported op

TensorFlow 2.x Version (vai_q_tensorflow2)

Installing vai_q_tensorflow2

You can install vai_q_tensorflow2 in the following two ways:

Install Using Docker Container

Vitis AI provides a Docker container for quantization tools, including vai_q_tensorflow. After running a container, activate the Conda environment vitis-ai-tensorflow2.

```
conda activate vitis-ai-tensorflow2
```

If there is a patch package, install the vitis-ai-tensorflow2 patch package inside the Docker container.

```
# [optional]
$ sudo env CONDA_PREFIX=/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/
PATH=/opt/vitis_ai/conda/bin:$PATH conda install patch_package.tar.bz2
```

Install from Source Code with the Wheel Package

vai_q_tensorflow2 is a fork of TensorFlow Model Optimization Toolkit. It is open source in Vitis_AI_Quantizer. To build vai_q_tensorflow2, run the following command:

```
$ sh build.sh
$ pip install pkgs/*.whl
```

Install from Source Code with the Conda Package

IMPORTANT! *This requires Anaconda.*

```
# CPU-only version
$ conda build vai_q_tensorflow2_cpu_feedstock --output-folder ./conda_pkg/
# GPU version
$ conda build vai_q_tensorflow2_gpu_feedstock --output-folder ./conda_pkg/
# Install conda package on your machine
$ conda install --use-local ./conda_pkg/linux-64/*.tar.bz2
```

Inspecting the Float Model

`VitisInspector` is an experimental new feature introduced in Vitis AI 2.5 to inspect a float model and show partition results for a given DPU target architecture, together with some indications on why the layers are not mapped to DPU. Without `target`, you can only show some general, target-independent inspection results. Assign `target` to get more detailed inspect results for it.

***Note*:** This feature in only available for default `pof2s` quantize strategy due to DPU limitations. This feature is experimental, please contact us if you encounter any bugs or problems.

The following codes show how to inspect a model.

```
model = tf.keras.models.load_model(‘float_model.h5’)
from tensorflow_model_optimization.quantization.keras import vitis_inspect
inspector = vitis_inspect.VitisInspector(target="DPUCADF8H_ISA0")
inspector.inspect_model(model,
                        plot=True,
                        plot_file="model.svg",
                        dump_results=True,
                        dump_results_file="inspect_results.txt",
                        verbose=0)
```

- **target:** string or None, the target DPU to deploy this model. It can be a name string (for example, DPUCAHX8L_ISA0), a JSON file path (for example, ./U50/arch.json) or a fingerprint. If set to None, no target will be applied and only some general, target-independent inspect results will be shown. The default value is None.
- **model:** tf.keras.Model instance, the float model to be inspected. Float model should have concrete input shapes. Build the float model with concrete input shapes or call inspect_model with the `input_shape` argument.
- **input_shape:** list(int) or list(list(int)) or tuple(int) or dictionary(int), contains the input shape for each input layer. Use default shape info in the input layers if not set. Use list of shapes for multiple inputs, for example inspect_model(model, input_shape=[1, 224, 224, 3]) or inspect_model(model, input_shape=\[[None, 224, 224, 3], [None, 64, 1]]). All dimensions should have concrete values, and batch_size dimension should be None or int. If the input shape of the model is variable like [None, None, None, 3], you need to specify a shape like [None, 224, 224, 3] to generate the final quantized model. The default value is None.
- **plot:** bool, whether to plot the model inspect results by `graphviz` and save image to disk. It is helpful when you need to visualize the model inspection results together with some modification hints. Note that only part of output file types can show the hints, such as `.svg`.The default value is False.
- **plot_file:** string, file path of model image file when plotting the model. The default value is `model.svg`.

- **dump_results:** bool, whether to dump the inspect results and save text to disk. More detailed layer-by-layer results than screen logging will be dumped to the text file. The default value is False.
- **dump_results_file:** string, file path of inspect results text file. The default value is `inspect_results.txt`.
- **verbose:** int, the logging verbosity level. More detailed logging results will be shown for higher verbose value. The default value is 0.

***Note*:** A known issue about multi-outputs pattern: 1) Due to Xcompiler's pattern matching problem, when the convolution or add layer has multiple output layers and one of which is a relu activation layer, the result of relu layer may be incorrect. 2) When the relu-like activation layer is followed by multiple convolutional layers, the result of convolutional layer may be incorrect. This issue will be fixed in a later version.

Running vai_q_tensorflow2

The TensorFlow2 quantizer supports two different approaches to quantize a deep learning model:

- **Post-training quantization (PTQ):** PTQ is a technique to convert a pre-trained float model into a quantized model with little degradation in model accuracy. A representative dataset is needed to run a few batches of inference on the float model to obtain the distributions of the activations. This is also called quantize calibration.
- **Quantization aware training (QAT):** QAT models the quantization errors in both the forward and backward passes during model quantization. For QAT, starting from a float-point pre-trained model with good accuracy is recommended over starting from scratch.

Preparing the Float Model and Calibration Set

Before running vai_q_tensorflow2, prepare the float model and calibration set, including the files listed in the following table.

Table 13: **Input Files for vai_q_tensorflow2**

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.

Quantizing Using the vai_q_tensorflow2 API

The following codes show how to perform post-training quantization with vai_q_tensorflow2 API. You can find a full example here.

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           calib_steps=100,
                                           calib_batch_size=10,
                                           **kwargs)
```

- **calib_dataset:** `calib_dataset` is used as a representative calibration dataset for calibration. You can use full or part of the `eval_dataset`, `train_dataset`, or other datasets.
- **calib_steps:** `calib_steps` is the total number of steps for calibration. It has a default value of None. If `calib_dataset` is a `tf.data dataset`, generator, or `keras.utils.Sequence` instance and steps is None, calibration will run until the dataset is exhausted. This argument is not supported with array inputs.
- **calib_batch_size:** calib_batch_size is the number of samples per batch for calibration. If the "calib_dataset" is in the form of a dataset, generator, or `keras.utils.Sequence` instances, the batch size is controlled by the dataset itself. If the `calib_dataset` is in the form of a `numpy.array` object, the default batch size is 32.
- **input_shape:** list(int) or list(list(int)) or tuple(int) or dictionary(int), contains the input shape for each input layer. Use default shape info in the input layers if not set. Use list of shapes for multiple inputs, for example input_shape=[1, 224, 224, 3] or input_shape=[[None, 224, 224, 3], [None, 64, 1]]. All dimensions should have concrete values, and batch_size dimension should be None or int. If the input shape of the model is variable like [None, None, None, 3], you need to specify a shape like [None, 224, 224, 3] to generate the final quantized model.
- ****kwargs:** dict of the user-defined configurations of quantize strategy. It will override the default built-in quantize strategy. For example, setting `bias_bit=16` will let the tool to quantize all the biases with 16bit quantizers. See the vai_q_tensorflow2 Usage section for more information of the user-defined configurations.

(Optional) vai_q_tensorflow2 Fast Finetuning

Generally, there is a small accuracy loss after quantization, but for some networks such as MobileNets, the accuracy loss can be large. Fast finetuning uses the AdaQuant algorithm to adjust the weights and quantize parameters layer-by-layer with the unlabeled calibration dataset to improve accuracy for some models. It takes longer than normal PTQ (still much shorter than QAT as the `calib_dataset` is smaller than the training dataset). Fast finetuning is disabled, by default. It can be turned on to improve the performance if you meet accuracy issues. A recommended workflow is to first try PTQ without fast finetuning and then try quantization with fast finetuning if the accuracy is not acceptable. QAT is another method to improve the accuracy, but it takes more time and needs the training dataset. You can activate fast finetuning by setting `include_fast_ft=True` during post-training quantization.

```
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_steps=None, calib_batch_size=None, include_fast_ft=True,
fast_ft_epochs=10)
```

Here,

- `include_fast_ft` indicates whether to do fast finetuning or not.
- `fast_ft_epochs` indicates the number of finetuning epochs for each layer.

Saving the Quantized Model

The quantized model object is a standard `tf.keras` model object. You can save it by running the following command:

```
quantized_model.save('quantized_model.h5')
```

The generated `quantized_model.h5` file can be fed to the vai_c_tensorflow compiler and then deployed on the DPU.

(Optional) Exporting the Quantized Model to ONNX

The following codes show how to perform post-training quantization and export the quantized model to onnx with vai_q_tensorflow2 API.

```

model = tf.keras.models.load_model(‘float_model.h5’)
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           output_format='onnx',
                                           onnx_opset_version=11,
                                           output_dir='./quantize_results',
                                           **kwargs)
```

- **output_format:** A string object, indicates what format to save the quantized model. Options are: '' for skip saving, 'h5' for saving .h5 file, 'tf' for saving saved_model file, 'onnx' for saving .onnx file. Default to ''.
- **onnx_opset_version:** An int object, the ONNX opset version. Take effect only when output_format is 'onnx'. Default to 11.
- **output_dir:** A string object, indicates the directory to save the quantized model in. Default to './quantize_results'.

(Optional) Evaluating the Quantized Model

If you have scripts to evaluate float models, like the models in Xilinx Model Zoo, you can replace the float model file with the quantized model for evaluation. To support the customized quantize layers, the vitis_quantize module should be imported, for example:

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantized_model = tf.keras.models.load_model('quantized_model.h5')
```

After that, evaluate the quantized model just as the float model, for example:

```
quantized_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy()
,
    metrics= keras.metrics.SparseTopKCategoricalAccuracy())
quantized_model.evaluate(eval_dataset)
```

(Optional) Dumping the Simulation Results

Sometimes after deploying the quantized model, it is necessary to compare the simulation results on the CPU/GPU and the output values on the DPU. You can use the `VitisQuantizer.dump_model` API of vai_q_tensorflow2 to dump the simulation results with the quantized model.

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantized_model = keras.models.load_model('./quantized_model.h5')
vitis_quantize.VitisQuantizer.dump_model(model=quantized_model,
                                         dataset=dump_dataset,
                                         output_dir='./dump_results')
```

Note: The `batch_size` of the dump_dataset should be set to the same batch_size on target device for DPU debugging. It is recommended to use CPU simulation results for DPU debugging since GPU results can be non-deterministic and slightly different for float value computation.

Dump results are generated in `${dump_output_dir}` after the command has successfully executed. Results for weights and activation of each layer are saved separately in the folder. For each quantized layer, results are saved in `*.bin` and `*.txt` formats. If the output of the layer is not quantized (such as for the softmax layer), the float activation results are saved in the `*_float.bin` and `*_float.txt` files. The / symbol is replaced by _ for simplicity. Examples for dumping results are shown in the following table.

Table 14: **Example of Dumping Results**

Batch No.	Quantized	Layer Name	Saved files		
			Weights	**Biases**	**Activation**
1	Yes	resnet_v1_50/conv1	{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.bin {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.txt	{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.bin {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.txt	{output_dir}/dump_results_0/quant_resnet_v1_50_conv1.bin {output_dir}/dump_results_0/quant_resnet_v1_50_conv1.txt
2	No	resnet_v1_50/softmax	N/A	N/A	{output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.bin {output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.txt

Note: The rounding mode in implementation of DPU is "HALF_UP" for all inputs and activations. Using other rounding modes in your implementation may lead to slight bit-level mismatch with dump results.

Configure the Quantize Strategy

Some default quantize strategies are provided, but sometimes users need to modify quantize configurations for different targets or to get better performance. For example, some target devices may need the biases to be quantized into 32 bit and some may need to quantize only part of the model. This part shows how to configure the quantizer to meet your needs.

Quantize Strategy

Three main configurable parts of the quantize tool are the quantize tool pipeline, what part of the model to be quantized and how to quantize them. Define all these thing in `quantize_strategy`. Internally, each `quantize_strategy` is a JSON file containing below configurations:

- **pipeline_config:** These configurations control the work pipeline of the quantize tool, including some optimizations during quantization, e.g., whether to fold Conv2D + BatchNorm layers, whether to perform `Cross-Layer-Equalization` algorithm and so on. It can be further divided into `optimize_pipeline_config`, `quantize_pipeline_config`, `refine_pipeline_config` and `finalize_pipeline_config`.
- **quantize_registry_config:** These configurations control what layer types are quantizable, where to insert the quantize ops and what kind of quantize op to be inserted. It includes some layer specific configurations and user-defined global configurations.

Below is an example configuration for the Conv2D layers:

```
{
  "layer_type": "tensorflow.keras.layers.Conv2D",
  "quantizable_weights": ["kernel"],
  "weight_quantizers": [
  {
    "quantizer_type": "Pof2SQuantizer",
    "quantizer_params": {"bit_width": 8,"method":0, "round_mode": 1,
"symmetry": true, "per_channel": true, "channel_axis": -1, "narrow_range":
False}
  ],
  "quantizable_biases": ["bias"],
  "bias_quantizers": [
  {
    "quantizer_type": "Pof2SQuantizer",
    "quantizer_params": {"bit_width": 8,"method":0, "round_mode": 1,
"symmetry": true, "per_channel": false, "channel_axis": -1, "narrow_range":
False}
  ],
  "quantizable_activations": ["activation"],
  "activation_quantizers": [
  {
    "quantizer_type": "FSQuantizer",
    "quantizer_params": {"bit_width": 8, "method":2,
"method_percentile":99.9999, "round_mode": 1, "symmetry": true,
"per_channel": false, "channel_axis": -1}
  ]
}
```

As you can see, by using this quantize configuration, you quantize the weight, bias and activations of the Conv2D layer. The weight and bias are using `Pof2SQuantizer`(power-of-2 scale quantizer) and the activation are using `FSQuantizer`(float scale quantizer). You can apply different quantizers for different objects in one layer.

Note: The `Quantizer` here in configurations means the quantize operation applied to each object. It consumes a float tensor and output a quantized tensor. Please note that the quantization is 'fake', which means that the input is quantized to int and then de-quantized to float.

Using Built-in Quantize Strategy

Users can use dump_quantize_strategy to see get the JSON file of current quantize strategy. To make things simple, four types of built-in quantize strategies for common user cases are provided, which users can extend or override for their need, including:

- `pof2s`: power-of-2 scale quantization, mainly used for DPU targets now. Default quantize strategy of the quantizer.
- `pof2s_tqt`: power-of-2 scale quantization with trained thresholds, mainly used for QAT in DPU now.
- `fs`: float scale quantization, mainly used for devices supporting floating-point calculation, such as CPU/GPUs.

- `fsx`: trained quantization threshold for power-of-2 scale quantization, mainly used for QAT for DPU now.

Users can switch between the built-in quantize strategies by assigning `quantize_strategy` argument in the contruct function of `VitisQuantizer`. Moreover, two handy ways to configure the quantize strategy are provided.

Configure by `kwargs` in `VitisQuantizer.quantize_model()`

This is a easy way for users who need to override the default pipeline configurations or do global modifications on the quantize operations. The `kwargs` here is a dict object which keys match the quantize configurations in the JSON file. See vitis_quantize.VitisQuantizer.quantize_model for more information about available keys.

Example codes below shows how to use it.

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantizer.quantize_model(calib_dataset,
                         input_layers=['conv2'],
                         bias_bit=32,
                         activation_bit=32,
                         weight_per_channel=True)
```

In this example, the quantizer is configured to quantize part of the model. Layers before `conv2` will be not be optimized or quantized. Moreover, all the activations and biases to 32 bit instead of 8 bit are quantized, and use per_channel quantization for all weights.

Configure by `VitisQuantizer.set_quantize_strategy()`

For advanced users who want full control of the quantize tool, this API is provided to set new quantize strategies JSON file. Users can first dump the current configurations to JSON file and make modifications on the it. This allows users to override the default configurations, make more fine-grained quantizer configurations or extend the quantize config to make more layer types quantizable. Then the user can set the new JSON file to the quantizer to apply these modifications.

Example codes below shows how to do it.

```
quantizer = VitisQuantizer(model)
# Dump the current quantize strategy
quantizer.dump_quantize_strategy(dump_file='my_quantize_strategy.json',
verbose=0)

# Make modifications of the dumped file 'my_quantize_strategy.json'
# Then, set the modified json to the quantizer and do quantization
quantizer.set_quantize_strategy(new_quantize_strategy='my_quantize_strategy.
json')
quantizer.quantize_model(calib_dataset)
```

Note: `verbose` is an `int` type argument which controls the verbosity of the dumped JSON file. Greater verbose value will dump more detailed quantize strategy. Setting verbose to value greater or equal to 2 will dump the full quantize strategy.

Quantizing with Float Scale

The quantization for DPU uses power-of-2 scales, symmetry, per-tensor quantizers and need some special processes to simulate DPU behaviors. For other devices supporting floating-point scales will need a different quantize strategy, so the float scale quantization is introduced.

- **The `fs` quantize strategy:** Do quantization for inputs and weights of `Conv2D`, `DepthwiseConv2D`, `Conv2DTranspose` and `Dense` layers. By default, it will not do Conv-BN folding.
- **The `fsx` quantize strategy:** Do quantization for more layer types than `fs` quantize strategy, such as `Add`, `MaxPooling2D` and `AveragePooling2D`. Moreover, it also quantizes the biases and activations of `Conv2D`, `DepthwiseConv2D`, `Conv2DTranspose` and `Dense` layers. By default, it will do Conv-BN folding.

Note: `fs` and `fsx` strategies are designed for target devices with floating-point supports. DPU does not have floating-point support now, so models quantized with these quantize strategies can not be deployed to them.

Users can switch to use float scale quantization by setting `quantize_strategy` to `fs` or `fsx` in the construct function of `VitisQuantizer`, example codes are showed as below:

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model, quantize_strategy='fs')
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           calib_step=100,
                                           calib_batch_size=10,
                                           **kwargs)
```

- **calib_dataset:** `calib_dataset` is used as a representative calibration dataset for calibration. You can use full or part of the `eval_dataset`, `train_dataset`, or other datasets.
- **calib_steps:** `calib_steps` is the total number of steps for calibration. It has a default value of None. If `calib_dataset` is a `tf.data dataset`, generator, or `keras.utils.Sequence` instance and steps is None, calibration will run until the dataset is exhausted. This argument is not supported with array inputs.
- **calib_batch_size:** calib_batch_size is the number of samples per batch for calibration. If the "calib_dataset" is in the form of a dataset, generator, or `keras.utils.Sequence` instances, the batch size is controlled by the dataset itself. If the `calib_dataset` is in the form of a `numpy.array` object, the default batch size is 32.

- ****kwargs:** dict of the user-defined configurations of quantize strategy. It will override the default built-in quantize strategy. For example, setting `bias_bit=16` will let the tool to quantize all the biases with 16 bit quantizers. See vai_q_tensorflow2 Usage section for more information of the user-defined configurations.

Converting to Float16 or BFloat16

vai_q_tensorflow2 supports data type conversions for float models, including Float16, BFloat16, Float, and Double. The following codes show how to perform the data type conversions with vai_q_tensorflow2 API.

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(convert_datatype='float16'
                                           **kwargs)
```

- **convert_datatype:** A string object, indicates the target data type for the float model. Options are 'float16', 'bfloat16', 'float32', and 'float64'. Default value is 'float16'

vai_q_tensorflow2 Quantization Aware Training

Generally, there is a small accuracy loss after quantization but for some networks such as MobileNets, the accuracy loss can be large. In this situation, quantization aware training (QAT) can be used to further improve the accuracy of quantized models.

QAT is similar to the float model training/finetuning except that vai_q_tensorflow2 rewrites the float graph to convert it to a quantized model before the training starts. The typical workflow is as follows. You can find a complete example here.

1. Preparing the float model, dataset, and training scripts:

   Before QAT, prepare the following files:

   Table 15: **Input Files for vai_q_tensorflow2 QAT**

No.	Name	Description
1	Float model	Floating-point model files to start from. Can be omitted if training from scratch.
2	Dataset	The training dataset with labels.
3	Training Scripts	The Python scripts to run float train/finetuning of the model.

2. (Optional) Evaluate the float model.

   Evaluate the float model first before QAT to check the correctness of the scripts and dataset. The accuracy and loss values of the float checkpoint can also be a baseline for QAT.

3. Modify the training scripts and run QAT.

Use the vai_q_tensorflow2 API, `VitisQuantizer.get_qat_model` to convert the model to a quantized model and then proceed to training/finetuning with it. The following is an example:

```
model = tf.keras.models.load_model('float_model.h5')


# *Call Vai_q_tensorflow2 api to create the quantize training model
from tensorflow_model_optimization.quantization.keras import
vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
qat_model = quantizer.get_qat_model(
    init_quant=True, # Do init PTQ quantization will help us to get a
better initial state for the quantizers, especially for `pof2s_tqt`
strategy. Must be used together with calib_dataset
    calib_dataset=calib_dataset)

# Then run the training process with this qat_model to get the quantize
finetuned model.
# Compile the model
qat_model.compile(
        optimizer= RMSprop(learning_rate=lr_schedule),
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),
        metrics=keras.metrics.SparseTopKCategoricalAccuracy())


# Start the training/finetuning
qat_model.fit(train_dataset)
```

***Note*:** Vitis AI supports pof2s_tqt quantize strategy since 2.0. It uses trained threshold in quantizers and may result in better results for QAT. By default, the Straight-Through-Estimator is used. 8bit_tqt strategy should only be used in QAT with `'init_quant=True'` to get best performance. Initialization with PTQ quantization can generate a better initial state for quantizer parameters, especially for pof2s_tqt. Otherwise, the training may not converge.

4. Save the model.

   Call `model.save()` to save the trained model or use callbacks in `model.fit()` to save the model periodically. For example:

```
# save model manually
qat_model.save('trained_model.h5')

# save the model periodically during fit using callbacks
qat_model.fit(
    train_dataset,
    callbacks = [
              keras.callbacks.ModelCheckpoint(
              filepath='./quantize_train/'
              save_best_only=True,
              monitor="sparse_categorical_accuracy",
              verbose=1,
      )])
```

5. Convert to deployable quantized model.

Modify the trained/finetuned model to meet the compiler requirements. For example, if "train_with_bn" is set to TRUE, it means that the BatchNormalization layers are not folded during training and must be folded before deployment. Some of the quantizer parameters may vary during training and exceed the compiler limitation ranges. These must be corrected before deployment.

A `get_deploy_model()` function is provided to perform these conversions and generate a deployable model as shown in the following example.

```
quantized_model = vitis_quantizer.get_deploy_model(qat_model)
quantized_model.save('quantized_model.h5')
```

6. (Optional) Evaluate the quantized model

   Call `model.evaluate()` on the `eval_dataset` to evaluate the quantized model, just like evaluation of the float model.

```
from tensorflow_model_optimization.quantization.keras import
vitis_quantize
quantized_model = tf.keras.models.load_model('quantized_model.h5')

quantized_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentrop
y(),
        metrics= keras.metrics.SparseTopKCategoricalAccuracy())
quantized_model.evaluate(eval_dataset)
```

RECOMMENDED: *Use the float model training and finetuning before proceeding to QAT.*

Quantizing with Custom Layers

Tensorflow 2 provides a lot of common built-in layers to build the machine learning models, as well as easy ways to for you to write your own application-specific layers either from scratch or as the composition of existing layers. `Layer` is one of the central abstractions in `tf.keras`, subclassing `Layer` is the recommended way to create custom layers. Please refer to tensorflow user guide for more information.

Vai_q_tensorflow2 provides support for new custom layers via subclassing, including quantizing models with custom layers and an experimental support for quantizing the custom layers with custom quantize strategies.

***Note*:** Custom model via subclassing `tf.keras.Model` is not supported by vai_q_tensorflow2 in this release, please flatten it to layers.

Quantizing models with custom layers

vai_q_tensorflow2 provides interfaces to load the custom layers that are available in some models. For example:

```
class MyCustomLayer(keras.layers.Layer):

    def __init__(self, units=32, **kwargs):
        super(MyLayer, self).__init__(kwargs)
        self.units = units


    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
            name='w')
        self.b = self.add_weight(
            shape=(self.units,), initializer="zeros", trainable=True,
name='b')


    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b


    def get_config(self):
        base_config = super(MyLayer, self).get_config()
        config = {"units": self.units}
        return dict(list(base_config.items()) + list(config.items()))


# Here is a float model with custom layer "MyCustomLayer", use
custom_objects argument in tf.keras.models.load_model to load it.
float_model = tf.keras.models.load_model(‘float_model.h5’,
custom_objects={'MyCustomLayer': MyCustomLayer})
```

Here, a float model contains a custom layer named `"MyCustomLayer"`. The `custom_objects` argument in the `tf.keras.model.load_model` API is needed to load it. Similarly, the `VitisQuantizer` class provides the `'custom_objects'` argument to handle the custom layers. The following code is an example. The argument `custom_objects` is a dict containing the `{"custom_layer_class_name":"custom_layer_class"}`, multiple custom layers should be separated by a comma. Moreover, `add_shape_info` should also be set to True for the `quantize_model` API when quantizing models with custom layers to add shape inference information for them.

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
# Register the custom layer to VitisQuantizer by custom_objects argument.
quantizer = vitis_quantize.VitisQuantizer(float_model,
custom_objects={'MyCustomLayer': MyCustomLayer})
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_step=100, calib_batch_size=10, add_shape_info=True)
```

During the quantization, these custom layers will be wrapped by `CustomLayerWrapper` and kept unquantized. You can find a complete example here.

Note: When calling the `dump_model` API to dump golden results for data checking during deployment, set `dump_float=True` to dump float weights and activation for the custom layers, since they are not quantized.

(Experimental) Quantizing custom layers with custom quantize strategy

With the default quantize strategy, the custom layers are not quantized and continue to exist as float during the quantization as they are not in the list of supported APIs of vai_q_tensorflow2. An interface named `'custom_quantize_strategy'` is provided for advanced users to build custom quantize strategies to run quantize experiments. The custom quantize strategy is a Dict object containing the quantize strategy items as a JSON file of the Dict.

The default quantize strategy provides an example of the quantize strategy. The custom quantize strategy follows the same format. However, the same item in the custom quantize strategy will override the one in the default strategy, but new items will be added to the quantize strategy.

With this feature, you can quantize the `'MyCustomLayer'` layer from the previous example:

```
# Define quantizer with custom quantize strategy, which quantizes w,b and
outputs 0 of MyCustomLayer objects.
my_quantize_strategy = {
    "quantize_registry_config": {
        "layer_quantize_config": [{
            "layer_type": "__main__.MyCustomLayer",
            "quantizable_weights": ["w", "b"],
            "weight_quantizers": [
                "quantizer_type":
"LastValueQuantPosQuantizer","quantizer_params": {"bit_width": 8, "method":
1, "round_mode": 0},
                "quantizer_type": "LastValueQuantPosQuantizer",
"quantizer_params": {"bit_width": 8, "method": 1, "round_mode": 0}
            ],
            "quantizable_outputs": ["0"],
            "output_quantizers": [
                "quantizer_type": "LastValueQuantPosQuantizer",
"quantizer_params": {"bit_width": 8, "method": 1, "round_mode": 1}
            ]
        }]
    }
}
quantizer = vitis_quantize.VitisQuantizer(model, custom_objects={'MyLayer':
MyLayer}, custom_quantize_strategy=my_quantize_strategy)


# The following quantization process are all the same as before, here we do
normal PTQ as an example
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_step=100, calib_batch_size=10)
```

vai_q_tensorflow2 Supported Operations and APIs

The following table lists the supported operations and APIs for vai_q_tensorflow2.

Table 16: **vai_q_tensorflow2 Supported Layers**

Layer Types	Supported Layers	Description
Core	tf.keras.layers.InputLayer	
Core	tf.keras.layers.Dense	
Core	tf.keras.layers.Activation	If 'activation' is 'relu' or 'linear', will be quantized. If 'activation' is 'sigmoid' or 'swish', will be converted to hard-sigmoid or hard-swish and then be quantized by default. Otherwise will not be quantized.
Convolution	tf.keras.layers.Conv2D	
Convolution	tf.keras.layers.DepthwiseConv2D	
Convolution	tf.keras.layers.Conv2DTranspose	
Convolution	tf.keras.layers.SeparableConv2D	
Pooling	tf.keras.layers.AveragePooling2D	
Pooling	tf.keras.layers.MaxPooling2D	
Pooling	tf.keras.layers.GlobalAveragePooling	
Normalization	tf.keras.layers.BatchNormalization	By default, BatchNormalization layers are fused with the previous convolution layers. If they cannot be fused, they are converted to depthwise convolutions. In the QAT mode, BatchNormalization layers are pseudo fused if train_with_bn is set to TRUE. They are fused when the get_deploy_model function is called.
Regularization	tf.keras.layers.Dropout	By default, the dropout layers are removed. In the QAT mode, dropout layers are retained if remove_dropout is set FALSE. It is removed when the get_deploy_model function is called.
Reshaping	tf.keras.layers.Reshape	
Reshaping	tf.keras.layers.Flatten	
Reshaping	tf.keras.UpSampling2D	
Reshaping	tf.keras.ZeroPadding2D	
Merging	tf.keras.layers.Concatenate	
Merging	tf.keras.layers.Add	
Merging	tf.keras.layers.Muliply	
Activation	tf.keras.layers.ReLU	
Activation	tf.keras.layers.Softmax	The input for the Softmax layer is quantized. It can run on the standalone Softmax IP for acceleration.
Activation	tf.keras.layers.LeakyReLU	Only 'alpha'=0.1 is supported on the DPU (0.1 will be converted to 26/256 by the quantizer). For other values, it is not quantized and mapped to the CPU.

Table 16: **vai_q_tensorflow2 Supported Layers** *(cont'd)*

Layer Types	Supported Layers	Description
Hard_sigmoid	tf.keras.layer.ReLU(6.)(x + 3.) * (1. / 6.))	The supported hard_sigmoid is from Mobilenet_v3. tf.keras.Activation.hard_sigmoid is not supported now and will not be quantized.
Activation	tf.keras.layers.PReLU	

Note: The DPU may have limitations of these supported layers, they may be rolled back to CPU during compilation. See Supported Operators and DPU Limitations for more information.

vai_q_tensorflow2 Usage

vitis_inspect.VitisInspector

The construction function of class `VitisInspector`.

```
vitis_inspect.VitisInspector(
    target=None)
```

Arguments

- **target:** **target**: string or None, the target DPU to deploy this model. It can be a name string (for example, DPUCZDX8G_ISA1_B4096), a JSON file path (for example, ./U50/arch.json) or a fingerprint. The default value is None, if the target DPU is not specified, an error will be reported.

vitis_inspect.VitisInspector.inspect_model

This function performs float model inspection.

```
VitisInspector.inspect_model(model,
                             input_shape=None,
                             dump_model=True,
                             dump_model_file="inspect_model.h5",
                             plot=True,
                             plot_file="model.svg",
                             dump_results=True,
                             dump_results_file="inspect_results.txt",
                             verbose=0)
```

Arguments

- **model:** tf.keras.Model instance, the float model to be inspected. Float model should have concrete input shapes, please build the float model with concrete input shapes or call inspect_model with `input_shape` argument.

- **input_shape:** list(int) or list(list(int)), contains the input shape for each input layer. Use default shape info in input layers if not set. Use list of shapes for multiple inputs, e.g. inspect_model(model, input_shape=[224, 224, 3]) or inspect_model(model, input_shape=[[224, 224, 3], [64, 1]]). All dimensions should have concrete values, batch_size dimension should be omitted. Default to None.

- **dump_model:** bool, whether to dump the inspected model and save model to disk. The default value is False.

- **dump_model_file:** string, path of inspected model file. The default value is 'inspect_model.h5'.

- **plot:** bool, whether to plot the model inspect results by `graphviz` and save image to disk. It is helpful when you need to visualize the model inspect results together with some modification hints. Note that only part of output file types can show the hints, such as `.svg`. Default to False.

- **plot_file:** string, file path of model image file when plotting the model. Default to 'model.svg'.

- **dump_results:** bool, whether to dump the inspect results and save text to disk. More detailed layer by layer results than screen logging will be dumped to the text file. Default to False.

- **dump_results_file:** string, file path of inspect results text file. Default to 'inspect_results.txt'.

- **verbose:** int, the logging verbosity level, more detailed logging results will be showed for higher verbose value. Default to 0.

vitis_quantize.VitisQuantizer

The construction function of class `VitisQuantizer`.

```
vitis_quantize.VitisQuantizer(
    float_model,
    quantize_strategy='pof2s',
    custom_quantize_strategy=None,
    custom_objects={})
```

Arguments

- **float_model:** A `tf.keras.Model` object, containing the configurations for quantization.

- **quantize_strategy:** A string object of the quantize strategy type. Available values are *`pof2s`* , *`pof2s_tqt`*, *`fs`* and *`fsx`*. *`pof2s`* is the default strategy that uses power-of-2 scale quantizer and the Straight-Through-Estimator. *`pof2s_tqt`* is a strategy introduced in Vitis AI 1.4 which uses Trained-Threshold in power-of-2 scale quantizers and may generate better results for QAT. *`fs`* is a new quantize strategy introduced in Vitis AI 2.5, it do float scale quantization for inputs and weights of Conv2D, DepthwiseConv2D, Conv2DTranspose and Dense layers. *`fsx`* quantize strategy do quantization for more layer types than *fs* quantize straetgy, such as Add, MaxPooling2D and AveragePooling2D. Moreover, it also quantizes the biases and activations.

***Note*:** *pof2s_tqt* strategy should only be used in QAT and be used together with `init_quant=True` to get the best performance.

***Note*:** *`fs`* and *`fsx`* strategy are designed for target devices with floating-point supports. DPU does not have floating-point support now, so models quantized with these quantize strategies can not be deployed to them.

- **custom_quantize_strategy:** A string object, the file path of custom quantize strategy JSON file.
- **custom_objects:** A Dict object, mapping names (strings) to custom classes or functions.

vitis_quantize.VitisQuantizer.quantize_model

This function performs the post-training quantization (PTQ) of the float model, including model optimization, weights quantization, and activation quantize calibration.

```
vitis_quantize.VitisQuantizer.quantize_model(
    calib_dataset=None,
    calib_batch_size=None,
    calib_steps=None,
    verbose=0,
    add_shape_info=False,
    **kwargs)
```

Arguments

- **calib_dataset:** A `tf.data.Dataset`, `keras.utils.Sequence`, or `np.numpy` object, the representative dataset for calibration. You can use full or part of eval_dataset, train_dataset, or other datasets as calib_dataset.
- **calib_steps:** An int object, the total number of steps for calibration. Ignored with the default value of None. If "calib_dataset" is a `tf.data` dataset, generator, or `keras.utils.Sequence` instance and steps is None, calibration will run until the dataset is exhausted. This argument is not supported with array inputs.
- **calib_batch_size:** An int object, the number of samples per batch for calibration. If the "calib_dataset" is in the form of a dataset, generator, or `keras.utils.Sequence` instances, the batch size is controlled by the dataset itself. If the "calib_dataset" is in the form of a `numpy.array` object, the default batch size is 32.
- **verbose:** An `int` object, the verbosity of the logging. Greater verbose value will generate more detailed logging. Default to 0.
- **add_shape_info:** An `bool` object, whether to add shape inference information for custom layers. Must be set True for models with custom layers.
- ****kwargs:** A dict object, the user-defined configurations of quantize strategy. It will override the default built-in quantize strategy. Detailed user-defined configurations are listed below.

Arguments in `**kwargs`

`**kwargs` in this API is a dict of the user-defined configurations of quantize strategy. It will override the default built-in quantize strategy. For example, setting "bias_bit=16" will let the tool to quantize all the biases with 16bit quantizers. Detailed user-defined configurations are listed below.

- **separate_conv_act:** A `bool` object, whether to separate activation functions from the `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` layers. Default to `True`.
- **fold_conv_bn:** A `bool` object, whether to fold the batch norm layers into previous `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` layers.
- **convert_bn_to_dwconv:** Named `fold_bn` in Vitis-AI 2.0 and previous versions.A `bool` object, whether to convert the standalone BatchNormalization layer into DepthwiseConv2D layers.
- **convert_sigmoid_to_hard_sigmoid:** Named `replace_sigmoid` in Vitis-AI 2.0 previous versions.A `bool` object, whether to replace the Activation(activation='sigmoid') and Sigmoid layers into hard sigmoid layers and do quantization. If not, the sigmoid layers will be left unquantized and will be scheduled on CPU.
- **convert_relu_to_relu6:** Named `replace_relu6` in Vitis-AI 2.0 and previous versions.A `bool` object, whether to replace the ReLU6 layers with ReLU layers.
- **include_cle:** A `bool` object, whether to do Cross-Layer Equalization before quantization.
- **cle_steps:** A `int` object, the iteration steps to do Cross-Layer Equalization.
- **cle_to_relu6:** Named `forced_cle` in Vitis-AI 2.0 and previous versions.A `bool` object, whether to do forced Cross-Layer Equalization for ReLU6 layers.
- **include_fast_ft:** A `bool` object, whether to do fast fine-tuning or not. Fast fine-tuning adjust the weights layer by layer with calibration dataset and may get better accuracy for some models. Fast fine-tuning is disabled by default. It takes longer than normal PTQ (still much shorter than QAT as calib_dataset is much smaller than the training dataset). Turn on to improve the performance if you meet accuracy issues.
- **fast_ft_epochs:** An `int` object, the iteration epochs to do fast fine-tuning for each layer.
- **output_format:** A string object, indicates what format to save the quantized model. Options are: '' for skip saving, 'h5' for saving .h5 file, 'tf' for saving saved_model file, 'onnx' for saving .onnx file. Default to ''.
- **onnx_opset_version:** An int object, the ONNX opset version. Take effect only when output_format is 'onnx'. Default to 11.
- **output_dir:** A string object, indicates the directory to save the quantized model in. Default to './quantize_results'.

- **convert_datatype:** A string object, indicates the target data type for the float model. Options are 'float16', 'bfloat16', 'float32', and 'float64'. Default value is 'float16'.
- **input_layers:** A `list(string)` object, names of the start layers to be quantized. Layers before these layers in the model will not be optimized or quantized. For example, this argument can be used to skip some pre-processing layers or stop quantizing the first layer. Default to `[]`.
- **output_layers:** A `list(string)` object, names of the end layers to be quantized. Layers after these layers in the model will not be optimized or quantized. For example, this argument can be used to skip some post-processing layers or stop quantizing the last layer. Default to `[]`.
- **ignore_layers:** A `List(string)` object, names of the layers to be ignored during quantization. For example, this argument can be used to skip quantizing some sensitive layers to improve accuracy. Default to `[]`.
- **input_bit:** An `int` object, the bit width of all inputs. Default to 8.
- **input_method:** An `int` object, the method to calculate scale factors in quantization of all inputs. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, 3 for `Percentile`. Default to 0.
- **input_symmetry:** A `bool` object, whether to do symmetry or asymmetry quantization for all inputs. Default to `True`.
- **input_per_channel:** A `bool` object, whether to do per-channel or per-tensor quantization for all inputs. Default to `False`.
- **input_round_mode:** An `int` object, the rounding mode used in quantization of all inputs. Options are: 0 for `HALF_TO_EVEN`, 1 for `HALF_UP`, 2 for `HALF_AWAY_FROM_ZERO`. Default to 1.
- **input_unsigned:** An `bool` object, whether to use unsigned integer quantization for all inputs. It is usually used for non-negative numeric inputs (such as range from 0 to 1) when input_unsigned is true. Default to `False`.
- **weight_bit:** An `int` object, the bit width of all weights. Default to 8.
- **weight_method:** An `int` object, the method to calculate scale factors in quantization of all weights. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, 3 for `Percentile`. Default to 1.
- **weight_symmetry:** A `bool` object, whether to do symmetry or asymmetry quantization for all weights. Default to `True`.
- **weight_per_channel:** An `bool` object, whether to do per-channel or per-tensor quantization for all weights. Default to `False`.

- **weight_round_mode:** An `int` object, the rounding mode used in quantization of all weights. Options are: 0 for `HALF_TO_EVEN`, 1 for `HALF_UP`, 2 for `HALF_AWAY_FROM_ZERO`. Default to 0.
- **weight_unsigned:** An `bool` object, whether to use unsigned integer quantization for all weights. It is usually used when weight_symmetry is false. Default to `False`.
- **bias_bit:** An `int` object, the bit width of all biases. Default to 8.
- **bias_method:** An `int` object, the method to calculate scale factors in quantization of all biases. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, 3 for `Percentile`. Default to 0.
- **bias_symmetry:** A `bool` object, whether to do symmetry or asymmetry quantization for all biases. Default to `True`.
- **bias_per_channel:** An `bool` object, whether to do per-channel or per-tensor quantization for all biases. Default to `False`.
- **bias_round_mode:** An `int` object, the rounding mode used in quantization of all biases. Options are: 0 for `HALF_TO_EVEN`, 1 for `HALF_UP`, 2 for `HALF_AWAY_FROM_ZERO`. Default to 0.
- **bias_unsigned:** An `bool` object, whether to use unsigned integer quantization for all bias. It is usually used when bias_symmetry is false. Default to `False`.
- **activation_bit:** An `int` object, the bit width of all activations. Default to 8.
- **activation_method:** An `int` object, the method to calculate scale factors in quantization of all activations. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, 3 for `Percentile`. Default to 1.
- **activation_symmetry:** A `bool` object, whether to do symmetry or asymmetry quantization for all activations. Default to `True`.
- **activation_per_channel:** An `bool` object, whether to do per-channel or per-tensor quantization for all activations. Default to `False`.
- **activation_round_mode:** An `int` object, the rounding mode used in quantization of all activations. Options are: 0 for `HALF_TO_EVEN`, 1 for `HALF_UP`, 2 for `HALF_AWAY_FROM_ZERO`. Default to 1.
- **activation_unsigned:** An `bool` object, whether to use unsigned integer quantization for all activations. It is usually used for non-negative numeric activations (such as ReLU or ReLU6) when activation_symmetry is true. Default to `False`.

vitis_quantize.VitisQuantizer.dump_model

This function dumps the simulation results of the quantized model, including weights and activation results.

```
vitis_quantize.VitisQuantizer.dump_model(
    model,
    dataset=None,
    output_dir='./dump_results',
    dump_float=False,
    weights_only=False)
```

Arguments

- **model:** A `tf.keras.Model` object, the quantized model to dump.
- **dataset:** A `tf.data.Dataset`, `keras.utils.Sequence` or `np.numpy` object, the dataset used to dump, not needed if weights_only is set to `True`.
- **output_dir:** A `string` object, the directory to save the dump results.
- **weights_only:** A `bool` object, set to `True` to only dump the weights, set to `False` will also dump the activation results.

vitis_quantize.VitisQuantizer.dump_quantize_strategy

This function dumps current quantize strategy configurations to JSON file.

```
vitis_quantize.VitisQuantizer.dump_quantize_strategy(
    dump_file='quantize_strategy.json',
    verbose=0)
```

Arguments

- **dump_file:** A `string` object, file path of the dumped quantize strategy JSON file.
- **verbose:** An `int` object, the verbosity of the dumped JSON file. Greater verbose value will dump more detailed quantize strategy. Setting verbose to value greater or equal to 2 will dump the full quantize strategy. Default to 0.

vitis_quantize.VitisQuantizer.set_quantize_strategy

This function updates the quantize strategy with the new configurations in the JSON file.

```
vitis_quantize.VitisQuantizer.set_quantize_strategy(
    new_quantize_strategy='quantize_strategy.json')
```

Arguments

- **new_quantize_strategy:** A `string` object, file path of the new quantize strategy JSON file.

vitis_quantize.VitisQuantizer.get_qat_model

This function gets the float model for QAT.

```
vitis_quantize.VitisQuantizer.get_qat_model(
    init_quant=False,
    calib_dataset=None,
    calib_batch_size=None,
    calib_steps=None,
    train_with_bn=False,
    freeze_bn_delay=-1)
```

Arguments

- **init_quant:** A `bool` object to notify whether or not to run initial quantization before QAT. Running an initial PTQ quantization yields an improved initial state for the quantizer parameters, especially for 8bit_tqt strategy. Otherwise, the training may not converge.
- **calib_dataset:** A `tf.data.Dataset`, `keras.utils.Sequence` or `np.numpy` object, the representative dataset for calibration. Must be set when "init_quant" is set `True`. You can use full or part of eval_dataset, train_dataset or other datasets as calib_dataset.
- **calib_steps:** An int object, the total number of steps for initial PTQ. Ignored with the default value of None. If "calib_dataset" is a `tf.data dataset`, generator or `keras.utils.Sequence` instance and steps is None, calibration will run until the dataset is exhausted. This argument is not supported with array inputs.
- **calib_batch_size:** An int object, the number of samples per batch for initial PTQ. If the "calib_dataset" is in the form of a dataset, generator or `keras.utils.Sequence` instances, the batch size is controlled by the dataset itself. If the "calib_dataset" is in the form of a `numpy.array` object, the default batch size is 32.
- **train_with_bn:** A `bool` object, whether to keep bn layers during QAT. If set to True, bn parameters are updated during quantize-aware training and help the model to converge. These trained bn layers are then fused into previous convolution-like layers in the get_deploy_model() function. If the float model has no bn layers, this option has have no affect. The default value is False.
- **freeze_bn_delay:** An int object, the train steps before freezing the bn parameters. After the delayed steps, model will switch inference bn parameters to avoid instability in training. Only take effect when train_with_bn is True. Default value is -1, which means never do bn freezing.

vitis_quantize.VitisQuantizer.get_deploy_model

This function converts the QAT model and generates the deployable model. The results can be fed into the `vai_c_tensorflow` compiler.

```
vitis_quantize.VitisQuantizer.get_deploy_model(model)
```

Arguments

- **model:** A `tf.keras.Model` object, the QAT model to deploy.

Examples

Quantize

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset)
```

Evaluate the Quantized Model

```
quantized_model.compile(loss=your_loss, metrics=your_metrics)
quantized_model.evaluate(eval_dataset)
```

Load the Quantized Model

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
with vitis_quantize.quantize_scope():
    model = keras.models.load_model('./quantized_model.h5')
```

Dump the Quantized Model

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
with vitis_quantize.quantize_scope():
    quantized_model = keras.models.load_model('./quantized_model.h5')
    vitis_quantize.VitisQuantizer.dump_model(quantized_model, dump_dataset)
```

vai_q_tensorflow2 Error Codes

Table 17: **vai_q_tensorflow2 Error Codes**

Error Description	Error Types	Causes and Solutions
Quantizer_TF2_Unsupported_Layer	Unsupported layer type	Layer is not a `tf.keras.layers.Layer` or this layer is not yet supported. By default, this layer will not be quantized and will be mapped to run on CPU. You can use the experimental support for customizing quantize strategy to define the quantization behaviour of it.

Table 17: **vai_q_tensorflow2 Error Codes** *(cont'd)*

Error Description	Error Types	Causes and Solutions
Quantizer_TF2_Unsupported_Model	Unsupported model type	Only tf.keras sequential or functional models can be supported. Subclassing model is not supported now, please convert it to sequential or functional model and try again.
Quantizer_TF2_Invalid_Input_Shape	Invalid input shape	The input_shape parameter is not valid, please check and set correct value for it.
Quantizer_TF2_Invalid_Calib_Dataset	Invalid calibration dataset	The calibration dataset is not valid, please check and set correct value for it.
Quantizer_TF2_Invalid_Target	Invalid target	The target parameter is not valid, please check and set correct value for it.

PyTorch Version (vai_q_pytorch)

Installing vai_q_pytorch

vai_q_pytorch has GPU and CPU versions. It supports PyTorch version 1.2~1.12but does not support PyTorch data parallelism. There are two ways to install vai_q_pytorch:

Install Using Docker Containers

The Vitis AI provides a Docker container for quantization tools, including vai_q_pytorch. After running a GPU/CPU container, activate the Conda environment, vitis-ai-pytorch.

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

***Note*:** In some cases, if you want to install some packages in the Conda environment and encounter permission problems, you can create a separate Conda environment based on `vitis-ai-pytorch` instead of using `vitis-ai-pytorch` directly. The `pt_pointpillars_kitti_12000_100_10.8G_1.3` model in Xilinx Model Zoo is an example of this.

A new Conda environment with a specified PyTorch version (1.2~1.12) can be created using the https://github.com/Xilinx/Vitis-AI/blob/v3.0/docker/common/replace_pytorch.sh script. This script clones a Conda environment from vitis-ai-pytorch, uninstalls the original PyTorch, Torchvision and vai_q_pytorch packages, and then installs the specified version of PyTorch, Torchvision, and re-installs vai_q_pytorch from source code. The following is the command line to create a new Conda environment with the script:

```
replace_pytorch.sh new_conda_env_name
```

***Note*:** Before running the script, you must check the version of Python, PyTorch, and cuda-toolkit version in the `replace_pytorch.sh` script and edit them according to your requirement. When choosing PyTorch version and editing the command line, it needs to follow the instructions on pytorch official webpage.

Install from the Source Code

vai_q_pytorch is a Python package designed to work as a PyTorch plugin. It is an open source in Vitis_AI_Quantizer. It is recommended to install vai_q_pytorch in the Conda environment. To do so, follow these steps:

1. Add the CUDA_HOME environment variable in .bashrc.

   For the GPU version, if the CUDA library is installed in `/usr/local/cuda`, add the following line into .bashrc. If CUDA is in other directory, change the line accordingly.

   ```
   export CUDA_HOME=/usr/local/cuda
   ```

   For the CPU version, remove all CUDA_HOME environment variable setting in your .bashrc. It is recommended to cleanup it in command line of a shell window by running the following command:

   ```
   unset CUDA_HOME
   ```

2. Install PyTorch (1.2~1.12) and Torchvision.

   The following code takes PyTorch 1.7.1 and torchvision 0.8.2 as an example. You can find detailed instructions for other versions on the PyTorch website.

   ```
   pip install torch==1.7.1 torchvision==0.8.2
   ```

3. Install other dependencies.

   ```
   pip install -r requirements.txt
   ```

4. Install vai_q_pytorch.

   ```
   cd ./pytorch_binding
   python setup.py install
   ```

5. Verify the installation.

   ```
   python -c "import pytorch_nndct"
   ```

***Note*:** If the installed PyTorch version is 1.4 or higher, import pytorch_nndct before importing torch in your script. This is caused by a PyTorch bug in versions prior to 1.4. Refer to PyTorch GitHub issue 28536 and 19668 for details.

```
import pytorch_nndct
import torch
```

Inspect Float Model Before Quantization

Vai_q_pytorch provides a function called inspector to help you 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.

Take resnet18_quant.py to demonstrate how to edit model code and apply this feature:

1. Import vai_q_pytorch module

   ```
   from pytorch_nndct.apis import Inspector
   ```

2. Create a inspector with target name or fingerprint

   ```
   inspector = Inspector("0x603000b16013831") # by target fingerprint
   or
   inspector = Inspector("DPUCAHX8L_ISA0_SP") # by target name
   ```

3. Inspect float model

   ```
   input = torch.randn([batch_size, 3, 224, 224])
   inspector.inspect(model, input)
   ```

Run the following command line to inspect model:

```
python resnet18_quant.py --quant_mode float --inspect
```

Inspector will display some special messages on screen with special color and special keyword prefix "VAIQ_*" according to the verbose_level setting. Note the messages displayed between "[VAIQ_NOTE]: =>Start to inspect model..." and "[VAIQ_NOTE]: =>Finish inspecting."

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
inspetion result is generated
```

Note:

- The inspector relies on 'xcompiler' package. In conda env vitis-ai-pytorch in Vitis-AI docker, xcompiler is ready. But if vai_q_pytorch is installed by source code, it needs to install xcompiler in advance.
- Visualization of inspection results relies on the dot engine. If you don't install dot successfully, set 'image_format = None' when inspecting.

- If you need more detailed guidance, you can refer to example/jupyter_notebook/inspector/inspector_tutorial.ipynb. Install jupyter notebook in advance. Run the following command:

```
jupyter notebook example/jupyter_notebook/inspector/
inspector_tutorial.ipynb
```

Running vai_q_pytorch

vai_q_pytorch is designed to work as a PyTorch plugin. Xilinx provides the simplest APIs to introduce the FPGA-friendly quantization feature. For a well-defined model, you only need to add a few lines to get a quantize model object. To do so, follow these steps:

Preparing Files for vai_q_pytorch

Prepare the following files for vai_q_pytorch.

Table 18: **Input Files for vai_q_pytorch**

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

Modifying the Model Definition

To make a PyTorch model quantizable, it is necessary to modify the model definition to make sure the modified model meets the following conditions. An example is available in Vitis AI GitHub.

1. The model to be quantized should include forward method only. All other functions should be moved outside or move to a derived class. These functions usually work as pre-processing and post-processing. If they are not moved outside, the API removes them in the quantized module, which causes unexpected behavior when forwarding the quantized module.
2. The float model should pass the jit trace test. Set the float module to evaluation status, then use the `torch.jit.trace` function to test the float model. For more details, please refer to example/jupyter_notebook/jit_trace_test/jit_trace_test.ipynb.
3. The most common operators in pytorch are supported in vai_q_pytorch. For more information, go to doc/support_op.md.

Adding vai_q_pytorch APIs to Float Scripts

If there is a trained float model and some Python scripts to evaluate accuracy/mAP of the model before quantization, the Quantizer API replaces the float module with a quantized module. The normal evaluate function encourages quantized module forwarding. Quantize calibration determines quantization steps of tensors in evaluation process if flag quant_mode is set to "calib". After calibration, evaluate the quantized model by setting quant_mode to "test".

1. Import the vai_q_pytorch module.

```
from pytorch_nndct.apis import torch_quantizer, dump_xmodel
```

2. Generate a quantizer with quantization needed input and get the converted model.

```
input = torch.randn([batch_size, 3, 224, 224])
quantizer = torch_quantizer(quant_mode, model, (input))
quant_model = quantizer.quant_model
```

3. Forward a neural network with the converted model.

```
acc1_gen, acc5_gen, loss_gen = evaluate(quant_model, val_loader, loss_fn)
```

4. Output the quantization result and deploy the model.

```
if quant_mode == 'calib':
    quantizer.export_quant_config()
if deploy:

    quantizer.export_torch_script()
    quantizer.export_onnx_model()
    quantizer.export_xmodel(deploy_check=False)
```

Running Quantization and Getting the Result

Note: vai_q_pytorch log messages have special colors and a special keyword prefix, "VAI_Q_*.". vai_q_pytorch log message types include "error", "warning", and "note." Pay attention to vai_q_pytorch log messages to check the flow status.

1. Run command with "--quant_mode calib" to quantize model.

   ```
   python resnet18_quant.py --quant_mode calib --subset_len 200
   ```

   When calibrating forward, borrow the float evaluation flow to minimize code change from float script. If you encounter loss and accuracy messages displayed in the end, you can ignore them.

   It is important to control iteration numbers during quantization and evaluation. Generally, 100-1000 images are enough for quantization and the whole validation set is required for evaluation. The iteration numbers can be controlled in the data loading part. In this case, the `subset_len` argument controls the number of images that are used for network forwarding. If the float evaluation script does not have an argument with a similar role, you must add one.

   If this quantization command runs successfully, two important files are generated in the output directory `./quantize_result`.

   - **ResNet.py:** Converted vai_q_pytorch format model.
   - **Quant_info.json:** Quantization steps of tensors. Retain this file for evaluating quantized models.

2. To evaluate the quantized model, run the following command:

   ```
   python resnet18_quant.py --quant_mode test
   ```

The accuracy displayed after the command has executed successfully is the accuracy for the quantized model.

3. To generate the XMODEL for compilation (and ONNX format quantized model) , the batch size should be 1. Set `subset_len=1` to avoid redundant iterations and run the following command:

   ```
   python resnet18_quant.py --quant_mode test --subset_len 1 --batch_size=1
   --deploy
   ```

   Skip loss and accuracy displayed in the log during running. The xmodel file for the Vitis AI compiler is generated in the output directory, `./quantize_result`. It is further used to deploy to the FPGA.

   ```
   ResNet_int.xmodel: deployed XIR format model
   ResNet_int.onnx:   deployed onnx format model
   ResNet_int.pt:     deployed torch script format model
   ```

   Note: XIR is ready in "vitis-ai-pytorch" conda environment in the Vitis AI docker but if vai_q_pytorch is installed from the source code, you have to install XIR in advance. If XIR is not installed, the xmodel file cannot be generated and the command will return an error. However, you can still check the accuracy in the output log.

Hardware-Aware Quantization Strategy

Inspector provides device assignments to operators in the neural network based on the target device. vai_q_pytorch can use the power of inspector to perform hardware-aware quantization.

Example code in example/resnet18_quant.py:

```
quantizer = torch_quantizer(quant_mode=quant_mode,
                            module=model,
                            input_args=(input),
                            device=device,
                            quant_config_file=config_file,
                            target=target)
```

For example/resnet18_quant.py, command line to do hardware-aware calibration:

```
python resnet18_quant.py --quant_mode calib --target DPUCAHX8L_ISA0_SP
```

command line to test hardware-aware quantized model accuracy:

```
python resnet18_quant.py --quant_mode test --target DPUCAHX8L_ISA0_SP
```

command line to deploy quantized model:

```
python resnet18_quant.py --quant_mode test --target DPUCAHX8L_ISA0_SP --
subset_len 1 --batch_size 1 --deploy
```

Module Partial Quantization

You can use module partial quantization if not all the sub-modules in a model need to be quantized. Besides using general vai_q_pytorch APIs, the `QuantStub/DeQuantStub` operator pair can be used to realize it. The following example demonstrates how to quantize `subm0` and `subm2`, but not quantize `subm1`.

```
from pytorch_nndct.nn import QuantStub, DeQuantStub

class WholeModule(torch.nn.module):
    def __init__(self,...):
        self.subm0 = ...
        self.subm1 = ...
        self.subm2 = ...

        # define QuantStub/DeQuantStub submodules
        self.quant = QuantStub()
        self.dequant = DeQuantStub()

    def forward(self, input):
        input = self.quant(input) # begin of part to be quantized
        output0 = self.subm0(input)
        output0 = self.dequant(output0) # end of part to be quantized

        output1 = self.subm1(output0)

        output1 = self.quant(output1) # begin of part to be quantized
        output2 = self.subm2(output1)
        output2 = self.dequant(output2) # end of part to be quantized
```

Register Custom Operation

In order to convert a quantized model to an xmodel，vai_q_pytorch provides a decorator to register an operation or a group of operations as a custom operation which is unknown for XIR.

```
# Decorator API
def register_custom_op(op_type: str, attrs_list: Optional[List[str]] =
None):
  """The decorator is used to register the function as a custom operation.
  Args:
  op_type(str) - the operator type registered into quantizer.
  The type should not conflict with pytorch_nndct

  attrs_list(Optional[List[str]], optional) -
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as padding,
dilation, stride and groups.
  The order of name in attrs_list should be consistent with that of the
arguments list.
  Default: None

  """
```

Perform the following steps:

1. Aggregate some operations as a function. The first argument name of this function should be ctx. The meaning of ctx is the same as that in torch.autograd.Function
2. Decorate this function with the decorator described above.

```
from pytorch_nndct.utils import register_custom_op

@register_custom_op(op_type="MyOp", attrs_list=["scale_1", "scale_2"])
def custom_op(ctx, x: torch.Tensor, y:torch.Tensor, scale_1:float,
scale_2:float) -> torch.Tensor:
  return scale_1 * x + scale_2 * y

class MyModule(torch.nn.Module):
  def __init__(self):
  ...

  def forward(self, x, y):
    return custom_op(x, y, scale_1=2.0, scale_2=1.0)
```

Limitations:

1. Loop operation is not allowed in a custom operation.
2. The number of return values for a custom operation can only be one.

vai_q_pytorch Fast Finetuning

Generally, there is a small accuracy loss after quantization, but for some networks such as MobileNets, the accuracy loss can be large. In this situation, first try fast finetune. If fast finetune still does not yield satisfactory results, QAT can be used to further improve the accuracy of the quantized models.

The AdaQuant algorithm[1] uses a small set of unlabeled data. It not only calibrates the activations but also finetunes the weights. The Vitis AI quantizer implements this algorithm and under the alias "fast finetuning". Though slightly slower, fast finetuning can achieve better performance than quantize calibration. Similar to QAT, each run of fast finetuning may produce a different result.

Fast finetuning does not train the model, and only needs a limited number of iterations. For classification models on the Imagenet dataset, 5120 images are enough in experiment. Data annotation information is not needed in fast finetuning flow, so data without annotation can be input and it still works fine. Fast finetuning only needs some modification based on the model evaluation script. There is no need to set up the optimizer for training. To use fast finetuning, a function for model forwarding iteration is needed and will be called during fast finetuning. Re-calibration with the original inference code is recommended.

You can find a complete example in the open source example.

```
# fast finetune model or load finetuned parameter before test
  if fast_finetune == True:
      ft_loader, _ = load_data(
          subset_len=5120,
          train=False,
          batch_size=batch_size,
          sample_method='random',
          data_dir=args.data_dir,
          model_name=model_name)
      if quant_mode == 'calib':
          quantizer.fast_finetune(evaluate, (quant_model, ft_loader,
loss_fn))
      elif quant_mode == 'test':
          quantizer.load_ft_param()
```

For parameter finetuning and re-calibration of this ResNet18 example, run the following command:

```
python resnet18_quant.py --quant_mode calib --fast_finetune
```

To test the finetuned quantized model accuracy, run the following command:

```
python resnet18_quant.py --quant_mode test --fast_finetune
```

To deploy the finetuned quantized model, run the following command:

```
python resnet18_quant.py --quant_mode test --fast_finetune --subset_len 1 --
batch_size 1 --deploy
```

***Note*:**

1. Itay Hubara et.al., Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming, arXiv:2006.10518, 2020.

Configuration of Quantization Strategy

For multiple quantization strategy configurations, vai_q_pytorch supports quantization configuration file in JSON format.

1. **Usage**

   In order to make the customized configuration take effect, you only need to pass the configuration file to torch_quantizer API.

   ```
   config_file = "./pytorch_quantize_config.json"
   quantizer = torch_quantizer(quant_mode=quant_mode,
                               module=model,
                               input_args=(input),
                               device=device,
                               quant_config_file=config_file)
   ```

There is example code in example/resnet18_quant.py, which could use the file example/pytorch_quantize_config.json as its configuration file. Run command with "--config_file pytorch_quantize_config.json" to quantize model.

```
python resnet18_quant.py --quant_mode calib --config_file
pytorch_quantize_config.json
python resnet18_quant.py --quant_mode test --config_file
pytorch_quantize_config.json
```

In the example configuration file, the model configuration in "overall_quantizer_config" is set to entropy calibration method and per_tensor quantization.

```
"overall_quantize_config": {
  ...
  "method": "entropy",
  ...
  "per_channel": false,
  ...
},
```

And the configuration of weights in "tensor_quantize_config" is maxmin calibration method and per_tensor quantization, which means weights use different quantization method from model configuration.

```
"tensor_quantize_config": {
  ...
  "weights": {
    ...
    "method": "maxmin",
    ...
    "per_channel": false,
    ...
    }
```

Besides, there is one layer quantization configuration in "layer_quantize_config" list. The configuration is based on layer_type, and set torch.nn.Conv2d layer to per_channel quantization.

```
"layer_quantize_config": [
  {
    "layer_type": "torch.nn.Conv2d",
    ...
    "overall_quantize_config": {
      ...
      "per_channel": false,
```

2. **The configurations that can be set in the file:**

   - **convert_relu6_to_relu:** (Global quantizer setting) Whether to convert ReLU6 to ReLU. Options: True or False.
   - **include_cle:** (Global quantizer setting) Whether to use cross layer equalization. Options: True or False.
   - **include_bias_corr:** (Global quantizer setting) Whether to use bias correction. Options: True or False

- **target_device:** (Global quantizer setting) Device to deploy quantized model, options: DPU, CPU, GPU
- **quantizable_data_type:** (Global quantizer setting) tensor types to be quantized in model
- **bit_width:** (Tensor quantization setting)Bit width used in quantization
- **method:** (Tensor quantization setting)Method used to calibrate the quantization scale. Options: Maxmin, Percentile, Entropy, MSE, diffs.
- **round_mode:** (Tensor quantization setting)Rounding method in quantization process. Options: half_even, half_up, half_down, std_round
- **symmetry:** (Tensor quantization setting)Whether to use symmetric quantization. Options: True or False
- **per_channel:** (Tensor quantization setting)Whether to use per_channel quantization. Options: True or False
- **signed:** (Tensor quantization setting)Whether to use signed quantization. Options: True or False
- **narrow_range:** (Tensor quantization setting)Whether to use symmetric integer range for signed quantization. Options: True or False
- **scale_type:** (Tensor quantization setting)Scale type used in quantization process. Options: Float, poweroftwo
- **calib_statistic_method:** (Tensor quantization setting)Method to choose one optimal quantization scale if got different scales using multiple batch data. Options: modal, max, mean, median

3. **Hierarchical Configuration**

   Quantization configuration is in hierarchical structure.

   - If configuration file is not provided in the torch_quantizer API, the default configuration will be used, which is adapted to DPU device and uses poweroftwo quantization method.
   - If configuration file is provided, model configuration, including global quantizer settings and global tensor quantization settings are required.
   - If only model configuration is provided in the configuration file, all tensors in the model will use the same configuration.
   - Layer configuration could be used to set some layers to specific configuration parameters.

   a. **Default Configurations**

      Details of default configuration are shown below.

```
"convert_relu6_to_relu": false,
"include_cle": true,
"include_bias_corr": true,
"target_device": "DPU",
"quantizable_data_type": [
  "input",
  "weights",
```

```
  "bias",
  "activation"],
"bit_width": 8,
"method": "diffs",
"round_mode": "std_round",
"symmetry": true,
"per_channel": false,
"signed": true,
"narrow_range": false,
"scale_type": "poweroftwo",
"calib_statistic_method": "modal"
```

b. **Model Configurations**

In the example configuration file "example/pytorch_quantize_config.json", the global quantizer settings are set under their respective keywords. And global quantization parameters must be set under the "overall_quantize_config" keyword. As shown below.

```
  "convert_relu6_to_relu": false,
  "include_cle": false,
  "keep_first_last_layer_accuracy": false,
  "keep_add_layer_accuracy": false,
  "include_bias_corr": false,
  "target_device": "CPU",
  "quantizable_data_type": [
    "input",
    "weights",
    "bias",
    "activation"],
"overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
}
```

Optionally, the quantization configuration of different tensors in the model can be set separately. And the configurations must be set in "tensor_quantize_config" keyword. And in the example configuration file, just change the quantization method of activation to "mse". The rest of the parameters are used the same as the global parameters.

```
"tensor_quantize_config": {

    "activation": {

        "method": "mse",

    }
}
```

c. **Layer Configurations**

Layer quantization configurations must be added in the "layer_quantize_config" list. And two parameter configuration methods, layer type and layer name, are supported. There are five notes to do layer configuration.

- Each individual layer configuration must be in dictionary format.
- In each layer configuration, the "quantizable_data_type" and "overall_quantize_config" parameter are required. And in "overall_quantize_config" parameter, all quantization parameters for this layer need to be included.
- If the setting is based on layer type, the “layer_name” parameter should be null.
- If the setting is based on layer name, the model needs to run the calibration process firstly, then pick the required layer name from the generated python file in quantized_result directory. Besides, the “layer_type” parameter should be null.
- Same as the model configuration, the quantization configuration of different tensors in the layer can be set separately. And they must be set in "tensor_quantize_config" keywords.

In the example configuration file, there are two layer configurations. One is based on layer type, the other is based on layer name. In the layer configuration based on layer type, torch.nn.Conv2d layer need to set to specific quantization parameters. And the "per_channel" parameter of weight is set to "true", "method" parameter of activation is set to "entropy".

```
{
  "layer_type": "torch.nn.Conv2d",
  "layer_name": null,
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
  },
  "tensor_quantize_config": {
    "weights": {
      "per_channel": true
    },
    "activation": {
      "method": "entropy"
    }
  }
}
```

In the layer configuration based on layer name, the layer named "ResNet::ResNet/Conv2d[conv1]/input.2" need to set to specific quantization parameters. And the round_mode of activation in this layer is set to "half_up".

```
{
  "layer_type": null,
  "layer_name": "ResNet::ResNet/Conv2d[conv1]/input.2",
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
  },
  "tensor_quantize_config": {
    "activation": {
      "round_mode": "half_up"
    }
  }
}
```

The layer name "ResNet::ResNet/Conv2d[conv1]/input.2" is picked from generated file "quantize_result/ResNet.py" of example code "example/resnet18_quant.py".

- Run the example code with command "python resnet18_quant.py --subset_len 100". The quantize_result/ResNet.py file is generated.
- In the file, the name of first convolution layer is "ResNet::ResNet/Conv2d[conv1]/input.2".
- Copy the layer name to quantization configuration file if this layer is set to specific configuration.

```
import torch
import pytorch_nndct as py_nndct
class ResNet(torch.nn.Module):
  def __init__(self):
    super(ResNet, self).__init__()
    self.module_0 = py_nndct.nn.Input() #ResNet::input_0
    self.module_1 = py_nndct.nn.Conv2d(in_channels=3,
out_channels=64, kernel_size=[7, 7], stride=[2, 2], padding=[3, 3],
dilation=[1, 1], groups= 1, bias=True) #ResNet::ResNet/Conv2d[conv1]/
input.2
```

d. **Configuration Restrictions**

Due to the restriction of DPU device design, if quantized models need to be deployed in DPU device, the quantization configuration should meet the restrictions as below:

```
method: diffs or maxmin
round_mode: std_round for weights, bias, and input; half_up for
activation.
symmetry: true
per_channel: false
signed: true
narrow_range: true
scale_type: poweroftwo
calib_statistic_method: modal.
```

And for CPU and GPU device, there is no restriction as DPU device. However, there are some conflicts when using different configurations. For example, if calibration method is 'maxmin', 'percentile', 'mse' or 'entropy', the calibration statistic method 'modal' is not supported. If symmetry mode is asymmetry, the calibration method 'mse' and 'entropy' are not supported. Quantization tool will give error message if there are configuration conflicts.

vai_q_pytorch QAT

Assuming that there is a pre-defined model architecture, use the following steps to do quantization aware training. Take the ResNet18 model from Torchvision as an example. The complete model definition is here.

1. Check if there are non-module operations to be quantized. ResNet18 uses `'+'` to add two tensors. Replace them with `pytorch_nndct.nn.modules.functional.Add`.
2. Check if there are modules to be called multiple times. Usually such modules have no weights; the most common one is the `torch.nn.ReLu` module. Define multiple such modules and then call them separately in a forward pass. The revised definition that meets the requirements is as follows:

```
class BasicBlock(nn.Module):
  expansion = 1

  def __init__(self,
               inplanes,
               planes,
               stride=1,
               downsample=None,
               groups=1,
               base_width=64,
               dilation=1,
               norm_layer=None):
    super(BasicBlock, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    if groups != 1 or base_width != 64:
      raise ValueError('BasicBlock only supports groups=1 and
base_width=64')
    if dilation > 1:
      raise NotImplementedError("Dilation > 1 not supported in
BasicBlock")
```

```
    # Both self.conv1 and self.downsample layers downsample the input
when stride != 1
    self.conv1 = conv3x3(inplanes, planes, stride)
    self.bn1 = norm_layer(planes)
    self.relu1 = nn.ReLU(inplace=True)
    self.conv2 = conv3x3(planes, planes)
    self.bn2 = norm_layer(planes)
    self.downsample = downsample
    self.stride = stride

    # Use a functional module to replace '+'
    self.skip_add = functional.Add()

    # Additional defined module
    self.relu2 = nn.ReLU(inplace=True)

  def forward(self, x):
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu1(out)

    out = self.conv2(out)
    out = self.bn2(out)

    if self.downsample is not None:
      identity = self.downsample(x)

    # Use function module instead of '+'
    # out += identity
    out = self.skip_add(out, identity)
    out = self.relu2(out)

    return out
```

3. Insert `QuantStub` and `DeQuantStub`.

   Use `QuantStub` to quantize the inputs of the network and `DeQuantStub` to de-quantize the outputs of the network. Any sub-network from `QuantStub` to `DeQuantStub` in a forward pass will be quantized. Multiple QuantStub-DeQuantStub pairs are allowed.

```
class ResNet(nn.Module):

  def __init__(self,
               block,
               layers,
               num_classes=1000,
               zero_init_residual=False,
               groups=1,
               width_per_group=64,
               replace_stride_with_dilation=None,
               norm_layer=None):
    super(ResNet, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    self._norm_layer = norm_layer

    self.inplanes = 64
    self.dilation = 1
    if replace_stride_with_dilation is None:
      # each element in the tuple indicates if we should replace
```

```
      # the 2x2 stride with a dilated convolution instead
      replace_stride_with_dilation = [False, False, False]
    if len(replace_stride_with_dilation) != 3:
      raise ValueError(
          "replace_stride_with_dilation should be None "
          "or a 3-element tuple, got
{}".format(replace_stride_with_dilation))
    self.groups = groups
    self.base_width = width_per_group
    self.conv1 = nn.Conv2d(
        3, self.inplanes, kernel_size=7, stride=2, padding=3, bias=False)
    self.bn1 = norm_layer(self.inplanes)
    self.relu = nn.ReLU(inplace=True)
    self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    self.layer1 = self._make_layer(block, 64, layers[0])
    self.layer2 = self._make_layer(
        block, 128, layers[1], stride=2,
dilate=replace_stride_with_dilation[0])
    self.layer3 = self._make_layer(
        block, 256, layers[2], stride=2,
dilate=replace_stride_with_dilation[1])
    self.layer4 = self._make_layer(
        block, 512, layers[3], stride=2,
dilate=replace_stride_with_dilation[2])
    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(512 * block.expansion, num_classes)

    self.quant_stub = nndct_nn.QuantStub()
    self.dequant_stub = nndct_nn.DeQuantStub()

    for m in self.modules():
      if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
      elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

    # Zero-initialize the last BN in each residual branch,
    # so that the residual branch starts with zeros, and each residual
block behaves like an identity.
    # This improves the model by 0.2~0.3% according to https://
arxiv.org/abs/1706.02677
    if zero_init_residual:
      for m in self.modules():
        if isinstance(m, Bottleneck):
          nn.init.constant_(m.bn3.weight, 0)
        elif isinstance(m, BasicBlock):
          nn.init.constant_(m.bn2.weight, 0)

  def forward(self, x):
    x = self.quant_stub(x)

    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
```

```
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    x = self.dequant_stub(x)
    return x
```

4. Use QAT APIs to create the quantizer and train the model.

```
def _resnet(arch, block, layers, pretrained, progress, **kwargs):
  model = ResNet(block, layers, **kwargs)
  if pretrained:
    #state_dict = load_state_dict_from_url(model_urls[arch],
progress=progress)
    state_dict = torch.load(model_urls[arch])
    model.load_state_dict(state_dict)
  return model

def resnet18(pretrained=False, progress=True, **kwargs):
  r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://
arxiv.org/pdf/1512.03385.pdf>'_

    Args:
        pretrained (bool): If True, returns a model pre-trained on
ImageNet
        progress (bool): If True, displays a progress bar of the
download to stderr
    """
  return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained,
progress,
                 **kwargs)

model = resnet18(pretrained=True)

# Generate dummy inputs.
input = torch.randn([batch_size, 3, 224, 224], dtype=torch.float32)

# Create a quantizer
from pytorch_nndct import QatProcessor
qat_processor = QatProcessor(model, inputs, bitwidth=8)
quantized_model = qat_processor.trainable_model()optimizer =
torch.optim.Adam(
        quantized_model.parameters(),
        lr,
        weight_decay=weight_decay)

# Use the optimizer to train the model, just like a normal float model.
…
```

5. Get the deployable model and test it.

   Convert the quantized model to a deployable model after training is complete. The accuracy of the deployable model may differ slightly from the accuracy of the quantized model.

```
output_dir = 'qat_result'
deployable_model = qat_processor.to_deployable(quantized_model,
output_dir)
validate(val_loader, deployable_model, criterion, gpu)
```

6. Export xmodel from the deployable model.

`batch size=1` is a must for the compilation of xmodel.

```
# Use cpu mode to export xmodel.
deployable_model.cpu()
val_subset = torch.utils.data.Subset(val_dataset, list(range(1)))
subset_loader = torch.utils.data.DataLoader(
    val_subset,
    batch_size=1,
    shuffle=False,
    num_workers=8,
    pin_memory=True)
# Must forward deployable model at least 1 iteration with batch_size=1
for images, _ in subset_loader:
  deployable_model(images)
qat_processor.export_xmodel(output_dir)
```

vai_q_pytorch QAT Requirements

Generally, there is a small accuracy loss after quantization, but for some networks such as MobileNets, the accuracy loss can be large. In this situation, first try fast finetune. If fast finetune does not yield satisfactory results, QAT can be used to further improve the accuracy of the quantized models.

The QAT APIs have some requirements for the model to be trained.

1. All operations to be quantized must be instances of the `torch.nn.Module` object, rather than Torch functions or Python operators. For example, it is common to use `'+'` to add two tensors in PyTorch. However, this is not supported in QAT. Thus, replace `'+'` with `pytorch_nndct.nn.modules.functional.Add`. Operations that need replacement are listed in the following table.

Table 19: **Operation-Replacement Mapping**

Operation	Replacement
`+`	`pytorch_nndct.nn.modules.functional.Add`
`-`	`pytorch_nndct.nn.modules.functional.Sub`
`torch.add`	`pytorch_nndct.nn.modules.functional.Add`
`torch.sub`	`pytorch_nndct.nn.modules.functional.Sub`

IMPORTANT! *A module to be quantized cannot be called multiple times in the forward pass.*

2. Use `pytorch_nndct.nn.QuantStub` and `pytorch_nndct.nn.DeQuantStub` at the beginning and end of the network to be quantized. The network can be the complete network or a partial sub-network.

Guidelines for Better Training Results

The following are some tips for getting better training results:

- Load the pre-trained floating-point weights as initial values to start the quantization aware training if possible. It is possible to train from scratch with random initial values, but this will make training more difficult and long.
- If pre-trained floating-point weights are loaded, then different initial learning rates and learning rate decrease strategies need to be used for the network parameters and quantizer parameters, respectively. In general, the learning rate of network parameters needs to be set small, while the learning rate of quantizer parameters needs to be larger.

```
model = qat_processor.trainable_model()
param_groups = [{
    'params': model.quantizer_parameters(),
    'lr': 1e-2,
    'name': 'quantizer'
}, {
    'params': model.non_quantizer_parameters(),
    'lr': 1e-5,
    'name': 'weight'
}]
optimizer = torch.optim.Adam(param_groups)
```

- For the choice of optimizer, avoid using torch.optim.SGD, as this optimizer may prevent the training from converging. Xilinx recommends using torch.optim.Adam or torch.optim.RMSprop and their variants.

vai_q_pytorch Usage

This section introduces the usage of execution tools and APIs to implement quantization and generate a model to be deployed on the target hardware. The APIs in the module `pytorch_binding/pytorch_nndct/apis/quant_api.py` are as follows:

class torch_quantizer()

Class `torch_quantizer` creates a quantizer object.

```
class torch_quantizer():
  def __init__(self,
                 quant_mode: str, # ['calib', 'test']
                 module: torch.nn.Module,
                 input_args: Union[torch.Tensor, Sequence[Any]] = None,
                 state_dict_file: Optional[str] = None,
                 output_dir: str = "quantize_result",
                 bitwidth: int = 8,
                 device: torch.device = torch.device("cuda"),
                 quant_config_file: Optional[str] = None,
                 target: Optional[str]=None):
```

Arguments

- **Quant_mode:** An integer that indicates which quantization mode the process is using. "calib" for calibration of quantization, and "test" for evaluation of quantized model.
- **Module:** Float module to be quantized.
- **Input_args:** Input tensor with the same shape as real input of float module to be quantized, but the values can be random numbers.
- **State_dict_file:** Float module pretrained parameters file. If float module has read parameters in, the parameter is not needed to be set.
- **Output_dir:** Directory for quantization result and intermediate files. Default is "quantize_result".
- **Bitwidth:** Global quantization bit width. Default is 8.
- **Device:** Run model on GPU or CPU.
- **Quant_config_file:** Json file path for quantization strategy configuration.
- **Target:** If target device is specified, the hardware-aware quantization is on. Default is None.

def export_quant_config(self)

This function exports information related to the quantization steps.

```
def export_quant_config(self):
```

def export_xmodel(self, output_dir, deploy_check)

This function exports the xmodel and dumps the output data of the operators for detailed data comparison.

```
def export_xmodel(self, output_dir, deploy_check):
```

Arguments

- **Output_dir:** Directory for quantization result and intermediate files. Default is “quantize_result.”
- **Deploy_check:** Flags to control dump of data for detailed data comparison. Default is FALSE. If it is set to TRUE, binary format data is dumped in the `output_dir/deploy_check_data_int/` location.

def export_onnx_model(self, output_dir, verbose)

The function is to export onnx format quantized model

```
def export_onnx_model(self, output_dir, verbose):
```

Arguments

- **Output_dir:** Directory for quantization result and intermediate files. The default value is “quantize_result”
- **Verbose:** Flag to control the display of verbose log.

def export_torch_script(self, output_dir, verbose)

The function is to export torch script format quantized model

```
def export_torch_script(self, output_dir, verbose):
```

Arguments

- **Output_dir:** Directory for quantization result and intermediate files. The default value is “quantize_result”
- **Verbose:** Flag to control the display of verbose log.

Class Inspector

Class Inspector creates a inspector object as follows:

```
class Inspector():
def __init__(self, name_or_fingerprint: str):
```

Arguments

- **name_or_fingerprint:** Specify the hardware target name or fingerprint

def inspect(...)

The function is to inspect a float model

```
def inspect(self,
            module: torch.nn.Module,
            input_args: Union[torch.Tensor, Tuple[Any]],
            device: torch.device = torch.device("cuda"),
            output_dir: str = "quantize_result",
            verbose_level: int = 1,
            image_format: Optional[str] = None):
```

Arguments

- **module:** Float module to be depolyed
- **input_args:** Input tensor with the same shape as real input of float module, but the value can be random number
- **device:** Trace model on GPU or CPU
- **output_dir:** Directory for inspection results
- **verbose_level:** Control the level of detail of the inspection results displayed on the screen. The default value is 1.0: turn off printing inspection results1: print summary report of operations assigned to CPU2: print summary report of device allocation of all operations
- **image_format:** Export visualized inspection result. Supports SVG and PNG image formats.

vai_q_pytorch message

In this part, some important messages are listed and can be searched by message ID. For every message, it can help users to identify the issues among their model deployment, and gives possible solution for the issue.

VAIQ_WARN

Vai_q_pytorch prints warning message on screen when there is issue may causing the quantization result has problem or incomplete (check according to the message text), but the process can be performed to its end, the format of this kind of message is "[VAIQ_WARN][MESSAGE_ID]: message text"

List important warning messages in the following table:

Table 20: **Vai_q_pytorch warning message table**

Message ID	Description
QUANTIZER_TORCH_BATCHNORM_AFFINE	BatchNorm OP attribute affine=False has been replaced by affine=True when parsing the model.

Table 20: **Vai_q_pytorch warning message table** *(cont'd)*

Message ID	Description
QUANTIZER_TORCH_BITWIDTH_MISMATCH	Bit width setting in configuration file is conflict with that from torch_quantizer API, will use that in configuration file.
QUANTIZER_TORCH_CONVERT_XMODEL	Convert to xmodel failed. Check message text to locate the reason.
QUANTIZER_TORCH_CUDA_UNAVAILABLE	CUDA (HIP) is not available, change device to CPU
QUANTIZER_TORCH_DATA_PARALLEL	Data parallel is not supported. The wrapper 'torch.nn.DataParallel' has been removed in vai_q_pytorch.
QUANTIZER_TORCH_DEPLOY_MODEL	Only quantization aware training process has deployable model.
QUANTIZER_TORCH_DEVICE_MISMATCH	The Device of input arguments mismatch with quantizer device type.
QUANTIZER_TORCH_EXPORT_XMODEL	Failed to generate xmodel due to some reasons. Refer to the message text.
QUANTIZER_TORCH_FINETUNE_IGNORED	Fast fine-tune function will be ignored in test mode!
QUANTIZER_TORCH_FLOAT_OP	vai_q_pytorch recognize the list OP as a float operator by default.
QUANTIZER_TORCH_INSPECTOR_PATTERN	The OP may be fused by compiler and will be assigned to DPU.
QUANTIZER_TORCH_LEAKYRELU	Force to change negative_slope of LeakyReLU to 0.1015625 because DPU only supports this value. It is recommended to change all negative_slope of LeakyReLU to 0.1015625 and re-train the float model for better deployed model accuracy.
QUANTIZER_TORCH_MATPLOTLIB	matplotlib is needed for visualization but not found. It needs to be installed.
QUANTIZER_TORCH_MEMORY_SHORTAGE	There is no enough memory for fast fine-tune and this process will be ignored!. Try to use a smaller calibration dataset.
QUANTIZER_TORCH_NO_XIR	Can't find XIR package in environment. It needs to be installed.
QUANTIZER_TORCH_REPLACE_RELU6	ReLU6 has been replaced by ReLU.
QUANTIZER_TORCH_REPLACE_SIGMOID	Sigmoid has been replaced by Hardsigmoid.
QUANTIZER_TORCH_REPLACE_SILU	SiLU has been replaced by Hardswish.
QUANTIZER_TORCH_SHIFT_CHECK	Quantization scale is too large or too small.
QUANTIZER_TORCH_TENSOR_NOT_QUANTIZED	Some tensors are not quantized, please check their particularity.
QUANTIZER_TORCH_TENSOR_TYPE_NOT_QUANTIZABLE	The tensor type of the node cannot be quantized. Only support float32/double/float16 quantization.
QUANTIZER_TORCH_TENSOR_VALUE_INVALID	The tensor has "inf" or "nan" value. Quantization for this tensor is ignored.
QUANTIZER_TORCH_TORCH_VERSION	Only support exporting torch script with pytorch 1.10 and later version.
QUANTIZER_TORCH_XIR_MISMATCH	XIR version does not match current vai_q_pytorch.
QUANTIZER_TORCH_XMODEL_DEVICE	Not support to dump xmodel when target device is not DPU.
QUANTIZER_TORCH_REUSED_MODULE	Reused module may lead to low accuracy of QAT, make sure this is what you expect. Refer to the message text to locate the module with issue.
QUANTIZER_TORCH_DEPRECATED_ARGUMENT	The argument "device" is no longer needed. Device information is get from the model directly.
QUANTIZER_TORCH_SCALE_VALUE	Exported scale values are not trained.

VAIQ_ERROR

Vai_q_pytorch prints error message on screen when there is issue causing the process cannot be performed anymore (check and solve the problem according to the message text), the format of this kind of message is "[VAIQ_ERROR][MESSAGE_ID]: message text"

List important error messages in the following table:

Table 21: **Vai_q_pytorch error message table**

Message ID	Description
QUANTIZER_TORCH_BIAS_CORRECTION	Bias correction file in quantization result directory does not match current model.
QUANTIZER_TORCH_CALIB_RESULT_MISMATCH	Node name mismatch is found when loading quantization steps of tensors. Please make sure vai_q_pytorch version and pytorch version for test mode are the same as those in calibration (or QAT training) mode.
QUANTIZER_TORCH_EXPORT_ONNX	The quantized module, which is based pytorch traced model, can not be exported to ONNX due to pytorch internal failure. The pytorch internal failure reason is listed in message text. May needs adjust float model code.
QUANTIZER_TORCH_EXPORT_XMODEL	Fail to convert graph to xmodel. Needs check the reasons in message text.
QUANTIZER_TORCH_FAST_FINETINE	Fast fine-tuned parameter file does not exist. Call load_ft_param in model code to load them.
QUANTIZER_TORCH_FIX_INPUT_TYPE	Data type or value is illegal in arguments of quantization OP when exporting ONNX format model.
QUANTIZER_TORCH_ILLEGAL_BITWIDTH	The configuration of tensors quantization is illegal. It should be integer, and in range given in message text.
QUANTIZER_TORCH_IMPORT_KERNEL	Importing vai_q_pytorch library file error. Check pytorch version matching vai_q_pytorch version (pytorch_nndct.__version__) or not.
QUANTIZER_TORCH_NO_CALIB_RESULT	Quantization result file does not exist. Please check calibration is done or not.
QUANTIZER_TORCH_NO_CALIBRATION	Quantization calibration is not performed completely, check if module FORWARD function is called! FORWARD function of torch_quantizer.quant_model needs to be called in user code explicitly. Please refer to the example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_NO_FORWARD	torch_quantizer.quant_model FORWARD function must be called before exporting quantization result. Please refer to example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_OP_REGIST	The type of OP can't be registered multiple times.
QUANTIZER_TORCH_PYTORCH_TRACE	Failed to get pytorch traced graph from model and input arguments. The pytorch internal failure reason is reported in message text. May needs adjust float model code.
QUANTIZER_TORCH_QUANT_CONFIG	Quantization configuration items are illegal. Refer to the message text.
QUANTIZER_TORCH_SHAPE_MISMATCH	Tensors shape are mismatch. Refer to the message text.
QUANTIZER_TORCH_TORCH_VERSION	Pytorch version is not supported for the function or does not match vai_q_pytorch version (pytorch_nndct.__version__). Refer to the message text.
QUANTIZER_TORCH_XMODEL_BATCHSIZE	Batch size must be 1 when exporting xmodel.
QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT	Inspector only support dump svg or png format.

Table 21: **Vai_q_pytorch error message table** *(cont'd)*

Message ID	Description
QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT	Inspector no longer support fingerprint. Please provide architecture name instead.
QUANTIZER_TORCH_UNSUPPORTED_OPS	The quantization of the op is not supported.
QUANTIZER_TORCH_TRACED_NOT_SUPPORT	The model produced by 'torch.jit.script' is not supported in vai_q_pytorch.
QUANTIZER_TORCH_NO_SCRIPT_MODEL	vai_q_pytorch does not find any script model.
QUANTIZER_TORCH_REUSED_MODULE	The quantized module has been called multiple times in forward pass. If you want to share quantized parameters in multiple calls, call trainable_model with "allow_reused_module=True"
QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED	torch.nn.DataParallel object is not allowed.
QUANTIZER_TORCH_INPUT_NOT_QUANTIZED	Input is not quantized. Please use QuantStub/DeQuantStub to define quantization scope.
QUANTIZER_TORCH_NOT_A_MODULE	Quantized operation must be instance of "torch.nn.Module", please replace the function by a "torch.nn.Module" object. Original source range is indicated in message text.
QUANTIZER_TORCH_QAT_PROCESS_ERROR	Must call "trainable_model" first before getting deployable model.
QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR	The given trained model has BN fused to CONV and cannot be converted to a deployable model. Make sure model.fuse_conv_bn() is not called.
QUANTIZER_TORCH_XMODEL_DEVICE	Xmodel can only be exported in CPU mode, use deployable_model(src_dir, used_for_xmodel=True) to get a CPU model.

Chapter 4

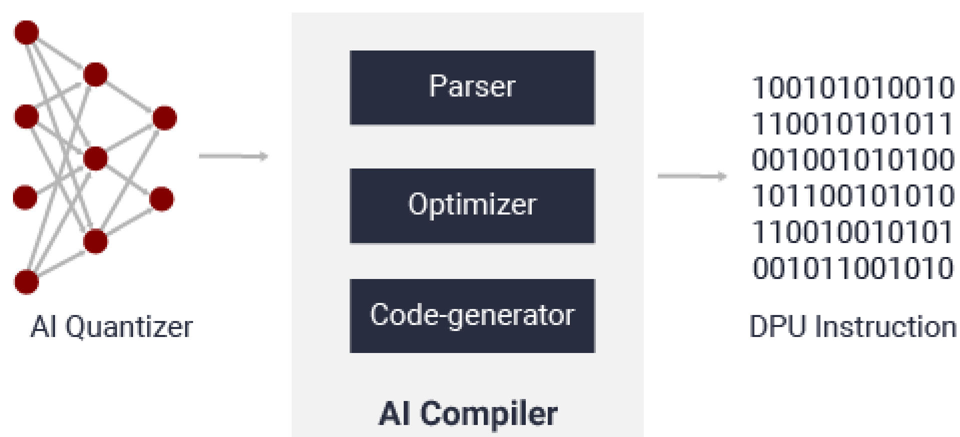
Compiling the Model

Vitis AI Compiler

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.

Figure 21: **Vitis AI Compiler Framework**



The Vitis AI Compiler generates the compiled model based on the DPU microarchitecture. Vitis AI supports several DPUs for different platforms and applications.

Table 22: **DPUs on Different Hardware Platforms**

DPU Name	Hardware platform
DPUCZDX8G	Zynq® UltraScale+™ MPSoC
DPUCVDX8G	Versal® ACAP VCK190 evaluation board, Versal AI Core Series
DPUCVDX8H	Versal ACAP VCK5000 evaluation kit
DPUCV2DX8G	Versal® ACAP VEK280 evaluation board, Versal AI Core Series

Compiling with an XIR-based Toolchain

Xilinx Intermediate Representation (XIR) is a graph-based intermediate representation of the AI algorithms which is designed for compilation and efficient deployment of the DPU on the FPGA platform. If you are an advanced user, you can apply whole application acceleration to allow the FPGA to be used to its maximum potential by extending the XIR to support customized IPs in the Vitis AI flow. It is the current foundation for the Vitis AI quantizer, compiler, runtime, and other tools.

XIR

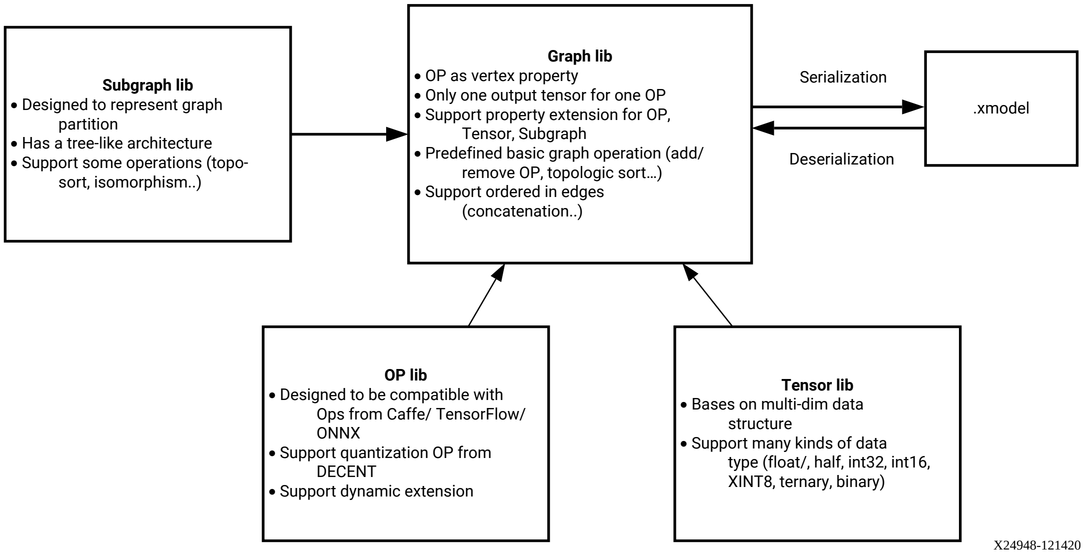
XIR includes the Op, Tensor, Graph, and Subgraph libraries, which provide a clear and flexible representation of the computational graph. XIR has in-memory format and file format for different usage. The in-memory format XIR is a graph object and the file format is an xmodel. A graph object can be serialized to an XMODEL while an XMODEL can be deserialized to a graph object.

In the Op library, there is a well-defined set of operators to cover the popular deep learning frameworks, e.g., TensorFlow, PyTorch and Caffe[1], and all of the built-in DPU operators. This enhances the expression ability and achieves one of the core goals, which is eliminating the difference between these frameworks and providing a unified representation for users and developers.

XIR also provides Python APIs named PyXIR, which enables Python users to fully access the XIR in a Python environment, e.g., co-develop and integrate users' Python projects with the current XIR-based tools without having to perform a huge amount of work to fix the gap between different languages.

[1] Caffe is deprecated from VAI 2.5 release. For more information, see Vitis AI 2.0 User Guide.

Figure 22: **XIR Based Flow**



xir::Graph

Graph is the core component of the XIR. It obtains serveral significant APIs, e.g., the `xir::Graph::serialize`, `xir::Graph::deserialize` and `xir::Graph::topological_sort`.

The Graph is like a container, which maintains the Op as its vertex, and uses the producer-consumer relation as the edge.

xir::Op

Op in XIR is the instance of the operator definition either in XIR or extended from XIR. All Op instances can only be created or added by the Graph according to the predefined built-in/ extended op definition library. The Op definition mainly includes the input arguments and intrinsic attributes.

Besides the intrinsic predefined attributes, an Op instance is also able to carry more extrinsic attributes by applying `xir::Op::set_attr` API. Each Op instance can only obtain one output tensor, but more than one fanout ops.

xir::Tensor

Tensor is another important class in XIR. Unlike other frameworks' tensor definition, XIR's Tensor is only a description of the data block it representes. The real data block is excluded from the Tensor.

The key attributes for Tensor is the data type and shape.

xir::Subgraph

XIR's Subgraph is a tree-like hierarchy, which divides a set of ops into several non-overlapping sets. The Graph's entire op set can be seen as the root. The Subgraph can be nested but it must be non-overlapping. The nested insiders must be the children of the outer one.

Compiling for DPU

The XIR-based compiler takes the quantized Caffe[2], TensorFlow, TensorFlow2.x or PyTorch model as the input. First, it transforms the input models into the XIR format as the foundation for the following processes. Most of the variations among different frameworks are eliminated and transferred to a unified representation in XIR. Then, it applies various optimizations to the graph and breaks up the graph into several subgraphs on the basis of whether the operation can be executed on the DPU. Architecture-aware optimizations are applied for each subgraph, as required. For the DPU subgraph, the compiler generates the instruction stream and attaches to it. Finally, the optimized graph with the necessary information and instructions for VART is serialized into a compiled xmodel file.

The XIR-based compiler can support the DPUCZDX8G series on the Edge Zynq UltraScale+ MPSoC platforms, DPUCADF8H on the Alveo platform, DPUCAHX8H on the Alveo HBM platform optimized for high-throughput applications, DPUCVDX8G and DPUCV2DX8G on the Versal Edge platform, and DPUCVDX8H on the Versal Cloud platform. You can find the`arch.json` files for these platforms in `/opt/vitis_ai/compiler/arch`.

Steps to compile Caffe or TensorFlow models with VAI_C are the same as for the previous DPUs. It is assumed that you have successfully installed the Vitis AI package including VAI_C and compressed your model with the vai_quantizer.

TensorFlow

For TensorFlow, vai_q_tensorflow generates a pb file (`quantize_eval_model.pb`). There are two pb files generated by vai_q_tensorflow. The `quantize_eval_model.pb` file is the input file for the XIR-based compiler. The compilation command is as follows.

```
vai_c_tensorflow -f /PATH/TO/quantize_eval_model.pb -a /PATH/TO/arch.json -
o /OUTPUTPATH -n netname
```

The outputs is the same as the output for Caffe.

Sometimes, the TensorFlow model does not contain input tensor shape information because it might cause the compilation to fail. You can specify the input tensor shape with an extra option like `--options '{"input_shape": "1,224,224,3"}'`.

[2] Caffe will be deprecated from the Vitis AI 2.5 release. For information on quantized Caffe, see Vitis AI 2.0 User Guide.

TensorFlow 2.x

For TensorFlow 2.x, the quantizer generates the quantized model in the hdf5 format.

```
vai_c_tensorflow2 -m /PATH/TO/quantized.h5 -a /PATH/TO/arch.json -o /
OUTPUTPATH -n netname
```

Currently, vai_c_tensorflow2 only supports Keras functional APIs.

PyTorch

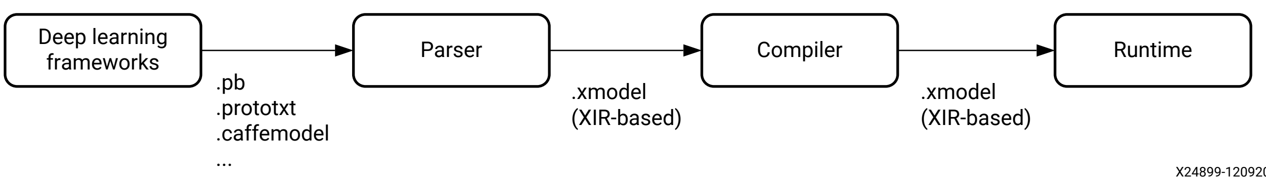
For PyTorch, the quantizer NNDCT outputs the quantized model in the XIR format directly. Use vai_c_xir to compile it.

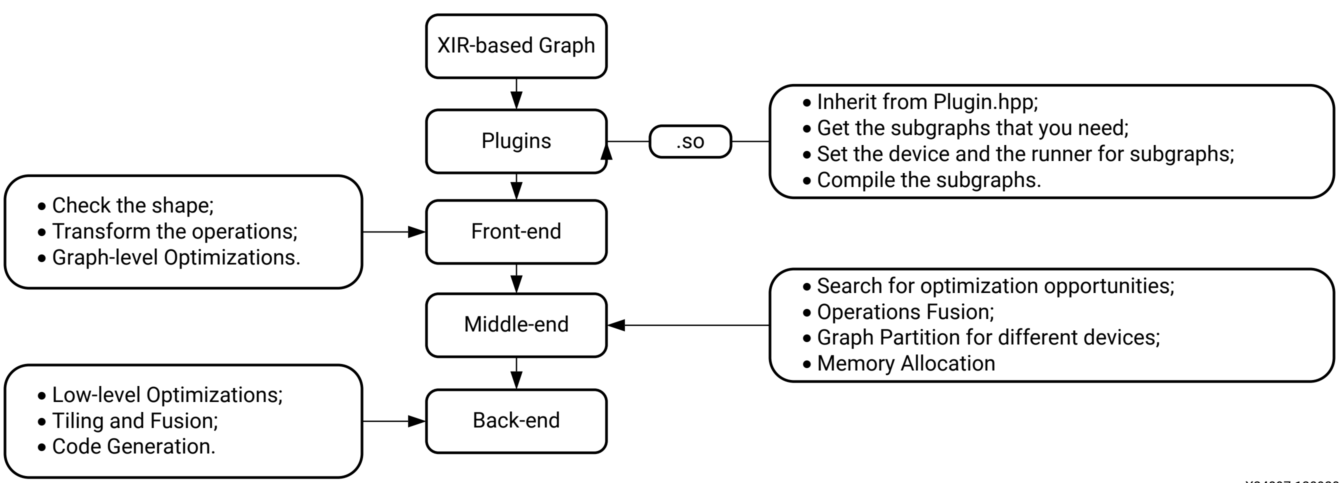
```
vai_c_xir -x /PATH/TO/quantized.xmodel -a /PATH/TO/arch.json -o /OUTPUTPATH
-n netname
```

Compiling for Customized Accelerator

The XIR-based compiler works in the context of a framework-independent XIR graph generated from deep learning frameworks. The parser removes the framework-specific attributes in the CNN models and transforms the models into XIR-based computing graphs. The compiler divides the computing graph into different subgraphs, leverages heterogeneous optimizations, and generates corresponding optimized machine codes for subgraphs.

Figure 23: **Compilation Flow**





When the model contains operations that the DPU cannot support, some subgraphs are created and mapped to the CPU. The FPGA is so powerful that you can create a specific IP to accelerate those operations for improved end-to-end performance. To enable customized accelerating IPs with an XIR-based toolchain, leverage a pipeline named plugin to extend the XIR and compiler.

In `Plugin.hpp`, the interface class Plugin is declared. Plugins are executed sequentially before the compiler starts to compile the graph for the DPU. At first, a child subgraph is created for each operator and the plugin picks the operators that it can accelerate. It merges them into larger subgraphs, maps them to the customized IP, and attaches necessary information for runtime (VART::Runner) such as the instructions on the subgraphs.

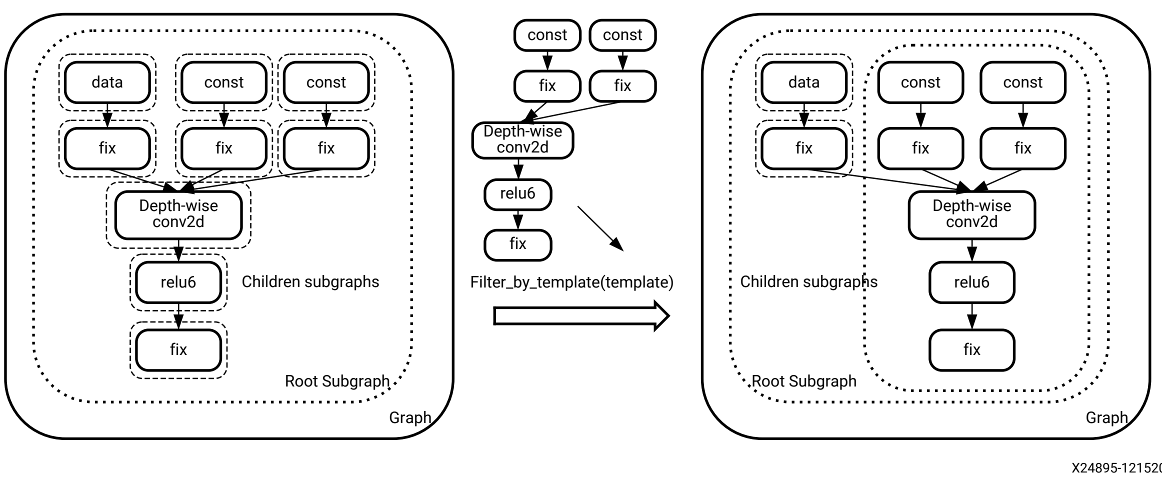
Implementing a Plugin

1. Implement `Plugin::partition()`

   In `std::set<xir::Subgraph*> partition(xir::Graph* graph)`, pick the desired operations and merge them into device level subgraphs using the following helper functions.

   - `xir::Subgraph* filter_by_name(xir::Graph* graph, const std::string& name)` returns the subgraph with a specific name
   - `std::set<xir::Subgraph*> filter_by_type(xir::Graph* graph, const std::string& type)` returns subgraphs with a specific type.
   - `std::set<xir::Subgraph*> filter_by_template(xir::Graph* graph, xir::GraphTemplate* temp)` returns subgraphs with a specific structure.

   Figure 24: **Filter by Templates**

   - `std::set<xir::Subgraph*> filter(xir::Graph* graph, std::function<std::set<xir::Subgraph*>(std::set<xir::Subgraph*>)> func)` allows you to filter the subgraphs by customized function. This method helps you to find all uncompiled subgraphs.

To merge the child subgraphs, use the `merge_subgraph()` helper function. However, this function can only merge subgraphs at the same level. If the subgraph list can not be merged into one subgraph, the helper function will merge them as far as possible.

2. Specify the name, device, and runner for the subgraphs you picked in the `Plugin::partition()` function.
3. Implement `Plugin::compile(xir::Subgraph*)`. This function is called for all the subgraphs returned by the `partition()` function. You can attach information on subgraphs for runtime.

Building the Plugin

Create an extern `get_plugin()` function and build the implementations into a shared library.

```
extern "C" plugin* get_plugin() { return new YOURPLUGIN(); }
```

Using the Plugin

Use `--options '{"plugins": "plugin0,plugin1"}'` in the vai_c command line option to pass your plugin library to compiler. When executing your plugin, the compiler opens the library and makes an instance of your plugin by loading your extern function named 'get_plugin'. If more than one plugin is specified, they are executed sequentially in the order defined by the command line option. Compilation for DPU and CPU are executed after all the plugins have been implemented.

Samples

Check https://github.com/Xilinx/Vitis-AI/tree/v3.0/src/vai_runtime/plugin-samples for samples.

Supported Operators and DPU Limitations

Xilinx is continuously improving the DPU IP and the compiler to support more operators with better performance. The following table lists some typical operations and the configurations such as kernel size, stride, etc. that the DPU can support. If the operation configurations exceed these limitations, the operator will be assigned to the CPU. Additionally, the operators that the DPU can support are dependent on the DPU types, ISA versions, and configurations.

You can configure the DPUs to suit your requirements. You can choose engines, adjust intrinsic parameters, and create your own DPU IP with TRD projects but this means that the limitations can be very different between configurations. Either use the following product guides for information on configuration or compile the model with your own DPU configuration. The compiler tells you which operators can be assigned to the CPU. The table shows a specific configuration of each DPU architecture.

- *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide*(PG338)
- *DPUCAHX8H for Convolutional Neural Networks Product Guide* (PG367)

- *DPUCVDX8G for Versal ACAPs Product Guide* (PG389)
- *DPUCVDX8H for Convolutional Neural Networks v1.0 LogiCORE IP Product Guide* (PG403)
- *DPUCV2DX8G for Versal ACAPs Product Guide* (PG425)

The following operators are primitively defined in different deep learning frameworks. The compiler can automatically parse these operators, transform them into the XIR format, and distribute them to DPU or CPU. These operators are partially supported by the tools, and they are listed here for your reference. According to the limitations, you can use Inspecting the Float Model to automatically check operators in your models.

Currently Supported Operators

Table 23: **Currently Supported Operators**

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
conv2d	Kernel size	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16] w * h * ceil(input_channel/ 2048) <= 64	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16] 256 * h * w <= 13760
	Strides	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]
	Dilation	dilation * input_channel <= 256 * channel_parallel						
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]						
		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]						
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth						kernel_w * kernel_h * ceil(input_channel / channel_parallel) * ceil(channel_parallel / 4) + 4 <= bank_depth
		input_channel <= 256 * channel_parallel		input_channel <= 256 * channel_parallel				input_channel <= 256 * channel_parallel
	Out Size	output_channel <= 256 * channel_parallel						
	Activation	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
	Group* (Caffe)	group==1						

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	**Parameters**	**DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)**	**DPUCAHX8L_ISA0 (U50, U50LV, U280)**	**DPUCVDX8G_ISA3_C32B3 [4](VCK190)**	**DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)**	**DPUCADF8H_ISA0 (U200, U250)**	**DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)**	**DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)**
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-conv2d	Kernel size	w, h: [1, 256]	w, h: [3]	w, h: [1, 256]	w, h: {1, 2, 3, 5, 7}	Not supported	w, h: [1, 8]	w, h: [1, 256] h * w <= 431
	Strides	w, h: [1, 256]	w, h: [1, 2]	w, h: [1, 256]	w, h: [1, 4]		w, h : [1, 4]	w, h: [1, 256]
	dilation	dilation * input_channel <= 256 * channel_parallel					dilation * input_channel <= 256 * channel_parallel	
	Paddings	pad_left, pad_right: [0, min((kernel_w - 1), 15) * dilation_w]	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]	pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w]	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]		pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]	pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w]
		pad_top, pad_bottom: [0, min((kernel_h - 1), 15) * dilation_h]	pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h]	pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h]
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth					kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth	(6 * stride_w + kernel_w) * kernel_h + 4 <= 512
	Out Size	output_channel <= 256 * channel_parallel					output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6, LeakyReLU[6], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, ReLU6, LeakyReLU[7], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6		ReLU, ReLU6	ReLU, ReLU6, LeakyReLU, Hard-Swish, Hard-Sigmoid
	Group* (Caffe)	group==input_channel					group==input_channel	

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]						
	Strides							
	Paddings	pad_left, pad_right: [0, kernel_w-1]						
		pad_top, pad_bottom: [0, kernel_h-1]						
	Out Size	output_channel <= 256 * channel_parallel						
	Activation	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
depthwise-transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 256]	kernel_w/stride_w, kernel_h/stride_h: [3]	kernel_w/stride_w, kernel_h/stride_h: [1, 256]	kernel_w/stride_w, kernel_h/stride_h: {1,2, 3, 5, 7}	Not supported	kernel_w/stride_w, kernel_h/stride_h: [1, 8]	kernel_w/stride_w, kernel_h/stride_h: [1, 256]
	Strides							
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1),15)]	pad_left, pad_right: [1, kernel_w-1]		pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1),15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]		pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Out Size	output_channel <= 256 * channel_parallel					output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6, LeakyReLU[6], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, ReLU6, LeakyReLU[7], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6		ReLU, ReLU6	ReLU, ReLU6, LeakyReLU, Hard-Swish, Hard-Sigmoid

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
max-pooling	Kernel size	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: {2, 3, 5, 7, 8}	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: [1, 8]	w, h: [1, 16]	w, h: [1, 128]	w, h: [1, 256] h * w <= bank_depth
	Strides	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 128]	w, h: [1, 256]
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]			pad_left, pad_right: [0, min((kernel_w-1), 15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]			pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Activation	ReLU, ReLU6	not supported	ReLU, ReLU6	not supported	ReLU	not supported	ReLU, ReLU6
average-pooling	Kernel size	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: {2, 3, 5, 7, 8} w==h	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: [1, 8] w==h	w, h: [1, 16]	w, h: [1, 128] w==h	w, h: [1, 256] h * w <= bank_depth
	Strides	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 128]	w, h: [1, 256]
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]			pad_left, pad_right: [0, min((kernel_w-1), 15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]			pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Activation	ReLU, ReLU6	not supported	ReLU, ReLU6	not supported	ReLU	not supported	ReLU, ReLU6

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
eltwise	type	sum, prod	sum	sum, prod	sum	sum	sum, prod	sum, prod
	Input Channel	input_channel <= 256 * channel_parallel						
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU, Hard-Sigmoid	ReLU
concat	Network-specific limitation, which relates to the size of feature maps, quantization results and compiler optimizations.							
reorg	Strides	reverse==false : stride ^ 2 * input_channel <= 256 * channel_parallel reverse==true : input_channel <= 256 * channel_parallel						
pad	In Size	input_channel <= 256 * channel_parallel						
	Mode	"SYMMETRIC" ("CONSTANT" pad(value=0) would be fused into adjacent operators during compiler optimization process)					"SYMMETRIC", "CONSTANT" (all padding value are identical)	"SYMMETRIC" ("CONSTANT" pad(value=0) would be fused into adjacent operators during compiler optimization process)
global pooling	Global pooling will be processed as general pooling with kernel size equal to input tensor size.							
InnerProduct, Fully Connected, Matmul	These ops will be transformed into conv2d op							

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
resize	scale	NEAREST: ceil(scale/bank_num) * scale * ceil(input_channel/channel_parallel) <= bank_depth BILINEAR: only for 4-D feature maps. This would be transformed into pad and depthwise-transposed-conv2d. TRILINEAR: only for 5-D feature maps. This would be transformed into pad and transposed-conv3d.						
	mode	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR, TRILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
conv3d	kernel size	Not supported	Not supported	w, h, d: [1, 16] w * h * ceil(ceil(input_channel/16) * 16 * d / 2048) <= 64	Not supported	Not supported	Not supported	Not supported
	strides			w, h, d: [1, 8]				
	paddings			pad_left, pad_right: [0, kernel_w-1] pad_top, pad_bottom: [0, kernel_h-1] pad_front, pad_back: [0, kernel_d-1]				
	In size			kernel_w * kernel_h * kernel_d * ceil(input_channel/channel_parallel) <= bank_depth, input_channel <= 256 * channel_parallel				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid				

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-conv3d	kernel size	Not supported	Not supported	w, h: [1, 256] d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides			w, h: [1, 256] d=1				
	paddings			pad_left, pad_right: [0, min((kernel_w-1), 15)] pad_top, pad_bottom: [0, min((kernel_h-1), 15)] pad_front, pad_back: [0, min((kernel_d-1), 15)]				
	In size			kernel_w * kernel_h * kernel_d * ceil(input_channel/ channel_parallel) <= bank_depth				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, ReLU6				

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
transposed-conv3d	kernel size	Not supported	Not supported	kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides							
	paddings			pad_left, pad_right: [0, kernel_w-1] pad_top, pad_bottom: [0, kernel_h-1] pad_front, pad_back: [0, kernel_d-1]				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid				

Table 23: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA0_C16M4B1 [5](VEK280)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-transposed-conv3d	kernel size	Not supported	Not supported	kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides							
	paddings			pad_left, pad_right: [0, min((kernel_w-1), 15)] pad_top, pad_bottom: [0, min((kernel_h-1), 15)] pad_front, pad_back: [0, min((kernel_d-1), 15)]				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, ReLU6				

Notes:

1. For DPUCAHX8H, only list *DPUCAHX8H_ISA2_DWC* here. For more IP configurations, see *DPUCAHX8H for Convolutional Neural Networks Product Guide* (PG367)
2. For DPUCVDX8H, only list *DPUCVDX8H_ISA1_F2W4_4PE* here. For more IP configurations, see *DPUCVDX8H for Convolutional Neural Networks LogiCORE IP* (PG403)
3. For DPUCZDX8G, only list DPUCZDX8G_ISA1_B4096 here. For more IP Configurations, see *DPUCZDX8G for Zynq UltraScale+ MPSoCs* (PG338)
4. For DPUCVDX8G, only list DPUCVDX8G_ISA3_C32B3 here. For more IP Configurations, see *DPUCVDX8G for Versal ACAPs Product Guide* (PG389)
5. For DPUCV2DX8G, only list DPUCV2DX8G_ISA0_C16M4B1 here. For more IP Configurations, see *DPUCV2DX8G for Versal ACAPs Product Guide* (PG425)
6. For DPUCZDX8G, the activation LeakyReLU for depthwise-conv like operators is not enabled by default. About how to enable this activation, please refer to *DPUCZDX8G for Zynq UltraScale+ MPSoCs* (PG338)
7. For DPUCVDX8G, the activation LeakyReLU for depthwise-conv like operators is not enabled by default. About how to enable this activation, please refer to *DPUCVDX8G for Versal ACAPs Product Guide* (PG389)

Operators Supported by TensorFlow

Table 24: **Operators Supported by TensorFlow**

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
placeholder / inputlayer*	shape	data	shape	Allocate memory for input data.
			data_type	
const		const	data	Allocate memory for const data.
			shape	
			data_type	
conv2d	filter	conv2d	kernel	Convolution Engine.
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
	dilations		dilation	
conv2d*	kernel_size	conv2d	kernel	
	strides		stride	
	padding		pad([0, 0, 0, 0])	
	dilation_rate		dilation	
	use_bias			
	group		group	
depthwiseconv2dnative	filter	depthwise-conv2d	kernel	Depthwise-Convolution Engine.
	strides		stride	
	explicit_paddings		pad	
	padding		pad_mode(SAME or VALID)	
	dilations		dilation	
conv2dbackpropinput / conv2dtranspose*	filter	transposed-conv2d	kernel	Convolution Engine.
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
	dilations		dilation	
spacetobacthnd + conv2d + batchtospacend	block_shape	conv2d	dilation	Spacetobatch, Conv2d and Batchtospace would be mapped to Convolution Engine when specific requirements that Xilinx sets have been met.
	padding		pad	
	filter		kernel	
	strides		stride	
	padding		pad_mode(SAME)	
	dilations		dilation	
	block_shape			
	crops			

Table 24: **Operators Supported by TensorFlow** *(cont'd)*

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
matmul / dense*	transpose_a	conv2d / matmul	transpose_a	The matmul would be transformed to a conv2d operation once the equivalent conv2d meets the hardware requirements and can be mapped to DPU.
	transpose_b		transpose_b	
maxpool / maxpooling2d* / globalmaxpool2d*	ksize	maxpool2d	kernel	Pooling Engine. Attribute global will be set true when original pooling operator requires global reduction.
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
			global	
avgpool / averagepooling2d* / globalavgeragepooling2d*	pool_size	avgpool2d	kernel	Pooling Engine. Attribute global will be set true when original pooling operator requires global reduction.
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
			count_include_pad (false)	
			count_include_invalid (true)	
			global	
mean	axis	avgpool / reduction_mean	axis	Mean operation would be transformed to avgpool if the equivalent avgpool meets the hardware requirements and can be mapped to DPU.
	keep_dims		keep_dims	
relu		relu		Activations would be fused to adjacent operations such as convolution, add, etc.
relu6		relu6		
leakyrelu	alpha	leaky_relu	alpha	
fixneuron / quantizelayer*	bit_width	fix	bit_width	It would be divided into float2fix and fix2float during compilation, then the float2fix and fix2float operations would be fused with adjacent operations into course-grained operations.
	quantize_pos		fix_point	
			if_signed	
			round_mode	
identity		identity		Identity would be removed.

Table 24: **Operators Supported by TensorFlow** *(cont'd)*

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
add, addv2		add		If the add is an element-wise add, the add would be mapped to DPU Element-wise Add Engine, if the add is an channel-wise add, Xilinx searches for opportunities to fuse the add with adjacent operations such as convolutions.
mul		mul		Mul can be mapped to Depthwise-Convolution Engine if one of its input is constant. If its two inputs are in same shape, it may be mapped to Misc Engine as Element-wise multiplication. For some other mul operation that is a part of special operators combination, then this mul can be fused into these combination. Otherwise it will be mapped to CPU.
concatv2 / concatenate*	axis	concat	axis	Xilinx reduces the overhead resulting from the concat by special reading or writing strategies and allocating the on-chip memory carefully.
pad / zeropadding2d*	paddings	pad	paddings	First compiler will try to fuse "CONSTANT" padding into adjacent operations, e.g. convolution and pooling. If there is no such operator, it still can be mapped to DPU when padding dimension equals 4 and meets the hardware requirements. For "SYMMETRIC" padding, it would be mapped to DPU. But "REFLECT" padding is not supported by DPU.
	mode		mode	
			constant_values	
shape		shape		The shape operation would be removed.

Table 24: **Operators Supported by TensorFlow** *(cont'd)*

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
stridedslice	begin	strided_slice	begin	If they are shape-related operations, they would be removed during compilation. If they are components of a coarse-grained operation, they would be fused with adjacent operations. Otherwise, they would be compiled into CPU implementations.
	end		end	
	strides		strides	
pack	axis	stack	axis	
neg		neg		
realdiv		div		
sub		sub		
prod	axis	reduction_product	axis	
	keep_dims		keep_dims	
sum	axis	reduction_sum	axis	
	keep_dims		keep_dims	
max	axis	reduction_max	axis	
	keep_dims		keep_dims	
resizebilinear	size	resize	size	If the mode of the resize is 'BILINEAR', align_corner=false, half_pixel_centers = false, size = 2, 4, 8; align_corner=false, half_pixel_centers = true, size = 2, 4 can be transformed to DPU implementations (pad +depthwise-transposed conv2d). If the mode of the resize is 'NEAREST' and the size is an integer, the resize would be mapped to DPU implementations.
	align_corners		align_corners	
	half_pixel_centers		half_pixel_centers	
			mode="BILINEAR"	
resizenearestneighbor	size	resize	size	
	align_corners		align_corners	
	half_pixel_centers		half_pixel_centers	
			mode="NEAREST"	
upsample2d/ upsampling2d*	size	resize	scale	
			align_corners	
			half_pixel_centers	
	interpolation		mode	
reshape	shape	reshape	shape	They would be transformed to the reshape operation in some cases. Otherwise they would be mapped to CPU.
reshape*	target_shape			
transpose	perm	transpose	order	
squeeze	axis	squeeze	axis	
exp		exp		They would only be compiled into CPU implementations.
softmax	axis	softmax	axis	
sigmoid		sigmoid		

Table 24: **Operators Supported by TensorFlow** *(cont'd)*

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
square+ rsqrt+ maximum		l2_normalize	axis	output = x / sqrt(max(sum(x ^ 2), epsilon)) would be fused into a l2_normalize in XIR.
			epsilon	

Notes:

1. The OPs in TensorFlow listed above are supported in XIR. All of them have CPU implementations in the tool-chain.
2. Operators with * represent that the version of TensorFlow > 2.0.

Operators Supported by PyTorch

Table 25: **Operators Supported by PyTorch**

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
Parameter/tensor/ zeros	data	const	data	Allocate memory for input data.
			shape	
			data_type	
Conv2d	in_channels	conv2d (groups = 1) / depthwise-conv2d (groups = input channel)		If groups == input channel, the convolution would be compiled into Depthwise-Convolution Engine. If groups == 1, the convolution would be mapped to Convolution Engine. Otherwise, it would be mapped to the CPU.
	out_channels			
	kernel_size		kernel	
	stride		stride	
	padding		pad	
	padding_mode('zeros')		pad_mode (FLOOR)	
	groups			
	dilation		dilation	
ConvTranspose2d	in_channels	transposed-conv2d (groups = 1) / depthwise-transposed-conv2d (groups = input channel)		If groups == input channel, the convolution would be compiled into Depthwise-Convolution Engine. If groups == 1, the convolution would be mapped to Convolution Engine. Otherwise, it would be mapped to the CPU. For the output_padding feature, DPU is not supported yet, so, if the value is not all 0, this operator will be assigned to CPU.
	out_channels			
	kernel_size		kernel	
	stride		stride	
	padding		pad	
	output_padding		output_padding	
	padding_mode('zeros')		pad_mode (FLOOR)	
	groups			
	dilation		dilation	
matmul		conv2d / matmul	transpose_a	The matmul would be transformed to conv2d and compiled to Convolution Engine. If the matmul fails to be transformed, it would be implemented by CPU.
			transpose_b	
MaxPool2d / AdaptiveMaxPool2d	kernel_size	maxpool2d	kernel	Pooling Engine
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	output_size (adaptive)		global	

Table 25: **Operators Supported by PyTorch** (cont'd)

PyTorch		XIR		DPU Implementation
API	Attributes	OP name	Attributes	
AvgPool2d / AdaptiveAvgPool2d	kernel_size	avgpool2d	kernel	Pooling Engine
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	count_include_pad		count_include_pad	
			count_include_invalid (true)	
	output_size (adaptive)		global	
ReLU		relu		Activations would be fused to adjacent operations such as convolution and add.
LeakyReLU	negative_slope	leakyrelu	alpha	
ReLU6		relu6		
Hardtanh	min_val = 0			
	max_val = 6			
Hardsigmoid		hard-sigmoid		
Hardswish		hardswish		
ConstantPad2d / ZeroPad2d	padding	pad	paddings	First compiler will try to fuse "CONSTANT" padding into adjacent operations, e.g. convolution and pooling. If there is no such operator, it still can be mapped to DPU when padding dimension equals 4 and meets the hardware requirements.
	value = 0		constant_values	
			mode ("CONSTANT")	

Table 25: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
add		add		If the add is an element-wise add, the add would be mapped to DPU Element-wise Add Engine. If the add is a channel-wise add, search for opportunities to fuse the add with adjacent operations such as convolutions. If they are shape-related operations, they would be removed during compilation. If they are components of a coarse-grained operation, they would be fused with adjacent operations. Otherwise, they would be compiled into CPU implementations. Mul can be mapped to Depthwise-Convolution Engine if one of its input is constant. If its two inputs are in same shape, it may be mapped to Misc Engine as Element-wise multiplication. For some other mul operation that is a part of special operators combination, then this mul can be fused into these combination. Otherwise it will be mapped to CPU.
sub / rsub		sub		
mul		mul		
neg		neg		
sum	dim	reduction_sum	axis	
	keepdim		keep_dims	
max	dim	reduction_max	axis	
	keepdim		keep_dims	
mean	dim	reduction_mean	axis	
	keepdim		keep_dims	
interpolate / upsample / upsample_bilinear / upsample_nearest	size	resize	size	If the mode of the resize is 'BILINEAR', align_corner=false, half_pixel_centers = false, size = 2, 4, 8; align_corner=false, half_pixel_centers = true, size = 2, 4 can be transformed to DPU implementations (pad +depthwise-transposed conv2d). If the mode of the resize is 'NEAREST' and the size are integers, the resize would be mapped to DPU implementations.
	scale_factor			
	mode		mode	
	align_corners		align_corners	
			half_pixel_centers = ! align_corners	

Table 25: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
transpose	dim0	transpose	order	These operations would be transformed to the reshape operation in some cases. Additionally, search for opportunities to fuse the dimension transformation operations into special load or save instructions of adjacent operations to reduce the overhead. Otherwise, they would be mapped to CPU.
	dim1			
permute	dims			
view/reshape	size	reshape	shape	
flatten	start_dim	reshape / flatten	start_axis	
	end_dim		end_axis	
squeeze	dim	reshape / squeeze	axis	
cat	dim	concat	axis	Reduce the overhead resulting from the concat by special reading or writing strategies and allocating the on-chip memory carefully.
aten::slice*	dim	strided_slice		If the strided_slice is shape-related or is the component of a coarse-grained operation, it would be removed. Otherwise, the strided_slice would be compiled into CPU implementations.
	start		begin	
	end		end	
	step		strides	
BatchNorm2d	eps	depthwise-conv2d / scale	epsilon	If the batch_norm is quantized and can be transformed to a depthwise-conv2d equivalently, it would be transformed to depthwise-conv2d and the compiler would search for compilation opportunities to map the batch_norm into DPU implementations. Otherwise, the batch_norm would be executed by CPU.
			axis	
			moving_mean	
			moving_var	
			gamma	
			beta	
softmax	dim	softmax	axis	They would only be compiled into CPU implementations.
Tanh		tanh		
Sigmoid		sigmoid		

Table 25: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
PixelShuffle	upscale_factor	pixel_shuffle	scale	They would be transformed to tile if there's convolution as its input.
			upscale=True	
PixelUnshuffle	downscale_factor	pixel_shuffle	scale	
			upscale=False	

Notes:

1. If the slice of tensor in PyTorch is written in the Python syntax, it is transformed into `aten::slice`.

VAI_C Usage

The corresponding Vitis AI compiler for Caffe and TensorFlow frameworks are `vai_c_caffe`, `vai_c_tensorflow`, `vai_c_tensorflow2`, and `vai_c_xir` across Cloud-to-Edge DPU. The common options for VAI_C are illustrated in the following table.

Table 26: **VAI_C Common Options for Cloud and Edge DPU**

Parameters	Description
--arch	The DPU architecture configuration file for the VAI_C compiler in JSON format. For pre-built DPU xclbins in Vitis AI releases, you can find the corresponding `arch.json` file in Vitis AI docker (`/opt/vitis_ai/compiler/arch`). The contents should be something like {"target": "DPUCZDX8G_ISA0_B4096"}. For customized DPU IPs, the corresponding `arch.json` files are generated by the DPU TRD along with the DPU IPs. The contents should be something like {“fingerprint”:"0x0101000016010407"}. The fingerprint is a 64-bit digital signature to identify a DPU target. It consists of 1 byte to indicate the DPU type, 1 byte to indicate the ISA version, and 6 bytes to indicate specific configurations. The fingerprint is unique to each DPU configuration and runtime relies on it to identify DPU instance running on the current platform and to verify that the model is compiled for the same DPU target. "DPUCZDX8G_ISA0_B4096" is an alias for a specific fingerprint which is pre-defined in the compiler.
--output_dir	Path of output directory for vai_c_caffe and vai_c_tensorflow after compilation process.
--net_name	Name of DPU kernel for network model after compiled by VAI_C.
--options	The list for the extra options in the format of 'key':'value'. If there are multiple options to be specified, they are separated by ‘,’. Use --options '{"input_shape": "1,224,224,3"}' to specify input shape manually. Use --options '{"plugins": "plugin0,plugin1"}' to specify plugin libraries. Use --options '{"output_ops": "op_name0,op_name1"}' to specify output ops. Use --options '{"prefetch": "true"}' to enable cross layer prefetch. Use --options '{"hd_opt": "true"}' to enable special optimization for HD input. ***Note***: Arguments specified with “--options” have the highest priorities and will override the values specified in other places.

Deploying and Running the Model

Programming with VART

Vitis™ AI provides a C++ DpuRunner class with the following interfaces:

1. ```
   std::pair<uint32_t, int> execute_async(
                            const std::vector<TensorBuffer*>& input,
                            const std::vector<TensorBuffer*>& output);
   ```

   Submit input tensors for execution and output tensors to store results. The host pointer is passed using the TensorBuffer object. This function returns a job ID and the status of the function call.

2. ```
   int wait(int jobid, int timeout);
   ```

   The job ID returned by `execute_async` is passed to `wait()` to block until the job is complete and the results are ready.

3. ```
   TensorFormat get_tensor_format()
   ```

   Query the DpuRunner for the Tensor format it expects.

   Returns DpuRunner::TensorFormat::NCHW or DpuRunner::TensorFormat::NHWC

4. ```
   std::vector<Tensor*> get_input_tensors()
   std::vector<Tensor*> get_output_tensors()
   ```

   Query the DpuRunner for the shape and name of the input and output tensors it expects for its loaded Vitis AI model.

5. To create a DpuRunner object, call the following: function

   ```
   create_runner(const xir::Subgraph* subgraph, const std::string& mode =
   "")
   ```

   It returns the following:

   ```
   std::unique_ptr<Runner>
   ```

   The input to create_runner is a XIR subgraph generated by the Vitis AI compiler.

> **TIP:** *To enable multi-threading with VART, create a runner for each thread.*

Note: If the model has multiple subgraph, you can refer to

https://github.com/Xilinx/Vitis-AI-Tutorials/tree/1.4/Feature_Tutorials/pytorch-subgraphs.

C++ Example

```
// get dpu subgraph by parsing model file
auto runner = vart::Runner::create_runner(subgraph, "run");
// populate input/output tensors
auto job_data = runner->execute_async(inputs, outputs);
runner->wait(job_data.first, -1);
// process outputs
```

For more C++ examples, refer to Vitis AI Examples.

Vitis AI also provides a Python ctypes Runner class that mirrors the C++ class, using the C DpuRunner implementation:

```
class Runner:
def __init__(self, path)
def get_input_tensors(self)
def get_output_tensors(self)
def get_tensor_format(self)
def execute_async(self, inputs, outputs)
# differences from the C++ API:
# 1. inputs and outputs are numpy arrays with C memory layout
#    the numpy arrays should be reused as their internal buffer
#    pointers are passed to the runtime. These buffer pointers
#    may be memory-mapped to the FPGA DDR for performance.
# 2. returns job_id, throws exception on error
def wait(self, job_id)
```

Python Example

```
dpu_runner = runner.Runner(subgraph, "run")
# populate input/output tensors
jid = dpu_runner.execute_async(fpgaInput, fpgaOutput)
dpu_runner.wait(jid)
# process fpgaOutput
```

DPU Debug with VART

This section aims to demonstrate how to verify DPU inference result with VART tools. TensorFlow ResNet50, and PyTorch ResNet50 networks are used as examples. Following are the four steps for debugging the DPU with VART:

1. Generate a quantized inference model and reference result.
2. Generate a DPU xmodel.

3. Generate a DPU inference result.
4. Crosscheck the reference result and the DPU inference result.

Before you start to debug the DPU result, ensure that you have set up the environment according to the instructions in the Chapter 2: Getting Started section.

Note: Caffe has been deprecated from Vitis™ AI 2.5. For information on Caffe, see Vitis AI 2.0 User Guide.

TensorFlow Workflow

To generate the quantized inference model and reference result, follow these steps:

1. Generate the quantized inference model by running the following command to quantize the model.

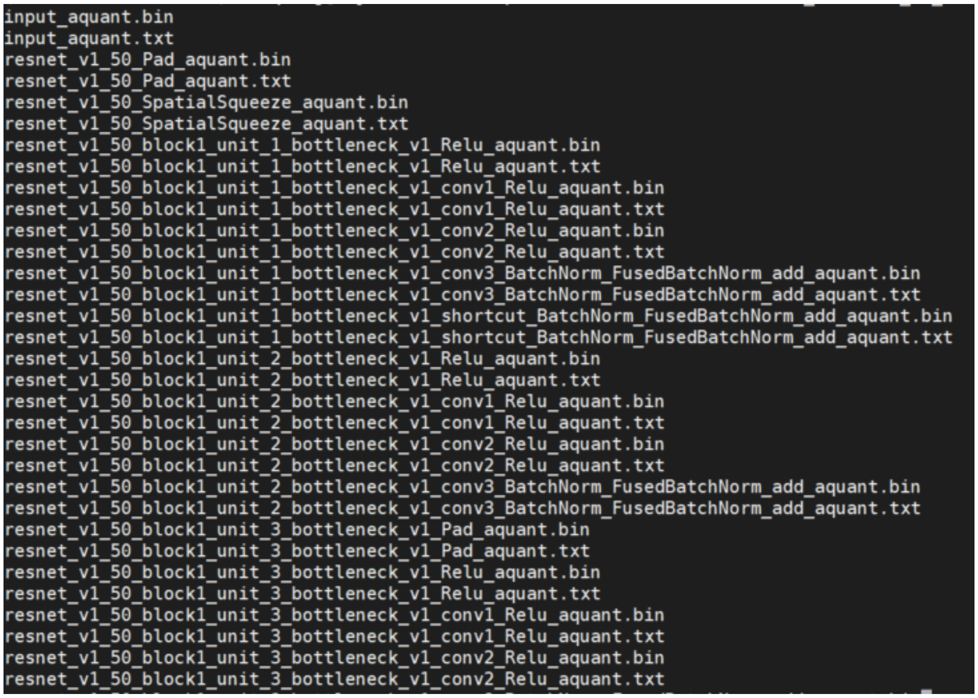
   The quantized model, `quantize_eval_model.pb`, is generated in the `quantize_model` folder.

   ```
   vai_q_tensorflow quantize                                     \
       --input_frozen_graph ./float/resnet_v1_50_inference.pb    \
       --input_fn input_fn.calib_input                           \
       --output_dir quantize_model                               \
       --input_nodes input                                       \
       --output_nodes resnet_v1_50/predictions/Reshape_1         \
       --input_shapes    ?,224,224,3                             \
       --calib_iter    100
   ```

2. Generate the reference result by running the following command to generate reference data.

   ```
   vai_q_tensorflow dump --input_frozen_graph         \
               quantize_model/quantize_eval_model.pb \
        --input_fn input_fn.dump_input                \
        --output_dir=dump_gpu
   ```

   The following figure shows part of the reference data.



```
input_aquant.bin
input_aquant.txt
resnet_v1_50_Pad_aquant.bin
resnet_v1_50_Pad_aquant.txt
resnet_v1_50_SpatialSqueeze_aquant.bin
resnet_v1_50_SpatialSqueeze_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.txt
```

3. Generate the DPU xmodel by running the following command to generate the DPU xmodel file, for example, U50LV.

```
vai_c_tensorflow --frozen_pb quantize_model/quantize_eval_model.pb \
  --arch /opt/vitis_ai/compiler/arch/DPUCAHX8H/U50LV/arch.json       \
  --output_dir compile_model                                        \
  --net_name resnet50_tf
```

4. Generate the DPU inference result by running the following command to generate the DPU inference result and compare the DPU inference result with the reference data automatically.

```
env XLNX_ENABLE_DUMP=1 XLNX_ENABLE_DEBUG_MODE=1 XLNX_GOLDEN_DIR=./
dump_gpu/dump_results_0 \
   xdputil run ./compile_model/resnet_v1_50_tf.xmodel            \
   ./dump_gpu/dump_results_0/input_aquant.bin                    \
   2>result.log 1>&2
```

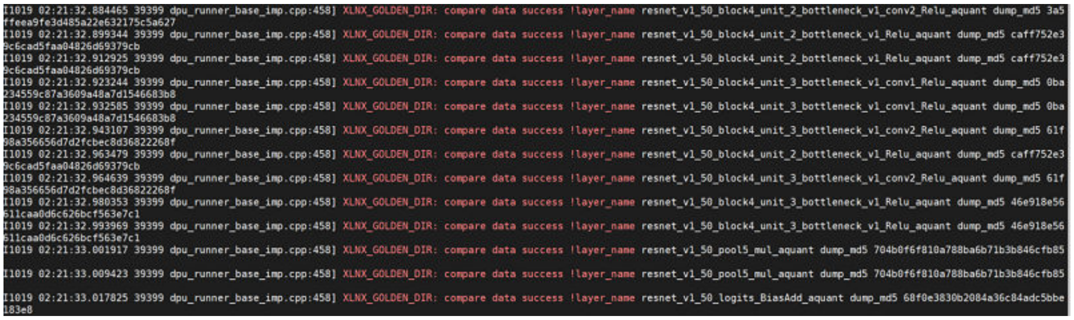
For `xdputil` more usage, execute `xdputil --help` command.

After the above command runs, the DPU inference result and the comparing result `result.log` are generated. The DPU inference results are located in the `dump` folder.

5. Crosscheck the reference result and the DPU inference result.

    a. View comparison results for all layers.

    ```
    grep --color=always 'XLNX_GOLDEN_DIR.*layer_name' result.log
    ```



```
I1019 02:21:32.884465 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_conv2_Relu_aquant dump_md5 3a5
ffeea9fe3d485a22e632175c5a627
I1019 02:21:32.899344 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_Relu_aquant dump_md5 caff752e3
9c6cad5faa04826d69379cb
I1019 02:21:32.912925 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_Relu_aquant dump_md5 caff752e3
9c6cad5faa04826d69379cb
I1019 02:21:32.923244 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv1_Relu_aquant dump_md5 0ba
234559c87a3609a48a7d1546683b8
I1019 02:21:32.932585 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv1_Relu_aquant dump_md5 0ba
234559c87a3609a48a7d1546683b8
I1019 02:21:32.943107 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv2_Relu_aquant dump_md5 61f
98a356656d7d2fcbec8d36822268f
I1019 02:21:32.963479 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_Relu_aquant dump_md5 caff752e3
9c6cad5faa04826d69379cb
I1019 02:21:32.964639 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv2_Relu_aquant dump_md5 61f
98a356656d7d2fcbec8d36822268f
I1019 02:21:32.980353 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_Relu_aquant dump_md5 46e918e56
611caa0d6c626bcf563e7c1
I1019 02:21:32.993969 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_Relu_aquant dump_md5 46e918e56
611caa0d6c626bcf563e7c1
I1019 02:21:33.001917 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_pool5_mul_aquant dump_md5 704b0f6f810a788ba6b71b3b846cfb85

I1019 02:21:33.009423 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_pool5_mul_aquant dump_md5 704b0f6f810a788ba6b71b3b846cfb85

I1019 02:21:33.017825 39399 dpu_runner_base_imp.cpp:458] XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_logits_BiasAdd_aquant dump_md5 68f0e3830b2084a36c84adc5bbe
183e8
```

b. View only the failed layers.

```
grep --color=always 'XLNX_GOLDEN_DIR.*fail ! layer_name' result.log
```

If the crosscheck fails, use the following methods to further check from which layer the crosscheck fails.

a. Check the input of DPU and GPU, make sure they use the same input data.

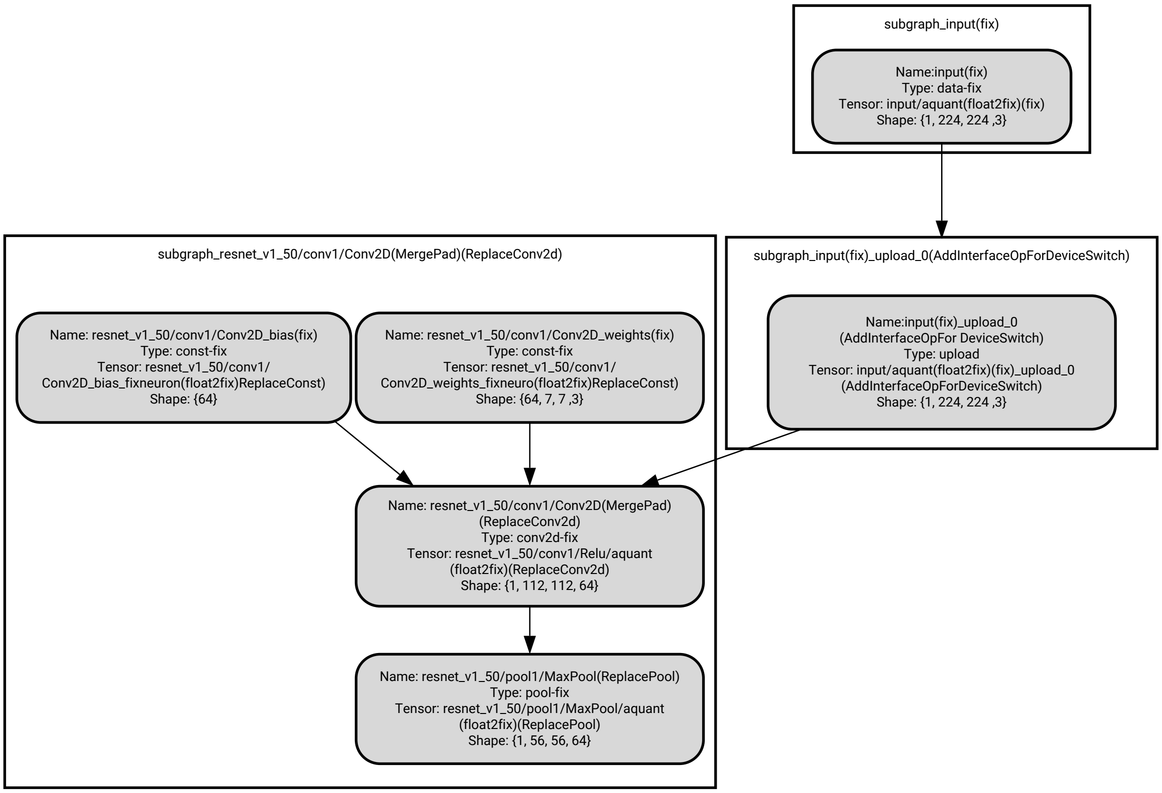
b. Use `xdputil` tool to generate a picture for displaying the network's structure.

```
Usage: xdputil xmodel <xmodel> -s <svg>
```

Note: In the Vitis AI docker environment, execute the following command to install the required library.

```
sudo apt-get install graphviz
```

When you open the picture you created, you can see many little boxes around these ops. Each box means a layer on DPU. You can use the last op's name to find its corresponding one in GPU dump-result. The following figure shows parts of the structure.



X24898-120920

c. Submit the files to Xilinx.

   If certain layer proves to be wrong on DPU, prepare the quantized model, such as `quantize_eval_model.pb` as one package for further analysis by factory and send it to Xilinx with a detailed description.

PyTorch Workflow

To generate the quantized inference model and reference result, follow these steps:

1. Generate the quantized inference model by running the following command to quantize the model.

   ```
   python resnet18_quant.py --quant_mode calib --subset_len 200
   ```

2. Set `deploy_check` to `True` in `export_xmodel` API.

   ```
   quantizer.export_xmodel(deploy_check=True)
   ```

3. Generate the reference result by running the following command to generate reference data.

   ```
   python resnet18_quant.py --quant_mode test  --deploy
   ```

4. Generate the DPU xmodel by running the following command to generate DPU xmodel file.

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

5. Generate the DPU inference result.

   This step is same as the step in TensorFlow workflow.

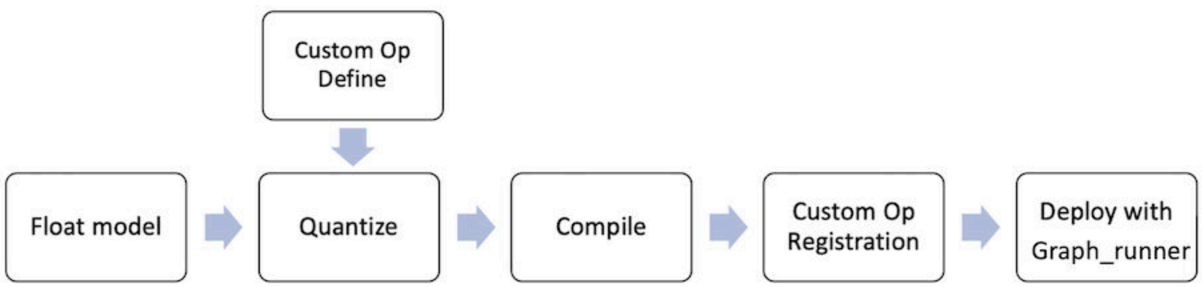
6. Crosscheck the reference result and the DPU inference result.

   This step is same as the step in TensorFlow workflow.

Custom OP Workflow

In VAI2.5 release, Pytorch model and Tensorflow2 model with custom op are supported. The basic workflow for custom op is shown below.

Figure 25: **Custom Op Workflow**



The following are the steps in the workflow:

1. Define the OP as a custom OP which is unknown to XIR and then quantize the model.
2. Compile the quantized model.
3. Register and implement the custom OP.
4. Deploy the model with `graph_runner` APIs.

Step 3 supports both `C++` and `Python` to implement and register the custom OP. There are more than 50 supported common OPs by the Vitis AI library. You can find the source code of the common OPs in https://github.com/Xilinx/Vitis-AI/tree/v3.0/src/vai_library/cpu_task/ops.

***Note*:** If you want to implement an accelerated (PL or AIE) function for a custom op, make it as a CPU OP, but implement the PL/AIE calling codes in this CPU OP's implementation.

For the step 4, `graph_runner` APIs support both `C++` and `Python`. When using the Graph_runner API to deploy Custom OP, its runtime has been optimized, including Zero-copy technology between different DPU OPs and CPU OPs. It means address sharing between different layers without data copying.

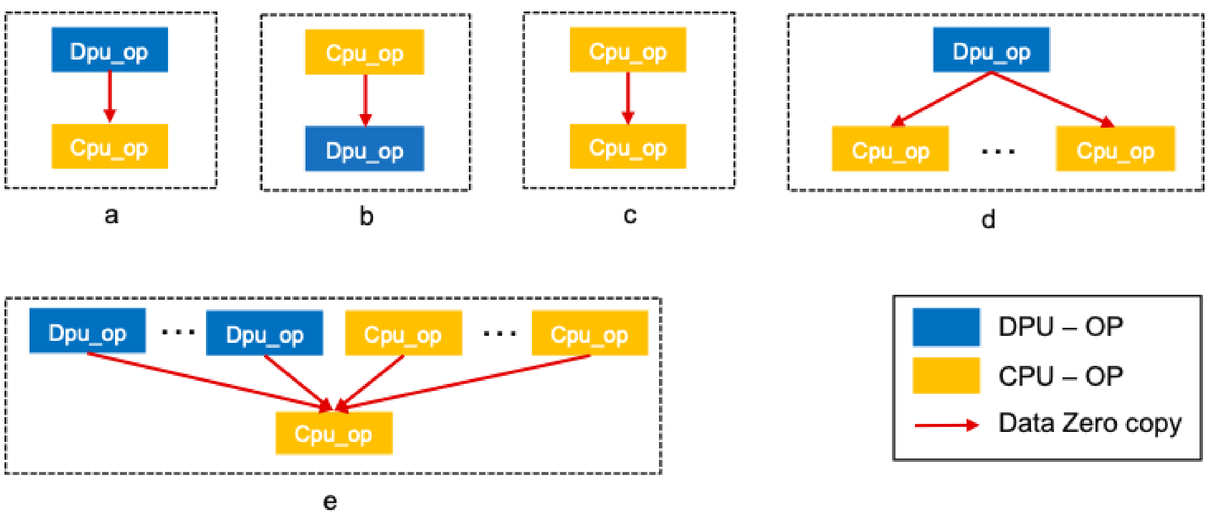
The following model structure is supported by Zero copy.

Table 27: **Model structure supported by Zero copy**

Type	Output of OP	Input of OP	Using Zero copy
a	Single dpu OP	Single cpu OP	Yes
b	Single cpu OP	Single dpu OP	Yes
c	Single cpu OP	Single cpu OP	Yes
d	Single dpu OP	Multiple cpu OP	Yes
e	Multiple cpu OP and multiple dpu OP	Single cpu OP	Yes

Note: Model structure types a-e are shown in the figure below.

Figure 26: **Model Structure Types**



Note: The application of Zero copy for the other model structures depends on the situation.

The following are examples for the two models respectively.

- MNIST model based on Tensorflow2
- Pointpillars model based on Pytorch

Quick Start Custom OP Workflow

In this section, a TensorFlow2 model with custom OP will be used to show how to quick start custom OP workflow on the edge ZCU102 platform.

Quantizing

1. Launch the docker image

```
[Host]$ cd Vitis-AI
[Host]$ ./docker_run.sh xilinx/vitis-ai-cpu:latest
```

2. Download the model source code package tf2_custom_op_demo.tar.gz

```
[Docker]$ wget https://www.xilinx.com/bin/public/openDownload?
filename=tf2_custom_op_demo.tar.gz -O tf2_custom_op_demo.tar.gz
[Docker]$ tar -xzvf tf2_custom_op_demo.tar.gz
[Docker]$ cd tf2_custom_op_demo
```

3. Quantize

```
[Docker]$ conda activate vitis-ai-tensorflow2
[Docker]$ bash 1_run_train.sh
[Docker]$ bash 3_run_quantize.sh
```

After quantizing, the quantized model named `quantized.h5` will be generated in the `./quantized/` directory.

Compiling

```
[Docker]$ vai_c_tensorflow2 -m ./quantized/quantized.h5 -a /opt/vitis_ai/
compiler/arch/DPUCZDX8G/ZCU102/arch.json -o ./ -n tf2_custom_op
```

OP Registration

1. Copy the `tensorflow2_example` folder to the ZCU102 board.

```
[Host]$ scp -r Vitis-AI/examples/custom_operator/tensorflow2_example
root@[BOARD_IP]:~
```

2. Run the following commands to register the custom OP on the target.

```
[Target]# cd ~/tensorflow2_example/op_registration/cpp
[Target]# bash op_registration.sh
```

Deployment

1. Copy the compiled model to the board.

```
[Host]$ scp tf2_custom_op.xmodel root@[BOARD_IP]:~
```

2. Download the test image sample.jpg and copy it to the board.
3. Compile the application code on the target.

```
[Target]# cd ~/tensorflow2_example/deployment/cpp
[Target]# bash build.sh
```

4. Run the demo.

```
[Target]# ./tf2_custom_op_graph_runner ~/tf2_custom_op.xmodel ~/
sample.jpg
```

Tensorflow2 Custom OP Model Example

Using the Tensorflow2 model as an example, download the code package from here. Refer to `readme.md` in the package to generate and quantize the model.

Model Quantizing

Tensorflow 2 provides a lot of common built-in layers to build the machine learning models, as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers. Layer is one of the central abstractions in `tf.keras`, subclassing `Layer` is the recommended way to create custom layers. Please refer to tensorflow user guide for more information.

Vai_q_tensorflow2 provides support for new custom layers via subclassing. This tutorial will demonstrate how to quantize models with custom operations step-by-step.

***Note*:** Custom model via subclassing `tf.keras.Model` is not supported by vai_q_tensorflow2 in this release, please flatten it to layers.

1. Train custom layer model

This example defines the custom layer named `MyLayer` to perform a "PReLU" function in `train_eval.py`. This custom layer performs the function below with a trainable weight `alpha`.

```
f(x) = alpha * x , if x < 0
f(x) = x , if x >= 0
```

where `alpha` is a learned array with the same shape as `x`.

A CNN model is built next to classify the MNIST dataset as an example. Before you start to train and quantize the model, please launch the docker and activate `vitis-ai-tensorflow2` environment. Run `1_run_train.sh` to train the model and you will get the float model `my_model.h5` and the accuracy of the model should be >90%.

```
bash 1_run_train.sh
```

```
Epoch 9/10
32/32 [==============================] - 0s 5ms/step - loss: 0.0108 - accuracy: 0.9960
Epoch 10/10
32/32 [==============================] - 0s 4ms/step - loss: 0.0078 - accuracy: 0.9990
313/313 [==============================] - 1s 3ms/step - loss: 0.0530 - accuracy: 0.9237

***************** Summary *****************
Trained float model accuracy:  0.9236999750137329
Trained float model is saved in  ./my_model.h5
```

This float model contains both the model structure and the weights, with a custom layer named `custom_layer`. You can get this information from the printed summary.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 22, 22, 32)        1600
_________________________________________________________________
batch_normalization (BatchNo (None, 22, 22, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 32)        50208
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
custom_layer (Mylayer)       (None, 2, 2, 64)          256
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 10)                2570
=================================================================
Total params: 106,154
Trainable params: 106,026
Non-trainable params: 128
_________________________________________________________________
```

2. (Optional) Evaluate the float model

You can run the script `2_run_eval_float.sh` to test the trained float model.

```
bash 2_run_eval_float.sh
```

3. Quantize the float model

You can quantize the float model with custom layers using the vai_q_tensorflow2 `quantize_model` API. Example code is shown below:

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quant_model = vitis_quantize.VitisQuantizer(loaded_model,
custom_objects={'MyLayer': MyLayer}).quantize_model(calib_dataset=x_test,
add_shape_info=True)
```

The `custom_objects` argument must be passed into the class `VitisQuantizer` when quantizing models with custom layers. The `custom_objects` argument is a dict containing the `{"custom_layer_class_name":"custom_layer_class"}`, multiple custom layers should be separated by a comma. Moreover, `add_shape_info` should also be set to True for models with custom layers to add shape inference information for them.

During quantization, these custom layers will be kept untouched in the quantized model. Run `3_run_quantize.sh` to do quantization:

```
bash 3_run_quantize.sh
```

If everything runs correctly, the quantized model named `quantized.h5` will be generated in `./quantized/` directory. This model can be used as the input of the xcompiler and then deployed on boards.

```
2022-01-06 12:43:53.428589: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 8s 17ms/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quantize Adjustment...
[VAI INFO] Post-Quantize Adjustment Done.
[VAI INFO] Quantization Finished.
[VAI INFO] Start Getting Shape Information...
[VAI INFO] Getting model layer shape information
[VAI INFO] Getting Shape Information Done.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` wi
ll be empty until you train or evaluate the model.

***************** Summary *****************
Quantized model is saved in  ./quantized/quantized.h5
```

4. (Optional) Evaluate the quantized model

Use `model.evaluate` API to evaluate the quantized model. Remember to recompile the model with correct losses and metrics because these information are ignored during the quantization process.

```
quantized_model.compile(loss="binary_crossentropy", metrics=["accuracy"])
quantized_model.evaluate(x_test, y_test)
```

Run `4_run_eval_quant` to evaluate the quantized model.

```
bash 4_run_eval_quant.sh
```

It can be seen that the quantized model has close accuracy to the float model.

```
2022-01-06 12:44:56.418449: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimizati
on Passes are enabled (registered 2)
2022-01-06 12:44:57.448941: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 3s 4ms/step - loss: 0.0897 - accuracy: 0.9207

***************** Summary *****************
Quantized model accuracy:  0.9207000136375427
```

5. Dump the golden results

Golden results are used to check the data correctness or debug the deployed models. Vai_q_tensorflow2 provides `dump_model` API to dump the weights/biases and intermediate activations of the quantized model with a sample input. Since the DPU dumping results are batch by batch, set the batch_size of dataset to 1 when dumping golden results.

```
vitis_quantize.VitisQuantizer.dump_model( model=quant_model,
dataset=x_test[0:1], output_dir="./dump_results", dump_float=True)
```

Since the custom layer is not quantized, set `dump_float=True` to dump float weights and activation for them. Run `5_run_dump.sh` to dump the quantized model.

```
bash 5_run_dump.sh
```

You can see the generated golden results in the folder `./dump_results`. `./dump_results/dump_results_weights` are the save weights, while `./dump_results/dump_results_0` are the saved activation, where the number 0 represents the index of the dataset.

Compiling the Model

For TensorFlow2, the commands are shown below.

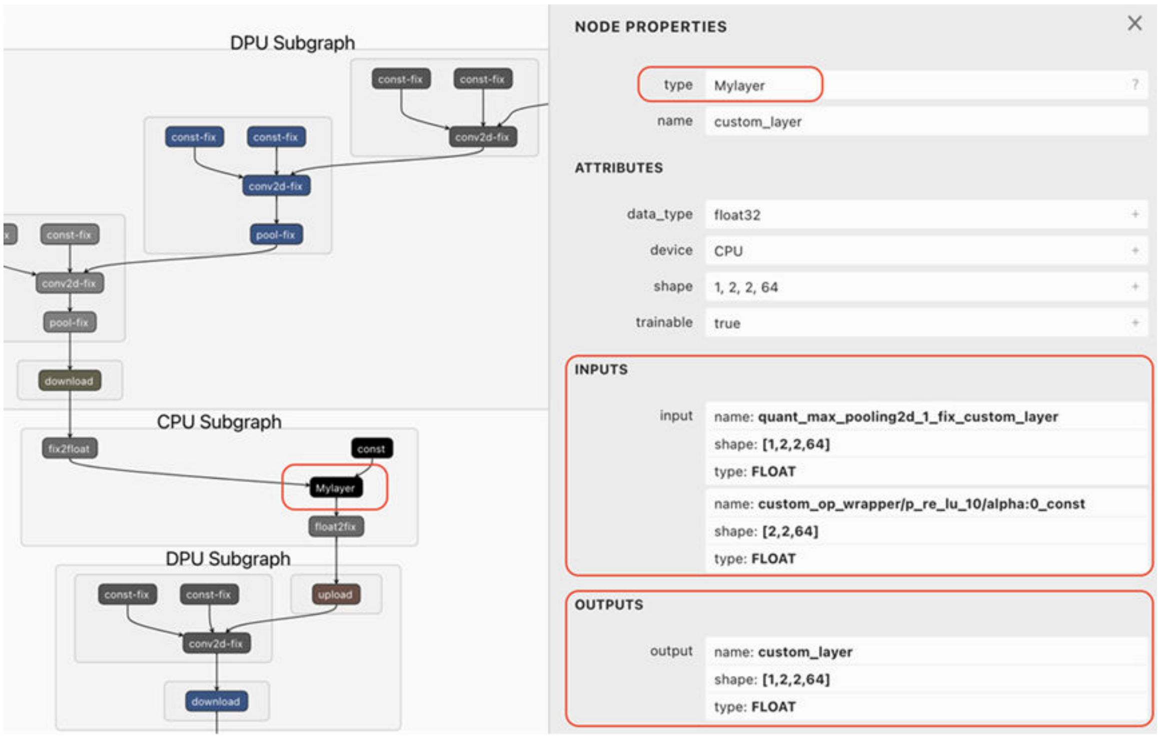
```
conda activate vitis-ai-tensorflow2
cd <path of Vitis-AI>/examples/custom_operator/tensorflow2_example/model/
quantized
vai_c_tensorflow2 -m ./quantized.h5 -a /opt/vitis_ai/compiler/arch/
DPUCZDX8G/ZCU102/arch.json -o ./ -n tf2_custom_op
```

Custom OP Registration

This section will illustrate how to register a custom op step-by-step.

First of all, you can use the `Netron` to check the custom op's properties, such as inputs and outputs.

Figure 27: **Custom op's properties**



You can also use `xdputil` to check the OP's detailed information. Run the following command to check the `custom_layer` OP.

```
xdputil xmodel tf2_custom_op.xmodel --op custom_layer
```

Then, you can create a directory for the new op and all your coding and building can be done under this new folder. Since `Mylayer` op has been realized in VAI2.0, you may use `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer` for reference.

Code

Custom OP registration supports both C++ and Python. The following shows how to implement the OP in C++. For the OP implementation in Python, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/python/`

1. Create cpp file for the new op. It's my_Mylayer_op.cpp in this example.
2. Include the header `<vart/op_imp.h>`.
3. Define a class with a constructor and a function named calculate. In this example, the class name is `MylayerOp`.
4. Implement your algorithm in the `calculate` function. For this example, `PReLU` is implemented.

5. Register your class by DEF_XIR_OP_IMP(className), for Mylayer, className is `MylayerOp`.

   Tips: You can copy the cpp file from my_Mylayer_op.cpp and modify the class name as desired, and then you can focus on step 4 and 5. The details of cpp file for Mylayer op are shown below.

```
#include <vart/op_imp.h>
class MylayerOp {
 public:
  MylayerOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {}
  int calculate(vart::simple_tensor_buffer_t<float> output,
                std::vector<vart::simple_tensor_buffer_t<float>> inputs) {
    CHECK_EQ(inputs.size(), 2);
    auto input_data_shape = inputs[0].tensor->get_shape();
    auto input_alpha_shape = inputs[1].tensor->get_shape();
    auto output_shape = output.tensor->get_shape();
    auto dims = output_shape.size();

    CHECK_EQ(input_data_shape.size(), 4);
    CHECK_EQ(input_alpha_shape.size(), 3);
    for (auto i = 1u; i < dims; i++)
      CHECK_EQ(input_data_shape[i], input_alpha_shape[i - 1]);

    auto element_num = inputs[0].tensor->get_element_num();
    auto alpha_size = inputs[1].tensor->get_element_num();
    for (auto i = 0; i < element_num; i++) {
      if (inputs[0].data[i] < 0) {
        output.data[i] = inputs[0].data[i] * inputs[1].data[i % alpha_size];
      } else {
        output.data[i] = inputs[0].data[i];
      }
    }

    return 0;
  }

 public:
  const xir::Op* const op;
};

DEF_XIR_OP_IMP(MylayerOp)
```

Build

1. Create a Makefile to build the op library.

   Refer to `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer/Makefile`.

2. Set the output directory and add the dependency between target (output .so), object (.o) and source (.cpp) files.

   If you use the reference Makefile in step 1, you only need to replace the file names with yours, including `libvart_op_imp_Mylayer.so`, `my_Mylayer_op.o` and `my_Mylayer_op.cpp`.

3. Execute "make" to complete the build of the library.

   After building, `libvart_op_imp_Mylayer.so` library will be generated.

The details of Makefile for Mylayer op are shown below.

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_Mylayer.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_Mylayer_op.o: my_Mylayer_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_Mylayer.so: $(OUTPUT_DIR)/my_Mylayer_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -lglog -
lvitis_ai_library-runner_helper -lvart-runner -lxir
```

***Note*:** The name of the output library must keep the same format as libvart_op_imp_Mylayer.so and in this format, “Mylayer”, which is the type of Mylayer op as shown in the figure of custom op's properties, must be changed to the type of your custom op, otherwise, the library can’t be linked when debugging the op or running the graph.

Debug

This section will introduce how to debug the custom op after you generate the custom op library. Before you debug the custom op, please copy the `libvart_op_imp_Mylayer.so` to `/usr/lib` on the target. Then, use `run_op` command in `xdputil` to test the op on the target. The usage of `run_op` is shown below.

```
xdputil run_op <model_file> <op_name> [-r ref] [-d dump]
```

***Note*:** For edge, run `xdputil run_op` on the board.

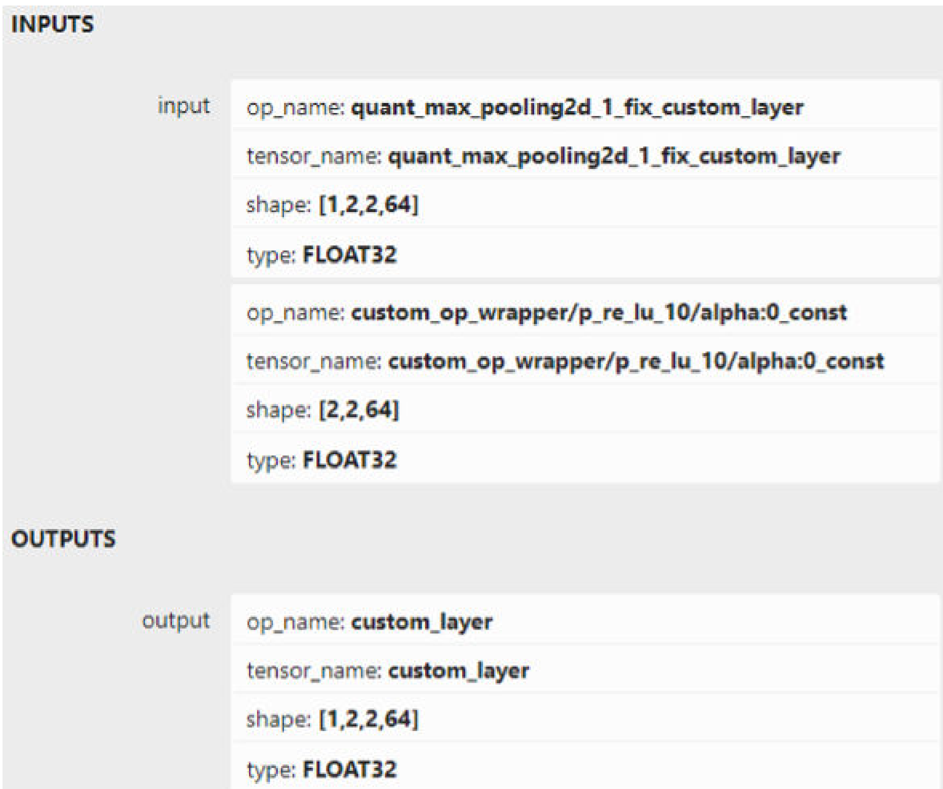
Execute `xdputil run_op -h` to view the valid arguments for `run_op`.

```
root@xilinx-zcu102-2021_2:~# xdputil run_op -h
usage: xdputil.py run_op [-h] [-r REF_DIR] [-d DUMP_DIR] xmodel op_name

positional arguments:
  xmodel                xmodel file name
  op_name               op name, this op_name should be consistent with the name in xmodel

optional arguments:
  -h, --help            show this help message and exit
  -r REF_DIR, --ref_dir REF_DIR
                        reference directory, this directory default as "ref" should contain inputs tensor file like <TENSOR_NAME>.bin
  -d DUMP_DIR, --dump_dir DUMP_DIR
                        dump directory, this directory default as "dump" will be the dump destination of output tensor file
```

In the ref directory, you should provide all the input bin files with the same names as the input tensors of the custom op. For `Mylayer`, it has two inputs and the input tensor names are `quant_max_pooling2d_1_fix_custom_layer` and `custom_op_wrapper/p_re_lu_10/alpha:0_const`, as shown in the following figure. Thus, the input files stored in the ref directory should be `quant_max_pooling2d_1_fix_custom_layer.bin` and `custom_op_wrapper_p_re_lu_10_alpha:0_const.bin`.

Note: You can dump the golden files when you do the model quantization.

Note: The slash ("/") marks in the name should be replace by underscore ("_").



If you still don't know what the names should be, you can just put your input files in ref, then try to execute run_op, then you can also find the file names expected in the resulting error message as shown in the following figure.

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
default directory ./ref has been created, please put the input tensor file in.
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:22:48.982190 16636 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:22:48.987767 16636 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:22:48.987846 16636 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:22:48.987871 16636 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
F1202 04:22:48.988036 16636 test_op_run.cpp:52] Check failed: std::ifstream(filename).read((char*)data.data, data.size).good() fail to
read! filename=./ref/quant_max_pooling2d_1_fix_custom_layer.bin;tensor=quant_max_pooling2d_1_fix_custom_layer
*** Check failure stack trace: ***
/usr/bin/xdputil: line 16: 16636 Aborted                 /usr/bin/python3 -m xdputil $*
```

After the successfully execution of `run_op`, as shown in the following figure, you can find the output bin file with the same name as the output tensor in the dump directory. Then, you can compare it with the golden output and debug your code until they are the same. For Mylayer, the output tensor name is `custom_layer`, so the output bin file name should be `custom_layer.bin` as shown in the following figure.

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:17:31.793207 16626 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:17:31.798681 16626 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:17:31.798761 16626 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:17:31.798786 16626 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
I1202 04:17:31.799113 16626 test_op_run.cpp:55] read ./ref/quant_max_pooling2d_1_fix_custom_layer.bin to 0xaaab004831e0 size=1024
I1202 04:17:31.799202 16626 test_op_run.cpp:55] read ./ref/custom_op_wrapper_p_re_lu_10_alpha:0_const.bin to 0xaaab00506a90 size=1024
I1202 04:17:31.799890 16626 test_op_run.cpp:114] graph name:functional_1testing op: {
        {args: input= TensorBuffer{@0xaaab004d2fa0,tensor=xir::Tensor{name = quant_max_pooling2d_1_fix_custom_layer, type = FLOAT32, sh
ape = {1, 2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab004831e0, 1024)]} TensorBuffer{@0xaaab004d2220,tensor=xir::Tensor{name = cust
om_op_wrapper/p_re_lu_10/alpha:0_const, type = FLOAT32, shape = {2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab00506a90, 1024),(Virt=
0xaaab00506c90, 512)]}}
{
I1202 04:17:31.800941 16626 test_op_run.cpp:68] write output to dump/custom_layer.bin from 0xaaab004e11e0 size=1024
test pass
```

Finally, you can compare the `custom_layer.bin` with the golden output file of the custom op. If they are same, that means the op library you implement is correct.

Note: Since the floating point numbers on different platforms may be different, this will cause the results of your dump and the golden results to not exactly match. Thus, the following tool for floating-point number comparison is supplied. It the difference is within a certain threshold, then it can be considered to be consistent.

```
xdputil comp_float <golden_file> <dump_file> [-t threshold] [--verbose]

-t: threshold, the default value is 0.5, in %
```

Deployment

This section will introduce how to deploy the tensorflow2 model with custom op in `graph_runner` APIs. For `graph_runner` APIs, it supports both C++ and Python. For C++ example, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp` . For Python example, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/python`.

Tips: You can create a new folder for your application, then all your code and build files can be placed under this folder.

Code

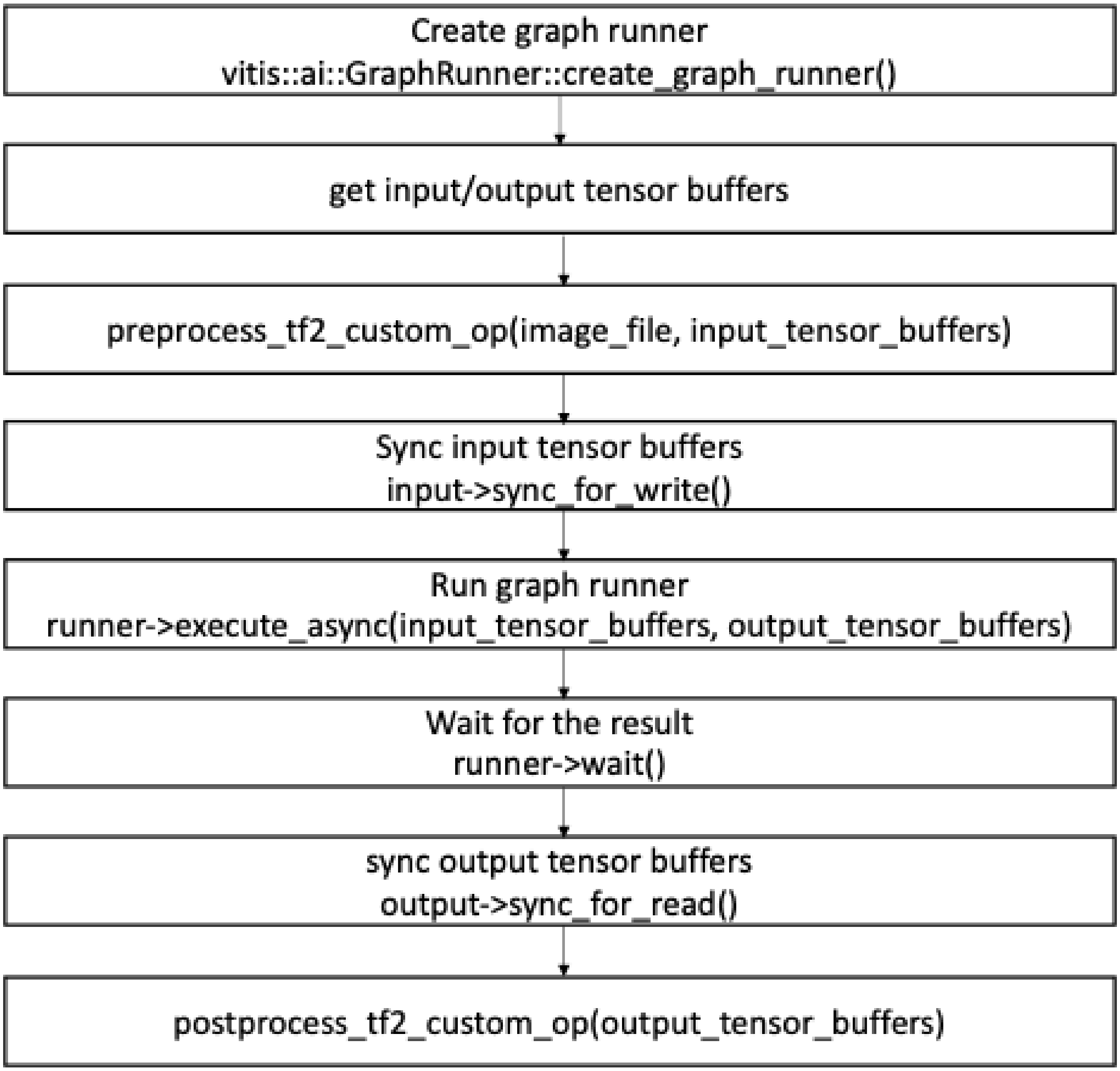
1. Create cpp source file, such as `tf2_custom_op_graph_runner.cpp`
2. Include the header: `<vitis/ai/graph_runner.hpp>`
3. Implement your model's pre-process, post-process and result-display-process in the following functions.

   ```
   static void preprocess_tf2_custom_op(…)
   static void postprocess_tf2_custom_op(…)
   static void print_result(…)
   ```

   Tips: You can copy the cpp source file from `tf2_custom_layer_graph_runner.cpp` and focus on the step 3 to implement the above 3 functions.

   The following shows the flow of `tf2_custom_op_graph_runner.cpp`

Figure 28: **The work flow of tf2_custom_op_graph_runner**



Part of the code is shown below.

```
  // create graph runner
  auto graph = xir::Graph::deserialize(xmodel_file);
  auto attrs = xir::Attrs::create();
  auto runner =
      vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
  CHECK(runner != nullptr);

  // get input/output tensor buffers
  auto input_tensor_buffers = runner->get_inputs();
  auto output_tensor_buffers = runner->get_outputs();

  // preprocess
```

```
  preprocess_tf2_custom_op(image_file, input_tensor_buffers);

  // sync input tensor buffers
  for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
                             input->get_tensor()->get_shape()[0]);
  }

  // run graph runner
  auto v = runner->execute_async(input_tensor_buffers,
output_tensor_buffers);
  auto status = runner->wait((int)v.first, -1);
  CHECK_EQ(status, 0) << "failed to run the graph";

  // sync output tensor buffers
  for (auto output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
                             output->get_tensor()->get_shape()[0]);
  }

  // postprocess
  postprocess_tf2_custom_op(output_tensor_buffers);
```

Build

1. Create `build.sh` to build the code, as shown below. You can also refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp/build.sh` , as shown below.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
    -o tf2_custom_op_graph_runner \
    tf2_custom_op_graph_runner.cpp \
    -lglog \
    -lxir \
    -lvart-runner \
    -lvitis_ai_library-graph_runner \
    ${OPENCV_FLAGS} \
    -lopencv_core \
    -lopencv_imgcodecs \
    -lopencv_imgproc
```

2. Execute `bash build.sh` to build the program on the target and the executable program `tf2_custom_op_graph_runner` will be generated.

Run

Before you run the demo, make sure the environment of the board has been set up correctly. Also, make sure the following files are generated or ready. Then copy them to the target.

- compiled model, such as tf2_custom_op.xmodel

- custom op library, such as libvart_op_imp_Mylayer.so
- test image from here
- executable program, such as tf2_custom_op_graph_runner

Copy the custom op library to `/usr/lib` on the target. Then, run the following command to test the model on the target.

```
./tf2_custom_op_graph_runner tf2_custom_op.xmodel sample.jpg
```

The following figure shows the result of running `tf2_custom_op_graph_runner`.

Figure 29: **tf2_custom_op_graph_runner example**

```
root@xilinx-zcu104-2021_1:~/tf2_custom_op_demo# ./tf2_custom_op_graph_runner tf2_custom_op.xmodel sample.jpg
model_file: tf2_custom_op.xmodel
image_file: sample.jpg
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1128 14:12:55.312837 27054 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: M
ylayer, is not defined in XIR. XIR creates the definition of this operator automatically. You should specify
the shape and the data_type of the output tensor of this operation by set_attr("shape", std::vector<int>) and
 set_attr("data_type", std::string)
score[0]    =  0
score[1]    =  0
score[2]    =  0
score[3]    =  0
score[4]    =  0
score[5]    =  0
score[6]    =  0
score[7]    =  0.992188
score[8]    =  0
score[9]    =  0
```

Pytorch Custom OP Model Example

Using Pointpillars model as an example, download the float model and code package from here and refer to `README.md` in the package to set up the environment .

Quantizing the Model

vai_q_pytorch provides a decorator to register an operation or a group of operations as a custom operation which is unknown to XIR.

```
# Decorator API
def register_custom_op(op_type: str, attrs_list: Optional[List[str]] =
None):
  """The decorator is used to register the function as a custom operation.
  Args:
  op_type(str) - the operator type registered into quantizer.
  The type should not conflict with pytorch_nndct
```

```
  attrs_list(Optional[List[str]], optional) -
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as padding,
dilation, stride and groups.
  The order of name in attrs_list should be consistent with that of the
arguments list.
  Default: None

"""
```

To quantize a model with custom op, two steps are requires to edit the code:

1. Move the target code into a function and change its calling accordingly. To Pointpillar model, replace the PointPillarsScatter model with a PPScatterV2 function. Check related code in `code/test/models/voxelnet.py` file.
2. Decorate this function with decorator API:

```
from pytorch_nndct.utils import register_custom_op
...

@register_custom_op("PPScatterV2", attrs_list=['ny', 'nx', 'nchannels'])
def PPScatterV2(ctx, voxel_features, coords, ny, nx, nchannels):
    '''
    input:
    voxel_features: B x 64 x 12000 x 1
    coords: B x 12000 x 4, 4 channels: [batch_idx, z_idx, y_idx, x_idx]
    '''
    batch_size = voxel_features.shape[0]
    # batch_canvas will be the final output.
    batch_canvas = []

    for b_idx in range(batch_size):
        # Create the canvas for this sample
        canvas = torch.zeros(nchannels, nx * ny,
dtype=voxel_features.dtype,
                             device=voxel_features.device)
        # Only include non-empty pillars

        batch_mask = coords[b_idx, :, 0] > -1
        this_coords = coords[b_idx, batch_mask, :]
        indices = this_coords[:, 2] * nx + this_coords[:, 3]
        indices = indices.type(torch.long)

        voxels = voxel_features[b_idx, :, batch_mask, 0]

        # Now scatter the blob back to the canvas.
        canvas[:, indices] = voxels
        # Append to a list for later stacking.
        batch_canvas.append(canvas)

    # Stack to 3-dim tensor (batch-size, nchannels, nrows*ncols)
    batch_canvas = torch.stack(batch_canvas, 0)
    # Undo the column stacking to final 4-dim tensor
    batch_canvas = batch_canvas.view(batch_size, nchannels, ny, nx)
    return batch_canvas
```

After the target custom op code has been prepared and decorated, add general vai_q_pytorch API functions (check the related code in code/test/test.py)

```
if quant_mode != 'float':
    max_voxel_num = config.eval_input_reader.max_number_of_voxels
    max_point_num_per_voxel =
model_cfg.voxel_generator.max_number_of_points_per_voxel
    aug_voxels = torch.randn((1, 4, max_voxel_num,
max_point_num_per_voxel)).to(device)
    # coors = torch.randn((max_voxel_num, 4)).to(device)
    coors = torch.randn((1, max_voxel_num, 4)).to(device)
    quantizer = torch_quantizer(quant_mode=quant_mode,
                                module=net,
                                input_args=(aug_voxels, coors),
                                output_dir=output_dir,
                                device=device,
                                )
    net = quantizer.quant_model
...
...
for example in iter(eval_dataloader):
...
    if quant_mode == 'test' and args.dump_xmodel:
        quantizer.export_xmodel(output_dir=output_dir, deploy_check=True)
        sys.exit()
...
...
if quant_mode == 'calib':
    quantizer.export_quant_config()
```

After all changes are ready, run script code/test/run_quant.sh to get quantization result files, including xmodel file to compiler (./quantized/VoxelNet_int.xmodel):

```
sh ./code/test/run_quant.sh
```

Compiling the Model

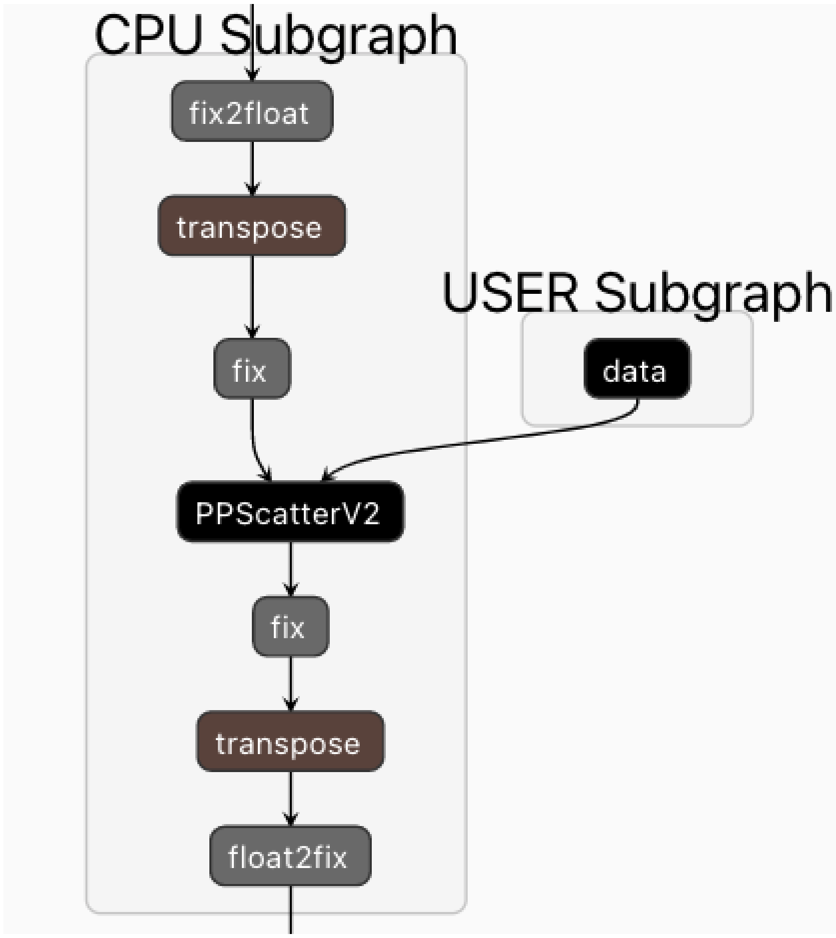
The following commands apply to PyTorch.

```
conda activate vitis-ai-pytorch
cd <path of Vitis-AI>/examples/custom_operator/pytorch_example/model/
quantized
vai_c_xir -x VoxelNet_int.xmodel -a /opt/vitis_ai/compiler/arch/DPUCZDX8G/
ZCU102/arch.json -o ./ -n pointpillars_custom_op
```

Custom OP Registration

Before custom op registration, you can use the latest Netron program to check the compiled model. From the following graph, PPScatter is assigned to the CPU. You have to implement and register `PPScatter` OP.

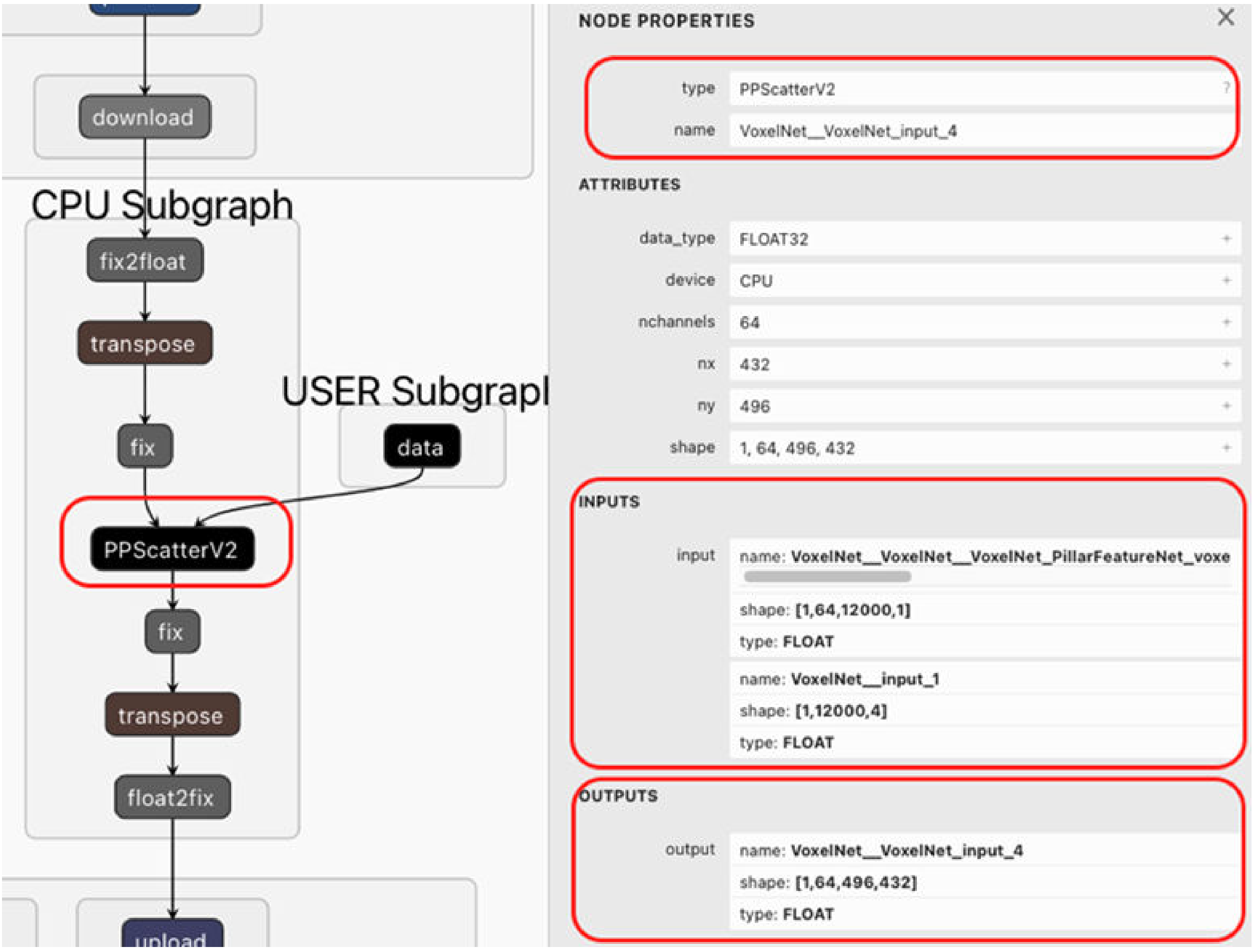
Figure 30: **PPScatter OP in CPU Subgraph**



Steps

1. Use `Netron` to open the compiled model and find the custom OP in CPU subgraph with op information.

Figure 31: **The inputs and outputs of PPScatter Op**



From the previous model structure image, you can find the OP type is `PPScatterV2`, which is the name of the custom OP that needs to be created.

You can also use `xdputil` to check the OP's detailed information. Run the following command to check the `custom_layer` OP.

```
xdputil xmodel pointpillars_custom_op.xmodel --op
VoxelNet__VoxelNet_input_4
```

2. Write your own implementation of this op.

   Custom OP registration supports both C++ and Python. The following shows how to implement the OP in C++. For the OP Python implementation, refer to `Vitis-AI/examples/custom_operator/pytorch_example/op_registration/python/`

   Note: There is `README.md` file in `Vitis-AI/examples/custom_operator/op_add` directory which illustrates the detailed steps on how to implement the custom op. You can refer to it on how to implement and register the custom OP.

   a. Create the `my_PPScatter_op.cpp` source file and put it under new folder `op_PPScatter`.

You can also copy one existed op and renamed to your op, as shown below. Then, rename `my_tanh_op.cpp` to `my_PPScatter_op.cpp`.

```
cp - r Vitis-AI/src/vai_library/cpu_task/examples/op_tanh/
op_PPScatter
```

b. Create the Makefile.

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_PPScatter_op.o: my_PPScatter_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so: $(OUTPUT_DIR)/
my_PPScatter_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -
lglog -lvitis_ai_library-runner_helper -lvart-runner -lxir
```

c. Write the implementation of the OP.

In my_PPScatter_op.cpp, use the construct function to initialize some variable; in this example, there is no variable need be initialized.

In the calculate() function, implementation your own logic. The logic is mainly getting input data from “inputs” variable, calculating the logic, writing output data to the “output” variable.

The code of `my_PPScatter_op.cpp` is shown below.

```
#include <vart/op_imp.h>

class MyPPScatterOp {
  public:
  MyPPScatterOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {
  // op and attrs is not in use.
}

int calculate(vart::simple_tensor_buffer_t output,
               std::vector<vart::simple_tensor_buffer_t<float>>
inputs) {
  CHECK_EQ(inputs.size(), 2);
  auto input_data_shape = inputs[0].tensor->get_shape();
  auto input_coord_shape = inputs[1].tensor->get_shape();
  auto output_shape = output.tensor->get_shape();
  CHECK_EQ(input_data_shape.size(), 4); // 1 12000 1 64 --> 1 64
12000 1
  CHECK_EQ(input_coord_shape.size(), 3); // 1 12000 4
  CHECK_EQ(output_shape.size(), 4); // 1 496 432 64 ---> 1 64 496 432

  auto coord_numbers = input_coord_shape[1];
  auto coord_channel = input_coord_shape[2];
  CHECK_EQ(coord_numbers, input_data_shape[2]);
```

```
  auto batch = output_shape[0];
  auto height = output_shape[2];
  auto width = output_shape[3];
  auto channel = output_shape[1];
  CHECK_EQ(input_data_shape[0], batch);
  CHECK_EQ(channel, input_data_shape[1]);

  auto output_idx = 0;
  auto input_idx = 0;
  auto x_idx = 0;

  memset(output.data, 0,
output_shape[0]*output_shape[1]*output_shape[2]*output_shape[3]*sizeo
f(float));

  for (auto n = 0; n < coord_numbers; n++) {
    auto x = (int)inputs[1].data[x_idx + 3];
    auto y = (int)inputs[1].data[x_idx + 2];
    if (x < 0) break; // stop copy data when coord x == -1 .
    for(int i=0; i < channel; i++) {
    output_idx =i*height*width + y*width+x;
    input_idx = n+i*coord_numbers;
    output.data[output_idx] = inputs[0].data[ input_idx ];
    }
    x_idx += coord_channel;
  }
  return 0;
}


public:
  const xir::Op* const op;
};

DEF_XIR_OP_IMP(MyPPScatterOp)
```

d. Build the library. The target directory is $(HOME)/build/custom_op/ . You can modify the path in Makefile.

   Running `make` with your Makefile, you'll see the custom defined op library is generated in $(HOME)/build/custom_op/, file name is `libvart_op_imp_PPScatterV2.so`.

e. Copy the `libvart_op_imp_PPScatterV2.so` to `/usr/lib` on the target.

3. Verify the Op on the target.

   a. Use `run_op` command in `xdputil` to test the op, as shown below.

   ```
   xdputil run_op pointpillars_op.xmodel VoxelNet__VoxelNet_input_4 -r
   ref -d dump
   ```

   Before running the above command, prepare the reference inputs of the op. After you run the command successfully, `VoxelNet__VoxelNet_input_4.bin` file will be generated.

   b. Compare the output with the golden file. The command is shown below.

   ```
    xdputil comp_float ref/VoxelNet__VoxelNet_input_4.bin dump/
   VoxelNet__VoxelNet_input_4.bin
   ```

If the OP implementation is successful, you will see the following result:

```
root@xilinx-zcu102-2021_2:~/pointpillars_custom_op# xdputil comp_float ref/VoxelNet__VoxelNet_input_4.bin dump/VoxelNet__VoxelNet_input_4.bin
float bin file comparation done.
golden file and dump file are the same!
```

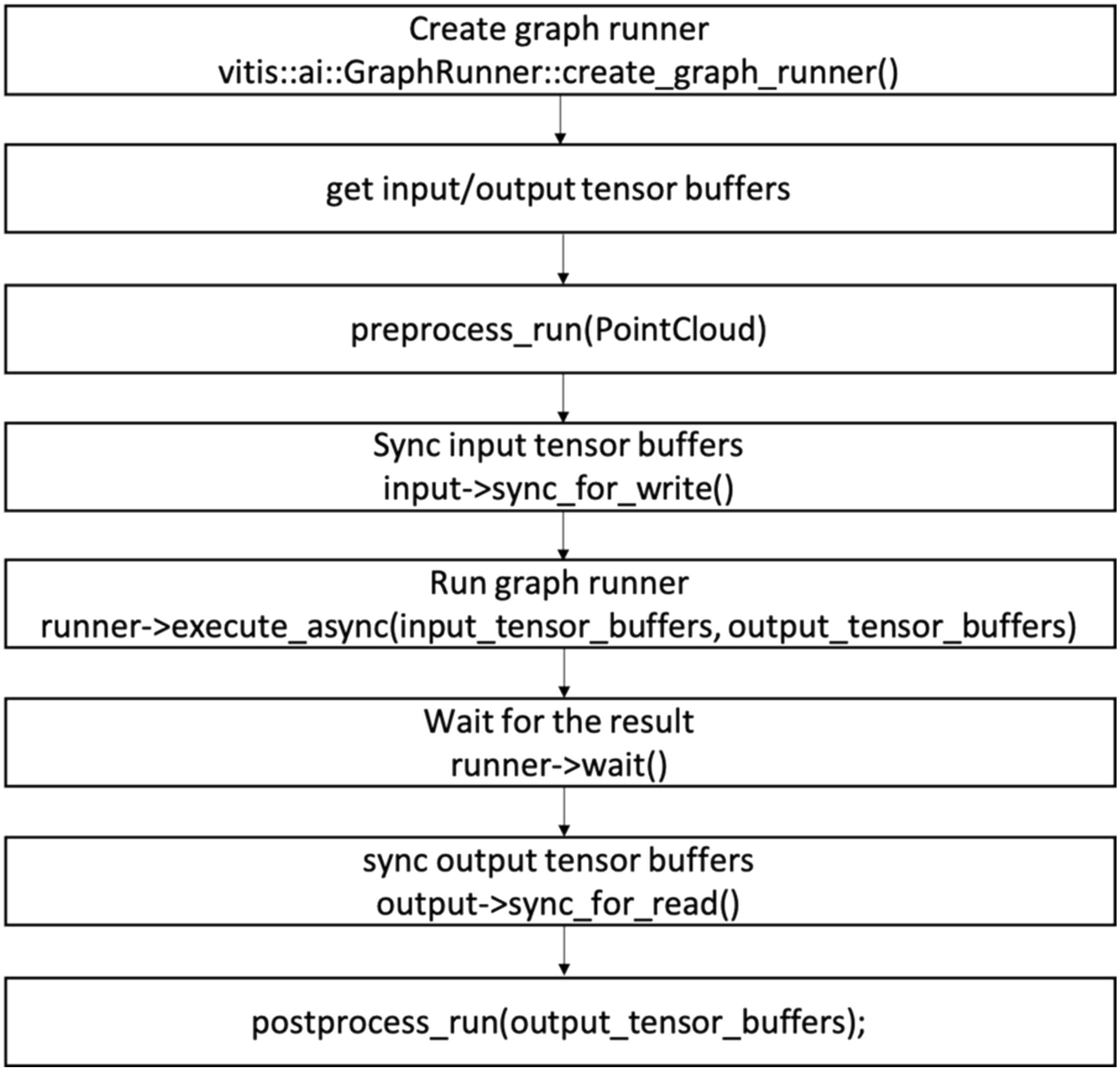
Deployment

This section describes the deployment of the pytorch model with custom op in `graph_runner` APIs. For `graph_runner` APIs, it supports both C++ and Python. You can refer to the `graph_runner` samples in `Vitis-AI/examples/vai_library/samples/graph_runner`.

1. Create a new directory to hold your test code.
2. Create source file and implement the following functions for this sample.
    a. Parameter parse and initialize
    b. Preprocess
    c. Model_run
    d. Postprocess

    The basic flow of this sample is shown in the following image.

Figure 32: **The basic flow of this sample**



3. Build the program.

   a. If your project is simple, such as only one .cpp source file, you can copy any existing build.sh from `Vitis-AI/examples/vai_library/samples/graph_runner` and modify it accordingly. Then, run the following command to build the program.

   ```
   cd <your sample folder>
   bash build.sh
   ```

   The following figure shows the build.sh of resnet50_graph_runner sample.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
        OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
        OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
        -o resnet50_graph_runner \
        resnet50_graph_runner.cpp \
        -lglog \
        -lxir \
        -lvart-runner \
        -lvitis_ai_library-graph_runner \
        ${OPENCV_FLAGS} \
        -lopencv_core \
        -lopencv_imgcodecs \
        -lopencv_imgproc
```

b. If your project is more complex, such as this sample, it's better to use CMakeLists.txt for easy compiling. For more details about CMakeLists.txt, refer to `Vitis-AI/examples/custom_operator/pytorch_example/deployment/cpp/pointpillars_graph_runner/CMakeLists.txt`

   Then, run the following command to build the program.

```
cd <your sample folder>
mkdir build
cd build
cmake ..
make
```

After a successful compilation, the executable program `sample_pointpillars_graph_runner` will be generated under `<your sample folder>`

4. Test the program

   Before you test the program, please copy the xmodel, test image, custom op library `libvart_op_imp_PPScatterV2.so` and executable program `sample_pointpillars_graph_runner` to the board. Put the custom op library under `/usr/lib`. Then, run the following command to do the test.

```
./sample_pointpillars_graph_runner ./
pointpillars_full_customer_op.xmodel sample_pointpillars.bin
```

The following shows the running result of this sample.

```
root@xilinx-zcu102-2021_2:~/pointpillars_graph_runner# ./
sample_pointpillars_graph_runner pointpillars_op.xmodel
sample_pointpillars.bin
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 05:59:20.517452 1307 tool_function.cpp:177] [UNILOG][WARNING] The
operator named VoxelNet__VoxelNet_input_4, type: PPScatterV2, is not
defined in XIR. XIR creates the definition of this operator
automatically. You should specify the shape and the data_type of the
output tensor of this operation by set_attr("shape", std::vector) and
set_attr("data_type", std::string)
result: 0
0 18.541065 3.999999 -1.732742 1.703191 4.419279 1.465484 1.679375
0.880797
0 34.522400 1.505865 -1.515198 1.503061 3.550991 1.420396 1.710625
0.851953
0 10.917599 4.705865 -1.622433 1.650789 4.350764 1.634866 1.632500
0.851953
1 21.338514 -2.400001 -1.681677 0.600000 1.963422 1.784916 4.742843
0.777300
0 57.891731 -4.188268 -1.536627 1.575194 3.780010 1.512004 2.007500
0.679179
```

If you want to profile the sample of custom op, use the environment variable `DEEPHI_PROFILING=1`, as shown below.

```
env DEEPHI_PROFILING=1 ./sample_pointpillars_graph_runner ./
pointpillars_full_customer_op.xmodel sample_pointpillars.bin
```

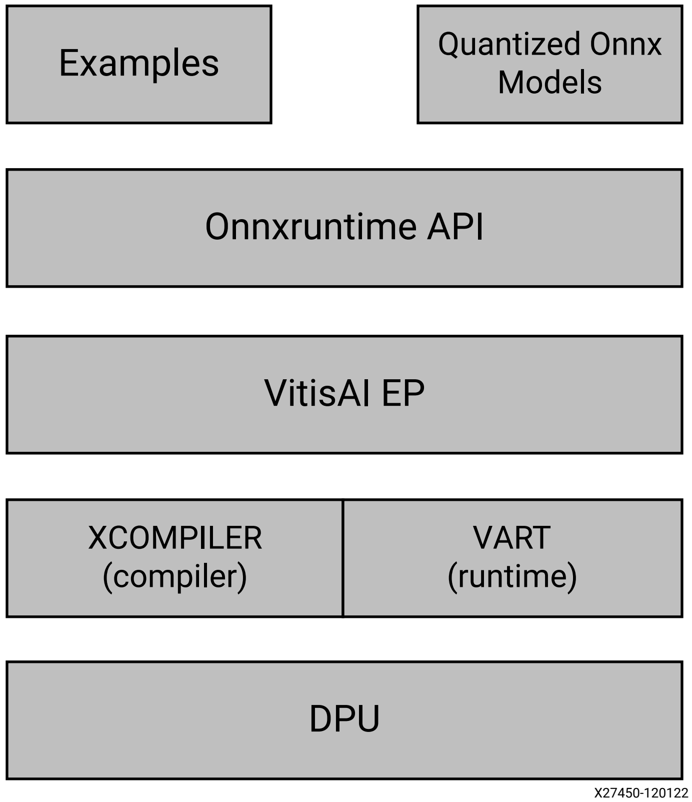
The profiling result is shown below.

```
I1130 01:29:53.038476 15571 cpu_task.cpp:163] CPU_UPDATE_INPUT : 5us
I1130 01:29:53.038684 15571 cpu_task.cpp:166] CPU_UPDATE_OUTPUT : 55us
I1130 01:29:53.038872 15571 cpu_task.cpp:169] CPU_SYNC_FOR_READ : 46us
I1130 01:29:53.039050 15571 cpu_task.cpp:181] CPU_OP_EXEC : 32us
I1130 01:29:53.039232 15571 cpu_task.cpp:181] CPU_OP_EXEC : 36us
I1130 01:29:53.039597 15571 cpu_task.cpp:181] CPU_OP_EXEC : 232us
I1130 01:29:53.066352 15571 cpu_task.cpp:181] CPU_OP_EXEC : 26575us
I1130 01:29:53.066745 15571 cpu_task.cpp:195] CPU_SYNC_FOR_WRITE : 1us
```

Programming with VOE

Vitis AI ONNX Runtime Engine, short for VOE, is a new feature in Vitis AI 3.0. It allows user to directly run the quantized ONNX model on the target board. `VitisAI EP` is provided to accelerate the inference with `Xilinx DPU`. The following is the overview of VOE in Vitis AI.

Figure 33: **VOE Overview**



In Vitis AI 3.0, there are more than 10 deployment examples based on ONNX runtime are provided. Users can find the examples in https://github.com/Xilinx/Vitis-AI/tree/v3.0/examples/vai_library/samples_onnx. The following shows how to use VOE to deploy the ONNX model step by step.

1. Prepare the quantized model in ONNX format. Users need to use the Vitis-AI quantizer to quantize the model and output the quantized model in ONNX format.
2. Download the ONNX runtime package vitis_ai_2022.2-r3.0.0.tar.gz and install it on the target board.

   ```
   tar -xzvf vitis_ai_2022.2-r3.0.0.tar.gz -C /
   ```

3. Use the ONNX Runtime `C++` API to create the application program. For the details of ONNX Runtime API, refer to https://onnxruntime.ai/docs/api/. The following shows the segmentation model deployment code snippet based on the C++ API.

   C++ example

   ```
   //Create a session
   //Select a set of execution provides(EP) if any, "VITISAI_EP" is selected
   env = Ort::Env(ORT_LOGGING_LEVEL_WARNING, "Segmentation");
   session_options = Ort::SessionOptions();
   CheckStatus(OrtSessionOptionsAppendExecutionProvider_VITISAI(session_opti
   ons,"")); std::string model_name_(model_name);
   session = std::unique_ptr<Ort::Experimental::Session>( new
   ```

```
Ort::Experimental::Session(env, model_name_, session_options));

//Do the pre-process and set the input
cv::Mat resize_image;
auto height = input_shapes[0][2];
auto width = input_shapes[0][3];
auto size = cv::Size((int)width, (int)height);
cv::resize(image[0], resize_image, size);
set_input_image(resize_image, input_tensor_values.data());
if (input_tensors.size())
{ input_tensors[0] = Ort::Experimental::Value::CreateTensor<float>
(input_tensor_values.data(), input_tensor_values.size(),
input_shapes[0]); }
else
{ input_tensors.push_back( Ort::Experimental::Value::CreateTensor<float>(
input_tensor_values.data(), input_tensor_values.size(),
input_shapes[0])); }

//Run the session
output_tensors = session->Run(session->GetInputNames(), input_tensors,
session->GetOutputNames());
output_tensor_ptr[0] = output_tensors[0].GetTensorMutableData<float>();

//Get the output and do the post-process
auto oc = output_shapes[0][1];
auto oh = output_shapes[0][2];
auto ow = output_shapes[0][3];
auto hwc = permute(output_tensor_ptr[0], oc, oh, ow);
cv::Mat result(oh, ow, CV_8UC1);
max_index_c(hwc.data(), oc, oh * ow, result.data);
```

4. Create a `build.sh` file as shown below, or copy one from the Vitis AI Library ONNX examples and modify it. Then, build the program.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

lib_x=" -lglog -lunilog -lvitis_ai_library-xnnpp -lvitis_ai_library-
model_config -lprotobuf -lxrt_core -lvart-xrt-device-handle -lvaip-core -
lxcompiler-core -labsl_city -labsl_low_level_hash -lvart-dpu-controller -
lxir -lvart-util -ltarget-factory -ljson-c"
lib_onnx=" -lonnxruntime"
lib_opencv=" -lopencv_videoio -lopencv_imgcodecs -lopencv_highgui -
lopencv_imgproc -lopencv_core "

if [[ "$CXX"  == *"sysroot"* ]];then
 inc_x="-I=/usr/include/onnxruntime -I=/install/Release/include/
onnxruntime -I=/install/Release/include -I=/usr/include/xrt"
 link_x="  -L=/install/Release/lib"
else
 inc_x=" -I/usr/include/onnxruntime  -I/usr/include/xrt"
 link_x="  "
fi

name=$(basename $PWD)

CXX=${CXX:-g++}
$CXX -O2 -fno-inline -I. \
 ${inc_x} \
```

```
${link_x} \
-o ${name}_onnx -std=c++17 \
$PWD/${name}_onnx.cpp \
${OPENCV_FLAGS} \
${lib_opencv} \
${lib_x} \
${lib_onnx}
```

5. Copy the executable program and the quantized ONNX model to the target. Then, run the program.

   Note: For the ONNX model deployment, the input model is the quantized ONNX model. It will do the model compiling online first when you run the program. It may take some time during compiling the model.

Multi-FPGA Programming

Most modern servers have multiple Xilinx® Alveo™ cards, and you would want to take advantage of scaling up and scaling out deep-learning inference. Vitis AI provides support for multi-FPGA servers using the following building blocks.

XRM

The Xilinx Resource Manager (XRM) manages and controls Xilinx FPGA resources on a machine. With the Vitis AI release, installing XRM is mandatory for running a deep-learning solution using XRM. XRM is implemented as a server-client paradigm. It is an add-on library on top of the XRT to facilitate multi-FPGA resource management. XRM is not a replacement for the Xilinx XRT. The feature list for XRM is as follows:

- Enables multi-FPGA heterogeneous support
- C++ API and CLI for the clients to allocate, use, and release resources
- Enables resource allocation at FPGA, compute unit (CU), and service granularity
- Auto-release resource
- Multi-client support: Enables multi-client/users/processes request
- XCLBIN-to-DSA auto-association
- Resource sharing amongst clients/users
- Containerized support
- User defined function
- Logging support

https://github.com/Xilinx/XRM

AI Kernel Scheduler

Real world deep learning applications involve multi-stage data processing pipelines which include many compute intensive preprocessing operations like data loading from disk, decoding, resizing, color space conversion, scaling, and cropping multiple ML networks of different kinds like CNN, and various post-processing operations like NMS.

The AI kernel scheduler (AKS) is an application to automatically and efficiently pipeline such graphs without much effort from the users. It provides various kinds of kernels for every stage of the complex graphs which are plug and play and are highly configurable. For example, preprocessing kernels like image decode and resize, CNN kernel like the Vitis AI DPU kernel and post processing kernels like SoftMax and NMS. You can create their graphs using kernels and execute their jobs seamlessly to get the maximum performance.

For more details and examples, see the Vitis AI GitHub (AI Kernel Scheduler).

Apache TVM, Microsoft ONNX Runtime, and TensorFlow Lite

In addition to VART and related APIs, Vitis AI has integrated with the Apache TVM and Microsoft ONNX Runtime and TensorFlow Lite frameworks for improved model support and automatic partitioning. This work incorporates community driven machine learning framework interfaces that are not available through the standard Vitis AI compiler and quantizers. In addition, it incorporates highly optimized CPU code for x86 and Arm® CPUs, when certain layers may not yet be available on Xilinx DPUs. These frameworks are supported on all Zynq® UltraScale+™ MPSoCs and Alveo™-based DPUs.

Apache TVM

Apache TVM is an open source deep learning compiler stack focusing on building efficient implementations for a wide variety of hardware architectures. It includes model parsing from TensorFlow, TensorFlow Lite (TFLite), Keras, PyTorch, MxNet, ONNX, Darknet, and others. Through the Vitis AI integration with TVM, Vitis AI is able to run models from these frameworks. TVM incorporates two phases. The first is a model compilation/quantization phase which produces the CPU/FPGA binary for your desired target CPU and DPU. Then by installing the TVM Runtime on your Cloud or Edge device, the TVM APIs in Python or C++ can be called to execute the model.

To read more about Apache TVM, see https://tvm.apache.org.

Vitis AI provides tutorials and installation guides on Vitis AI and TVM integration on theVitis AI GitHub repository: https://github.com/Xilinx/Vitis-AI/tree/v3.0/third_party/tvm.

Microsoft ONNX Runtime

Microsoft ONNX Runtime is an open source inference accelerator focused on ONNX models. It is the platform Vitis AI has integrated with to provide first-class ONNX model support, which can be exported from a wide variety of training frameworks. It incorporates very easy to use runtime APIs in Python and C++ and can support models without requiring the separate compilation phase that TVM requires. Included in ONNXRuntime is a partitioner that can automatically partition between the CPU and FPGA further enhancing the ease of model deployment. Finally, it also incorporates the Vitis AI quantizer in a way that does not require separate quantization setup.

To read more about Microsoft ONNX Runtime, see https://microsoft.github.io/onnxruntime/.

Vitis AI provides tutorials and installation guides on Vitis AI and ONNX Runtime integration on the Vitis AI GitHub repository: https://github.com/Xilinx/Vitis-AI/tree/v3.0/third_party/onnxruntime.

TensorFlow Lite

TensorFlow Lite (TFLite) is an open source inference accelerator focused on TensorFlow Lite models. It is the platform Vitis AI has integrated with to provide first-class TFLite model support, which can be exported from TensorFlow. It incorporates a very easy to use runtime APIs in Python and C++ and can support models without requiring the separate compilation phase that TVM requires. Included in TensorFlow Lite is a partitioner that can automatically partition between the CPU and FPGA further enhancing the ease of model deployment. Finally, it also incorporates the Vitis AI quantizer in a way that does not require separate quantization setup.

To read more about TensorFlow Lite, see https://tensorflow.org/lite.

Vitis AI provides tutorials and installation guides on Vitis AI and TensorFlow Lite integration on the GitHub repository: https://github.com/Xilinx/Vitis-AI/tree/v3.0/third_party/tflite.

Using WeGO

WeGO (Whole Graph Optimizer) is a Vitis AI early access feature that offers a smooth solution to deploy TensorFlow and PyTorch models on cloud DPU by integrating Vitis AI development kit with TensorFlow and PyTorch frameworks. In addition to TensorFlow 1.15, WeGO begins to support TensorFlow 2.8 and PyTorch 1.10 since VAI 2.5. WeGO is regarded as Vitis AI in-framework inference solution, compared with non-framework approach using VART APIs or AI Library.

WeGO automatically performs subgraph partitioning for the Vitis AI quantized models and applies optimizations and acceleration for the cloud DPU compatible subgraphs. The remaining DPU unsupported parts of the graph, called CPU subgraphs, are dispatched to TensorFlow or PyTorch framework for CPU native execution. WeGO takes care of the whole graph optimization, compilation, and run-time subgraphs' dispatch and execution. This process is entirely transparent to the end-users, making it easy to use.

Using WeGO is a straightforward transition from training to inference for model designers. WeGO provides a Python programming interface to deploy the quantized models over the TensorFlow or PyTorch framework. This makes it possible to maximally reuse the Python code (including pre-processing and post-processing) developed during models training phase with TensorFlow or PyTorch. It substantially improves the productivity for models' deployment and evaluation over cloud DPUs.

***Note*:** Currently, WeGO only supports cloud DPU target DPUCVDX8H on VCK5000 Prod platform.

For WeGO examples and more information on how to apply TensorFlow and PyTorch to deploy models, see Vitis AI GitHub repo.

WeGO Programming Interface

PyTorch

WeGO-Torch is a sub-project of WeGO, which is designed to improve Vitis AI EoU by integrating Vitis AI toolchain into PyTorch framework. WeGO-Torch follows standard WeGO's workflow, which means it will perform model partition, compilation and inference automatically without extra efforts from the users.

WeGO-Torch Python API will accept a quantized TorchScript module (which is generated by Vitis AI PyTorch quantizer) as an input and returns an optimized TorchScript module, which can be used for later inference immediately. The general steps to exploit WeGO-Torch for Vitis-AI acceleration in PyTorch are listed below:

1. Import WeGO-Torch python module into your application.
2. Load the quantized TorchScript module using standard PyTorch's Python API.
3. Compile the quantized TorchScript module using wego-torch module's API by providing the TorchScript module and input shape as inputs. It returns an optimized TorchScript module as the compiled result.
4. Run inference using the optimized TorchScript module.

The following code snippets show the basic usage of Python APIs of WeGO-Torch:

```
import torch
# Step 1: Import WeGO-Torch python module
import wego_torch

# Step 2: Load the quantized torchscript module generated by vitis-ai
PyTorch quantizer.
model_path = <quantized_torchscript_model_path>
mod = torch.jit.load(model_path)

# Step 3: Create an optimized TorchScript module through WeGO-Torch's API.
wego_mod = wego_torch.compile(mod,
  wego_torch.CompileOptions(
  inputs_meta = [ wego_torch.InputMeta(torch.float, [1, 3, 224, 224]) ]
  )
)

# Step 4: Run inference using the optmized TorchScript module.
result = wego_mod(input)
```

WeGO-Torch Python Classes and APIs

Core Python Classes

wego_torch.CompileOptions(accuracy_mode : wego_torch.AccuracyMode = wego_torch.AccuracyMode.Default, inputs_meta = [], partition_options : wego_torch.PartitionOptions = None)

A python class object representing WeGO-Torch compilation options. It will be created and passed into wego_torch.compile interface by users.

Table 28: **Constructor Parameters**

Parameter	Description	Values
accuracy_mode	Decides the accuracy mode.	• *wego_torch.AccuracyMode.Default*: This is the default value for accuracy mode. In this mode, WeGO-Torch will remove all redundant fixneurons to improve performance after the compilation process. The redundant fixneurons exist because even some operators are quantized. Since they are not supported by the DPU target on-board, they will be dispatched into CPU for inference. These fixneurons can be removed from the model to improve the end-to-end performance if the accuracy can meet our requirement. • *wego_torch.AccuracyMode.ReserveReduantFixNeurons*: If this value is provided, WeGO-Torch will keep all the redundant fixneurons in the model rather than removing them. Although removing redundant fixneurons improves performance, there is a possibility of accuracy issues in some cases. Users are encouraged to try this value if the end-to-end accuracy cannot meet the requirement after your model is compiled by WeGO-Torch.
inputs_meta	A list of `wego_torch.InputMeta` for each input of the model.	See the following section for more details about wego_torch.InputMeta type.
partition_options	Partition options with type `PartitionOptions`.	See the following section for more details about it.

wego_torch.InputMeta (dtype = None, input_shape = [])

Meta Information for describing inputs of the quantized model. Due to the limitations of Vitis AI toolchain, WeGO-Torch only supports compilation with static type and shape. The user is required to pass the date type and shape information for each input explicitly to enable WeGO-Torch for type and shape inference.

Table 29: **Constructor Parameters**

Parameter	Description	Values
dtype	Data type of the current input tensor. It can be torch.int32, torch.float or torch.bool.	
input_shape	Input shape of current input tensor.	

wego_torch.PartitionOptions (wego_subgraph_min_ops_number = 0, extra_accel_op_list = [])

Options for WeGO partiton configuration.

Table 30: **Constructor Parameters**

Parameter	Descriptions
wego_subgraph_min_ops_number	Currently WeGO uses greedy method to dispatch operators into DPU as long as they are supported by DPU. It may result in the following issues: • If there are a lot of operators not supported by DPU, the whole model may be partitioned into many DPU subgraphs and CPU subgraphs. If each DPU subgraph only contains a minor number of operators, then dispatching these subgraphs into DPU for execution may lead to performance issues due to frequent memory transfer between the host and device. • WeGO will allocate device buffer for each of DPU subgraphs. There might be buffer overflow issue when the model is large and there are many DPU subgrahs after partition. Added the following option: `wego_subgraph_min_ops_number` in WeGO to control dispatching a DPU subgraph into the DPU for execution. If the number of operators in a DPU subgraph is below or equal to the `wego_subgraph_min_ops_number` threshold, WeGO will dispatch the subgrah into the CPU side for execution even if all operators in the subgraph can be supported by DPU. ***Note***: If `wego_subgraph_min_ops_number` is 0, then there are no limitations.
extra_accel_op_list	DPU can support diverse DL operators but with some limitations(For example. DPUCVDX8H_ISA1_F2W4_4PE can only support convolution with kernel 1-16 and stride 1-4). WeGO leverages a DPU limitation check engine to decide a operator can be supported by DPU or not when performing partition. But for some operators, the rules to decide whether they are supported or not are very complicated. To avoid introducing too much overhead, WeGO won't dispatch them into DPU by default but relying on users to explicitly specify the operatos' type desiring for acceleration in the `extra_accel_op_list`. Currently the following operators can be specified through `extra_accel_op_list` for DPU execution: • aten::mul • aten::mean • aten::linear ***Note***: If errors occur during the compilation in WeGO after specifying the operator(s) in the `extra_accel_op_list` list, it indicates that the supplied operator(s) cannot be accelerated by DPU. Otherwise they can be.

wego_torch.TargetInfo()

A python class object which will wrap DPU target information through which you can get the batch, name, and the fingerprint of the DPU target on-board.

Note: users should not create this object by themselves but should rely on the API `wego_torch.get_target_info()` to return this object, which contains diverse property fields:

1. Batch: batch size supported by the DPU target on-board.
2. Name: name of the DPU target.
3. Fingerprint: fingerprint of the DPU target on-board.

The general way to use wego_torch.TargetInfo is shown below:

```
import wego_torch
...
# Detect the DPU target on-board and return an object with type
wego_torch.TargetInfo.
target_info = wego_torch.get_target_info()
# The target_info object can be printed directly.
print(target_info)
# Retrieve diverse propery fields of the DPU target.
batch, name, fingerprint = target_info.batch, target_info.name,
target_info.fingerprint
...
```

Core Python APIs

Table 31: **wego_torch.compile(module: Any, options: wego_torch.CompileOptions)**

Description	Parameters	Return
Compiles a pytorch torchscript module for Vitis AI acceleration.	• module: A quantized PyTorch module with type torch.jit.ScriptModule to be compiled. • options:Compiler options for WeGO-Torch compilation purpose, with type wego_torch.CompileOptions. See core classes section for more details about this object class type.	An optimized Torchscript Module.

Table 32: **wego_torch.get_target_info()**

Description	Parameters	Return
Detects DPU target on-board and return the target information.	None	A wego_torch.TargetInfo object. See core classes section for more details about this object class type.

Table 33: **wego_torch.version()**

Description	Parameters	Return
Get WeGO-Torch version.	None	A raw string representing wego_torch version information.

WeGO-PyTorch Limitations

The WeGO-Torch project is in early access state with a few known usage issues:

1. WeGO-Torch cannot support RCNN models(with control-flow) because:
    a. There is dynamic shape issue in such models(shape of the tensors in the model may change during runtime when different images are provided as model's input, such as RCNN models), to deploy them in WeGO. Some modifications must be performed manually to remove this constraint.
    b. Such models usually accept `Tensor []` as input type and it's not supported by WeGO's compile API. On the other hand, using `Tensor []` as input type means the float model itself is batch-sensitive and the quantized models through tracing are different when different batch size are used during torchscript tracing phase. To deploy these models in WeGO:
        i. Replace `Tensor []` with `Tensor` or `Tensor, Tensor, ...` (when the number of inputs is known) as input type in the original float model.
        ii. The batch size used for inference in WeGO must be the same as the one used in export phase during quantization.
2. WeGO-Torch currently only covers a subset of operators that cloud DPUs can support, which means WeGO-Torch will dispatch some operators into CPU for execution even if these operators can be supported by Cloud DPUs.

Examples

For WeGO-Torch examples, see Vitis AI GitHub page.

TensorFlow 2.x

WeGO-TensorFlow2.x is a sub-project of WeGO, which is designed to improve the Vitis AI EoU by integrating the Vitis AI toolchain into TensorFlow 2.x framework. For VAI 2.5, TensorFlow v2.8.0 is supported. The input for WeGO-TensorFlow2.x is a quantized model with HDF5 format usually named as quantized.h5, which is generated by vai_q_tensorflow2 quantizer. The WeGO core API create_wego_model() automatically converts the quantized Keras model into new concrete function where cloud DPU compatible subgraphs are transformed into TensorFlow operator with the kind of VaiWeGOOp.

The whole WeGO-TensorFlow2.x inference can be abstracted into the following for steps:

1. Import WeGO tensorflow2.x Python module into the application.

2. Get batch info of DPU target from vitis_vai.get_target_info() for inputs batching process.
3. Create wego model with vitis_vai.create_wego_model() to get concrete function.
4. Run concrete function.

WeGO-TensorFlow2.x Python APIs

Core Python Classes

DeviceInfo()

An object which will wrap DPU target information.

Note: Users should not create this object by themselves but should rely on the API vitis_vai.get_target_info() to return this object, which will contain diverse property fields:

- batch: batch size supported by the DPU target on-board.
- target: name of the DPU target.
- fingerprint: fingerprint of the DPU target on-board.

The general way to use DeviceInfo is:

```
from tensorflow.compiler import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

Core Python APIs

Table 34: **get_target_info()**

Description	Parameters	Return
Gets target information including batch, fingerprint, target name. You can use info for batching or get target name information.	None	A DeviceInfo object. See core classes section for more details about this object type.

Table 35: **create_wego_model(input_h5, feed_dict={}, accuracy_mode=vitis_vai.enums.AccuracyMode.Default)**

Description	Parameters	Return
Creates WeGO model, convert Keras h5 file into concrete function	1. input_h5: Path to the h5 file. 2. feed_dict: Infer shape configuration when input model without fixed input shape. 3. accuracy_mode: • vitis_vai.enums.AccuracyMode.Default: Inference without CPU FixNeuron. • vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons: Inference with CPU FixNeruon	New concrete function with VaiWeGOOps. ***Note*:** WeGO eliminates CPU FixNeurons operators within quantized model to achieve optimal performance by default. However for those models containing many CPU FixNeurons operators, the models' accuracy maybe decrease by deploying them with default value(vitis_vai.enums.AccuracyMode.Default).In such cases, you can switch to vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons to achieve better accuracy.

Environment Variable

WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE

This environment variable can be enabled when some operators need to rely on batchsize to infer static shapes. Export WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1 will set batch size to 1. For example, the static shape of the reshape operator cannot be obtained for some models (For example, ssd_resnet_50_fpn_coco_tf model with input shape [-1,640,640,3]), resulting in an error when some WeGO subgraph is compiled by Vitis AI toolchain. The error message is as follows:

```
AssertionError: [ERROR] Invalid shape of input layer: shape: [1, -1, -1,
256] (N,H,W,C), name: input1
[INFO] parse raw model     :  0%|          | 0/52 [00:00<?, ?it/
s]
*** Check failure stack trace: ***
```

To solve this problem, you need to set the environment variables as follows before running the sample to enable WeGO aggressive shape inference:

```
export WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1
```

Otherwise, you don't need this environment variable. Or, cancel the environment variable that has been set by the following command.

```
unset WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE
```

Examples

For WeGO-TensorFlow 2.x samples, see Vitis AI GitHub page.

TensorFlow 1.x

WeGO-TensorFlow1.x is a sub-project of WeGO, which is designed to improveVitis AI EoU by integrating the Vitis AI toolchain into TensorFlow 1.x framework. Vitis AI 2.5 supports TensorFlow v1.15. The input for WeGO-TensorFlow1.x is the quantized model usually named as quantize_eval_model.pb, which is generated by vai_q_tensorflow. The core WeGO API create_wego_graph() automatically converts the quantized graph into a new TensorFlow graph called as WeGO graph, where the cloud DPU compatible subgraphs are transformed into TensorFlow operator with the kind of VaiWeGOOp.

The whole WeGO-TensorFlow1.x inference can be abstracted into the following for steps

1. Execute some graph-level optimizations on the original graph to meet DPU-specific requirements.
2. Traverse the whole graph of the input quantized model and detect nodes which are supported by cloud DPU.
3. Perform graph auto-partitioning over the quantized graph over the node list detected in step 2.
4. Transform all cloud DPU compatible subgraphs into new TensorFlow nodes with kind of VaiWeGOOp within the input quantized model.
5. Return the optimized new WeGO graph and then invoke TensorFlow sess.run() to execute the whole graph.

WeGO-TensorFlow 1.x Python APIs

Core Python Classes

DeviceInfo()

An object which will wrap DPU target information.

***Note*:** Users should not create this object by themselves but should rely on the API vitis_vai.get_target_info() to return this object, which will contain diverse property fields:

- batch: batch size supported by the DPU target on-board.
- target: name of the DPU target.
- fingerprint: fingerprint of the DPU target on-board.

The general way to use DeviceInfo is:

```
from tensorflow.contrib import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

Core Python APIs

Table 36: **get_target_info()**

Description	Parameters	Return
Gets target information including batch, fingerprint, target name. You can use info for batching or get target name information.	None	A DeviceInfo object. See core classes section for more details about this object type.

Table 37: **create_wego_graph(input_graph_def, feed_dict={}, accuracy_mode= vitis_vai.enums.AccuracyMode.Default)**

Description	Parameters	Return
Python wrapper for the VAI transformation.	1. input_graph_def: GraphDef object containing a model to be transformed. 2. feed_dict: Infer shape configuration when input model without fixed input shape. 3. accuracy_mode: • vitis_vai.enums.AccuracyMode. Default: Running without CPU FixNeuron. • vitis_vai.enums.AccuracyMode. ReserveReduantFixNeurons: Running with CPU FixNeruon	New GraphDef with VaiWeGOOps placed in graph replacing subgraphs. ***Note*:** WeGO eliminates CPU FixNeurons operators within quantized model to achieve optimal performance by default. However for those models containing many CPU FixNeurons operators, the models' accuracy maybe decrease by deploying them with default value(vitis_vai.enums.AccuracyMode.Default).In such cases, you can switch to vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons to achieve better accuracy.

Environment Variable

WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE

This environment variable is used both by WeGO TensorFlow 1.x and WeGO TensorFlow 2.x. Refer to WeGO TensorFlow 2.x section for its usage.

Examples

For WeGO-TensorFlow 1.x samples, see Vitis AI GitHub page.

On-the-fly Quantization in WeGO

In the original WeGO workflow, since WeGO will only support a quantized INT8 model as its input, a separate quantization flow should be executed first by leveraging the Vitis AI quantizer explicitly to quantize the float32 model into an INT8 model. It creates the need to perform extra tasks for the users such performing conda environment switch operations between quantizer and WeGO, figuring out the relationship between Vitis AI quantizer and WeGO. To improve the ease of use and make the entire process from quantization to deployment smoother, WeGO has integrated Vitis AI quantizer into its flow, enabling on-the-fly quantization when a float32 model is offered as WeGO's input. Besides the original WeGO API for compilation, a new API is introduced in WeGO for quantization purposes and the quantizer details are transparent to the end users. The quantization integration in WeGO is in early stage, so there are some limitations:

1. Only PTQ (Post Training Quantization) is supported in the integration flow now. If accuracy is far from expected, fine-tuning or QAT (Quantization Aware Training) must be used to improve the accuracy by following the native Vitis AI quantization flow.
2. Only CPUs are adopted for quantization in WeGO and currently GPUs devices are not supported. This may introduce some issues when quantizing large models, which will consume a lot of time.

Quantization APIs

This section will introduce the WeGO's API for PTQ quantization targeting different frameworks.

PyTorch

Quantization API

```
wego_torch.quantize(
module: torch.nn.Module,
input_shapes: Sequence[Sequence],
dataloader: Iterable,
calibrator: Callable[[torch.nn.Module, Any, int, torch.device], None],
export_dataloader: Iterable = None,
device: torch.device = torch.device("cpu"),
output_dir: str = "quantize_result",
bitwidth: int = None,
quant_config_file: Optional[str] = None,
*args, **kwargs) -> torch.jit.ScriptModule
```

This function will quantize a torch float model with Post Training Quantization (PTQ) method and a quantized TorchScript Module will be returned for WeGO compilation usage.

If PTQ cannot achieve the required accuracy, you may need to consider using Quantization Aware Training (QAT) with Vitis AI Quantizer API. For in-depth understanding of the quantization process please see Quantizing the Model part in user guide.

Parameters

- **module:** `(torch.nn.Module)` An input pytorch float model.
- **input_shapes:** `(Sequence[Sequence]` Input shapes for the model- a sequence of lists or tuples.
- **dataloader:** `(Iterable)` Dataloader for calibration dataset. It must be an iterable. API will iterate through it and pass the returned values to calibrator.
- **calibrator:** `(Callable)` Callable object to do batch data pre-processing and forwarding. Get batch data from dataloader, preprocess it if necessary, and use module to forward it. This calibrator will be called N + 1 times in calibration and export stages.
    - Stage 1 is for calibration. In this stage your dataloader will be iterated, data passed through the module to collect quantization statistics. Calibrator will be called N times(N = len(dataloader)). At stage 1, if you didn't pass the optional export_dataloader(see below), first batch returned by dataloader will be saved and later used by stage 2. In this case, ensure the first batch is unchanged by calibrator or iteration side effects.
    - Stage 2 is for quantized torchscript module export. In this stage calibrator will only be called once with one batch of data. If you pass in an export_dataloader, this export_dataloader will be iterated and only the first batch will be used. Program breaks out of iteration after processing the first batch. If you didn't pass in an export_dataloader, the saved first batch from stage 1 will be used.

      Calibrator arguments:

        - **module:** `(torch.nn.Module)` Module for quantization. This will be a modified version of the module you passed in, with the necessary mechanisms to collect data statistics. You should use this module instead of the original float model to forward your data.
        - **batch_data:** `(Any)` Batch data returned from dataloader.
        - **batch_index:** `(int)` Index of the batch. Use it if necessary
        - **device:** `(torch.device)` Device to use for forward. Currently only support CPU.

      ***Note*:** Extra positional and keyword arguments to quantize API will be forwarded to calibrator. For more information, see Quantizing the Model.

- **export_dataloader:** `(Iterable)` An optional dataloader for the export stage. Default value is None. If None, will use first batch saved from stage 1.
- **device:** `(torch.device)` Device to use for calibration. Currently only support CPU.
- **output_dir:** `(str)` A temporary working directory. The default value is quantize_result. Some intermediary files will be saved here.
- **bitwidth:** `(int)` Global quantization bit width. The default value is 8.

- **quant_config_file:** `(str)` Path to quantizer configuration file. The default value is None.
- **args:** Extra positional arguments to pass to calibrator.
- **kwargs:** Extra keyword arguments to pass to calibrator.

For more information on how to use on-the-fly quantization in WeGO, see WeGO examples .

TensorFlow 2.x

Quantization API

```
vitis_vai.quantize(
input_float,
quantize_strategy = 'pof2s',
custom_quantize_strategy = None,
calib_dataset = None,
calib_steps = None,
calib_batch_size = None,
save_path = './vai_wego/quantized.h5',
verbose = 0,
add_shape_info = False,
dump = False,
dump_output_dir = './vai_dump/')
```

This function performs the post-training quantization (PTQ) of the float model, including model optimization, weights quantization, and activation quantize calibration.

Parameters

- **input_float:** A `tf.keras.Model` float object to be quantized.
- **quantize_strategy:** A string object of the quantize strategy type. Available values are *pof2s*, *pof2s_tqt*, *fs* , and *fsx*. *pof2s* is the default strategy that uses power-of-2 scale quantizer and the Straight-Through-Estimator. *pof2s_tqt* is a strategy introduced in Vitis AI 1.4 which uses Trained-Threshold in power-of-2 scale quantizers and may generate better results for QAT. *fs* is a new quantize strategy introduced in Vitis AI 2.5 that does float scale quantization for inputs and weights of Conv2D, DepthwiseConv2D, Conv2DTranspose, and Dense layers. *fsx* quantize strategy does quantization for more layer types than *fs* quantize strategy, such as Add, MaxPooling2D, and AveragePooling2D. Moreover, it also quantizes the biases and activations.

  Note:

  - *pof2s_tqt* strategy should only be used in QAT and be used together with init_quant=True to get the best performance.
  - *fs* and *fsx* strategy are designed for target devices with floating-point supports. DPU does not have floating-point support now, so models quantized with these quantize strategies cannot be deployed to them.
- **custom_quantize_strategy:** A string object, the file path of custom quantize strategy JSON file.

- **calib_dataset:** A *tf.data.Dataset*, *keras.utils.Sequence*, or *np.numpy* object, the representative dataset for calibration. You can use full or part of eval_dataset, train_dataset, or other datasets as calib_dataset.
- **calib_steps:** An int object, the total number of steps for calibration. Ignored with the default value of None. If *calib_dataset* is a *tf.data* dataset, generator, or *keras.utils.Sequence* instance and steps is None, calibration will run until the dataset is exhausted. This argument is not supported with array inputs.
- **calib_batch_size:** An int object, the number of samples per batch for calibration. If the "calib_dataset" is in the form of a dataset, generator, or *keras.utils.Sequence* instances, the batch size is controlled by the dataset itself. If the "calib_dataset" is in the form of a numpy.array object, the default batch size is 32.
- **save_path:** A string object, the directory to save the quantized model.
- **verbose:** An int object, the verbosity of the logging. Greater verbose value will generate more detailed logging. The default value is 0.
- **add_shape_info:** A bool object, whether to add shape inference information for custom layers. Must be set to True for models with custom layers.
- **dump:** A flag to enable/disable dump. If *dump=False*, dump is disabled, if *dump=True*, dump is enabled.
- **dump_output_dir:** A string object, the directory to save the dump results.

For more information on how to use on-the-fly quantization in WeGO TensorFlow 2.x, see WeGO examples.

TensorFlow 1.x

Quantization API

```
def quantize(
input_frozen_graph = "",
input_nodes = "",
input_shapes = "",
output_nodes = "",
input_fn = "",
method = 1,
calib_iter = 100,
output_dir = "./quantize_results",
**kargs)
```

This function will invoke `vai_q_tensorflow` command tool in WeGO TensorFlow r1.15 and converts the input floating-point model to fixed-point model for DPU deployment acceleration. To be fully compatible with native `vai_q_tensorflow` quantizer, all parameters received from this API will be forwarded to `vai_q_tensorflow` command tool directly. This function will return a quantized `GraphDef` object or `None` on failure.

Note: Only PTQ is supported now for on-the-fly quantization in WeGO. For more information on fast fine-tuning and QAT quantization, see vai_q_tensorflow Quantization Aware Training.

Parameters

- **input_frozen_graph:** string: path to input frozen graph(.pb) (default:)
- **input_nodes:** string: The comma-separated name list of input nodes of the subgraph to be quantized. Used together with output_nodes. When generating the model for deploy, only the subgraph between input_nodes and output_nodes will be included. Please set it to the beginning of the main body of the model to quantize, such as the nodes after data pre-processing and augmentation. (default:)
- **input_shapes:** string: the comma-separated shape list of input_nodes. The shape must be a 4-dimension shape for each node, comma separated, for example, `1,224,224,3`; Unknown size for batch size is supported, for example, `?,224,224,3`; In case of multiple input_nodes, please assign the shape list of each node, separated by `:`, for example, `?,224,224,3:?,300,300,1` (default:)
- **output_nodes:** string: the comma-speareted name list of output nodes of the subgraph to be quantized that is used together with input_nodes. When generating the model for deployment, only the subgraph between input_nodes and output_nodes will be included. Set it to the end of the main body of the model to quantize, such as the nodes before post-processing. (default:)
- **input_fn:** string: the python importable function that provides the input data. The format is `module_name.input_fn_name`, for example, `my_input_fn.input_fn`. The `input_fn` should take a `int` object as input indicating the calibration step, and should return a dict (`placeholder_node_name : numpy.Array`) object for each call, which will be fed into the model's placeholder nodes. (default:)
- **method:** int32: {0,1,2}, default: 1. The method for quantization, options are:
  - 0: non-overflow method. Ensure no values are saturated during quantization. It may get bad results in case of outliers.
  - 1: min-diffs method. It allows saturation for large values during quantization to get smaller quantization errors. This method is slower than method 0 but has higher endurance to outliers.
  - 2: min-diffs method with strategy for depthwise. It allows saturation for large values during quantization to get smaller quantization errors. Apply special strategy for depthwise weights, but implement method 1 to normal weights and activation. This method is slower than method 0 but has higher endurance to outliers.
- **calib_iter:** int32: the iterations of calibration. The total number of images for calibration = calib_iter * batch_size (default: 100)
- **output_dir:** string: the directory to save the quantization results (default: ./quantize_results).

Note: For more information on other parameters for `**kargs`, see vai_q_tensorflow Usage.

Note: For more information on the on-the-fly quantization examples for WeGO TensorFlow 1.x, see examples.

Optimize Performance with AMD ZenDNN

ZenDNN library, which includes APIs for basic neural network building blocks optimized for AMD CPU architecture, targets deep learning application and framework developers with the goal of improving deep learning inference performance on AMD CPUs. To improve the performance of DPU-unsupported operators on CPUs especially on AMD CPUs, ZenDNN is integrated into WeGO flow.

Note: This is just an experimental feature. The performance gain using ZenDNN in WeGO is not guaranteed currently. Enable/disable ZenDNN as you need.

ZenDNN in WeGO PyTorch

Enable ZenDNN in WeGO PyTorch

The ZenDNN is disabled by default and can be enabled through an extra option provided by WeGO-Torch's compile API:

```
wego_mod = wego_torch.compile(mod, wego_torch.CompileOptions(
...
optimize_options = wego_torch.OptimizeOptions(zendnn_enable = True))
)
```

After ZenDNN is enabled, the CPU operators (the operators not supported by DPU) in the compiled WeGO graph will be replaced with the ZenDNN operators, and they will be executed using ZenDNN kernels for acceleration.

Environment Variables

ZenDNN provides some environment variables for performance tuning purpose.

Table 38: **Environment Variables**

Name	Description
OMP_DYNAMIC	Set it explicitly with FALSE when you want to enable ZenDNN.
ZENDNN_GEMM_ALGO	Default is 3. You can set [0, 1, 2, 3] to tune different GEMM ALGO path.
OMP_NUM_THREADS	Default is the number of physical cores of user system. You need to tune as per the inference thread number to achieve better performance. See tuning guidelines for more details.

Tunning Guidelines

ZenDNN uses OpenMP as the underlying library. The environment variable `OMP_NUM_THREADS` is used to control intra-op parallelism which is multi-core parallelism in ZenDNN kernels. For OpenMP, different application threads or inter-op threads may use different OpenMP thread pools for intra-op tasks and thus a large number of OpenMP threads might be used in a multi-thread application, which will consume lots of CPU core resources and reduce the overall performance. So, the recommended tuning `OMP_NUM_THREADS` value is set as per the number of cores in the target CPU platform and the thread number used in your application to avoid over-subscription. For example, if you launch 16 threads in an application and you have 64 CPU cores on your platform, then you can set `OMP_NUM_THREADS <= 4` to avoid CPU cores contention.

ZenDNN in WeGO TensorFlow 2

Enable ZenDNN in WeGO TensorFlow 2

ZenDNN is disabled by default. Set `export ZENDNN_INFERENCE_ONLY=1` to enable it.

Environment Variables

You must export the following environment variables explicitly to enable ZenDNN working properly in WeGO TensorFlow 2.

Table 39: **Environment Varariables**

Name	Description
OMP_DYNAMIC	Set it to FALSE explicitly when ZenDNN is enabled.
OMP_NUM_THREADS	Set it explicitly to achieve a better performance. See tuning guidelines for more details.
ZENDNN_GEMM_ALGO	Default is 3. You can set [0, 1, 2, 3] to tune different GEMM ALGO path.
ZENDNN_TENSOR_POOL_LIMIT	Default is 32. See tuning guidelines for more details.
ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE	Default is 0. • 0: Enable reduced memory pool tensor. • 1: Enable increased memory pool tensor.
ZENDNN_INFERENCE_ONLY	Default is 0. Set 1 to enable ZenDNN.

Tuning Guidelines

Set `OMP_NUM_THREADS` as per the core number of user system. Xilinx recommends setting a small number like 1 or 2.

In some cases, set `ZENDNN_TENSOR_POOL_LIMIT` to a small number like 1, so some layers will use default memory allocation instead of tensor pool once it hits the pool limit with `ZEN_TENSOR_POOL_LIMIT`.

Profiling the Model

Vitis AI Profiler

The Vitis™ AI profiler is a set of tools that helps profile and visualize AI applications based on VART:

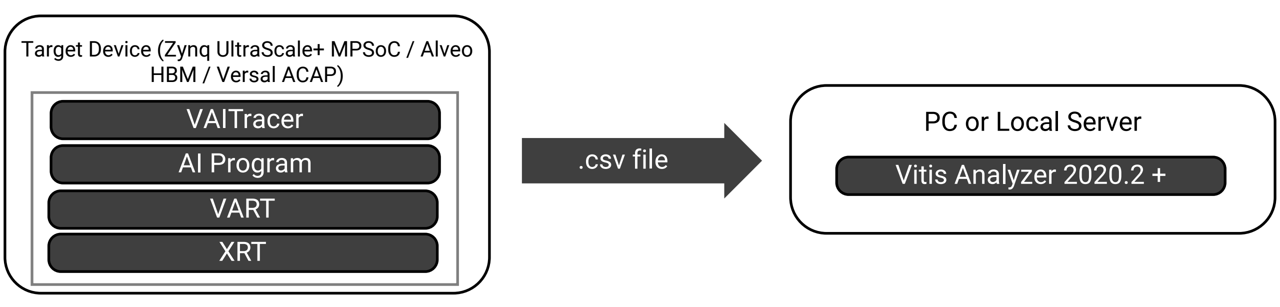
- Easy to use as it neither requires any change in the user code nor any re-compilation of the program.
- Visualize system performance bottlenecks.
- Illustrate the execution state of different compute units (CPU/DPU).

The Vitis AI Profiler is an all-in-one profiling solution for Vitis AI. It is an application level tool to profile and visualize AI applications based on VART. For an AI application, there are components that run on the hardware, for example, neural network computation usually runs on the DPU, and there are components that run on a CPU as a function that is implemented by C/C++ code-like image pre-processing. This tool helps you to put the running status of all these different components together.

Vitis AI Profiler Architecture

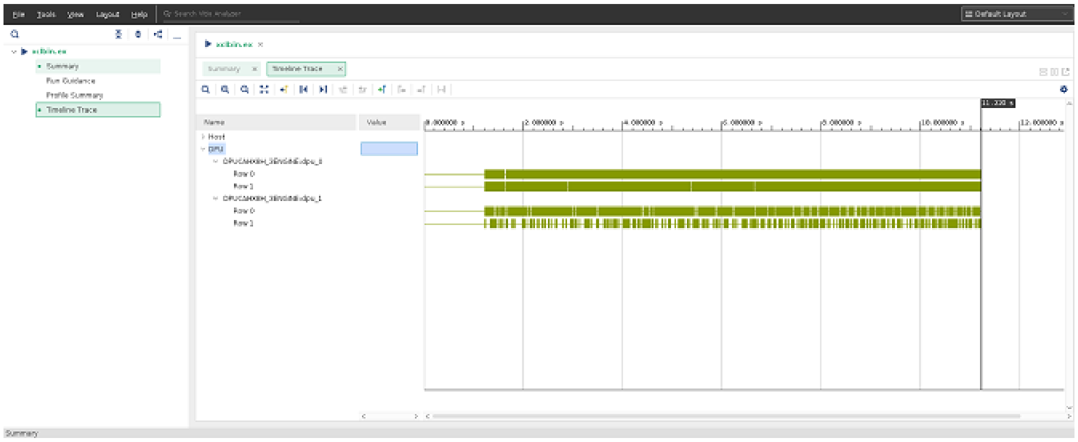
The Vitis AI Profiler architecture is shown in the following figure:

Figure 34: **Vitis AI Profiler Architecture**

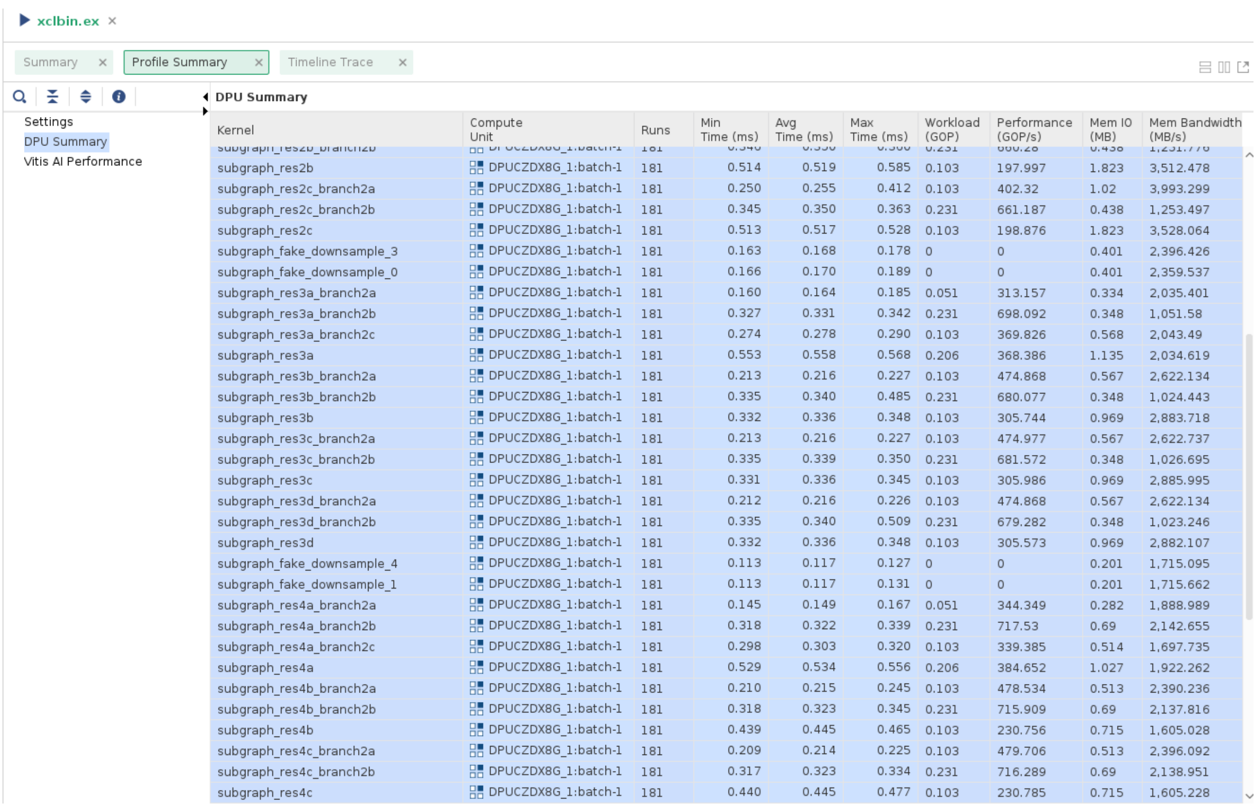


Vitis AI Profiler GUI Overview

Figure 35: **Vitis AI Profiler GUI Overview**

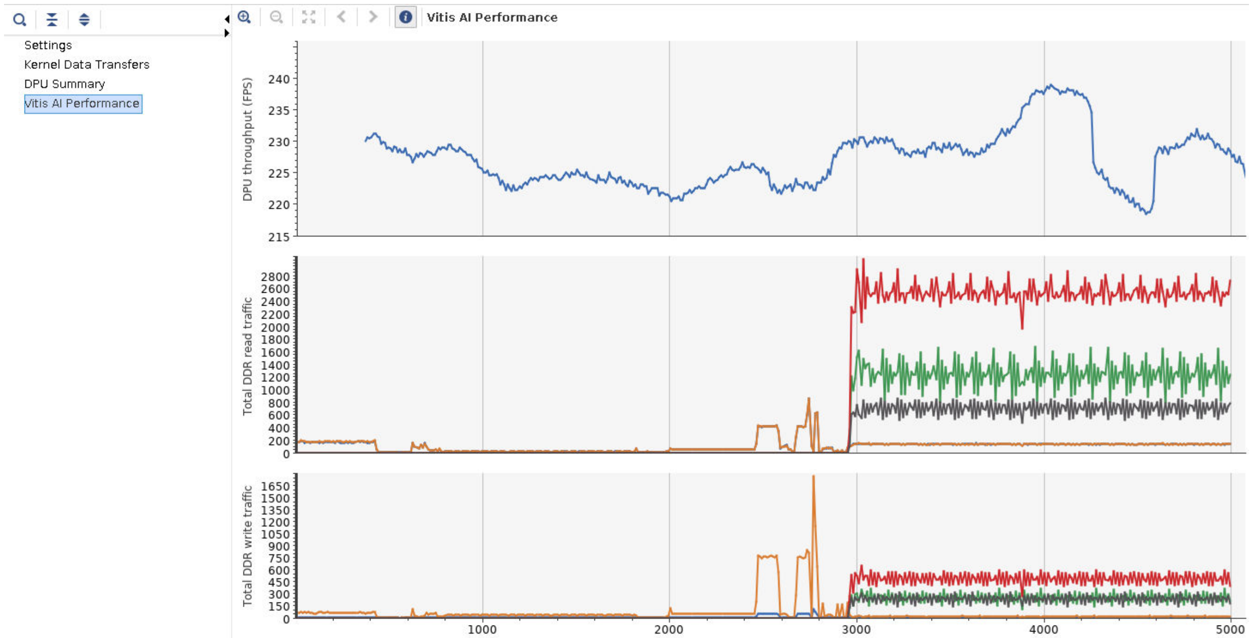


- **DPU Summary:** A table of the number of runs and minimum/average/maximum times (ms) for each kernel.

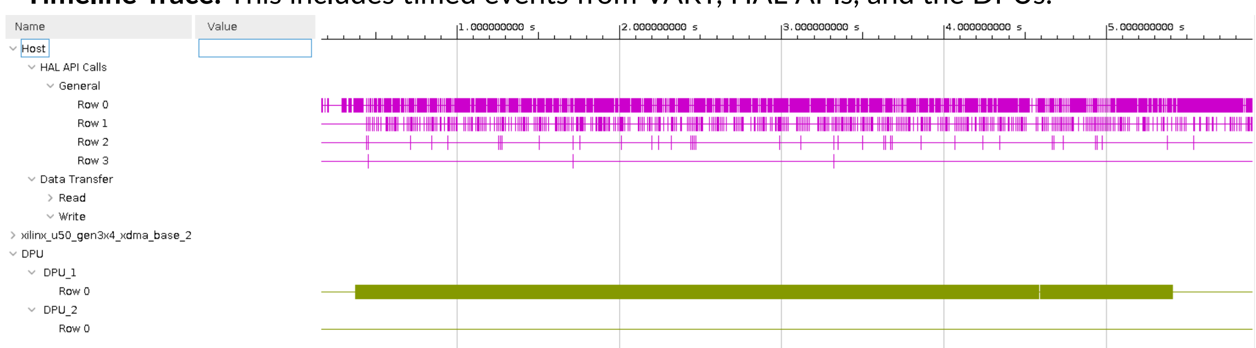


Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Workload (GOP)	Performance (GOP/s)	Mem IO (MB)	Mem Bandwidth (MB/s)
subgraph_res2b	DPUCZDX8G_1:batch-1	181	0.514	0.519	0.585	0.103	197.997	1.823	3,512.478
subgraph_res2c_branch2a	DPUCZDX8G_1:batch-1	181	0.250	0.255	0.412	0.103	402.32	1.02	3,993.299
subgraph_res2c_branch2b	DPUCZDX8G_1:batch-1	181	0.345	0.350	0.363	0.231	661.187	0.438	1,253.497
subgraph_res2c	DPUCZDX8G_1:batch-1	181	0.513	0.517	0.528	0.103	198.876	1.823	3,528.064
subgraph_fake_downsample_3	DPUCZDX8G_1:batch-1	181	0.163	0.168	0.178	0	0	0.401	2,396.426
subgraph_fake_downsample_0	DPUCZDX8G_1:batch-1	181	0.166	0.170	0.189	0	0	0.401	2,359.537
subgraph_res3a_branch2a	DPUCZDX8G_1:batch-1	181	0.160	0.164	0.185	0.051	313.157	0.334	2,035.401
subgraph_res3a_branch2b	DPUCZDX8G_1:batch-1	181	0.327	0.331	0.342	0.231	698.092	0.348	1,051.58
subgraph_res3a_branch2c	DPUCZDX8G_1:batch-1	181	0.274	0.278	0.290	0.103	369.826	0.568	2,043.49
subgraph_res3a	DPUCZDX8G_1:batch-1	181	0.553	0.558	0.568	0.206	368.386	1.135	2,034.619
subgraph_res3b_branch2a	DPUCZDX8G_1:batch-1	181	0.213	0.216	0.227	0.103	474.868	0.567	2,622.134
subgraph_res3b_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.340	0.485	0.231	680.077	0.348	1,024.443
subgraph_res3b	DPUCZDX8G_1:batch-1	181	0.332	0.336	0.348	0.103	305.744	0.969	2,883.718
subgraph_res3c_branch2a	DPUCZDX8G_1:batch-1	181	0.213	0.216	0.227	0.103	474.977	0.567	2,622.737
subgraph_res3c_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.339	0.350	0.231	681.572	0.348	1,026.695
subgraph_res3c	DPUCZDX8G_1:batch-1	181	0.331	0.336	0.345	0.103	305.986	0.969	2,885.995
subgraph_res3d_branch2a	DPUCZDX8G_1:batch-1	181	0.212	0.216	0.226	0.103	474.868	0.567	2,622.134
subgraph_res3d_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.340	0.509	0.231	679.282	0.348	1,023.246
subgraph_res3d	DPUCZDX8G_1:batch-1	181	0.332	0.336	0.348	0.103	305.573	0.969	2,882.107
subgraph_fake_downsample_4	DPUCZDX8G_1:batch-1	181	0.113	0.117	0.127	0	0	0.201	1,715.095
subgraph_fake_downsample_1	DPUCZDX8G_1:batch-1	181	0.113	0.117	0.131	0	0	0.201	1,715.662
subgraph_res4a_branch2a	DPUCZDX8G_1:batch-1	181	0.145	0.149	0.167	0.051	344.349	0.282	1,888.989
subgraph_res4a_branch2b	DPUCZDX8G_1:batch-1	181	0.318	0.322	0.339	0.231	717.53	0.69	2,142.655
subgraph_res4a_branch2c	DPUCZDX8G_1:batch-1	181	0.298	0.303	0.320	0.103	339.385	0.514	1,697.735
subgraph_res4a	DPUCZDX8G_1:batch-1	181	0.529	0.534	0.556	0.206	384.652	1.027	1,922.262
subgraph_res4b_branch2a	DPUCZDX8G_1:batch-1	181	0.210	0.215	0.245	0.103	478.534	0.513	2,390.236
subgraph_res4b_branch2b	DPUCZDX8G_1:batch-1	181	0.318	0.323	0.345	0.231	715.909	0.69	2,137.816
subgraph_res4b	DPUCZDX8G_1:batch-1	181	0.439	0.445	0.465	0.103	230.756	0.715	1,605.028
subgraph_res4c_branch2a	DPUCZDX8G_1:batch-1	181	0.209	0.214	0.225	0.103	479.706	0.513	2,396.092
subgraph_res4c_branch2b	DPUCZDX8G_1:batch-1	181	0.317	0.323	0.334	0.231	716.289	0.69	2,138.951
subgraph_res4c	DPUCZDX8G_1:batch-1	181	0.440	0.445	0.477	0.103	230.785	0.715	1,605.228

- **DPU Throughput and DDR Transfer Rates:** Line graphs of achieved FPS and read/write transfer rates (in MB/s) as sampled during the application.



- **Timeline Trace:** This includes timed events from VART, HAL APIs, and the DPUs.



***Note*:** The Vitis Analyzer is the default GUI for vaitrace in Vitis AI 1.3 and later releases.

Getting Started with the Vitis AI Profiler

System Requirements

- **Hardware:**
  - Supports Zynq® UltraScale+™ MPSoC (DPUCZDX8G)

    Supports Versal® ACAP (DPUCVDX8G/ DPUCVDX8H)
- **Software:**
  - Supports VART v1.2+

Installing the Vitis AI Profiler

1. Prepare the debug environment for vaitrace in the Zynq UltraScale+ MPSoC PetaLinux platform.
   a. Configure and build PetaLinux by running `petalinux-config -c kernel`.
   b. Enable the following settings for the Linux kernel.
      - General architecture-dependent options ---> [*] Kprobes
      - Kernel hacking ---> [*] Tracers
      - Kernel hacking ---> [*] Tracers --->

        [*] Kernel Function Tracer

        [*] Enable kprobes-based dynamic events

        [*] Enable uprobes-based dynamic events
   c. Run `petalinux-config -c rootfs` and enable the following setting for root-fs.

      Petalinux package Groups ---> packaggroup-petalinux-self-hosted ---> [*] packagegroup-petalinux-self-hosted
   d. Run `petalinux-build`.
2. Install vaitrace. vaitrace is integrated into the VART runtime. If VART runtime is installed, vaitrace will be installed into `/usr/bin/vaitrace`.

Starting a Simple Trace with vaitrace

The following example uses VART ResNet50 sample:

1. Download and set up Vitis AI.
2. Start testing and tracing.
   - For C++ programs, add `vaitrace` in front of the test command as follows:

     ```
     # cd ~/Vitis_AI/examples/vai_runtime/resnet50
     # vaitrace ./resnet50 /usr/share/vitis_ai_library/models/resnet50/
     resnet50.xmodel
     ```
   - For Python programs, add `-m vaitrace_py` to the Python interpreter command as follows:

     ```
     # cd ~/Vitis_AI/examples/vai_runtime/resnet50_mt_py
     # python3 -m vaitrace_py ./resnet50.py 2 /usr/share/vitis_ai_library/
     models/resnet50/resnet50.xmodel
     ```

   vaitrace and XRT generates some files in the working directory.
3. Copy all .csv files and `xclbin.ex.run_summary` to your system. You can open the `xclbin.ex.run_summary` using vitis_analyzer 2020.2 and above:

- If using the command line, run `# vitis_analyzer xclbin.ex.run_summary`.
- If using the GUI, select **File → Open Summary → xclbin.ex.run_summary**.

To know more about the Vitis Analyzer, see Using the Vitis Analyzer in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

vaitrace Usage

Command Line Usage

```
# vaitrace --help
usage: vaitrace [-h] [-c [CONFIG]] [-d] [-o [TRACESAVETO]] [-t [TIMEOUT]] [-
v]
                [-b] [-p] [--va] [--xat] [--txt_summary] [--fine_grained]
                ...

positional arguments:
  cmd

optional arguments:
  -h, --help          show this help message and exit
  -c [CONFIG]         Specify the config file
  -d                  Enable debug
  -o [TRACESAVETO]    Save report to, only avaliable for txt summary mode
  -t [TIMEOUT]        Tracing time limitation
  -v                  Show version
  -b                  Bypass vaitrace, just run command
  -p                  Trace python application
  --va                Generate trace data for Vitis Analyzer
  --xat               Save raw data, for debug usage
  --txt_summary       Display txt summary
  --fine_grained      Fine grained mode
```

Following are some important and frequently-used arguments:

- **cmd:** cmd is your executable program of Vitis AI that to be traced, including program name and arguments
- **-t:** Controlling the tracing time (in seconds) starting from the [cmd] being launched, the default value is 30. In other words, if no -t is specified for vaitrace, the tracing will stop after [cmd] running for 30 seconds. The [cmd] will continue to run as normal, but it will stop collecting tracing data.
- **-c:** You can start a tracing with more custom options by writing these options on a JSON configuration file and specify the configuration by -c. Details of configuration file will be explained in the next section.
- **-o:** Location of the report. This is only available for the text summary mode. By default, the test summary will output to STDOUT.
- **--va:** Generate trace data for Vitis Analyzer, enabled by default, cannot work together with --txt_summary

- **--txt_summary or --txt:** Output text summary. vaitrace does not generate a report for the Vitis Analyzer in this mode, cannot work together with --va.
- **--fine_grained:** Start trace in the fine grained mode. This mode generates a mass of trace data and the trace time is limited to 10 seconds.

Others arguments are used for debugging.

Configuration

It is recommended to use a configuration file to record trace options for vaitrace. You can start a trace with configuration by using `vaitrace -c trace_cfg.json`.

Configuration priority: **Configuration File → Command Line → Default**.

Here is an example of vaitrace configuration file.

```
{
  "trace": {
      "enable_trace_list": ["vitis-ai-library", "vart", "custom"]
  }
  "trace_custom": []
}
```

Table 40: **Contents of the Configuration File**

Key Name		Value Type	Description
trace		object	
	enable_trace_list	list	Built-in trace function list to be enabled, available value "vitis-ai-library", "vart", “opencv”, "custom", custom for function in trace_custom list.
trace_custom		list	The list of functions to be traced that are implemented by user. For the name of function, naming space are supported. You can see an example of using custom trace function later in this document.

Text Summary

When the `--txt` or `--txt_summary` option is used, vaitrace prints an ASCII table as shown in the following figure:

Figure 36: **ASCII Table**

```
DPU Summary:
================================================================================================================================
DPU Id        | Bat | SubGraph                                 | WL    | RT    | Perf    | LdWB  | LdFM  | StFM  | AvgBw
--------------+-----+------------------------------------------+-------+-------+---------+-------+-------+-------+----------
DPUCZDX8G_1   | 1   | conv1                                    | 0.239 | 0.612 | 389.930 | 0.009 | 0.507 | 0.191 | 1183.568
DPUCZDX8G_1   | 1   | res2a_branch2a                           | 0.026 | 0.189 | 136.988 | 0.004 | 0.191 | 0.191 | 2095.569
DPUCZDX8G_1   | 1   | res2a_branch2b                           | 0.231 | 0.285 | 811.971 | 0.035 | 0.191 | 0.191 | 1501.974
DPUCZDX8G_1   | 1   | res2a_branch2c                           | 0.104 | 0.270 | 383.568 | 0.016 | 0.191 | 0.766 | 3689.815
DPUCZDX8G_1   | 1   | res2a                                    | 0.105 | 0.566 | 185.811 | 0.016 | 0.957 | 0.766 | 3145.318
DPUCZDX8G_1   | 1   | res2b_branch2a                           | 0.103 | 0.242 | 425.459 | 0.016 | 0.766 | 0.191 | 4115.961
DPUCZDX8G_1   | 1   | res2b_branch2b                           | 0.231 | 0.334 | 692.849 | 0.035 | 0.191 | 0.191 | 1281.624
DPUCZDX8G_1   | 1   | res2b                                    | 0.105 | 0.491 | 214.193 | 0.016 | 0.957 | 0.766 | 3625.764
DPUCZDX8G_1   | 1   | res2c_branch2a                           | 0.103 | 0.243 | 423.708 | 0.016 | 0.766 | 0.191 | 4099.023
DPUCZDX8G_1   | 1   | res2c_branch2b_bias                      | 0.058 | 0.163 | 354.926 | 0.035 | 0.191 | 0.048 | 1724.310
DPUCZDX8G_1   | 1   | res2b_downsample_bp_by_fake_downsample_0 | 0.026 | 0.270 | 97.379  | 0.016 | 0.239 | 0.191 | 1693.519
DPUCZDX8G_1   | 1   | res3a_branch2a                           | 0.051 | 0.167 | 308.267 | 0.031 | 0.191 | 0.096 | 1952.844
DPUCZDX8G_1   | 1   | res3a_branch2b                           | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019
DPUCZDX8G_1   | 1   | res3a_branch2c                           | 0.103 | 0.266 | 387.827 | 0.063 | 0.096 | 0.383 | 2084.586
DPUCZDX8G_1   | 1   | res3a                                    | 0.207 | 0.527 | 392.268 | 0.125 | 0.574 | 0.383 | 2103.416
DPUCZDX8G_1   | 1   | res3b_branch2a                           | 0.103 | 0.209 | 492.157 | 0.063 | 0.383 | 0.096 | 2651.316
DPUCZDX8G_1   | 1   | res3b_branch2b                           | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019
DPUCZDX8G_1   | 1   | res3b                                    | 0.104 | 0.319 | 325.908 | 0.063 | 0.479 | 0.383 | 2967.085
DPUCZDX8G_1   | 1   | res3c_branch2a                           | 0.103 | 0.209 | 492.157 | 0.063 | 0.383 | 0.096 | 2651.316
DPUCZDX8G_1   | 1   | res3c_branch2b                           | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019
DPUCZDX8G_1   | 1   | res3c                                    | 0.104 | 0.320 | 324.890 | 0.063 | 0.479 | 0.383 | 2957.811
DPUCZDX8G_1   | 1   | res3d_branch2a                           | 0.103 | 0.208 | 494.523 | 0.063 | 0.383 | 0.096 | 2664.063
DPUCZDX8G_1   | 1   | res3d_branch2b_bias                      | 0.058 | 0.142 | 407.238 | 0.141 | 0.096 | 0.024 | 1877.641
DPUCZDX8G_1   | 1   | res3c_downsample_bp_by_fake_downsample_1 | 0.026 | 0.209 | 124.360 | 0.063 | 0.120 | 0.096 | 1363.636
DPUCZDX8G_1   | 1   | res4a_branch2a                           | 0.051 | 0.144 | 357.156 | 0.125 | 0.096 | 0.048 | 1911.458
DPUCZDX8G_1   | 1   | res4a_branch2b                           | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254
DPUCZDX8G_1   | 1   | res4a_branch2c                           | 0.103 | 0.289 | 356.267 | 0.251 | 0.048 | 0.191 | 1737.024
DPUCZDX8G_1   | 1   | res4a                                    | 0.206 | 0.504 | 408.974 | 0.501 | 0.287 | 0.191 | 1990.079
DPUCZDX8G_1   | 1   | res4b_branch2a                           | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252
DPUCZDX8G_1   | 1   | res4b_branch2b                           | 0.231 | 0.308 | 750.848 | 0.563 | 0.048 | 0.048 | 2189.121
DPUCZDX8G_1   | 1   | res4b                                    | 0.103 | 0.423 | 244.156 | 0.251 | 0.239 | 0.191 | 1650.118
DPUCZDX8G_1   | 1   | res4c_branch2a                           | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252
DPUCZDX8G_1   | 1   | res4c_branch2b                           | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254
DPUCZDX8G_1   | 1   | res4c                                    | 0.103 | 0.420 | 246.101 | 0.251 | 0.239 | 0.191 | 1661.005
DPUCZDX8G_1   | 1   | res4d_branch2a                           | 0.103 | 0.205 | 501.515 | 0.250 | 0.191 | 0.048 | 2445.122
DPUCZDX8G_1   | 1   | res4d_branch2b                           | 0.231 | 0.306 | 755.756 | 0.563 | 0.048 | 0.048 | 2203.431
DPUCZDX8G_1   | 1   | res4d                                    | 0.103 | 0.422 | 244.935 | 0.251 | 0.239 | 0.191 | 1654.028
DPUCZDX8G_1   | 1   | res4e_branch2a                           | 0.103 | 0.205 | 501.515 | 0.250 | 0.191 | 0.048 | 2445.122
DPUCZDX8G_1   | 1   | res4e_branch2b                           | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254
DPUCZDX8G_1   | 1   | res4e                                    | 0.103 | 0.421 | 245.517 | 0.251 | 0.239 | 0.191 | 1657.957
DPUCZDX8G_1   | 1   | res4f_branch2a                           | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252
DPUCZDX8G_1   | 1   | res4f_branch2b_bias                      | 0.058 | 0.159 | 363.618 | 0.563 | 0.048 | 0.012 | 4009.434
DPUCZDX8G_1   | 1   | res4e_downsample_bp_by_fake_downsample_2 | 0.026 | 0.248 | 104.196 | 0.251 | 0.203 | 0.048 | 2073.589
DPUCZDX8G_1   | 1   | res5a_branch2a                           | 0.051 | 0.160 | 321.283 | 0.500 | 0.048 | 0.024 | 3662.500
DPUCZDX8G_1   | 1   | res5a_branch2b                           | 0.231 | 0.391 | 591.397 | 2.250 | 0.024 | 0.024 | 6019.182
DPUCZDX8G_1   | 1   | res5a_branch2c                           | 0.103 | 0.341 | 301.645 | 1.002 | 0.024 | 0.096 | 3368.035
DPUCZDX8G_1   | 1   | res5a                                    | 0.206 | 0.438 | 469.913 | 2.002 | 0.144 | 0.096 | 5239.726
DPUCZDX8G_1   | 1   | res5b_branch2a                           | 0.103 | 0.234 | 439.254 | 1.000 | 0.096 | 0.024 | 4901.709
DPUCZDX8G_1   | 1   | res5b_branch2b                           | 0.231 | 0.394 | 586.894 | 2.250 | 0.024 | 0.024 | 5973.350
DPUCZDX8G_1   | 1   | res5b                                    | 0.103 | 0.377 | 273.373 | 1.002 | 0.120 | 0.096 | 3306.366
DPUCZDX8G_1   | 1   | res5c_branch2a                           | 0.103 | 0.232 | 443.041 | 1.000 | 0.096 | 0.024 | 4943.966
DPUCZDX8G_1   | 1   | res5c_branch2b                           | 0.231 | 0.391 | 591.397 | 2.250 | 0.024 | 0.024 | 6019.182
DPUCZDX8G_1   | 1   | res5c                                    | 0.103 | 0.374 | 275.566 | 1.002 | 0.120 | 0.096 | 3332.888
DPUCZDX8G_1   | 1   | pool5                                    | ~0    | 0.102 | 0.984   | ~0    | 0.096 | 0.002 | 980.392
DPUCZDX8G_1   | 1   | fc1000_bias                              | 0.004 | 0.312 | 13.131  | 1.954 | 0.002 | ~0    | 6422.927
================================================================================================================================

Notes:
"~0": Within range of (0, 0.001)
Bat: DPU batch number
WL(GOP): Workload
RT(ms): Run time
Perf(GOP/s)
LdFM(MB): External memory load size of feature map
LdBW(MB): External memory load size of bias and weight
StFM(MB): External memory store size of feature map
AvgBw(MB/s): External memory aaverage bandwidth
....

CPU Tasks in Graph(called by graph runner):
=============================================================
SubGraph        | OPs       | Device | Runs | AverageRunTime(ms)
----------------+-----------+--------+------+-------------------
fc1000_fixed_   | fix2float | CPU    | 1    | 0.063
=============================================================

CPU Functions(Not in Graph, e.g.: pre/post-processing, vai-runtime):
=====================================================
Function        | Device | Runs | AverageRunTime(ms)
----------------+--------+------+-------------------
xir::XrtCu::run | CPU    | 55   | 0.295
=====================================================
```

The fields are defined in the following list:

- **DPU Id:** Name of the DPU instance .
- **Bat:** Batch size of the DPU instance.
- **SubGraph:** Name of subgraph in the xmodel.
- **WL(Workload):** Computation workload (MAC indicates two operations), unit is GOP
- **RT(Run time):** The execution time in milliseconds, unit is ms.
- **Perf:** The DPU performance in unit of GOP per second, unit is GOP/s.

- **LdFM (Load Size of Feature Map):** External memory load size of feature map, unit is MB.
- **LdWB (Load Size of Weight and Bias):** External memory load size of bias and weight, unit is MB.
- **StFM (Store Size of Feature Map):** External memory store size of feature map, unit is MB.
- **AvgBw(Average bandwidth):** Average DDR memory access bandwidth.

  AvgBw = (total load size of the subgraph (including feature map and weight/bias, from DDR/HBM to DPU bank mem) + total store size of the subgraph (from DPU bank mem to DDR/HBM)) / subgraph runtime

DPU Profiling Examples

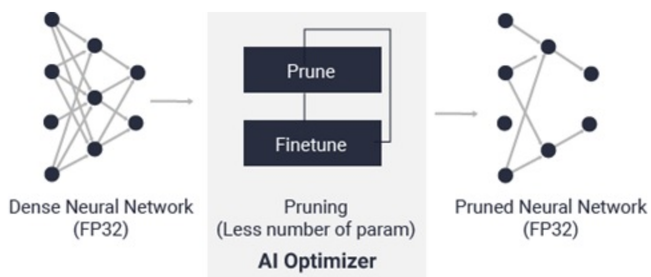
You can find advanced DPU profiling examples with the Vitis AI Profiler on the Vitis AI Profiler GitHub page.

Optimizing the Model

***Note*:** Optimizing the model is an optional step.

The Vitis AI optimizer provides the ability to optimize neural network models. Currently, the Vitis AI optimizer includes only one tool called the Vitis AI pruner (VAI pruner), which prunes redundant connections in neural networks and reduces the overall required operations. The pruned models produced by the VAI pruner can be further quantized by the VAI quantizer and deployed to an FPGA.

Figure 37: **Vitis AI Optimizer**



The VAI pruner supports four deep learning frameworks: TensorFlow, PyTorch, Caffe, and Darknet. The corresponding tool names are vai_p_tensorflow, vai_p_pytorch, vai_p_caffe, and vai_p_darknet, where the "p" in the middle stands for pruning.

For more information, see the *Vitis AI Optimizer User Guide* (UG1333).

The Vitis AI optimizer requires a commercial license to run. Contact a Xilinx sales representative for more information.

Chapter 8

Integrating the DPU into Custom Platforms

You can integrate the DPU into custom Vitis platforms to run AI applications with the Vitis™ software platform. There are some pre-compiled platforms that can be downloaded from the Xilinx® Vitis Embedded Platform Downloads. If you want to create a custom platform, see *Vitis Unified Software Platform Documentation* (UG1416).

To facilitate the DPU integration, Xilinx provides the DPU TRD and XVDPU TRD in which you can configure the DPU with different parameters to meet the performance and resource utilization requirements. For more details, see *DPUCZDX8G for Zynq UltraScale+ MPSoCs* (PG338) and Vitis DPU TRD flow. For DPUCVDX8G, see *DPUCVDX8G for Versal ACAPs Product Guide* (PG389) and the Vitis DPUCVDX8G TRD flow.

On the hardware side, the Vitis software platform integrates the DPU as an RTL kernel. It requires two clocks: `clk` and `clk2x`. One interrupt is needed. The DPU may also need multiple AXI HP interfaces.

On the software side, the platform needs to provide the XRT and ZOCL packages. The host application can use the XRT OpenCL™ API to control the kernel. The Vitis AI Runtime can control the DPU with XRT. ZOCL is the kernel module that talks to acceleration kernels. It needs a device tree node which has to be added.

For more details, see the Vitis AI Platform Creation tutorials.

If you use the Vivado® Design Suite for DPU integration, see the DPU TRD Vivado flow.

If you want to integrate the DPU into Versal Data Center accelerator cards and other non-embedded platforms, contact amd_ai_mkt@amd.com.

Error Codes

Error Code ID	Error Message
OPTIMIZER_DATA_PARALLEL_NOT_ALLOWED_ERROR	torch.nn.DataParallel module is not allowed.
OPTIMIZER_INVALID_ANA_RESULT_ERROR	Model analysis result is not valid. This is usually caused by PyTorch or Python version change.
OPTIMIZER_INVALID_ARGUMENT_ERROR	Invalid argument.
OPTIMIZER_TORCH_MODULE_ERROR	The operation is not an instance of torch.nn.Module.
OPTIMIZER_NOT_EXCLUDE_NODE_ERROR	Some nodes must be excluded from pruning.
OPTIMIZER_NO_ANA_RESULT_ERROR	Model analysis result not found.
OPTIMIZER_SUBNET_ERROR	Subnet candidates not found. Must do subnet searching first.
OPTIMIZER_UNSUPPORTED_OP_ERROR	The operation is not supported yet.
OPTIMIZER_KERAS_MODEL_ERROR	The given object is not an instance of keras.Model.
OPTIMIZER_KERAS_LAYER_ERROR	The operation is not an instance of keras.Layer.
OPTIMIZER_DATA_FORMAT_ERROR	The data format for saving weights is not allowed in pruning.
OPTIMIZER_INVALID_GRAPH	The parsed graph is invalid.
OPTIMIZER_IO_ERROR	IO error. Usually occurs during disk read/write.
OPTIMIZER_MODEL_ANALYSIS_ERROR	An error occurred while performing model analysis.
OPTIMIZER_PARSE_GRAPH_FAILED	Unable to parse the model to a computation graph.
OPTIMIZER_WEIGHTS_NOT_FOUND	The weights for the operation can not be found.
QUANTIZER_TF1_INVALID_BITWIDTH	invalid parameter
QUANTIZER_TF1_INVALID_METHOD	invalid parameter
QUANTIZER_TF1_INVALID_TARGET_DTYPE	invalid parameter
QUANTIZER_TF1_MISSING_QUANTIZE_INFO	not found
QUANTIZER_TF1_INVALID_INPUT	not found
QUANTIZER_TF1_UNSUPPORTED_OP	Unsupported Op type
QUANTIZER_TF1_LENGTH_MISMATCH	invalid parameter
QUANTIZER_TF1_INVALID_INPUT_FN	fail to import
QUANTIZER_TF2_UNSUPPORTED_MODEL	Unsupported model type
QUANTIZER_TF2_UNSUPPORTED_LAYER	Unsupported layer type
QUANTIZER_TF2_INVALID_CALIB_DATASET	Invalid calibration dataset
QUANTIZER_TF2_INVALID_INPUT_SHAPE	Invalid input shape
QUANTIZER_TF2_INVALID_TARGET	Invalid Target
QUANTIZER_TORCH_BIAS_CORRECTION	Bias correction file in quantization result directory does not match current model.

Error Code ID	Error Message
QUANTIZER_TORCH_CALIB_RESULT_MISMATCH	Node name mismatch is found when loading quantization steps of tensors. Please make sure vai_q_pytorch version and pytorch version for test mode are the same as those in calibration (or QAT training) mode.
QUANTIZER_TORCH_EXPORT_ONNX	The quantized module, which is based pytorch traced model, can not be exported to ONNX due to pytorch internal failure. The pytorch internal failure reason is listed in message text. May needs adjust float model code.
QUANTIZER_TORCH_EXPORT_XMODEL	Fail to convert graph to xmodel. Needs check the reasons in message text.
QUANTIZER_TORCH_FAST_FINETINE	Fast fine-tuned parameter file does not exist. Call load_ft_param in model code to load them.
QUANTIZER_TORCH_FIX_INPUT_TYPE	Data type or value is illegal in arguments of quantization OP when exporting ONNX format model.
QUANTIZER_TORCH_ILLEGAL_BITWIDTH	The configuration of tensors quantization is illegal. It should be integer, and in range given in message text.
QUANTIZER_TORCH_IMPORT_KERNEL	Importing vai_q_pytorch library file error. Check pytorch version matching vai_q_pytorch version (pytorch_nndct.__version__) or not.
QUANTIZER_TORCH_NO_CALIB_RESULT	Quantization result file does not exist. Please check calibration is done or not.
QUANTIZER_TORCH_NO_CALIBRATION	Quantization calibration is not performed completely, check if module FORWARD function is called! FORWARD function of torch_quantizer.quant_model needs to be called in user code explicitly. Please refer to the example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_NO_FORWARD	torch_quantizer.quant_model FORWARD function must be called before exporting quantization result. Please refer to example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_OP_REGIST	The type of OP can't be registered multiple times.
QUANTIZER_TORCH_PYTORCH_TRACE	Failed to get pytorch traced graph from model and input arguments. The pytorch internal failure reason is reported in message text. May needs adjust float model code.
QUANTIZER_TORCH_QUANT_CONFIG	Quantization configuration items are illegal. Refer to the message text.
QUANTIZER_TORCH_SHAPE_MISMATCH	Tensors shape are mismatch. Refer to the message text.
QUANTIZER_TORCH_VERSION	Pytorch version is not supported for the function or does not match vai_q_pytorch version (pytorch_nndct.__version__). Refer to the message text.
QUANTIZER_TORCH_XMODEL_BATCHSIZE	Batch size must be 1 when exporting xmodel.
QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT	Inspector only support dump svg or png format.
QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT	Inspector no longer support fingerprint. Please provide architecture name instead.
QUANTIZER_TORCH_UNSUPPORTED_OPS	The quantization of the op is not supported.
QUANTIZER_TORCH_TRACED_NOT_SUPPORT	The model produced by 'torch.jit.script' is not supported in vai_q_pytorch.
QUANTIZER_TORCH_NO_SCRIPT_MODEL	vai_q_pytorch does not find any script model.
QUANTIZER_TORCH_REUSED_MODULE	The quantized module has been called multiple times in forward pass. If you want to share quantized parameters in multiple calls, call trainable_model with "allow_reused_module=True"
QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED	torch.nn.DataParallel object is not allowed.

Error Code ID	Error Message
QUANTIZER_TORCH_INPUT_NOT_QUANTIZED	Input is not quantized. Please use QuantStub/DeQuantStub to define quantization scope.
QUANTIZER_TORCH_NOT_A_MODULE	Quantized operation must be instance of "torch.nn.Module", please replace the function by a "torch.nn.Module" object. Original source range is indicated in message text.
QUANTIZER_TORCH_QAT_PROCESS_ERROR	Must call "trainable_model" first before getting deployable model.
QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR	The given trained model has BN fused to CONV and cannot be converted to a deployable model. Make sure model.fuse_conv_bn() is not called.
QUANTIZER_TORCH_XMODEL_DEVICE	Xmodel can only be exported in CPU mode, use deployable_model(src_dir, used_for_xmodel=True) to get a CPU model.
WEGO_TORCH_UNKNOWN_ERROR	Unknown error
WEGO_TORCH_INTERNAL_ERROR	Internal error
WEGO_TORCH_INVALID_ARGUMENT	Invalid argument error
WEGO_TORCH_INVALID_MODEL	Invalid model error
WEGO_TORCH_OUT_OF_RANGE	Out of range error
WEGO_TORCH_UNIMPLEMENTED	Unimplemented error
WEGO_TORCH_RUNTIME_ERROR	Runtime error
XCOM_OP_CONV_PARAM_ERROR	convolution parameter out of range or error, including feature map height, width, depth, channel, dilation size, transposition size, kernel height, kernel width, stride height, stride width or padding left, right, top, bottom, depth or fixed-point shift range or other network designed parameters.
XCOM_OP_IO_TENSOR_TYPE_ERROR	error tensor type for io operator such as load and save.
XCOM_OP_MEM_TYPE_ERROR	The op's output tensor's memory type is error.
XCOM_OP_PAD_SMF_MISSING	failed to generate padding in pool since smf data missing.
XCOM_OP_POOL_SIZE_ERROR	failed to calculate pooling size with formula.
XCOM_OP_SIGMOID_HEIGHT_NE	sigmoid operator need input and output have same height.
XCOM_OP_SIGMOID_WIDTH_NE	sigmoid operator need input and output have same width.
XCOM_OP_SIGMOID_CHANNEL_NE	sigmoid operator need input and output have same channel.
XCOM_OP_REORG_HEIGHT_NE	reorg operator need input height and output height have scale multiple.
XCOM_OP_REORG_WIDTH_NE	reorg operator need input width and output width have scale multiple.
XCOM_OP_REORG_CHANNEL_NE	reorg operator need input channel and output channel have scale multiple.
XCOM_OP_REORG_CHANNEL_OVERFLOW	feature channel size overflows regorg channel input.
XCOM_OP_TYPE_UNMATCH	unmatch operator type, it could be unknown operator or inappropriate suffix operator type.
XCOM_OP_TYPE_ERROR	op involved type error, unrecognized op involvded type here.
XCOM_TILING_SIZE_ERROR	Tiling bank group size not enough and tiling failed. Perhaps input tensor or kernel size is too large or tiling bank aligned has calculation fault.
XCOM_DPUOP_DATA_SIZE_ERROR	size not enough or unaligned while mapping smf onto banks with channel, width, depth, height, stride_h and other dimension incompatible. Please check bank info
XCOM_ACGEN_POOL_KERNEL_OUTRANGE	pooling layer kernel size out of range. please check network design or input data size.
XCOM_OP_NONLINEAR_TYPE_ERROR	error of operator non-linear type.
XCOM_ACGEN_UNSUPPORT_QUANTIZATION	unsupport quantization bit shift while assembly code generation.

Error Code ID	Error Message
XCOM_ACGEN_NONLINEAR_TYPE_ERROR	unrecognized or unmatched type of non-linear type while assembly code generation
XCOM_ACGEN_BANK_IO_ERROR	bank input or output addr count error while assembly code generation, it may exceed hardware capapcity.
XCOM_ACGEN_PRELU_ERROR	parameter-relu data info error while convolution assembly code generating, might be error with parameter data input width or number error.
XCOM_ACGEN_CONV_WEIGHTS_OC_NE	convolution output channel number should be equal to convolution weights input, if not, please check data life-circle.
XCOM_ACGEN_BANK_OC_WEIGHTS_UNALIGNED	weights bank address need be aligned to convolution output channel.
XCOM_BANK_UNALIGNED_ADDRESSING	trying to address in the middle of bank addr which is illegal, bank adddressing only support aligned operation, for example, stride or address mod 16 == 0.
XCOM_ACGEN_CONV_FAKE_WEIGHTS_BANKID	convolution weights input bank id need be equal to base bank id, please check bank assignation of weights.
XCOM_ACGEN_CONV_FAKE_BIAS_BANKID	convolution bias input bank id need be equal to base bank id, please check bank assignation of bias.
XCOM_ACGEN_CONV_OCG_WEIGHTS_CNT_NE	convolution weights input number need be equal to output channel group size, please check weigths input number.
XCOM_ACGEN_KERNEL_ALL_PAD	kernel are fullfilled with pad value, this is an unexpected situation, please check kernel size and dilation value.
XCOM_ACGEN_BANK_ADDR_IN_OUTRANGE	bank address input number need be ordered in hardware limitation.
XCOM_ACGEN_BANK_ADDR_OUT_OUTRANGE	bank address output number need be ordered in hardware limitation.
XCOM_ACGEN_ELEW_IO_ERROR	element wise operator have more than hardware capability input number, or, input and output number is not equal. please check bank assignation.
XCOM_ACGEN_ELEW_IO_CHANNEL_NE	element wise operatore need input and output have same channel group size.
XCOM_ACGEN_ELEW_IO_LENGTH_NE	element wise operatore need input and output have same length.
XCOM_ACGEN_MUL_IO_ERROR	mul operator have more than hardware capability input number, or, input and output number is not equal. please check bank assignation.
XCOM_ACGEN_INPUT_MISSING	operator assembly code generation input bank address missing, please check bank assignation.
XCOM_ACGEN_BLOB_MISSING	blob shifting failed because cannot find specific blob id in blob area. please check blob assignation.
XCOM_ACGEN_OUTPUT_MISSING	operator assembly code generation ouptut bank address missing, please check bank assignation.
XCOM_ACGEN_IO_TUPLE_NE	some operator need input and output number be equal but here is not. please check bank assignation.
XCOM_ACGEN_WEIGHTS_NOT_UNIQ	some operator need uniq weights input bank, please check bank assignation.
XCOM_ACGEN_PRELU_NOT_UNIQ	some operator need uniq prelu input bank, please check bank assignation.
XCOM_ACGEN_PARAM_NOT_UNIQ	some operator need uniq param input bank like sigmoid, please check bank aggregation.
XCOM_ACGEN_BIAS_NOT_UNIQ	some opeartor need uniq bias input bank, please check bank assignation.
XCOM_ACGEN_V3ME_BANK_MISSING	operator's dest bank have no virtual bank or constant bank, on v3me conv operator at least need 1 type of bank.

Error Code ID	Error Message
XCOM_ACGEN_IC_WEIGHTS_CHANNEL_NE	depth operator input channel and weights channel are not equal, please check bank assignation or network structure.
XCOM_ACGEN_BIAS_WEIGHTS_CHANNEL_NE	depth operator weighs channel and bias channel are not equal, please check bank assignation.
XCOM_ACGEN_INVALIDE_STATUS	compiler internal error, some status is invalid for assembly code generation like object was not inited, missing input or output data on dpu bank. please check data flow and code logic.
XCOM_ACGEN_BANK_JUMP_READ_ERROR	the bank cannot jump read.
XCOM_ACGEN_BANK_JUMP_WRITE_ERROR	the bank cannot jump write.
XCOM_OP_ARGMAX_IO_HEIGHT_NE	argmax need input and output have equal height.
XCOM_OP_ARGMAX_IO_WIDTH_NE	argmax need input and output have equal width.
XCOM_OP_ARGMAX_IO_DEPTH_NE	argmax need input and output have equal depth.
XCOM_OP_ARGMAX_OC_NOT_UNIQ	argmax need output channel is 1.
XCOM_OP_CONCAT_IO_CHANNEL_NE	concat operator need input and output channel equal.
XCOM_OP_CORR_ELT_MUL_OUTPUT_ERROR	correlation eltwise multiply output depth calculate error.
XCOM_OP_CORR_ELT_MUL_IO_HEIGHT_NE	correlation eltwise multiply need input height is equal to output height
XCOM_OP_CORR_ELT_MUL_IO_WIDTH_NE	correlation eltwise multiply need input width is equal to output width
XCOM_OP_CORR_ELT_MUL_IO_CHANNEL_NE	correlation eltwise multiply need input channel is equal to output channel
XCOM_OP_CORR_ELT_MUL_INPUT_CHANNEL_NE	correlation eltwise multiply need all input channel are equal
XCOM_OP_COST_STRIDE_OUTPUT_DEPTH_NE	cost operator need stride is equal to output depth
XCOM_OP_COST_IO_HEIGHT_NE	cost operator need input and output have same height
XCOM_OP_COST_IO_WIDTH_NE	cost operator need input and output have same width
XCOM_OP_COST_INPUT_DEPTH_NOT_UNIQ	cost operator need input depth = 1
XCOM_OP_COST_IO_CHANNEL_NE	cost operator need input channel = 1/2 output channel
XCOM_OP_DOWNSAMPLE_IO_HEIGHT_NE	downsample need input height ceiling divide scale height equal to output height
XCOM_OP_DOWNSAMPLE_IO_WIDTH_NE	downsample need input width ceiling divide scale width equal to output width
XCOM_OP_DOWNSAMPLE_IO_CHANNEL_NE	downsample need input and output channnel equal.
XCOM_OP_TDPTCONV3D_ICG_NOT_ENOUGH	transposed depth conv 3d input channel group mult channel parallel is less than feature channel size.
XCOM_OP_TDPTCONV3D_KERNEL_HEIGHT_OVERFLOW	transposed depth conv 3d kernel height overflow.
XCOM_OP_TDPTCONV3D_KERNEL_WIDTH_OVERFLOW	transposed depthconv 3d kernel width overflow.
XCOM_OP_TDPTCONV3D_KERNEL_DEPTH_OVERFLOW	transposed depth conv 3d kernel depth overflow.
XCOM_OP_TDPTCONV3D_STRIDE_HEIGHT_OVERFLOW	transposed depth conv 3d stride height overflow.
XCOM_OP_TDPTCONV3D_STRIDE_WIDTH_OVERFLOW	transposed depth conv 3d stride width overflow.
XCOM_OP_TDPTCONV3D_STRIDE_DEPTH_OVERFLOW	transposed depth conv 3d stride depth overflow.
XCOM_OP_TDPTCONV3D_OCG_NOT_ENOUGH	transposed depth conv 3d output channel group mult channel parallel is less than feature channel size.

Error Code ID	Error Message
XCOM_OP_DPTCONV_IC_WEIGHT_DEPTH_NE	depth conv need kernel mult input channel group equal to weight bank depth
XCOM_OP_DPTCONV3D_KERNEL_HEIGHT_OVERFLOW	depth conv 3d kernel height overflow.
XCOM_OP_DPTCONV3D_KERNEL_WIDTH_OVERFLOW	depth conv 3d kernel width overflow.
XCOM_OP_DPTCONV3D_KERNEL_DEPTH_OVERFLOW	depth conv 3d kernel depth overflow.
XCOM_OP_DPTCONV3D_STRIDE_HEIGHT_OVERFLOW	depth conv 3d stride height overflow.
XCOM_OP_DPTCONV3D_STRIDE_WIDTH_OVERFLOW	depth conv 3d stride width overflow.
XCOM_OP_DPTCONV3D_STRIDE_DEPTH_OVERFLOW	depth conv 3d stride depth overflow.
XCOM_OP_DPTCONV3D_OCG_NOT_ENOUGHT	depth conv 3d output channel group mult channel parallel is less than feature channel size.
XCOM_OP_THRESHOLD_HEIGHT_NE	threshold operator need input and output have same height.
XCOM_OP_THRESHOLD_WIDTH_NE	threshold operator need input and output have same width.
XCOM_OP_THRESHOLD_CHANNEL_NE	threshold operator need input and output have same channel.
XCOM_OP_TILE_HEIGHT_NE	tile operator need input and output height have scale multiple relationship.
XCOM_OP_TILE_WIDTH_NE	tile operator need input and output width have scale multiple relationship.
XCOM_OP_TILE_CHANNEL_NE	tile operator need input and output channel have scale multiple relationship.
XCOM_OP_UPSAMPLE_HEIGHT_NE	upsample operator need input and output height have scale multiple relationship.
XCOM_OP_UPSAMPLE_WIDTH_NE	upsample operator need input and output width have scale multiple relationship.
XCOM_OP_UPSAMPLE_CHANNEL_NE	upsample operator need input and output channel have scale multiple relationship.
XCOM_OP_ELEW_IO_CHANNEL_NE	element wise operator need input and output channel equal
XCOM_OP_ELEW3D_IO_CHANNEL_NE	element wise 3d operator need input and output channel equal
XCOM_OP_MUL_IO_HEIGHT_NE	mul operator need input and output have same height.
XCOM_OP_MUL_IO_WIDTH_NE	mul operator need input and output have same width.
XCOM_OP_MUL_IO_DEPTH_NE	mul operator need input and output have same depth.
XCOM_OP_MUL_IO_CHANNEL_NE	mul operator need input and output have same height.
XCOM_OP_MVR_IO_HEIGHT_NE	mvr operator need input and output have same height.
XCOM_OP_MVR_IO_WIDTH_NE	mvr operator need input and output have same width.
XCOM_OP_MVR_IO_DEPTH_NE	mvr operator need input and output have same depth.
XCOM_OP_MVR_IO_CHANNEL_NE	mvr operator need input and output have same height.
XCOM_OP_CONV_KERNEL_WIDTH_OVERFLOW	kernel width is larger than input width plus padding. that makes window cannot slide
XCOM_OP_CONV_KERNEL_HEIGHT_OVERFLOW	kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_CONV_STRIDE_WIDTH_OVERFLOW	kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation

Error Code ID	Error Message
XCOM_OP_CONV_STRIDE_HEIGHT_OVERFLOW	kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_CONV_KERNEL_DEPTH_OVERFLOW	kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_KERNEL_HEIGHT_OVERFLOW	kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_STRIDE_WIDTH_OVERFLOW	kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_STRIDE_HEIGHT_OVERFLOW	kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_KERNEL_DEPTH_OVERFLOW	kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_DILATION_HEIGHT_OVERFLOW	dilation height is too large for input height
XCOM_OP_TCONV3D_DILATION_WIDTH_OVERFLOW	dilation height is too large for input height
XCOM_OP_TCONV3D_DILATION_DEPTH_OVERFLOW	dilation height is too large for input height
XCOM_OP_TCONV3D_ICG_WEIGHT_DEPTH_OVERFLOW	input channel group stride overflow the weight bank depth.
XCOM_OP_CONV_DILATION_WIDTH_ALL_PAD	padding width too large for dilation, make all value in slide window are padding without input.
XCOM_OP_CONV_DILATION_HEIGHT_ALL_PAD	padding height too large for dilation, make all value in slide window are padding without input.
XCOM_OP_CONV_DILATION_DEPTH_ALL_PAD	padding depth too large for dilation, make all value in slide window are padding without input.
XCOM_OP_CONV_STRIDE_OVERFLOW	input channel stride overflow the weight bank depth.
XCOM_OP_TCONV3D_KENREL_DEPTH_OVERLARGE	kernel depth too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_KENREL_WIDTH_OVERLARGE	kernel width too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_KENREL_HEIGHT_OVERLARGE	kernel height too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_DILATION_WIDTH_ALL_PAD	padding width too large for dilation, makes all value in slide window are padding without input feature.
XCOM_OP_TCONV3D_DILATION_HEIGHT_ALL_PAD	padding height too large for dilation, makes all value in slide window are padding without input feature.
XCOM_OP_TCONV3D_DILATION_DEPTH_ALL_PAD	padding dpeth too large for dilation, makes all value in slide window are padding without input feature.
XCOM_ACGEN_CONV_ERROR	error parameters while generating convolution, like some convolution parameter exceed hardware limitation or unexpected middle result generated.
XCOM_BANK_CONV_ERROR	error banking status or behavior for convolution operation wihle generating assembly code. please check conv op data flow and tensor aggregation.
XCOM_BANK_INVALID_ID	invalid id wihle parsing for finding bank id. please check bank name in target_factory.
XCOM_STR_PARSE_FAILED	Failed to parse specific string, perhaps it's an illegal string or empty string.
XCOM_DATA_SEGMENT_FAULT	data tensor or const tensor index exceed max tensor size, please check index value.

Error Code ID	Error Message
XCOM_OP_CONFIG_MISSING	Failed to get specific op config. please check op type config file.
XCOM_AIE_TARGET_INIT_FAILED	Failed to init aie target.
XCOM_AIE_SHIMTILE_OVERFLOW	aie tiling shim index or shim size out of range.
XCOM_AIE_MEMTILE_OVERFLOW	aie tiling memory index or memory size out of range.
XCOM_AIE_AIETILE_OVERFLOW	aie tiling index or bd index out of range.
XCOM_AIE_OUT_OF_BD	cannot find free bd for mem tiling.
XCOM_SLNODE_UNREGISTED	slnode target, type, name, or any info unregisted.
XCOM_INVOKE_BASE	An unproper function invoking occured!
XCOM_VALUE_UNMATCH	The value is not supposed!
XCOM_MEANINGLESS_VALUE	The value is meaningless.
XCOM_SIZE_UNMATCH	The object's size is not not matching the requirement.
XCOM_OPERATOR_UNSUPPORT	This operator is not supposed!
XCOM_LEAF_SUBGRAPH_REQUIRED	Here requires a leaf subgraph.
XCOM_UNACCEPTABLE_SUBGRAPH	The subgraph is not allowed or meeting the requirements.
XCOM_PASS_MISS	Some compiler pass is missed.
XCOM_PASS_DEPENDENCY	Something wrong about pass dependency.
XCOM_DEBUGMANAGER_NOT_RECORDING	Invalide status for DebugManager recording.
XCOM_NO_PASS_RECORDED	No Pass has been recorded in DebugManager.
XCOM_DEBUGMANAGER_UNRECORDED_OP	Unrecorded op found.
XCOM_DDR_ADDR_ASSIGNMENT_FAILED	DDR address assignment is failed.
XCOM_DDR_PARAM_SPACE_INITIALIZTION_SIZE_ERROR	DDR parameter space initialization size error. Please concat us.
XCOM_UNIMPLEMENT	This part of function is unimplement.
XCOM_UNDEFINED_STATE	This behavior is undefined.
XCOM_EXECUTE_SYSTEM_CMD_FAILED	Error occured when execute the system command.
XCOM_TENSOR_DIMENSION_UNMATCH	The tensor dimension is unexpected.
XCOM_DATA_OUTRANGE	Data value is out of range!
XCOM_TYPE_UNMATCH	Unmatched type!
XCOM_ITEM_UNDEFINED	The requested item was not found or defined!
XCOM_OPERATION_FAILED	The supposed operation is failed!
XCOM_DIR_OPEN_FILE_FAILED	The file can't be read or can't access it.
XCOM_INVALID_GRAPH	Subgraph is null or error subgraph type like cpu subgraph using for dpu
XCOM_UNREGISTED_STRATEGY	This error code only used in dead code
XCOM_INVALID_ARCH_PARAM	This error code only used in dead code
XCOM_ACGEN_ERROR	instuction generating fail, please contact us.
XCOM_UNEXPECTED_VALUE	Inappropriate value at this place like nullptr or non-one value
XCOM_UNEXPECTED_ARCH	Unknown architecture name, please check target_factory
XCOM_ARCH_UNREGISTED	Unknown architecture name, please check target_factory
XCOM_ARCH_INVALID_NAME	Failed to file arch name in config dict, target factory or arch name serialization, please check arch name.
XCOM_FILE_NOT_EXISTS	The file is not exists

Error Code ID	Error Message
XCOM_TARGET_REQUIRED	The compiler requires the target.
XCOM_GRAPH_REQUIRED	The compiler requires an input graph.
XCOM_LOGICAL_CONDITION_ERROR	The logical condition is wrong.
XCOM_INVALID_SUPERLAYER	The superlayer subgraph is invalid.
XCOM_UNSUPPORT_QUANTIZATION	The fix info is error or unsupported.
XCOM_SWIM_NODE_TYPE_ERROR	mismatch slnode type
XCOM_SWIM_OUTPUT_MISSING	swim lane output bank smf missing.
XCOM_SWIM_UNDEFINED_TYPE	undefined swim prgrame or lane type.
XCOM_ALLOCATE_BANK_FAIL	XCompiler occurs error when allocating bank. Please contact us.
XCOM_TILING_FAIL	XCompiler occurs error when tiling. Please contact us.
XCOM_PM_FAIL	The compiler occurs an error when generating instructions, please contact us.
XCOM_SUBGRAPH_ATTR_MISSING	The subgraph attribute is missing.
XCOM_ASSIGN_OUTPUT_OPS_FAILED	Assign output ops failed.
XCOM_DDR_REG_ID_SIZE_UNMATCH	The DDR reg id size unmatch. Please contact us.
XCOM_DDR_OPTIMIZATION_0_FAILED	DDR assignment optimization failed (code 0). Please contact us.
XCOM_DDR_OPTIMIZATION_1_FAILED	DDR assignment optimization failed (code 1). Please contact us.
XCOM_TRANSPOSED_CONV_WEIGHTS_DDR_OPTIMIZATION_ERROR	DDR assignment optimization failed during optimizing the transposed convolution's weights. Please contact us.
XCOM_DIRECTORY_EXIST	The dirctory is existing, can't be created multiple times.
XCOM_DIRECTORY_NOT_EXIST	The dirctory is not existing.
XCOM_INVALID_COMPILE_MODE	The compile mode is not supported now.
XCOM_INVALID_TARGET	Invalid DPU target
XCOM_DPU_MEMORY_ALLOCATION_FAILED	error mapping smfs onto dpu banks since error input group of smfs like no specific type (data, const data (weights, bias ...)) found in given Smf group. Or unaligned smf info on data width, height or their stride, channel and channel group stride. Unaligned smfs cannot be aggregated on aggregation dimension.
XCOM_UNSUPPORT_NONLIEAR_TYPE	The nonlinear type is unsupported by DPU.
XCOM_PAD_KERNEL_SIZE_UNMATCH	The pad size is not correct comparing with the kernel size.
XCOM_DATA_DEPENDCY_MISSING	Generate code failed.
XCOM_MULTIPLE_WEIGHTS	There are more than one weights for some ops.
XCOM_MULTIPLE_BIAS	There are more than one bias for some ops.
XCOM_MULTIPLE_PRELU	There are more than oen prelu for some ops.
XCOM_TOO_MANY_INPUTS	There are too many inputs for the op.
XCOM_TENSOR_SHAPE_UNMATCH	Tensor shapes for some ops are not matching.
XCOM_UNSUPPORT_KERNEL_SIZE	The op's kernel size is not supported.
XCOM_CODE_GEN_ERROR	Code generation fail.
XCOM_UNSUPPORT_ROUND_MODE	The round mode is not supported.
XCOM_ADDITION_OVERFLOW	The addtion is overflow.
XCOM_INT_COMPOSITION_INVALID_RANGE	The integer composition's output range constraint is invalid. Please contact us.
XCOM_TO_XINT_DATA_SIZE_UNMATCH	The input data vector's size is unmatching with the xint's bit width

Error Code ID	Error Message
XCOM_REVERT_XINT_DATA_SIZE_UNMATCH	The input data vector's size is unmatching with the xint's original bit width.
XCOM_DWCONV_PARAM_REORDER_SIZE_UNMATCH	size unmatching occured during reordering the DepthwiseConv2d's parameter.
XCOM_CONV_PARAM_REORDER_SIZE_UNMATCH	size unmatching occured during reordering the Conv2d's parameter.
XCOM_AIE_CORE_NUM_MISMATCH	The config of aie core number mismatchs.
XCOM_AIE_TIMING_CONFIG_MISMATCH	The config inside aie timing calculating mismatchs.
XCOM_AIE_TILING_FAIL	There is not enough bank space inside AIE local memory for this tensor
XCOM_AIE_UNSUPPORTED_OP	Unsupported op for aie tiling
XCOM_PARTITIONENGINE_HINTS_ERROR	The partition engine's hints are invalid. Please contact us.
XCOM_PARSE_FAIL	Failed to parse structured data!
XCOM_PARTITION_REPEATED_REGISTRATION	Repeated checker registration for the op_type and arch_type in partition.
XCOM_UNREGISTED_SLNODE	Unregisted Slnode
XCOM_GET_CHANNEL_FAILED	xGet channel failed.
XCOM_GET_PACKET_ID_FAILED	xGet packet id failed.
XCOM_GET_PACKET_TYPE_FAILED	xGet packet type failed.
XCOM_GET_LOCK_ID_FAILED	xGet lock id failed.
XCOM_GET_BD_FAILED	xGet bd failed.
XCOM_SMF_SPEC_MISSING	no found such spec for the smf
XCOM_SMF_MISSING	no found such smf
XCOM_SMF_Y_SIZE_ERROR	smf y direction size overflow bank height with padding.
XCOM_SMF_C_SIZE_ERROR	channel direction smf need height = 1, width = 1, pad top and bottom = 0.
XCOM_SMF_COORDINATE_ERROR	smf on coordinate have error size or missing.
XCOM_SMF_CONCAT_ERROR	smf on concatenate have error size or missing.
XCOM_SMF_BIAS_ERROR	bias smf size error or missing.
XCOM_SMF_PARAM_ERROR	param smf size error or missing.
XCOM_SMF_TYPE_ERROR	error smf type.
XCOM_SMF_TENSOR_TYPE_ERROR	tensor type error while smf arrangment
XCOM_SMF_RESERVED_ERROR	smf dynamic size error or missing or addressing failed.
XCOM_SMF_BANK_MANAGEMENT_ERROR	error happens in bank management, error name addressing or missing.
XCOM_BANK_SMF_EXIST	smf name already exists in bank.
XCOM_BANK_ADDR_MISSING	bank addressing missing.
XCOM_BANK_ADDR_OVERFLOW	bank addressing overflow.
XCOM_BANK_ADDR_ERROR	bank block addr have error sequence or non-exist addressing.
XCOM_USER_FILE_NOT_EXISTS	The file does not exist
XCOM_USER_FILE_OPERATION_ERR	The file operation failed
XCOM_USER_INVALID_COMPILE_MODE	The compile mode is not supported now.
XCOM_USER_DIRECTORY_NOT_EXIST	The dirctory does not exist, please create it first.
XCOM_USER_DIRECTORY_ALREADY_EXIST	The dirctory already exist, please remove it first.
XCOM_USER_INVALID_CMD_PARAM	Invalid cmdline parameter

Error Code ID	Error Message
XCOM_USER_INVALID_TARGET	Invalid DPU target is given by cmdline.
XCOM_USER_INVALID_OUTPUT_OPS	The ouput ops user specified can't be found in the network. Please check the op names.
XCOM_USER_INVALID_OUTPUT_TENSORS	The ouput tensors user specified can't be found in the network. Please check the tensor names.
XCOM_USER_UNSUPPORTED_SYSTEM	The function is not supported by current operating system.
XCOM_PASS_DEPENDENCY_ERROR	Something wrong about pass dependency.
XCOM_PASS_UNREGISTERED	Pass has not been registered.
XCOM_PASS_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_PASS_TARGET_UNIMPLEMENTED	The target has not been implemented yet.
XCOM_PASS_GRAPH_ATTR_MISSING	Necessary attribution has not been set for the graph.
XCOM_PASS_OP_INVALID_ATTR	The parameter of the operator is invalid. Please check the input network.
XCOM_PASS_OP_ATTR_MISMATCH	The parameter of the operator is unexcepted.
XCOM_PASS_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_PASS_OP_ATTR_MISSING	The requisite parameter is missing. Please check input network.
XCOM_PASS_TENSOR_SIZE_ERROR	Size of the tensor is invalid. Please check input network.
XCOM_PASS_TENSOR_SIZE_MISMATCH	Size mismatch between correlative tensors.
XCOM_PASS_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_COMGRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_COMGRAPH_OP_CONNECTION_MISSING	The connection is missing in the graph.
XCOM_COMGRAPH_OP_CONNECTION_INVALID	The connection between two specific op is unexpected.Please check input network.
XCOM_COMGRAPH_ATTR_NOT_ASSIGNED	The requisite attribution of xcomgraph has not be assigned.
XCOM_COMGRAPH_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_COMGRAPH_SUBGRAPH_MISSING	The subgraph is missing.
XCOM_COMGRAPH_BANK_UNEXPECTED_STATE	Bank assignment is rejected by a particular subgraph.
XCOM_COMGRAPH_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_FRONTEND_SUBGRAPH_MISSING	The subgraph is missing.
XCOM_FRONTEND_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_FRONTEND_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_FRONTEND_GRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_FRONTEND_GRAPH_TEMPLATE_MISMATCH	Pattern mismatch between template and replacing process.
XCOM_FRONTEND_GRAPH_LEVEL_MISMATCH	A unsuitable level of graph is given for current function.
XCOM_FRONTEND_GRAPH_ATTR_MISSING	Necessary attribution has not been set for the graph.
XCOM_FRONTEND_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_FRONTEND_OP_INVALID_ATTR	The parameter of the operator is invalid. Please check the input network.
XCOM_FRONTEND_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_FRONTEND_OP_UNSUPPORTED_BLOB_NUMBER	Blob number of the operator is unsupported by target DPU.

Error Code ID	Error Message
XCOM_FRONTEND_OP_UNSUPPORTED_ATTR	An attribution of the operator is unsupported by target DPU.
XCOM_FRONTEND_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_FRONTEND_OP_UNSUPPORTED_NONLIEAR	The nonlinear(or activation) is unsupported by target DPU
XCOM_FRONTEND_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_FRONTEND_OP_ATTR_MISMATCH	The parameter of the operator is unexcepted.
XCOM_FRONTEND_QUANT_ATTR_MISSING	Quantizing information for the op is mission
XCOM_FRONTEND_QUANT_ATTR_OUT_OF_RANGE	The shiftbit for quantizing is out of range, that the range is restricted by target DPU.
XCOM_FRONTEND_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_FRONTEND_TENSOR_SHAPE_MISMATCH	Shape mismatch between 2 correlative tensors.
XCOM_FRONTEND_TENSOR_SIZE_MISMATCH	Size mismatch between 2 correlative tensors.
XCOM_FRONTEND_DATA_TYPE_MISMATCH	Data type mismatch between 2 correlative tensors.
XCOM_FRONTEND_BITWIDTH_MISMATCH	The bit width of the tensor is unexcepted.
XCOM_GENINST_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_GENINST_INVALID_VALUE	A invalid value is given for specific function.
XCOM_GENINST_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_GENINST_OP_UNSUPPORTED	The operator has not been implemented by target dpu.
XCOM_GENINST_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_GENINST_OP_UNSUPPORTED_NONLIEAR	The nonlinear(or activation) is unsupported by target DPU
XCOM_GENINST_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_GENINST_TARGET_UNIMPLEMENTED	The target has not been implemented yet.
XCOM_GENINST_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_GENINST_GRAPH_TEMPLATE_MISMATCH	Pattern mismatch between template and replacing process.
XCOM_GENINST_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_GENINST_TILING_FAIL	Failed to get a available tiling scheme.
XCOM_GENINST_CODE_GEN_ERROR	Code generation fail.
XCOM_OPFACTORY_ILLEGAL_NAME	The object name is illegal.
XCOM_OPFACTORY_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_OPFACTORY_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_OPFACTORY_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_OPFACTORY_OP_UNSUPPORTED_BLOB_NUMBER	Blob number of the operator is unsupported by target DPU.
XCOM_OPFACTORY_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_OPFACTORY_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_OPFACTORY_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_OPTIMIZE_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_OPTIMIZE_GRAPH_OPERATION_FAILED	A graphic problem has caused by graphic operation.

Error Code ID	Error Message
XCOM_OPTIMIZE_OP_UNSUPPORTED_ATTR	An attribution of the operator is unsupported by target DPU.
XCOM_OPTIMIZE_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_OPTIMIZE_TENSOR_SHAPE_INVALID	Tensor shape is invalid. Please check the input network.
XCOM_OPTIMIZE_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_OPTIMIZE_OP_ATTR_MISMATCH	The parameter of the operator is unexcepted.
XCOM_UTIL_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_UTIL_OPERATION_DENIED	The operation is not permited.
XCOM_UTIL_PARSE_FAIL	Failed to parse structured data.
XCOM_UTIL_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_UTIL_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_UTIL_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_UTIL_TENSOR_SHAPE_INVALID	Tensor shape is invalid. Please check the input network.
XCOM_UTIL_DATA_TYPE_INVALID	Data type is invalid for current context.
XCOM_UTIL_ATTR_OUT_OF_RANGE	The value is out of range.
XCOM_UTIL_INVALID_VALUE	A invalid value is given for specific function.
XCOM_UTIL_INVALID_VALUE_RANGE	The data range is illegal.
XCOM_UTIL_ADDITION_OVERFLOW	The addtion is overflow.
XCOM_UTIL_DATA_SIZE_MISMATCH	Size mismatch between 2 data chunk.
XCOM_UTIL_BANK_ALLOC_FAILED	Failed to get a available scheme of bank assignment.
XCOM_BANKASSIGN_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_BANKASSIGN_INVALID_ITEM	The item is not available yet.
XCOM_BANKASSIGN_OUT_OF_BANK_SIZE	Insufficiency of bank size, input model might be too large.
XCOM_BANKASSIGN_TARGET_ENGINE_UNREGISTERED	The engine is not registered for dpu target.
XCOM_BANKASSIGN_INVALID_VALUE	A invalid value is given for specific function.
XCOM_BANKASSIGN_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_BANKASSIGN_BANK_UNEXPECTED_STATE	Bank space manager might be in disorder.
XCOM_BANKASSIGN_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_BANKASSIGN_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_PARTITION_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_PARTITION_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_DDRALLOC_NULL_POINTER	Accessing of uninitialized object or pointer.
XCOM_DDRALLOC_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_DDRALLOC_INVALID_ITEM	Failed to get a available scheme of bank assignment.
XCOM_DDRALLOC_ITEM_UNDEFINED	The item has not be defined.
XCOM_DDRALLOC_DATA_SIZE_MISMATCH	Size mismatch between 2 data chunk.
XCOM_DDRALLOC_MEM_ACCESS_OVERFLOW	Memory access overflow.
XCOM_DDRALLOC_OUT_OF_MEM	DDR space is not enough for current model.

Error Code ID	Error Message
XCOM_DDRALLOC_FEATURE_UNIMPLEMENT	The feature of DDR allocating optimization is not implemented.
XCOM_DDRALLOC_OPERATION_DENIED	The operation is not permited.
XCOM_DDRALLOC_TARGET_UNIMPLEMENTED	The target has not been implemented yet.
XCOM_DDRALLOC_ASSIGNMENT_STATE_ERROR	The state of DDR allocation is unexcepted.
XCOM_DDRALLOC_PARAM_SPACE_INITIALIZTION_SIZE_ERROR	DDR parameter space initialization size error. Please concat us.
XCOM_DDRALLOC_GRAPH_LEVEL_MISMATCH	A unsuitable level of graph is given for current function.
XCOM_DDRALLOC_GRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_DDRALLOC_GRAPH_INVALID_STRUCTURE	There is a unexcepted pattern in the graph. Please check the input network.
XCOM_DDRALLOC_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_DDRALLOC_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_DDRALLOC_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_DDRALLOC_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_DDRALLOC_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_DDRALLOC_TENSOR_SHAPE_MISMATCH	Shape mismatch between 2 correlative tensors.
XCOM_DDRALLOC_TENSOR_SIZE_ERROR	Size of the tensor is invalid. Please check input network.
XCOM_DDRALLOC_TENSOR_SIZE_MISMATCH	Size mismatch between 2 correlative tensors.
VAILIB_DPU_TASK_NOT_FIND	Model files not find
VAILIB_DPU_TASK_OPEN_ERROR	Open file failed
VAILIB_DPU_TASK_CONFIG_PARSE_ERROR	Parse model config file failed
VAILIB_DPU_TASK_TENSORS_EMPTY	Runner has no input tensors
VAILIB_DPU_TASK_SUBGRAPHS_EMPTY	Runner has no subgraphs
VAILIB_CPU_RUNNER_OPEN_LIB_ERROR	dlopen can not open lib
VAILIB_CPU_RUNNER_LOAD_LIB_SYM_ERROR	dlsym load symbol error
VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_FIND	Can not find tensor buffer with this name
VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_CONTINOUS	Tensor buffer not continous
VAILIB_CPU_RUNNER_READ_FILE_ERROR	Fail to read file
VAILIB_CPU_RUNNER_WRITE_FILE_ERROR	Fail to write file
VAILIB_CPU_RUNNER_CPU_OP_NOT_FIND	Can not find op with this name
VAILIB_GRAPH_RUNNER_NOT_FIND	GraphTask can not find tensor or tensor buffer
VAILIB_GRAPH_RUNNER_DPU_BATCH_ERROR	GraphTask get dpu batch not equal
VAILIB_GRAPH_RUNNER_NOT_SUPPORT	The function or value are not supported in graph runner
VAILIB_GRAPH_RUNNER_NOT_OVERRIDE	The funtion has not been overridden
VAILIB_MATH_NOT_SUPPORT	The function or value are not supported in vai-math
VAILIB_MATH_FIX_POS_ERROR	Softmax table not support the fix position value
VAILIB_MODEL_CONFIG_NOT_FIND	Model config info not find
VAILIB_MODEL_CONFIG_OPEN_ERROR	Model config file or directory open error
VAILIB_MODEL_CONFIG_CONFIG_PARSE_ERROR	Model config file parse error

Error Code ID	Error Message
VAILIB_BENCHMARK_LIST_EMPTY	Can not found images. List of images are empty
VAILIB_DEMO_CANVAS_ERROR	Canvas image size is too small
VAILIB_DEMO_GST_ERROR	Failed to open gstreamer
VAILIB_DEMO_IMAGE_LOAD_ERROR	Failed to load image
VAILIB_DEMO_OUT_OF_BOUNDARY	Gui rects out of boundary
VAILIB_DEMO_VIDEO_OPEN_ERROR	Cannot open video stream
VART_OPEN_DEVICE_FAIL	Cannot open device
VART_LOAD_XCLBIN_FAIL	Bitstream download failed
VART_LOCK_DEVICE_FAIL	Cannot lock device
VART_FUNC_NOT_SUPPORT	Function not support
VART_XMODEL_ERROR	Xmodel error
VART_GRAPH_ERROR	Graph error
VART_TENSOR_INFO_ERROR	Tensor info error
VART_DPU_INFO_ERROR	DPU info error
VART_SYSTEM_ERROR	File system error
VART_DEVICE_BUSY	Device busy
VART_DEVICE_MISMATCH	Device mismatch
VART_DPU_ALLOC_ERROR	DPU allocate error
VART_VERSION_MISMATCH	Version mismatch
VART_OUT_OF_RANGE	Array index out of range
VART_SIZE_MISMATCH	Array size not match
VART_NULL_PTR	Nullptr
VART_XRT_NULL_PTR	Nullptr
VART_XRT_DEVICE_BUSY	Device busy
VART_XRT_READ_ERROR	Read error
VART_XRT_READ_CU_ERROR	Read cu fatal
VART_XRT_FUNC_FAULT	XRT function fault
VART_XRT_DEVICE_AVAILABLE_ERROR	No devices available
VART_XRT_CU_AVAILABLE_ERROR	No CU available
VART_XRT_OPEN_CONTEXT_ERROR	xclOpenContext failed
VART_XRM_CREATE_CONTEXT_ERROR	failed to create XRM context
VART_XRM_CONNECT_ERROR	Failed to connect to XRM
VART_XRM_ACQUIRE_CU_ERROR	Could not acquire CU
VART_DEVICE_BUFFER_ALLOC_ERROR	Cannot alloc device buffer -- unknown datatype
VART_XCLBIN_READ_ERROR	Failed to open xclbin file for reading
VART_XCLBIN_DOWNLOAD_ERROR	Bitstream download failed !
VART_CONTROLLER_VIR_MEMORY_ALLOC_ERROR	not enough virtual space on host
VART_VERSION_MISMATCH_ERROR	subgraph's version is mismatch with xclbin
VART_CONTROLLER_DUMP_FOLDER_CREATE_ERROR	Create dump folder error

Error Code ID	Error Message
VART_CONTROLLER_DUMP_SUBFOLDER_CREATE_ERROR	Create sub dump folder error
VART_DEVICE_MEMORY_ALLOC_ERROR	Device memory not enough, alloc fail
VART_TENSOR_BUFFER_CREATE_ERROR	TensorBuffer create fail
VART_TENSOR_BUFFER_INVALID	invalid tensorbuffer input or output
VART_DPU_EXEC_ERROR	DPU run fail
VART_DPU_TIMEOUT_ERROR	DPU timeout
VART_CONTROLLER_DUMP_ERROR	dump failed
VART_XCLBIN_PATH_INVALID	xclbinPath is not set, please consider setting XLNX_VART_FIRMWARE.
VART_GRAPH_FINGERPRINT_ERROR	no hardware info in subgraph
VART_TENSOR_BUFFER_CHECK_ERROR	TensorBuffer size less than offset, input shpe invalid
VART_TENSOR_BUFFER_DIMS_ERROR	input dims shape is invalid
XIR_ACCESS_ADDRESS_OVERFLOW	The address you try to access does not exsit!
XIR_ADD_OP_FAIL	Failed to add an op!
XIR_FILE_NOT_EXIST	File does't exist!
XIR_INTERNAL_ERROR	it is an internal bug supposed never happen
XIR_INVALID_ARG_OCCUR	Invalid arg occurence!
XIR_INVALID_ATTR_DEF	Invalid attribute defination!
XIR_INVALID_ATTR_OCCUR	Invalid attr occurence!
XIR_INVALID_DATA_TYPE	The data type is invalid.
XIR_MEANINGLESS_VALUE	The value you set for this parameter makes no sense.
XIR_MULTI_DEFINED_OP	Multiple definition of OP!
XIR_MULTI_DEFINED_TENSOR	Multiple definition of Tensor!
XIR_MULTI_REGISTERED_ARG	Multiple registration of argument!
XIR_MULTI_REGISTERED_ATTR	Multiple registration of attribute!
XIR_MULTI_REGISTERED_EXPANDED_ATTR	Multiple registration of static attr!
XIR_MULTI_REGISTERED_OP	Multiple registration of operator!
XIR_OPERATION_FAILED	Fail to execute command!
XIR_OP_DEF_SHAPE_HINT_MISSING	A shape hint is required by the op definition
XIR_OP_NAME_CONFLICT	There're at least two ops assigned the same name, please check the ops' name.
XIR_OUT_OF_RANGE	idx out of range!
XIR_PROTECTED_MEMORY	The content in protected momory for tensor can not be modified!
XIR_READ_PB_FAILURE	failed to read pb file
XIR_REMOVE_OP_FAIL	Failed to remove an op!
XIR_SHAPE_UNMATCH	The shape is unmatching.
XIR_SUBGRAPH_ALREADY_CREATED_ROOT	Already created root subgraph for the graph!
XIR_SUBGRAPH_CREATE_CHILDREN_FOR_NONLEAF	
XIR_SUBGRAPH_HAS_CYCLE	Children from a same subgraph depend each other!
XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONCHILD	Cannot merge subgraphs which are not children!

Error Code ID	Error Message
XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONLEAF	Cannot merge subgraphs which are not leaves!
XIR_UNDEFINED_ATTR	Undefined attribute!
XIR_UNDEFINED_INPUT_ARG	Undefined input arg!
XIR_UNDEFINED_OP	Access undefined OP!
XIR_UNEQUIVALENT_ATTRIBUTE	These two attibutes/parameters are not equivalent.
XIR_UNEXPECTED_VALUE	Unexpected value!
XIR_UNKNOWNTYPE_TENSOR	The DataType of this tensor is not specified.
XIR_UNREGISTERED_ARG	Unregistered argument!
XIR_UNREGISTERED_ATTR	Unregistered attribute!
XIR_UNREGISTERED_OP	Unregistered operator!
XIR_UNSUPPORTED_ROUND_MODE	unsupported round mode.
XIR_UNSUPPORTED_TYPE	unsupported data type for attr value
XIR_VALUE_UNMATCH	Value unmatch!
XIR_WRITE_PB_FAILURE	failed to write pb file
XIR_XIR_UNDEFINED_OPERATION	Undefined operation!
TARGET_EXPLORER_XCLBIN_ERROR	No xclbin specified
TARGET_EXPLORER_XCLBIN_ENV_ERROR	DPU xclbin path specified by 'XLNX_VART_FIRMWARE' not exist, please check!
TARGET_EXPLORER_XCLBIN_ENV_VAL_ERROR	'XLNX_VART_FIRMWARE' need to be specified like /path/to/xxx.xclbin, please check!
TARGET_EXPLORER_SYS_DEVICE_CHECK_ERROR	The system has no device
TARGET_EXPLORER_XCLBIN_SET_ERROR	xclbinPath is not set, please consider setting XLNX_VART_FIRMWARE.
TARGET_EXPLORER_NO_DPU_ERROR	xclbin is not for DPU, can't find DPU kernel in xclbin
TARGET_EXPLORER_BATCH_ERROR	Only support multiple DPU cores with same batch and fingerprint
TARGET_EXPLORER_DEVICE_CHECK_ERROR	No device available for current xclbin
TARGET_FACTORY_INVALID_ARCH	Invalid target arch!
TARGET_FACTORY_INVALID_FEATURE_CODE	Invalid target feature code!
TARGET_FACTORY_INVALID_ISA_VERSION	Invalid target ISA version!
TARGET_FACTORY_INVALID_TYPE	Invalid target type!
TARGET_FACTORY_MULTI_REGISTERED_TARGET	Multiple registration of target!
TARGET_FACTORY_PARSE_TARGET_FAIL	Fail to parse target from prototxt!
TARGET_FACTORY_UNREGISTERED_TARGET	Unregistered target!

VART Programming APIs

This section describes all the programming APIs offered by the VART programming interface.

C++ APIs

Class 1

The class name is `vart::Runner`. The following table lists all the functions defined in the `vitis::vart::Runner` class.

Table 41: **Quick Function Reference**

Return Type	Name	Arguments
std::unique_ptr<Runner>	create_runner	const xir::Subgraph* subgraph const std::string& mode
std::vector<std::unique_ptr<Runner>>	create_runner	const std::string& model_directory
std::pair<uint32_t, int>	execute_async	const std::vector<TensorBuffer*>& input const std::vector<TensorBuffer*>& output
int	wait	int jobID int timeout
TensorFormat	get_tensor_format	
std::vector<const xir::Tensor*>	get_input_tensors	
std::vector<const xir::Tensor*>	get_output_tensors	

For `vart::Runner` example, refer to vart::Runner Example.

Class 2

The class name is `vart::TensorBuffer`. The following table lists all the functions defined in the `vart::TensorBuffer` class.

Table 42: **Quick Function Reference**

Return Type	Name	Arguments
location_t	get_location	
const xir::Tensor*	get_tensor	
std::pair<std::uint64_t, std::size_t>	data	const std::vector<std::int32_t> idx = {}
std::pair<uint64_t, size_t>	data_phy	const std::vector<std::int32_t> idx
void	sync_for_read	uint64_t offset, size_t size
void	sync_for_write	uint64_t offset, size_t size
void	copy_from_host	size_t batch_idx, const void* buf, size_t size, size_t offset
void	copy_to_host	size_t batch_idx, void* buf, size_t size, size_t offset
void	copy_tensor_buffer	vart::TensorBuffer* tb_from, vart::TensorBuffer* tb_to

Class 3

The class name is `vart::RunnerExt`. The following table lists all the functions defined in the vart::RunnerExt class.

Table 43: **Quick Function Reference**

Return Type	Name	Arguments
std::vector<vart::TensorBuffer*>	get_inputs	
std::vector<vart::TensorBuffer*>	get_outputs	

Class 4

The class name is `vitis::ai::GraphRunner`. The following table lists all the functions defined in the `vitis::ai::GraphRunner` class.

Table 44: **Quick Function Reference**

Return Type	Name	Arguments
std::unique_ptr<vart::RunnerExt>	create_graph_runner	const xir::Graph* graph, xir::Attrs* attrs

For `graph_runner` example, refer to Graph runner Example.

create_runner

Creates an instance of CPU/SIM/DPU runner by subgraph. This is a factory method.

Prototype

```
std::unique_ptr<Runner> create_runner(const xir::Subgraph* subgraph,
                                      const std::string& mode = "");
```

Parameters

The following table lists the `create_runner` function arguments.

Table 45: **create_runner Arguments**

Type	Name	Description
const xir::Subgraph*	subgraph	XIR Subgraph
const std::string&	mode	3 mode supported: 'ref' - CPU runner 'sim' - Simulation 'run' - DPU runner

Returns

An instance of CPU/SIM/DPU runner.

Usage

The runner instance has a number of member functions to control the execution and get the input and output tensors of the runner. This API can be used like:

```
auto runner = vart::Runner::create_runner(subgraph, "run");
```

create_runner

Creates a DPU runner by model_directory.

Prototype

```
std::vector<std::unique_ptr<Runner>> create_runner(const std::string&
model_directory);
```

Parameters

The following table lists the `create_runner` function arguments.

Table 46: **create_runner Arguments**

Type	Name	Description
conststd::string&	model_directory	The directory name which contains meta.json

Returns

A vector of DPU runner.

execute_async

Executes the runner. This is a blocking function.

Prototype

```
virtual std::pair<uint32_t, int> execute_async(
      const std::vector<TensorBuffer*>& input,
      const std::vector<TensorBuffer*>& output) = 0;
```

Parameters

The following table lists the `execute_async` function arguments.

Table 47: **execute_async Arguments**

Type	Name	Description
conststd::vector<TensorBuffer*>&	input	Vector of the input Tensor buffers containing the input data for inference.
conststd::vector<TensorBuffer*>&	output	Vector of the output Tensor buffers which will be filled with output data.

Returns

pair<jobID, status> status 0 for exit successfully, others for customized warnings or errors.

wait

Waits for the end of DPU processing.

Prototype

```
int wait(int jobid, int timeout)
```

Parameters

The following table lists the `wait` function arguments.

Table 48: **wait Arguments**

Type	Name	Description
int	jobid	job id, neg for any id, others for specific job id

Table 48: **wait Arguments** *(cont'd)*

Type	Name	Description
int	timeout	timeout, negative for block forever, 0 for non-block, pos for block with a limitation(ms).

Returns

Status 0 for exit successfully, others for customized warnings or errors.

get_tensor_format

Gets the tensor format of runner.

Prototype

```
TensorFormat get_tensor_format();
```

Parameters

None

Returns

TensorFormat: NHWC / HCHW

Usage

```
auto format = runner->get_tensor_format();
switch (format) {
    case vart::Runner::TensorFormat::NCHW:
        // do something
        break;
    case vart::Runner::TensorFormat::NHWC:
        // do something
        break;
}
```

get_input_tensors

Gets all input tensors of runner.

Prototype

```
std::vector<const xir::Tensor*> get_input_tensors()
```

Parameters

None

Returns

All input tensors. A vector of raw pointer to the input tensor.

Usage

Each element of the list returned by get_input_tensors() corresponds to a DPU runner input. It has a number of class attributes which can be used like:

```
inputTensors = runner->get_input_tensors();
for (auto input : inputTensor) {
    input->get_name();
input->get_shape();
input->get_element_num();
}
```

get_output_tensors

Gets all output tensors of runner.

Prototype

```
std::vector<const xir::Tensor*> get_output_tensors()
```

Parameters

None

Returns

All output tensors. A vector of raw pointer to the output tensor.

Usage

Each element of the list returned by get_output_tensors() corresponds to a DPU runner output. It has a number of class attributes which can be used like:

```
outputTensors = runner->get_output_tensors();
for (auto output : outputTensor) {
    output->get_name();
output->get_shape();
output->get_element_num();
}
```

vart::Runner Example

This example assumes that you have a DPU subgraph called dpu_subgraph.

The way to create a DPU runner to run dpu_subgraph is shown below.

```
// create runner
auto runner = vart::Runner::create_runner(dpu_subgraph, ”run”);
// get input tensors
auto input_tensors = runner->get_input_tensors();
// get input tensor buffers
auto input_tensor_buffers = std::vector<vart::TensorBuffer*>();
    for (auto input : input_tensors) {
        auto t = vart::alloc_cpu_flat_tensor_buffer(input);
        input_tensor_buffers.emplace_back(t.get());
}
// get output tensors
auto output_tensors = runner->get_output_tensors();
// get output tensor buffers
auto output_tensor_buffers = std::vector< vart::TensorBuffer*>();
for (auto output : output _tensors) {
    auto t = vart::alloc_cpu_flat_tensor_buffer(output);
            output_tensor_buffers.emplace_back(t.get());
}
// sync input tensor buffers
for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
            input->get_tensor()->get_shape()[0]);
}
// run runner
auto v = runner->execute_async(input_tensor_buffers, output_tensor_buffers);
auto status = runner->wait((int)v.first, 1000000000);
// sync output tensor buffers
for (auto& output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
    output->get_tensor()->get_shape()[0]);
}
```

get_location

Get where the tensor buffer located.

Prototype

```
location_t get_location();
```

Parameters

None.

Returns

The tensor buffer location, a location_t enum type value.

The following table lists the location_t enum type.

Table 49: **location_t enum type**

Name	Value	Description
HOST_VIRT	0	Only accessible by the host.
HOST_PHY	1	Continuous physical memory, shared among host and device.
DEVICE_0	2	Only accessible by device_*.
DEVICE_1	3	
DEVICE_2	4	
DEVICE_3	5	
DEVICE_4	6	
DEVICE_5	7	
DEVICE_6	8	
DEVICE_7	9	

Usage

```
vart::TensorBuffer* tb;
switch (tb->get_location()) {
            case vart::TensorBuffer::location_t::HOST_VIRT:
                  // do nothing
                  break;
            case vart::TensorBuffer::location_t::HOST_PHY:
                  // do nothing
                  break;
           default:
                  // do nothing
                  break;
      }
```

get_tensor

Get Tensor of TensorBuffer.

Prototype

```
const xir::Tensor* get_tensor()
```

Parameters

None.

Returns

A pointer to the Tensor.

Usage

```
vart::TensorBuffer* tb;
auto shape = tb->get_tensor()->get_shape();
```

data

Get the data address of the index and the size of the data available for use.

Prototype

```
std::pair<std::uint64_t, std::size_t> data(const std::vector<std::int32_t>
idx = {});
```

Parameters

The following table lists the `data` function arguments.

Table 50: **data Arguments**

Type	Name	Description
const std::vector<std::int32_t>	idx	The index of the data to be accessed, its dimension same to the Tensor shape

Returns

A pair of the data address of the index and the size of the data available for use in byte unit.

Usage

```
vart::TensorBuffer* tb;
std::tie(data_addr, tensor_size) = tb->data({0,0,0,0});
```

data_phy

Get the data physical address of the index and the size of the data available for use.

Prototype

```
std::pair<uint64_t, size_t> data_phy(const std::vector<std::int32_t> idx);
```

Parameters

The following table lists the `data_phy` function arguments.

Table 51: **data_phy Arguments**

Type	Name	Description
const std::vector<std::int32_t>	idx	The index of the data to be accessed, its dimension same to the tensor shape

Returns

A pair of the data physical address of the index and the size of the data available for use in byte unit.

Usage

```
vart::TensorBuffer* tb;
std::tie(phy_data, phy_size) = tb->data_phy({0, 0});
```

sync_for_read

Invalid cache for reading before read. It is no-op in case `get_location()` returns DEVICE_ONLY or HOST_VIRT.

Prototype

```
void sync_for_read(uint64_t offset, size_t size) {};
```

Parameters

The following table lists the `sync_for_read` function arguments.

Table 52: **sync_for_read Arguments**

Type	Name	Description
uint64_t	offset	The start offset address
size_t	size	The data size

Returns

None.

Usage

```
for (auto& output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
                                  output->get_tensor()->get_shape()[0]);
}
```

sync_for_write

Flush cache for writing after write. It is no-op in case `get_location()` returns DEVICE_ONLY or HOST_VIRT.

Prototype

```
void sync_for_write (uint64_t offset, size_t size) {};
```

Parameters

The following table lists the `sync_for_write` function arguments.

Table 53: **sync_for_write Arguments**

Type	Name	Description
uint64_t	offset	The start offset address
size_t	size	The data size

Returns

None.

Usage

```
for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
                             input->get_tensor()->get_shape()[0]);
}
```

copy_from_host

Copy data from the source buffer.

Prototype

```
void copy_from_host(size_t batch_idx, const void* buf, size_t size, size_t
offset);
```

Parameters

The following table lists the `copy_from_host` function arguments.

Table 54: **copy_from_host Arguments**

Type	Name	Description
size_t	batch_idx	The batch index

Table 54: **copy_from_host Arguments** *(cont'd)*

Type	Name	Description
const void*	buf	Source buffer start address
size_t	size	Data size to be copied
size_t	offset	The start offset to be copied

Returns

None.

Usage

```
vart::TensorBuffer* tb_from;
vart::TensorBuffer* tb_to;
for (auto batch = 0u; batch < batch_size; ++batch) {
           std::tie(data, tensor_size) = tb_from->data({(int)batch, 0, 0,
0});
           tb_to->copy_from_host(batch, reinterpret_cast<const void*>(data),
                          tensor_size, 0u);
}
```

copy_to_host

Copy data to the destination buffer.

Prototype

```
void copy_to_host(size_t batch_idx, void* buf, size_t size, size_t offset);
```

Parameters

The following table lists the `copy_to_host` function arguments.

Table 55: **copy_to_host Arguments**

Type	Name	Description
size_t	batch_idx	The batch index
void*	buf	Destination buffer start address
size_t	size	Data size to be copied
size_t	offset	The start offset to be copied

Returns

None.

Usage

```
vart::TensorBuffer* tb_from;
vart::TensorBuffer* tb_to;
for (auto batch = 0u; batch < batch_size; ++batch) {
       std::tie(data, tensor_size) = tb_to->data({(int)batch, 0, 0, 0});
    tb_from->copy_to_host(batch, reinterpret_cast<void*>(data),
                          tensor_size, 0u);
}
```

copy_tensor_buffer

Copy TensorBuffer from one to another.

Prototype

```
static void copy_tensor_buffer(vart::TensorBuffer* tb_from,
vart::TensorBuffer* tb_to);
```

Parameters

The following table lists the `copy_tensor_buffer` function arguments.

Table 56: **copy_tensor_buffer Arguments**

Type	Name	Description
vart::TensorBuffer*	tb_from	The source TensorBuffer
vart::TensorBuffer*	tb_to	The destination TensorBuffer

Returns

None.

Usage

```
vart::TensorBuffer* tb_from;
vart::TensorBuffer* tb_to;
vart::TensorBuffer::copy_tensor_buffer(tb_from.get(), tb_to.get());
```

get_inputs

Gets all input TensorBuffers of RunnerExt.

Prototype

```
std::vector<vart::TensorBuffer*> get_inputs();
```

Parameters

None.

Returns

All input TensorBuffers. A vector of raw pointer to the input TensorBuffer.

Usage

```
auto runner = vart::RunnerExt::create_runner(subgraph, attrs);
auto input_tensor_buffers = runner->get_inputs();
    for (auto input : input_tensor_buffers) {
        auto shape = input->get_tensor()->get_shape();
}
```

get_outputs

Gets all output TensorBuffers of RunnerExt.

Prototype

```
std::vector<vart::TensorBuffer*> get_outputs();
```

Parameters

None.

Returns

All output TensorBuffers. A vector of raw pointer to the output TensorBuffer.

Usage

```
auto runner = vart::RunnerExt::create_runner(subgraph, attrs);
auto output_tensor_buffers = runner->get_outputs();
    for (auto output : output_tensor_buffers) {
        auto shape = output->get_tensor()->get_shape();
}
```

create_graph_runner

Factory function to create an instance of runner by graph and attributes.

Prototype

```
static std::unique_ptr<vart::RunnerExt> create_graph_runner(const
xir::Graph* graph, xir::Attrs* attrs);
```

Parameters

The following table lists the `create_graph_runner` function arguments.

Table 57: **create_graph_runner Arguments**

Type	Name	Description
`const xir::Graph*`	graph	XIR Graph
`xir::Attrs*`	attrs	XIR attrs object, this object is shared among all runners on the same graph.

Returns

An instance of runner.

Usage

```
auto graph = xir::Graph::deserialize(xmodel_file);
auto attrs = xir::Attrs::create();
auto runner = vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
auto input_tensor_buffers = runner->get_inputs();
```

Graph runner Example

The way to create graph runner and the APIs usage of runner are shown below.

```
auto graph = xir::Graph::deserialize(xmodel_file);
auto attrs = xir::Attrs::create();
auto runner = vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
// get input and output tensor buffers
auto input_tensor_buffers = runner->get_inputs();
auto output_tensor_buffers = runner->get_outputs();
// sync input tensor buffers
for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
    input->get_tensor()->get_shape()[0]);
}
// run graph runner
auto v = runner->execute_async(input_tensor_buffers, output_tensor_buffers);
auto status = runner->wait((int)v.first, 1000000000);
// sync output tensor buffers
for (auto& output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
    output->get_tensor()->get_shape()[0]);
}
```

Python APIs

Class 1

The class name is `vart.Runner`. The following table lists all the functions defined in the `vart.Runner` class.

Table 58: **Quick Function Reference**

Type	Name	Arguments
vart.Runner	create_runner	xir.Subgraph subgraph string mode
List[xir.Tensor]	get_input_tensors	
List[xir.Tensor]	get_output_tensors	
tuple[uint32, int]	execute_async	List[vart.TensorBuffer] inputs List[vart.TensorBuffer] outputs Note: vart.TensorBuffer complete with buffer protocol .
int	wait	tuple[uint32, int] jobID

For `vart.Runner Example`, refer to vart.Runner Example.

Class 2

The class name is `vart.RunnerExt`. The following table lists all the functions defined in the `vart.RunnerExt` class.

`vart.RunnerExt` extends from `vart.Runner`.

Table 59: **Quick Function Reference**

Type	Name	Arguments
vart.RunnerExt	create_runner	xir.Subgraph subgraph String mode
List[vart.TensorBuffer]	get_inputs	
List[vart.TensorBuffer]	get_outputs	

For `vart.RunnerExt` example, refer to vart.RunnerExt Example.

Class 3

The class name is `vitis_ai_library.GraphRunner`. The following table lists all the functions defined in the `vitis_ai_library.GraphRunner` class.

Table 60: **Quick Function Reference**

Type	Name	Arguments
vart.RunnerExt	create_graph_runner	xir.Graph graph

For `graph runner` example, refer to Graph runner Example.

create_runner

Creates an instance of DPU runner by subgraph. This is a factory function.

Prototype

```
vart.Runner create_runner(xir.Subgraph subgraph, String mode)
```

Parameters

The following table lists the `create_runner` function arguments.

Table 61: **create_runner Arguments**

Type	Name	Description
xir.Subgraph	subgraph	XIR Subgraph
String	mode	'run' - DPU runner

Returns

An instance of DPU runner.

Usage

```
runner = vart.Runner.create_runner(subgraph, "run")
```

execute_async

Executes the runner. This is a blocking function.

Prototype

```
tuple[uint32_t, int] execute_async(
      List[vart.TensorBuffer] inputs,
      List[vart.TensorBuffer] outputs)
```

***Note*:** vart.TensorBuffer complete with buffer protocol.

Parameters

The following table lists the `execute_async` function arguments.

Table 62: **execute_async Arguments**

Type	Name	Description
List[vart.TensorBuffer]	inputs	A list of vart.TensorBuffer containing the input data for inference.
List[vart.TensorBuffer]	outputs	A list of vart.TensorBuffer which will be filled with output data.

Returns

tuple[jobID, status] status 0 for exit successfully, others for customized warnings or errors.

wait

Waits for the end of DPU processing.

Prototype

```
int wait(tuple[uint32_t, int] jobid)
```

Parameters

The following table lists the `wait` function arguments.

Table 63: **wait Arguments**

Type	Name	Description
tuple[uint32_t, int]	jobid	job id

Returns

Status 0 for exit successfully, others for customized warnings or errors.

get_input_tensors

Gets all input tensors of runner.

Prototype

```
List[xir.Tensor] get_input_tensors()
```

Parameters

None

Returns

A list of DPU runner inputs, each of which have type xir.Tensor.

Usage

Each element of the list returned by get_input_tensors() corresponds to a DPU runner input. Each list element has a number of class attributes which can be displayed like this:

```
inputTensors = dpu_runner.get_input_tensors()
print(dir(inputTensors[0])
```

The most useful of these attributes are name, dims and dtype:

```
for inputTensor in inputTensors:
print(inputTensor.name)
print(inputTensor.dims)
print(inputTensor.dtype)
```

Note that the dimensions (.dim) of an input tensor are in the form NHWC (batchsize, height,width,channels).

get_output_tensors

Gets all output tensors of runner.

Prototype

```
List[xir.Tensor] get_output_tensors()
```

Parameters

None

Returns

All output tensors. A vector of raw pointer to the output tensor.

Usage

```
outputTensors = runner.get_output_tensors()
shapeOut = tuple(outputTensors[0].dims)
```

vart.Runner Example

This example assumes creating a DPU runner from a DPU subgraph (called dpu_subgraph).

```
# create DPU runner
dpu_runner = vart.Runner.create_runner(dpu_subgraph, "run")

# get a list of runner inputs
inputTensors = dpu.get_input_tensors()

# optional - print names and shapes of each input tensor
for inputTensor in inputTensors:
print('Input tensor :',inputTensor.name, inputTensors.dims)


# create input buffer
# Important: Order of values passed to DPU thru' input data buffer must
match the order of tensor objects returned by get_input_tensor()
inputData = []
for inputTensor in inputTensors:
inputData.append(some_input_data.reshape(inputTensor.dims))


# pass input buffer to DPU runner, launch and wait for completion
job_id = dpu_runner.execute_async(inputData,outputData)
dpu_runner.wait(job_id)
```

get_inputs

Gets all input TensorBuffers of RunnerExt.

Prototype

```
List[vart.TensorBuffer] get_inputs()
```

Parameters

None.

Returns

All input TensorBuffers. A vector of raw pointer to the input TensorBuffer.

Usage

```
input_tensor_buffers = runner.get_inputs()
input_dim = tuple(input_tensor_buffers[0].get_tensor().dims)
```

get_outputs

Gets all output TensorBuffers of RunnerExt.

Prototype

```
List[vart.TensorBuffer] get_outputs()
```

Parameters

None.

Returns

All output TensorBuffers. A vector of raw pointer to the output TensorBuffer.

Usage

```
output_tensor_buffers = runner.get_outputs()
output_element_num =
tuple(output_tensor_buffers[0].get_tensor().get_element_num())
```

vart.RunnerExt Example

The vart.RunnerExt Class example is show below.

```
// create runner
runner = vart.RunnerExt.create_runner(subgraph, “run”)
// get input and output tensor buffers
input_tensor_buffers = runner.get_inputs()
output_tensor_buffers = runner.get_outputs()
// run graph runner
v = runner.execute_async(input_tensor_buffers, output_tensor_buffers)
runner.wait(v)
output_data = np.asarray(output_tensor_buffers[0])
```

create_graph_runner

Factory function to create an instance of runner by graph.

Prototype

```
vart.RunnerExt create_graph_runner(xir.Graph graph)
```

Parameters

The following table lists the `create_graph_runner` function arguments.

Table 64: **create_graph_runner Arguments**

Type	Name	Description
xir.Graph	graph	XIR Graph

Returns

An instance of runner.

Usage

```
graph = xir.Graph.deserialize(xmodel_file)
runner = vitis_ai_library.GraphRunner.create_graph_runner(graph)
input_tensor_buffers = runner.get_inputs()
```

Graph runner Example

The Graph runner class example is shown below.

```
// create graph runner
graph = xir.Graph.deserialize(xmodel_file)
runner = vitis_ai_library.GraphRunner.create_graph_runner(graph)
// get input and output tensor buffers
input_tensor_buffers = runner.get_inputs()
output_tensor_buffers = runner.get_outputs()
// run graph runner
v = runner.execute_async(input_tensor_buffers, output_tensor_buffers)
runner.wait(v)
output_data = np.asarray(output_tensor_buffers[0])
```

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. Release Notes and Known Issues - https://xilinx.github.io/Vitis-AI/docs/reference/release_notes_3.0.html
2. *Vitis AI Optimizer User Guide* (UG1333)
3. *Vitis AI Library User Guide* (UG1354)
4. *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide* (PG338)
5. *DPUCAHX8H for Convolutional Neural Networks Product Guide* (PG367)
6. *DPUCVDX8G for Versal ACAPs Product Guide* (PG389)
7. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
8. *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393)
9. *PetaLinux Tools Documentation: Reference Guide* (UG1144)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/24/2023 Version 3.0	
General Updates	Updated the links.
01/12/2023 Version 3.0	
General Updates	• Made technical updates for the new release • Editorial updates
06/15/2022 Version 2.5	
General Updates	• Made technical updates for the new release • Editorial updates
01/20/2022 Version 2.0	
General Updates	• Updated minor technical details • Updated the supported card versions • Editorial updates
Deep-Learning Processor Unit	Updated to include the Versal DPUs
vitis_quantize.VitisQuantizer.get_qat_model	Updated the argument descriptions
Supported Operators and DPU Limitations	Updated the supported operators table
vaitrace Usage	Updated command line usage text
12/13/2021 Version 1.4.1	
Chapter 3: Quantizing the Model	Updated vai_q_tensorflow2 Supported Operations and APIs section
07/22/2021 Version 1.4	
Chapter 1: Vitis AI Overview	Added Versal AI Core Series: DPUCVDX8G section

Section	Revision Summary
TensorFlow 2.x Version (vai_q_tensorflow2)	Added vai_q_tensorflow2 Quantization Aware Training, Quantizing with Custom Layers, and vai_q_tensorflow2 Usage sections
PyTorch Version (vai_q_pytorch)	Updated vai_q_pytorch QAT
Chapter 5: Deploying and Running the Model	Updated Apache TVM, Microsoft ONNX Runtime, and TensorFlow Lite
Chapter 6: Profiling the Model	Added Text Summary Updated vaitrace Usage
02/03/2021 Version 1.3	
Entire document	Updated links
12/17/2020 Version 1.3	
Entire document	Minor changes
Deep-Learning Processor Unit	Added new topics: Alveo U200/U250 Card: DPUCADF8H and Versal AI Core Series: DPUCVDX8G.
TensorFlow 2.x Version (vai_q_tensorflow2)	Added new section
PyTorch Version (vai_q_pytorch)	Added new topics: Module Partial Quantization, vai_q_pytorch Fast Finetuning, and vai_q_pytorch QAT.
Chapter 4: Compiling the Model	Added new section: Compiling with an XIR-based Toolchain.
Chapter 8: Integrating the DPU into Custom Platforms	Added new chapter.
Appendix B: VART Programming APIs	Added new section: VART APIs.
07/21/2020 Version 1.2	
Entire document	Minor changes
07/07/2020 Version 1.2	
Entire document	• Added Vitis AI Profiler topic. • Added Vitis AI unified API introduction.
DPU Naming	Added new topic
Chapter 2: Getting Started	Updated the chapter
03/23/2020 Version 1.1	
DPUCAHX8H	Added new topic
Entire document	Added contents for Alveo U50 support, U50 DPUV3 enablement, including compiler usage and model deployment description.

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright