Vitis AI User Guide

UG1414 (v1.3) February 3, 2021

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
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: DPUCADF8H, Alveo U50/U50LV/U280: DPUCAHX8L, 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 Quantize Finetuning.
Chapter 5: Compiling the Model	Added new section: Compiling with an XIR-based Toolchain.
Chapter 10: Integrating the DPU into Custom Platforms	Added new chapter.
Appendix A: Vitis AI Programming Interface	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.

Table of Contents

Revision History......2

Chapter 1: Vitis AI Overview......5

Navigating Content by Design Process......6

Features......7

Vitis AI Tools Overview......7

Vitis AI Containers......22

Minimum System Requirements......23

Development Flow Overview......23

Chapter 2: Getting Started......25

Installation and Setup......25

Running Examples......36

Support......40

Chapter 3: Understanding the Vitis AI Model Zoo Networks......41

Chapter 4: Quantizing the Model......43

Overview......43

Vitis AI Quantizer Flow......45

TensorFlow 1.x Version (vai_q_tensorflow)......46

TensorFlow 2.x Version (vai_q_tensorflow2)......58

PyTorch Version (vai_q_pytorch)......65

Caffe Version (vai_q_caffe)......76

Chapter 5: Compiling the Model......82

Vitis AI Compiler......82

Compiling with an XIR-based Toolchain......83

Compiling with DPUCADX8G......103

VAI_C Usage......105

Chapter 6: Deploying and Running the Model......107

Deploying and Running Models on Alveo U200/250......107

Programming with VART.......107
DPU Debug with VART.......109
Multi-FPGA Programming.......115
Apache TVM and Microsoft ONNX Runtime.......118

Chapter 7: Profiling the Model.......120
Vitis AI Profiler.......120

Chapter 8: Optimizing the Model.......127

Chapter 9: Accelerating Subgraph with ML Frameworks.......128
Partitioning Functional API Call in TensorFlow.......128
Partitioner API.......128
Partitioning Support in Caffe.......130

Chapter 10: Integrating the DPU into Custom Platforms.......132

Appendix A: Vitis AI Programming Interface.......133
VART APIs.......133

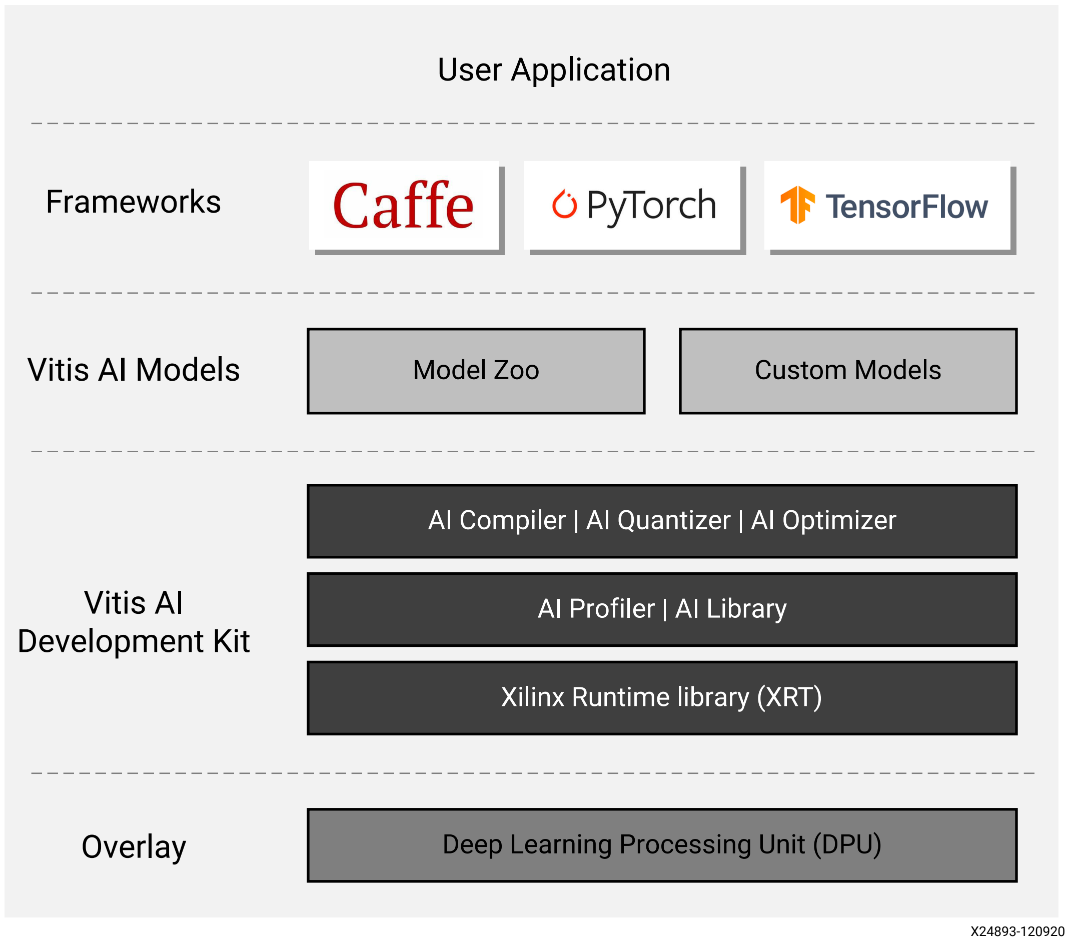
Appendix B: Legacy DNNDK.......147
DNNDK N2Cube Runtime.......147
DNNDK Examples.......149
DNNDK Programming for Edge.......153
DNNDK Utilities.......159
Profiling Using the DNNDK Profiler.......166
DNNDK Programming APIs.......170

Appendix C: Additional Resources and Legal Notices.......229
Xilinx Resources.......229
Documentation Navigator and Design Hubs.......229
References.......229
Please Read: Important Legal Notices.......230

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 FPGAs and on adaptive compute acceleration platforms (ACAPs). It makes it easier for users without FPGA knowledge to develop deep-learning inference applications, by abstracting the intricacies of the underlying FPGA and ACAP.

Figure 1: **Vitis AI Stack**



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. This document covers the following design processes:

- **Machine Learning and Data Science:** Importing a machine learning model from a Caffe, 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 4: Quantizing the Model
  - Chapter 5: Compiling the Model
- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. Topics in this document that apply to this design process include:
  - Chapter 3: Understanding the Vitis AI Model Zoo Networks
- **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 10: Integrating the DPU into Custom Platforms
- **Host Software Development:** Developing the application code, accelerator development, including library, XRT, and Graph API use. Topics in this document that apply to this design process include:
  - Chapter 6: Deploying and Running the Model
  - Chapter 9: Accelerating Subgraph with ML Frameworks
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem 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 10: 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 7: 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%.
- The AI profiler provides layer by layer analysis to help with bottlenecks.
- The AI library 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

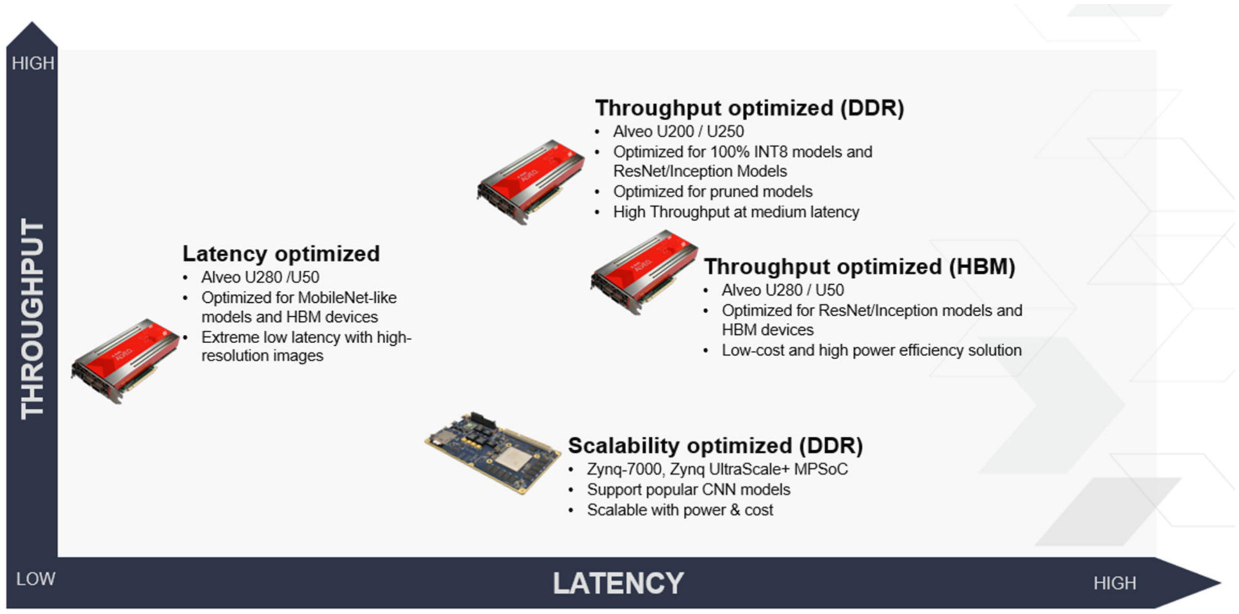
The Deep-Learning Processor Unit (DPU) is a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required. It is designed to accelerate the computing workloads of deep learning inference algorithms widely 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®-7000 devices, Zynq UltraScale+ MPSoCs, 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. In the DPU-TRD, 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®-7000, Zynq® UltraScale+™ MPSoC, and Alveo cards such as U50, U200, U250, and U280, enabling unique differentiation and flexibility in terms of throughput, latency, scalability, and power.

Figure 2: **DPU Options**

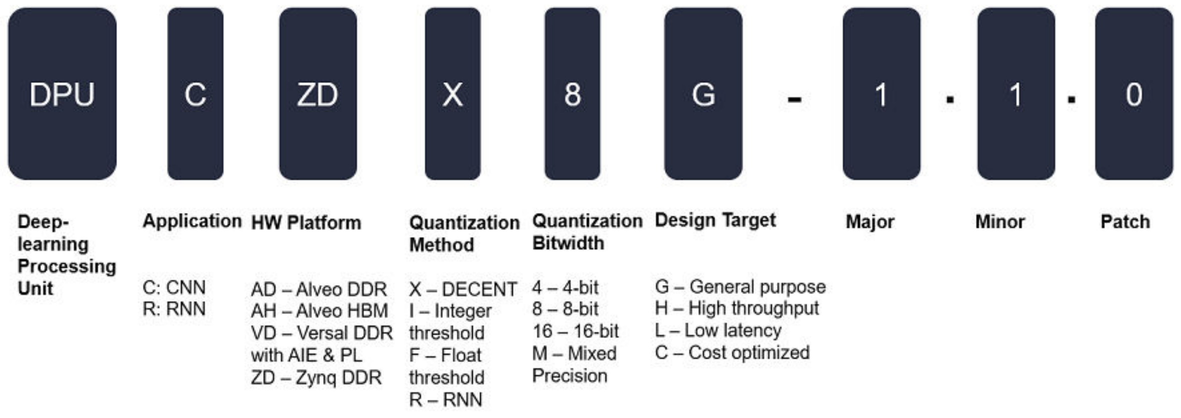

HIGH
THROUGHPUT
LOW
LATENCY
HIGH
Throughput optimized (DDR)
• Alveo U200 / U250
• Optimized for 100% INT8 models and ResNet/Inception Models
• Optimized for pruned models
• High Throughput at medium latency
Latency optimized
• Alveo U280 /U50
• Optimized for MobileNet-like models and HBM devices
• Extreme low latency with high-resolution images
Throughput optimized (HBM)
• Alveo U280 / U50
• Optimized for ResNet/Inception models and HBM devices
• Low-cost and high power efficiency solution
Scalability optimized (DDR)
• Zynq-7000, Zynq UltraScale+ MPSoC
• Support popular CNN models
• Scalable with power & cost

DPU Naming

Vitis AI 1.2 and later releases use a new DPU naming scheme to differentiate various DPUs designed for different purposes. The old DPUv1/v2/v3 naming is deprecated.

The new DPU naming convention is shown in the following figure:

Figure 3: **DPU Naming Convention**



DPU Naming Example

To understand the mapping between the old DPU naming scheme and the current naming scheme, see the following table:

Table 1: **DPU Naming Examples**

Example	DPU	Application	Hardware Platform	Quantization Method	Quantization Bitwidth	Design Target	Major	Minor	Patch	DPU Name
DPUv1	DPU	C	AD	X	8	G	3	0	0	DPUCADX8G-3.0.0
DPUv2	DPU	C	ZD	X	8	G	1	4	1	DPUCZDX8G-1.4.1
DPUv3e	DPU	C	AH	X	8	H	1	0	0	DPUCAHX8H-1.0.0
DPUv3me	DPU	C	AH	X	8	L	1	0	0	DPUCAHX8L-1.0.0
DPUv3int8	DPU	C	AD	F	8	H	1	0	0	DPUCADF8H-1.0.0
XRNN	DPU	R	AH	R	16	L	1	0	0	DPURAHR16L-1.0.0

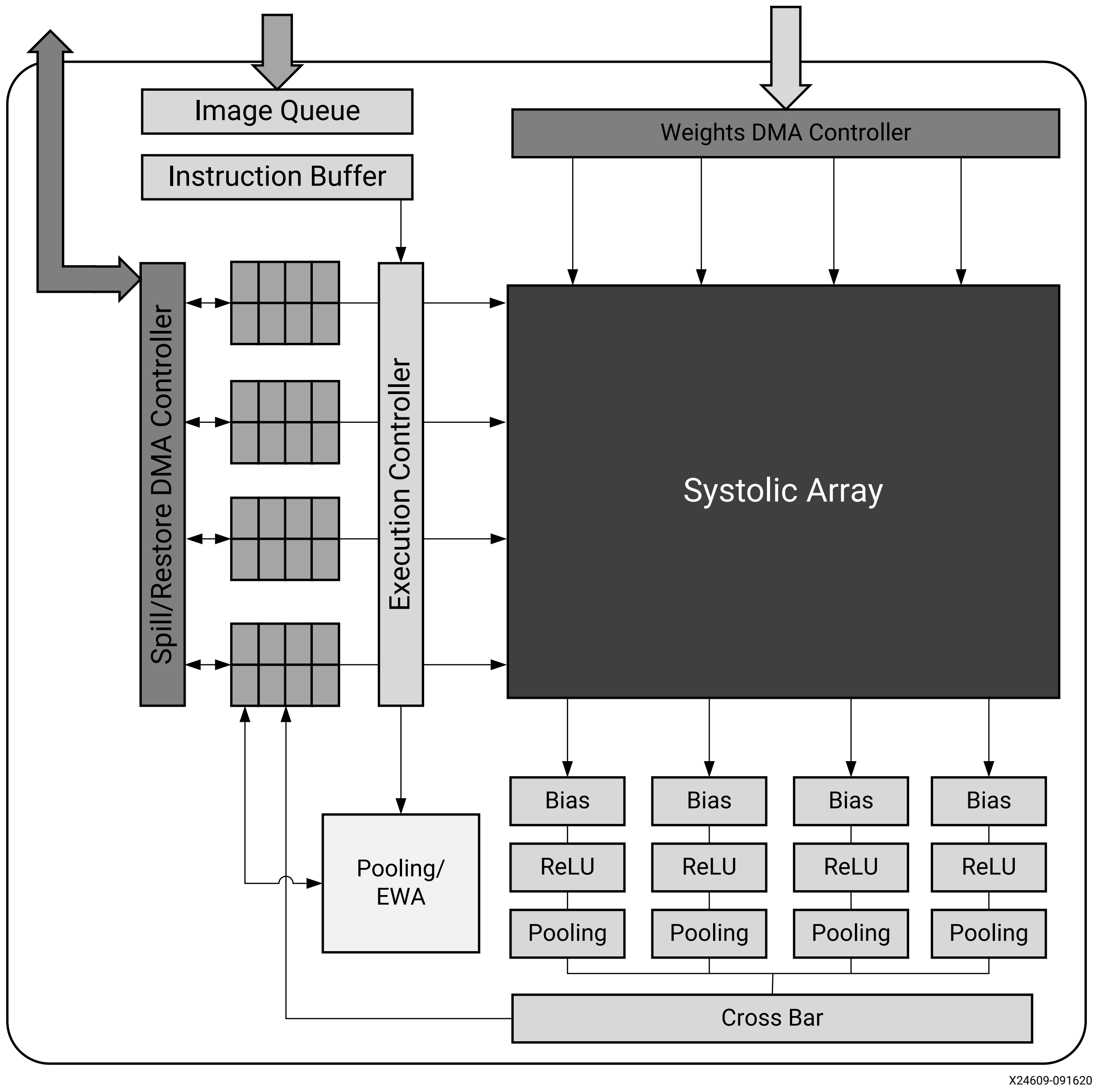
Notes:

1. For Application: C-CNN, R-RNN
2. For Hardware Platform: AD-Alveo DDR; AH-Alveo HBM; VD-Versal DDR with AI Engine and PL; ZD-Zynq DDR
3. For Quantization method: X-Decent; F- Float threshold; I-Integer threshold; R-RNN
4. For Quantization bandwidth: 4-4 bit; 8-8 bit; 16-16 bit; M- Mixed precision
5. For Design target: G-General purpose; H-High throughput; L-Low latency; C-Cost optimized

Alveo U200/U250: DPUCADX8G

DPUCADX8G (previously known as xDNN) IP cores are high performance general CNN processing engines (PE).

Figure 4: **DPUCADX8G Architecture**



The key features of this engine are:

- 96x16 DSP Systolic Array operating at 700 MHz
- Instruction-based programming model for simplicity and flexibility to represent a variety of custom neural network graphs.
- 9 MB on-chip Tensor Memory composed of UltraRAM
- Distributed on-chip filter cache
- Utilizes external DDR memory for storing Filters and Tensor data

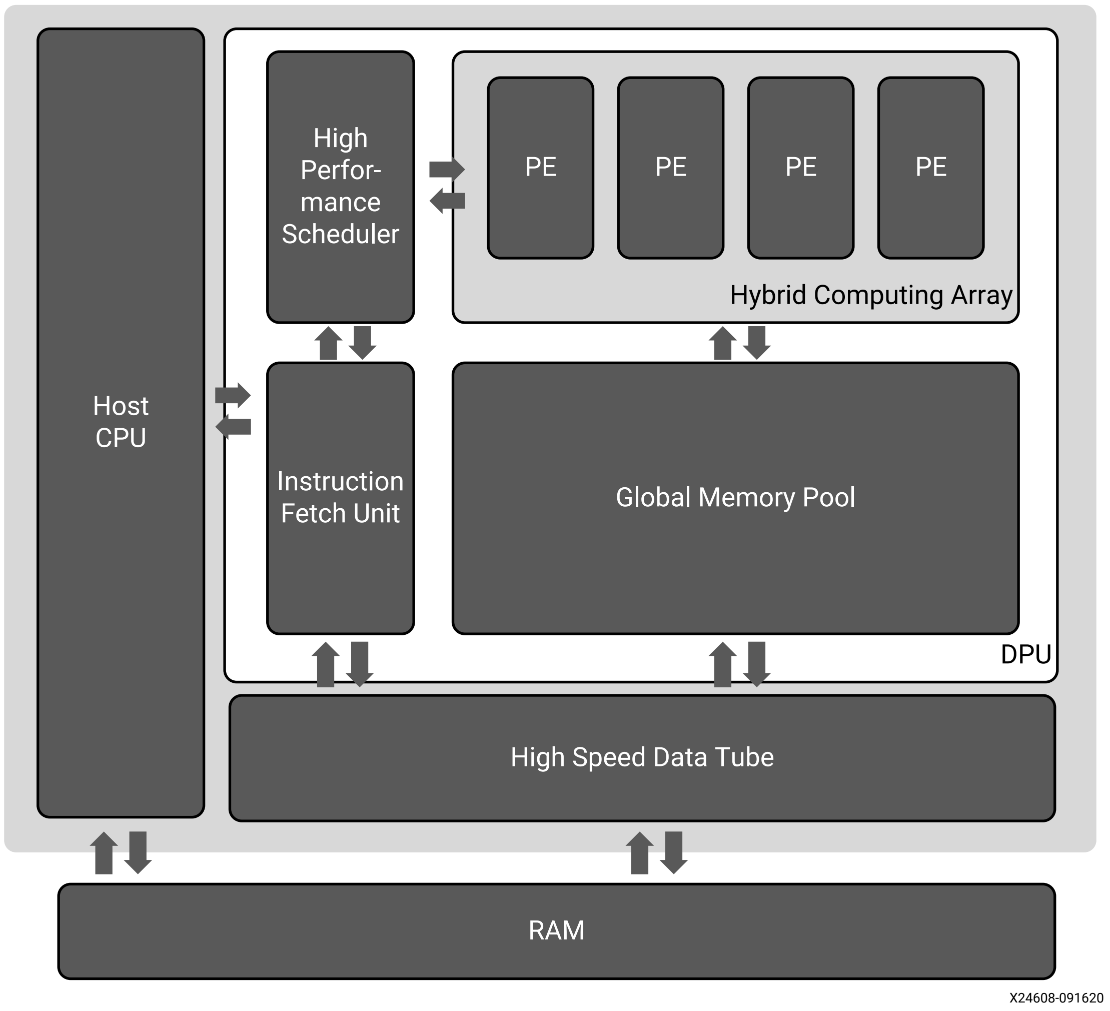
- Pipelined Scale, ReLU, and Pooling Blocks for maximum efficiency
- Standalone Pooling/Eltwise execution block for parallel processing with Convolution layers
- Hardware-Assisted Tiling Engine to sub-divide tensors to fit in on-chip Tensor Memory and pipelined instruction scheduling
- Standard AXI-MM and AXI4-Lite top-level interfaces for simplified system-level integration
- Optional pipelined RGB tensor Convolution engine for efficiency boost

***Note*:** For increased throughput in Cloud applications, a new DPU, DPUCADF8H, for Alveo U200/U250 is supported in Vitis AI 1.3 and later releases.

Zynq MPSoC: DPUCZDX8G

The DPUCZDX8G IP has been optimized for Xilinx MPSoC devices. This IP can be integrated as a block in the programmable logic (PL) of the selected Zynq-7000 SoC and Zynq UltraScale+ MPSoCs with direct connections to the processing system (PS). The configurable version DPU IP is released together with Vitis AI. 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 the https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD.

Figure 5: **DPUCZDX8G Architecture**



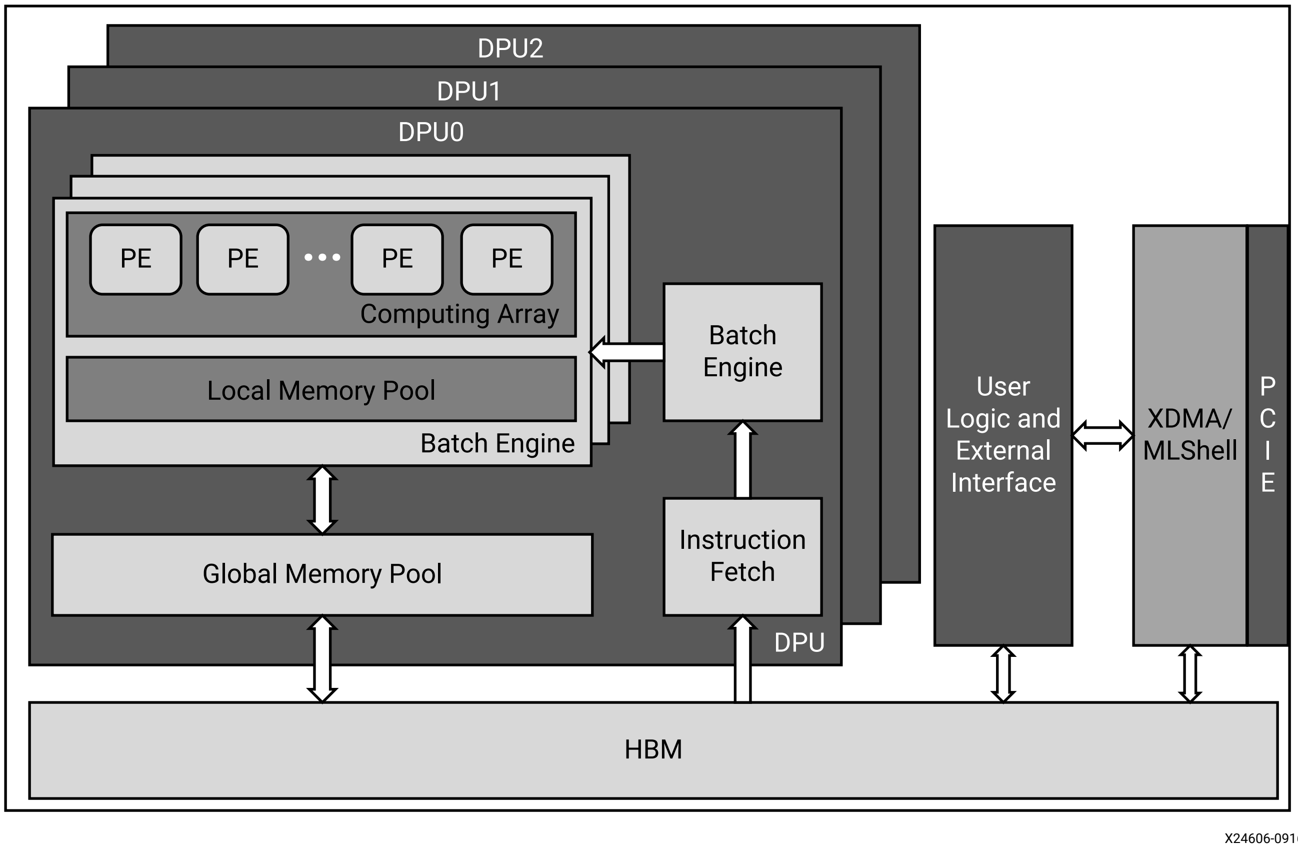
Alveo U50/U280: DPUCAHX8H

The Xilinx DPUCAHX8H DPU is a programmable engine optimized for convolutional neural networks, mainly for high throughput applications. This unit includes a high performance scheduler module, a hybrid computing array module, an instruction fetch unit module, and a global memory pool module. The DPU uses a specialized instruction set, which allows efficient implementation of many convolutional neural networks. Some examples of convolutional neural networks that are deployed include VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN, and many others.

The DPU IP can be implemented in the PL of the selected Alveo board. The DPU requires instructions to implement a neural network and accessible memory locations for input images as well as temporary and output data. A user-defined unit running on PL is also required to do necessary configuration, inject instructions, service interrupts and coordinate data transfers.

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

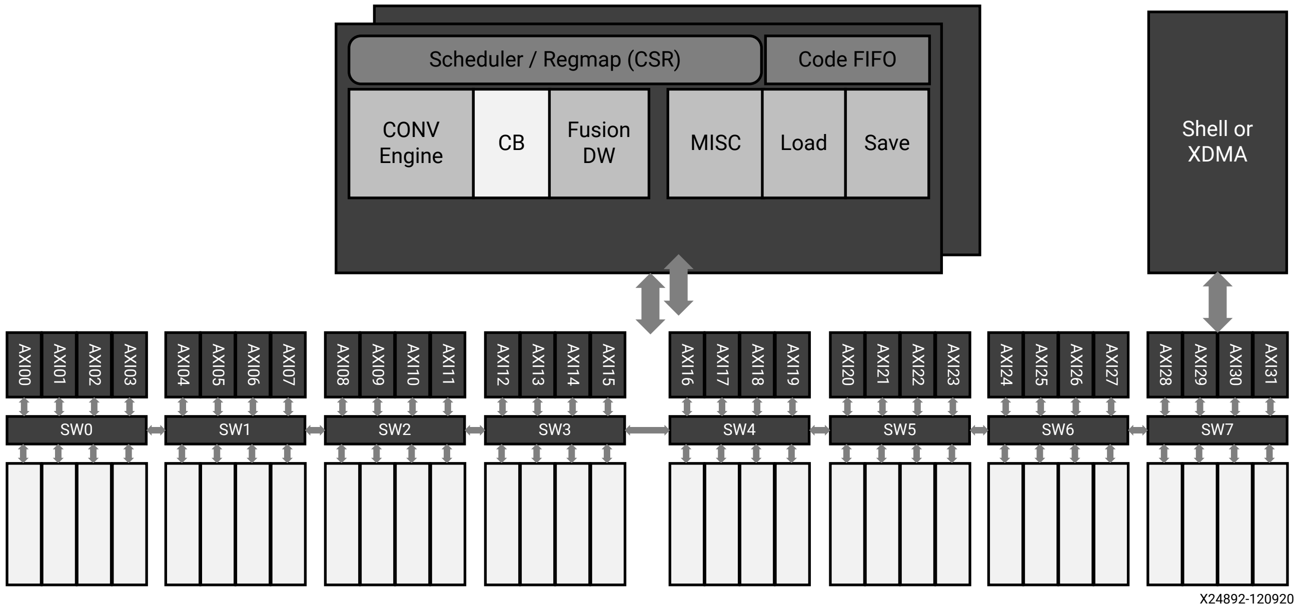
Figure 6: **DPUCAHX8H Top-Level Block Diagram**



Alveo U50/U50LV/U280: DPUCAHX8L

The DPUCAHX8L IP is a new general purpose CNN accelerator which is optimized for HBM cards, such as U50/U50LV and U280, and designed for low latency applications. It has a new low latency DPU micro-architecture with an HBM memory sub-system supporting 4TOPs to 5.3TOPs MAC array. It supports the back-to-back convolution and depthwise convolution engines to increase computing parallelism. It also supports hierarchical memory system, URAM and HBM, to maximize data movement. With this low latency DPU IP, the Xcompiler supports the super layer interface and many new compiling strategies for kernel fusion and graph partition.

Figure 7: **DPUCAHX8L Architecture**



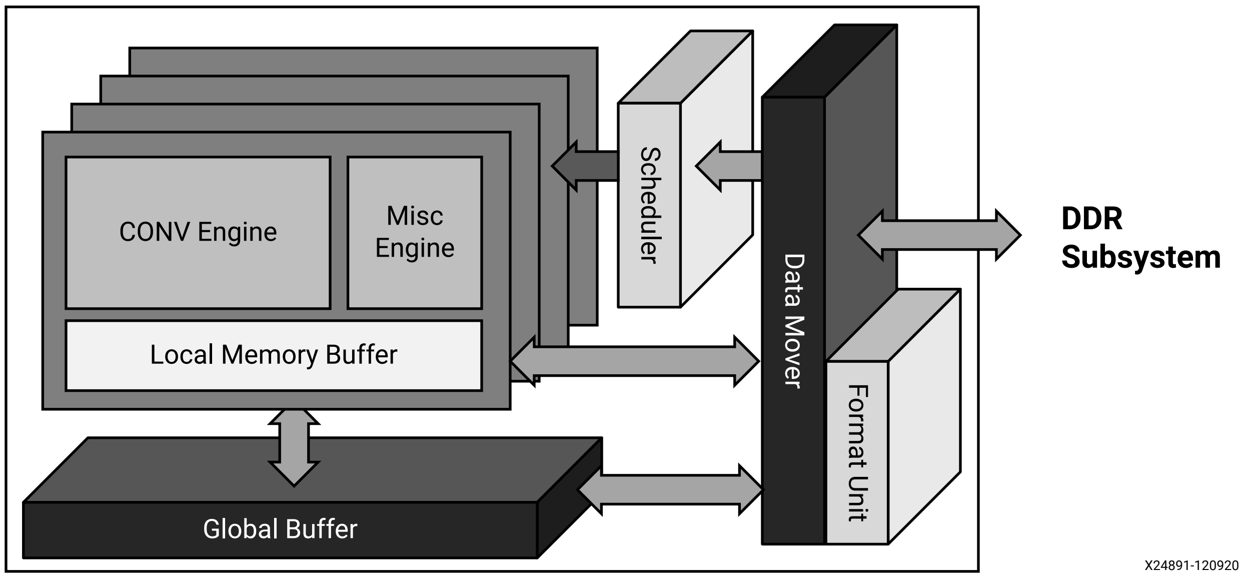
Alveo U200/U250: DPUCADF8H

The DPUCADF8H is the DPU optimized for Alveo U200/U250 and targeted for high-throughput applications. The key features of the DPUCADF8H are as follows:

- Throughput oriented and high-efficiency computing engines: throughput is improved by 1.5X~2.0X on different workloads
- Wide range of convolution neural network support
- Friendly to compressed convolution neural networks
- Optimized for high-resolution images

The top-level block diagram is shown in the following figure:

Figure 8: **DPUCADF8H Architecture**

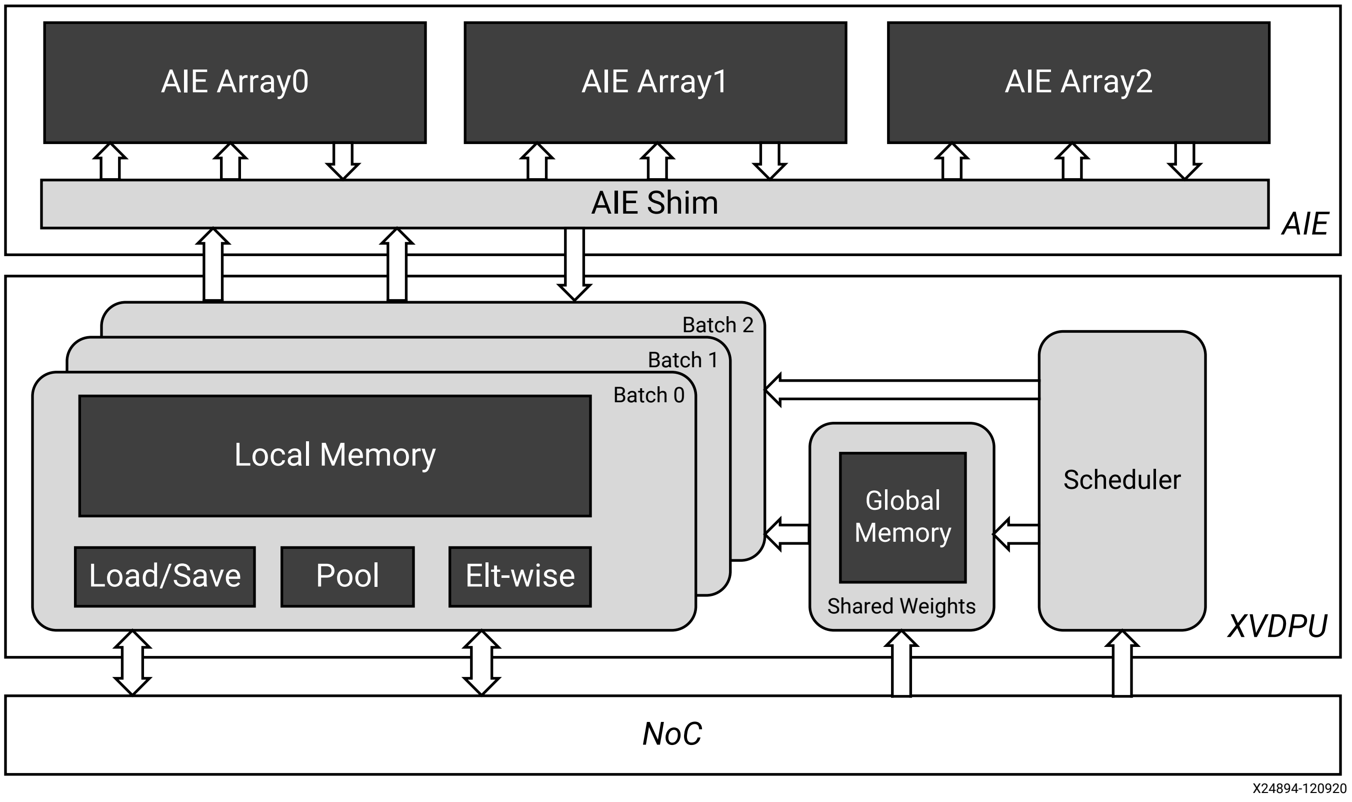


Versal AI Core Series: DPUCVDX8G

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 comprises of AI Engines and PL. This IP is user-configurable and exposes several parameters which can be specified to optimize AI Engines and PL resources or customize enable features.

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

Figure 9: **DPUCVDX8G Architecture**

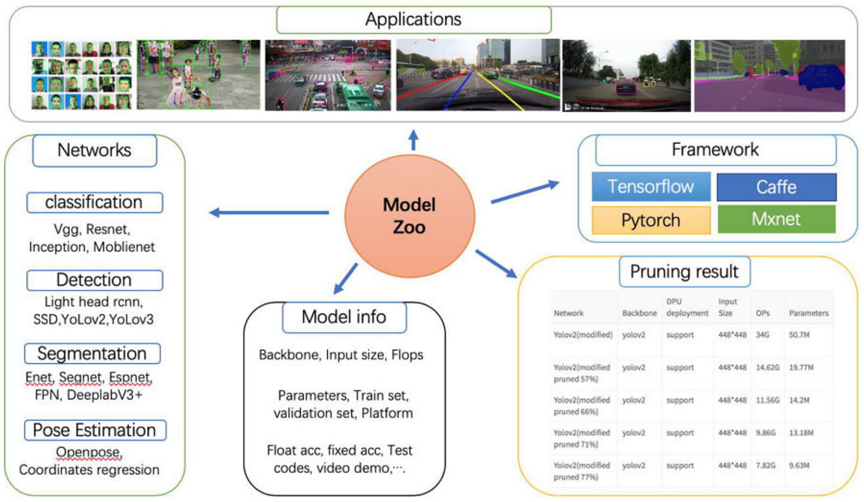


AI Model Zoo

The 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 10: **AI Model Zoo**



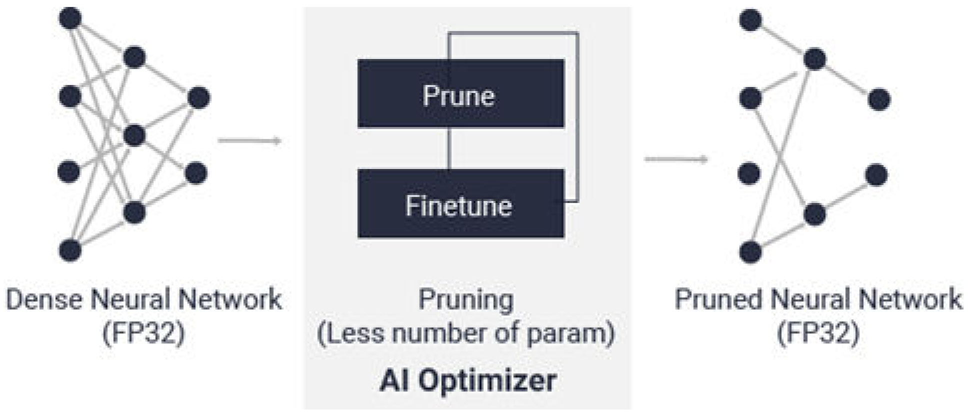
Network	Backbone	DPU deployment	Input Size	OPs	Parameters
Yolov2(modified)	yolov2	support	448*448	34G	50.7M
Yolov2(modified pruned 57%)	yolov2	support	448*448	14.62G	19.77M
Yolov2(modified pruned 66%)	yolov2	support	448*448	11.56G	14.2M
Yolov2(modified pruned 71%)	yolov2	support	448*448	9.86G	13.18M
Yolov2(modified pruned 77%)	yolov2	support	448*448	7.82G	9.63M

AI Optimizer

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

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

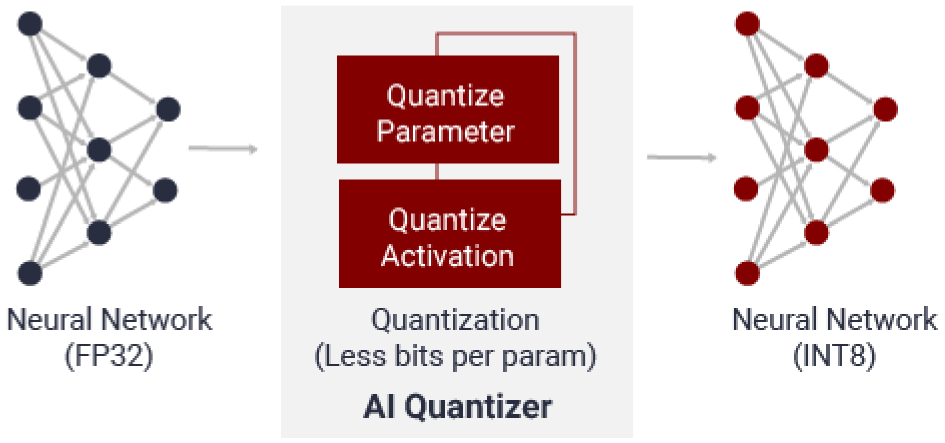
Figure 11: **AI Optimizer**



AI Quantizer

By converting the 32-bit floating-point weights and activations to fixed-point like INT8, the 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.

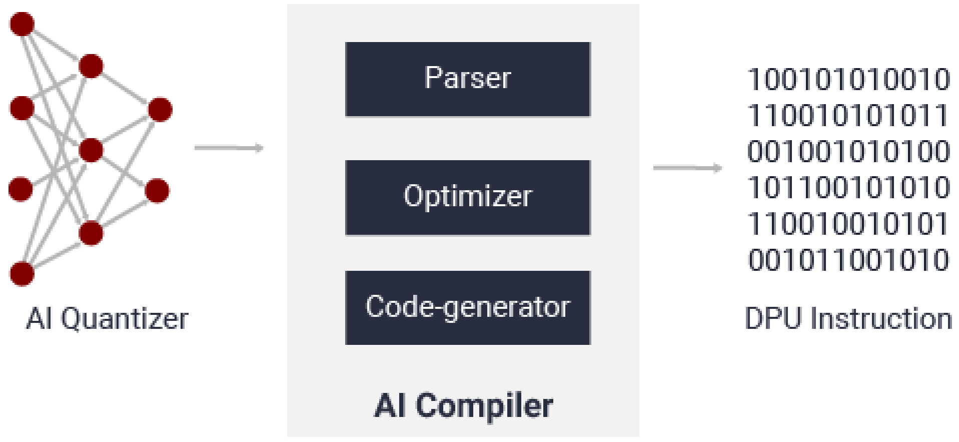
Figure 12: **AI Quantizer**



AI Compiler

The 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.

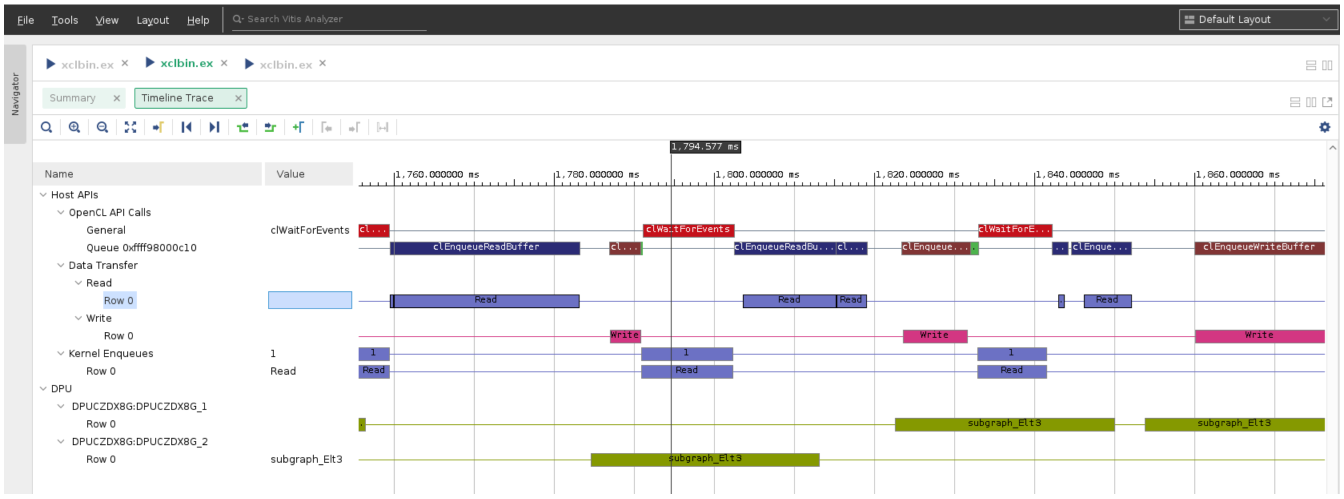
Figure 13: **AI Complier**



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 14: **AI Profiler**

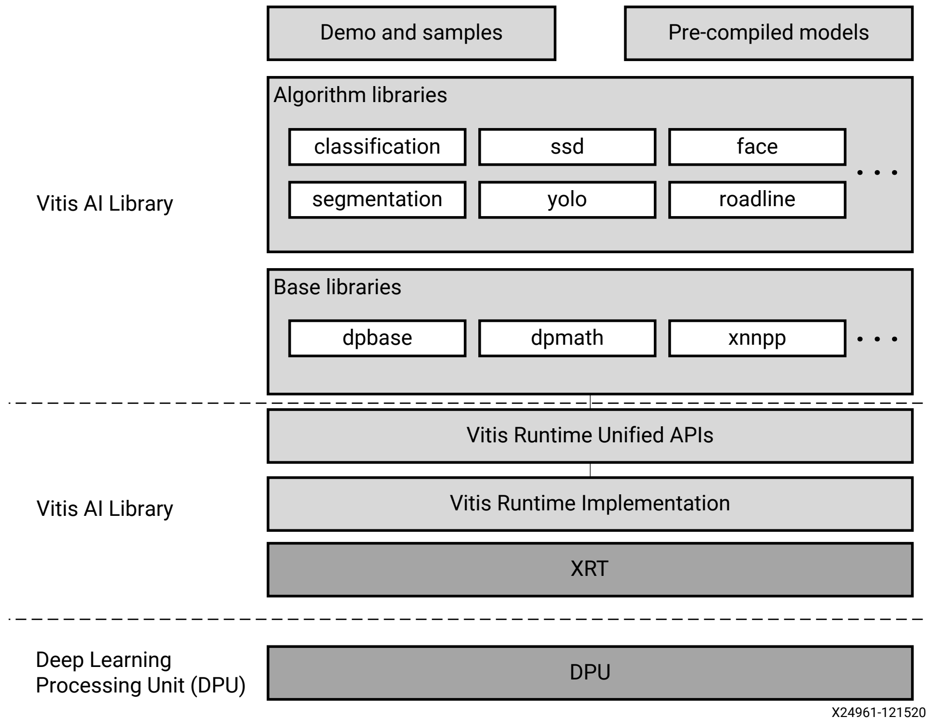


AI Library

The Vitis AI Library is a set of high-level libraries and APIs built for efficient AI inference with DPUs. It fully supports 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 15: **AI Library**



AI Runtime

The 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

For Cloud

The cloud accelerator has multiple independent Compute Units (CU) that can be programmed to each work on a different AI model, or to work on the same AI model for maximum throughput.

The cloud runtime introduces a new AI resource manager, to simplify scaling applications over multiple FPGA resources. The application no longer needs to designate a specific FPGA card to be used. Applications can make requests for a single CU or a single FPGA, and the AI resource manager returns a free resource compatible with your request. The AI resource manager works with Docker containers, as well as with multiple users and multiple tenants on the same host.

Vitis AI Runtime

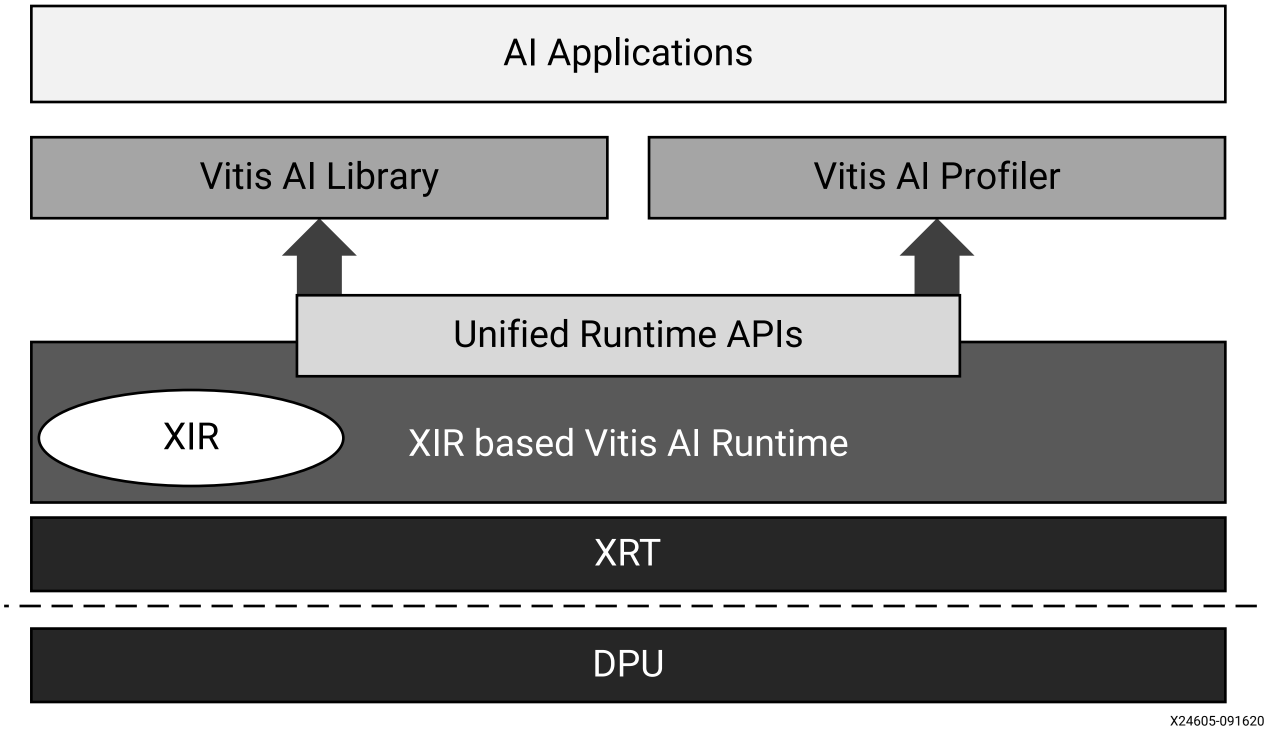
The Vitis AI Runtime (VART) is the next generation runtime suitable for devices based on DPUCZDX8G, DPUCADX8G, DPUCADF8H, and DPUCAHX8H. DPUCZDX8G and DPUCADF8H are used for Edge devices, such as ZCU102 and ZCU104. DPUCADX8G is used for cloud devices, such as Alveo U200 and U250. DPUCAHX8H is used for cloud devices, such as Alveo U50, U50LV, and U280. DPUCVDX8G is used for Versal evaluation boards, such as VCK190. The framework of VART is shown in the following figure. For the Vitis AI release, VART is based on the XRT.

Currently, Vitis AI ships with two kinds of runtime:

- **VART:** Based on Xilinx intermediate representation (XIR). This is a graph-based intermediate representation, which is the official data exchange standard for Vitis AI.
- **n2cube:** Included in the legacy Deep Neural Network Development Kit (DNNDK) and is available for compatibility.

***Note*:** DNNDK is deprecated in the Vitis AI 1.4 and future releases.

Figure 16: **VART Stack**



Vitis AI Containers

Vitis AI 1.3 release uses container technology to distribute the AI software. The release consists of the following components.

- Tools container
- Runtime package for Zynq UltraScale+ MPSoC
- Public GitHub for examples (https://github.com/Xilinx/Vitis-AI)
- Vitis AI Model Zoo (https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-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 (Embedded)
    - Compiler flow for DPUCAHX8H (Cloud)
    - Compiler flow for DPUCAHX8L (Cloud)
    - Compiler flow for DPUCVDX8G (Edge)
    - Compiler flow for DPUCVDX8H (Cloud)
    - Compiler flow for DPUCADX8G (Cloud)
    - Compiler flow for DPUCADF8H (Cloud)
- Pre-built conda environment to run frameworks:
    - conda activate vitis-ai-caffe for Caffe-based flows
    - 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
- Alveo Runtime tools

Runtime Package for MPSoC Devices

- Container path URL: https://www.xilinx.com/bin/public/openDownload?filename=vitis_ai_2020.2-r1.3.0.tar.gz

- Contents
  - PetaLinux SDK and Cross compiler tool chain
  - Vitis AI board packages based on 2020.2 release, including Vitis AI new generation runtime VART.
- Models and overlaybins at https://github.com/Xilinx/Vitis-AI
  - All public pre-trained models
  - All Zynq UltraScale+ MPSoCs and Alveo accelerator cards overlays
  - Scripts are included to automate download and install models and overlays.

Minimum System Requirements

The following table lists system requirements for running containers as well as Alveo boards.

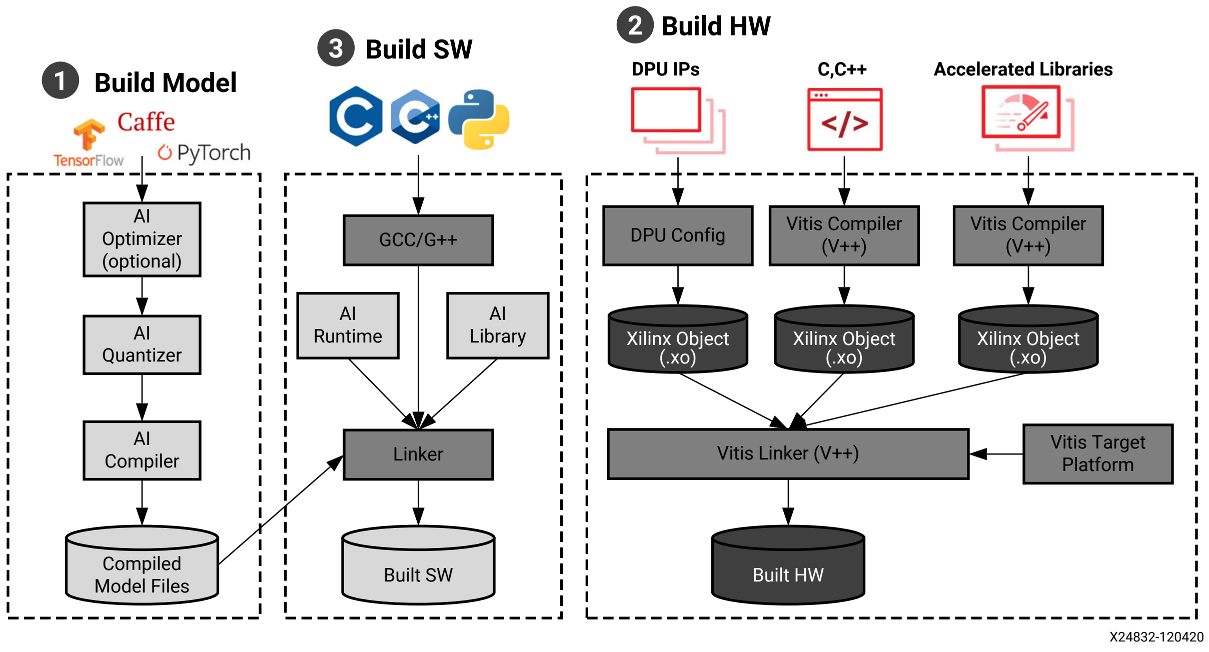
Table 2: **Minimum System Requirements**

Component	Requirement
FPGA	Xilinx Alveo U50, U50LV, U200, U250, U280, Xilinx ZCU102, ZCU104, VCK190
Motherboard	PCI Express® 3.0-compliant with one dual-width x16 slot.
System Power Supply	225W
Operating System	• Linux, 64-bit • Ubuntu 16.04, 18.04 • CentOS 7.4, 7.5 • RHEL 7.4, 7.5
GPU (Optional to accelerate quantization)	NVIDIA GPU supports CUDA 9.0 or higher, like NVIDIA P100, V100
CUDA Driver (Optional to accelerate quantization)	Driver compatible to CUDA version, NVIDIA-384 or higher for CUDA 9.0, NVIDIA-410 or higher for CUDA 10.0
Docker version	19.03 or higher

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 17: **Vitis AI Flow**



1. 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).
2. 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) and Alveo™ 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 10: Integrating the DPU into Custom Platforms.
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

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:latest
- Vitis AI runtime package for Edge

The tools container contains the Vitis AI quantizer, AI compiler, and AI runtime for cloud DPU. The Vitis AI runtime package for edge is for edge DPU development, which holds Vitis AI runtime installation package for Xilinx® ZCU102 and ZCU104 evaluation boards, and Arm® GCC cross-compilation toolchain.

Xilinx FPGA devices and evaluation boards supported by the Vitis AI development kit v1.3 release are:

- Cloud: Alveo™ cards U200, U250, U280, U50, U50LV, and Versal ACAP evaluation board VCK190.
- Edge: Zynq® UltraScale+™ MPSoC evaluation boards ZCU102 and ZCU104.

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

Use the following steps for installation:

1. Install Docker, if Docker 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 --recurse-submodules https://github.com/Xilinx/Vitis-AI

   cd Vitis-AI
   ```

4. Run Docker Container
   - Run the CPU image from Docker Hub

     ```
     docker pull xilinx/vitis-ai:latest
     ./docker_run.sh xilinx/vitis-ai
     ```

   - Build the CPU image locally and run it

     ```
     cd docker
     ./docker_build_cpu.sh

     # After build finished
     cd ..
     ./docker_run.sh xilinx/vitis-ai-cpu:latest
     ```

   - Build the GPU image locally and run it

     ```
     cd docker
     ./docker_build_gpu.sh

     # After build finished
     cd ..
     ./docker_run.sh xilinx/vitis-ai-gpu:latest
     ```

Setting Up the Host (Using VART)

For Edge (DPUCZDX8G)

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

1. Download `sdk-2020.2.0.0.sh` from here.
2. Install the cross-compilation system environment.

   ```
   ./sdk-2020.2.0.0.sh
   ```

3. Follow the prompts to install. The following figure shows the installation process.

   ✓ **RECOMMENDED:** *The `~/petalinux_sdk` path is recommended for installation. Regardless of the path you choose for the installation, ensure that your chosen path has write permissions. In this section, it is installed in `~/petalinux_sdk`.*

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

   ```
   source ~/petalinux_sdk/environment-setup-aarch64-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.

5. Download the `vitis_ai_2020.2-r1.3.0.tar.gz` from here and install it to the PetaLinux system.

```
tar -xzvf vitis_ai_2020.2-r1.3.0.tar.gz -C ~/petalinux_sdk/sysroots/
aarch64-xilinx-linux
```

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

```
cd Vitis-AI/demo/VART/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 Cloud (DPUCAHX8H)

Use the following steps to set up the host for cloud. These steps apply to U50, U50LV, and U280 cards.

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. Download the xclbin files from here. Untar it, choose the Alveo card and install it. Take `U50` as an example.

```
tar -xzvf alveo_xclbin-1.3.0.tar.gz
cd alveo_xclbin-1.3.0/U50/6E300M
sudo cp dpu.xclbin hbm_address_assignment.txt /usr/lib
```

For `DPUCAHX8L`, take `U50lv` as an example.

```
tar -xzvf alveo_xclbin-1.3.0.tar.gz
cd alveo_xclbin-1.3.0/U50lv-V3ME/1E250M
sudo cp dpu.xclbin /opt/xilinx/overlaybins/
export XLNX_VART_FIRMWARE=/opt/xilinx/overlaybins/dpu.xclbin
```

Note: If there is more than one card installed on the server and you want to specify some cards to run the program, 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.

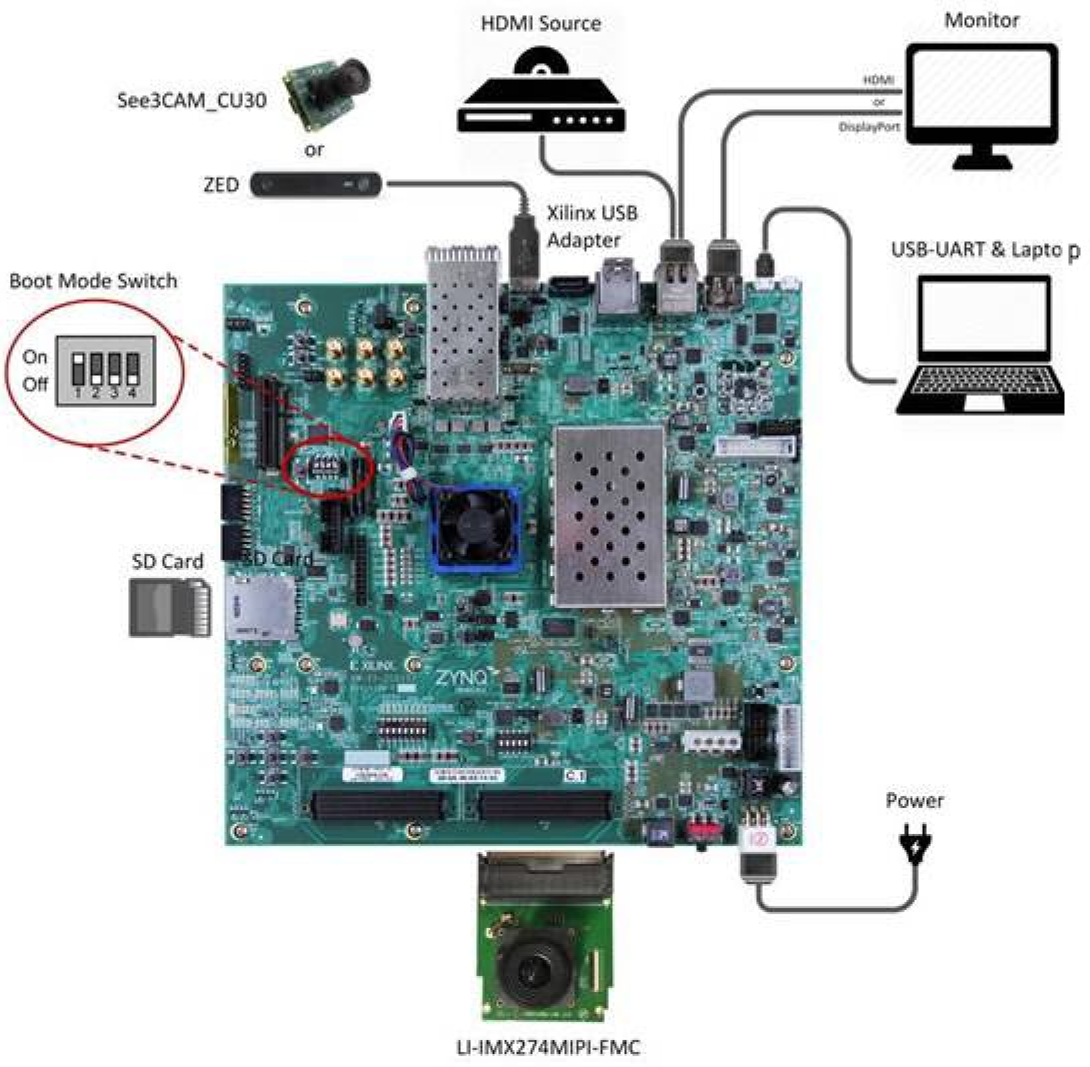
Setting Up the Evaluation Board

Setting Up the ZCU102/104 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 main connectivity interfaces for ZCU102 are shown in the following figure.

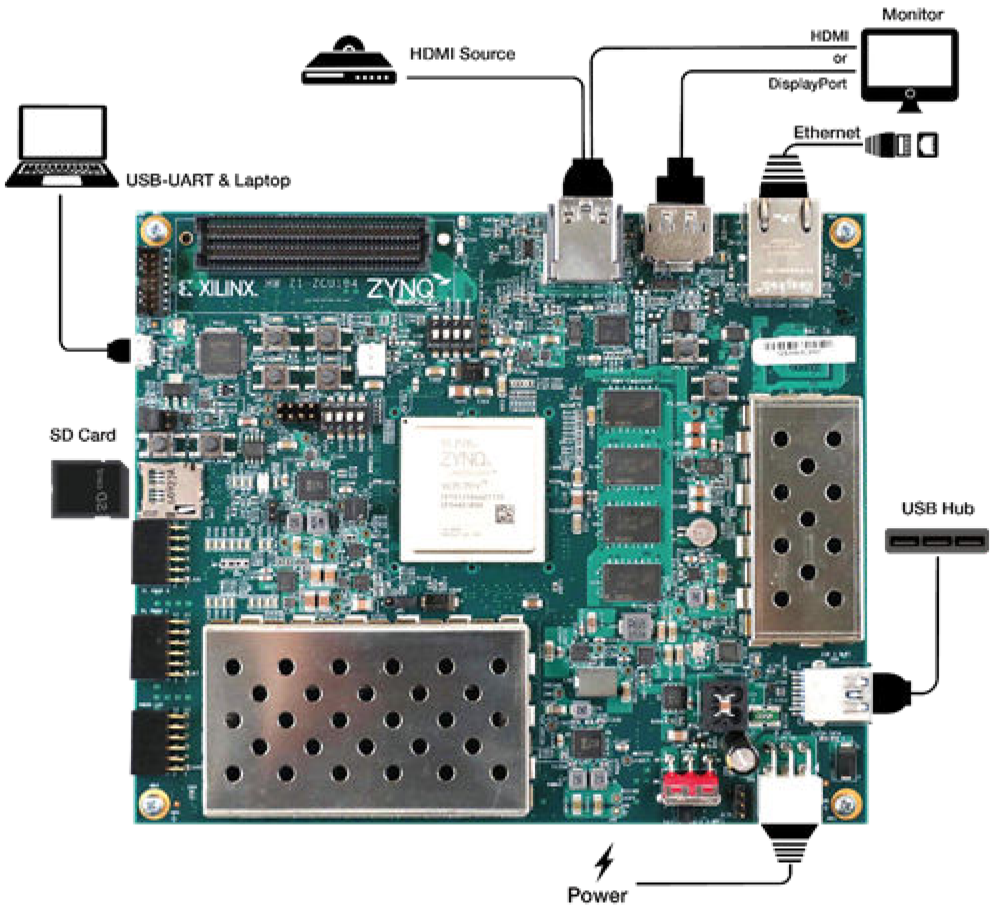
Figure 18: **Xilinx ZCU102 Evaluation Board and Peripheral Connections**



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 main connectivity interfaces for ZCU104 are shown in the following figure.

Figure 19: **Xilinx ZCU104 Evaluation Board and Peripheral Connections**

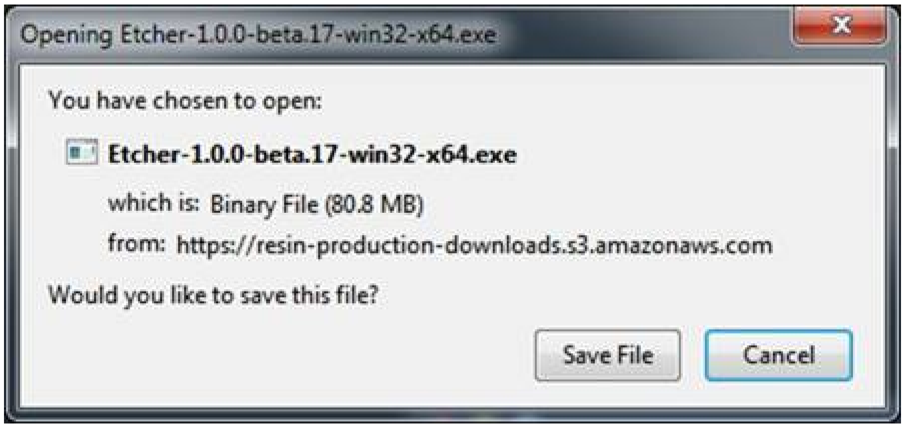


In the following sections, ZCU102 is used as an example to show the steps to setup the Vitis AI running environment on the evaluation boards.

Flashing the OS Image to the SD Card

For ZCU102, the system images can be downloaded from here; for ZCU104, it can be downloaded from here. One suggested software application for flashing the SD card is 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.



Opening Etcher-1.0.0-beta.17-win32-x64.exe

You have chosen to open:

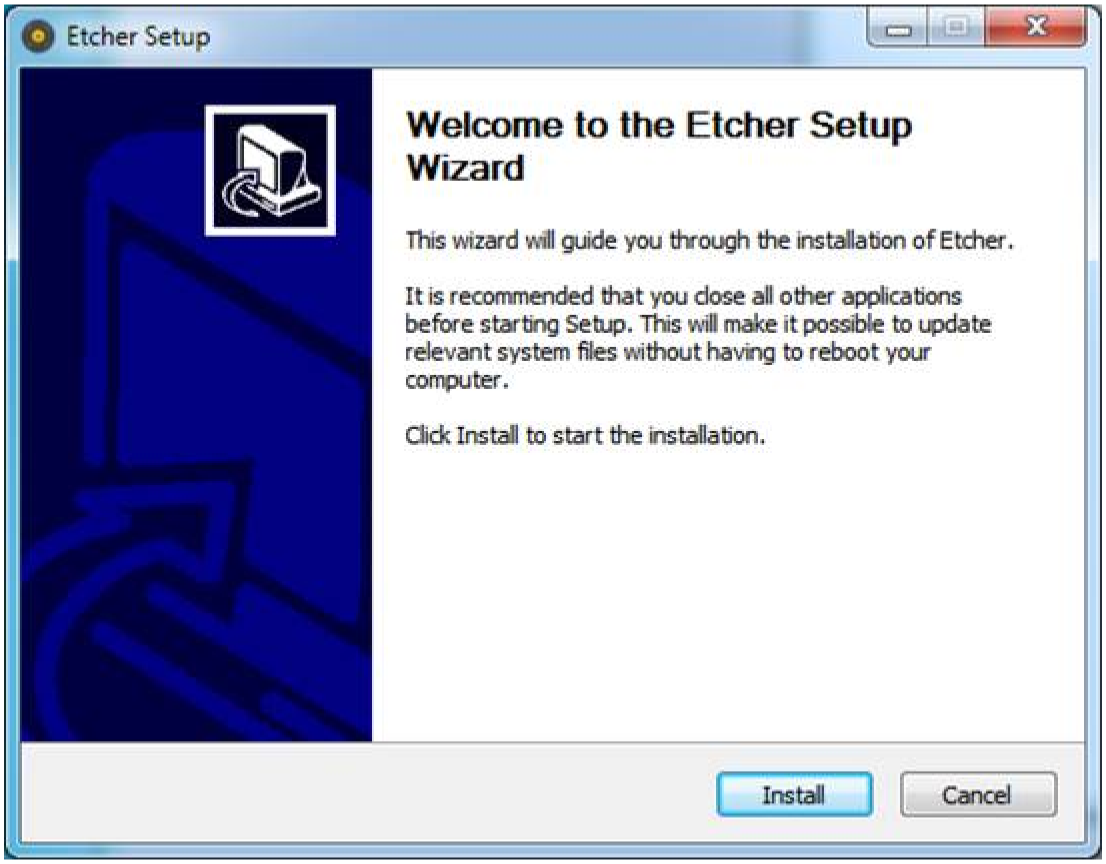
Etcher-1.0.0-beta.17-win32-x64.exe

which is: Binary File (80.8 MB)

from: https://resin-production-downloads.s3.amazonaws.com

Would you like to save this file?

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



Etcher Setup

Welcome to the Etcher Setup Wizard

This wizard will guide you through the installation of Etcher.

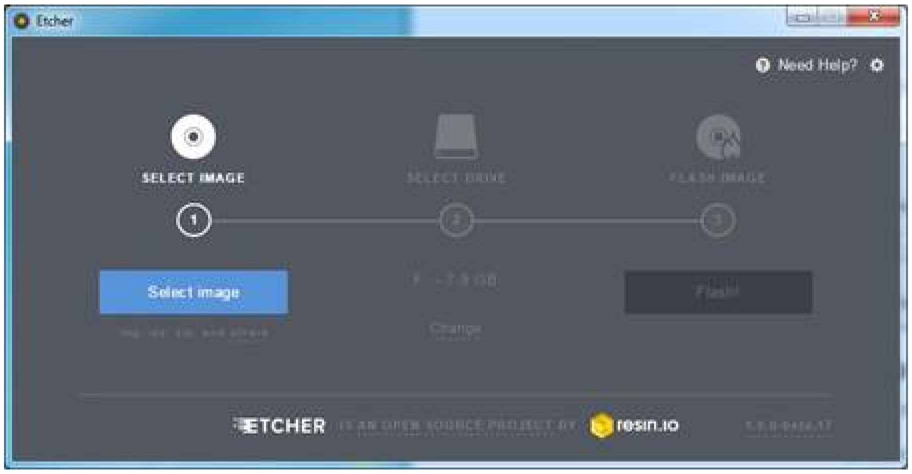
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Install to start the installation.

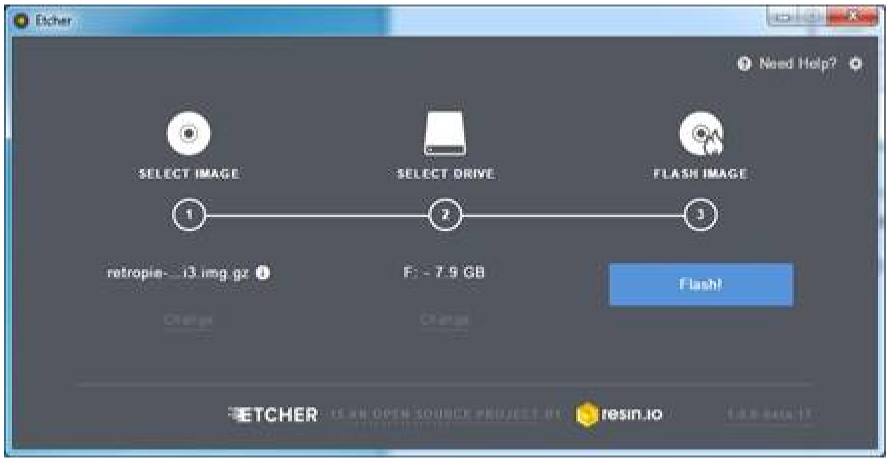
3. Eject any external storage devices such as USB flash drives and backup hard disks. This makes it easier to identify the SD card. Then, insert the SD card into the slot on your computer, or into the reader.

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 the steps below to boot the evaluation board.

1. Connect the power supply (12V ~ 5A).
2. Connect the UART debug interface to the host and other peripherals as required.
3. Turn on the power and wait for the system to boot.

4. Login to the system.
5. The system needs to perform some configurations for its first boot. Then reboot the board for these configurations to take effect.

Accessing the Evaluation Board

There are three ways to access the ZCU102 board:

- UART port
- Ethernet connection
- Standalone

UART Port

Apart from monitoring the boot process and checking Linux kernel messages for debugging, you can login 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 precompiled. 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.

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

1. Download the `vitis-ai-runtime-1.3.x.tar.gz` from here. Untar it and copy the following files to the board using scp.

   ```
   tar -xzvf vitis-ai-runtime-1.3.x.tar.gz
   scp -r vitis-ai-runtime-1.3.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-1.3.0-r<x>.aarch64.rpm | cpio -idmv
   ```

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

   ```
   cd ~/centos
   rpm -ivh --force libunilog-1.3.0-r<x>.aarch64.rpm
   rpm -ivh --force libxir-1.3.0-r<x>.aarch64.rpm
   rpm -ivh --force libtarget-factory-1.3.0-r<x>.aarch64.rpm
   rpm -ivh --force libvart-1.3.0-r<x>.aarch64.rpm
   ```

   If you want to run the example based on Vitis-AI-Library, execute the following command to install the Vitis-AI-Library runtime package.

   ```
   rpm -ivh --force libvitis_ai_library-1.3.0-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, you need to preform the following steps in order. 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 DPU IP. Refer to https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD to complete the integration of DPU IP.

   Note: After this step is completed, an `sd_card` directory and an `sd_card.img` image with DPU are created. For more known issues, see to Known issues.

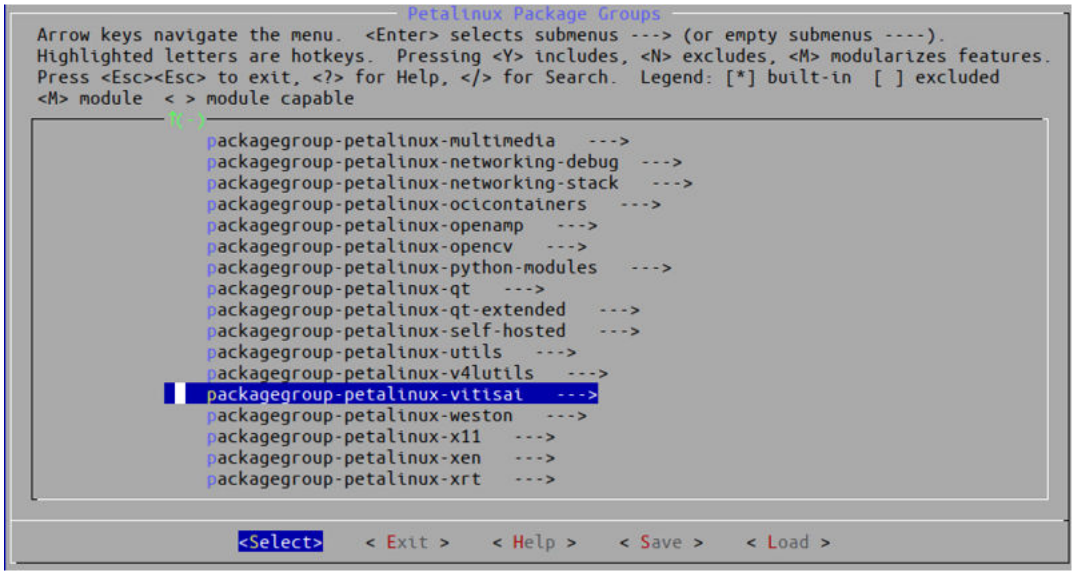
3. Install the dependent libraries of Vitis AI.

   There are two ways to install the dependent libraries of Vitis AI. One is to rebuild system through the configuration of PetaLinux, the other is to install the Vitis AI dependent libraries online.

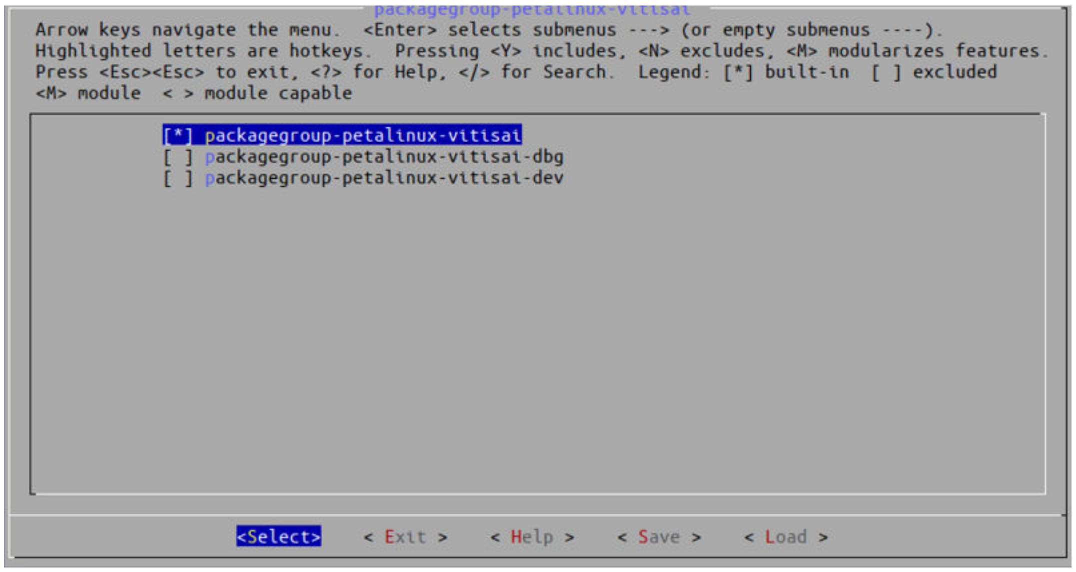
   a. Rebuild the system using PetaLinux configuration by executing the following command:

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

   b. Ensure that **packagegroup-petalinux-vitisai** is selected, as shown in the following figure:



```
Petalinux Package Groups
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.
Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
<M> module  < > module capable

            ↑(-)
            packagegroup-petalinux-multimedia  --->
            packagegroup-petalinux-networking-debug  --->
            packagegroup-petalinux-networking-stack  --->
            packagegroup-petalinux-ocicontainers  --->
            packagegroup-petalinux-openamp  --->
            packagegroup-petalinux-opencv  --->
            packagegroup-petalinux-python-modules  --->
            packagegroup-petalinux-qt  --->
            packagegroup-petalinux-qt-extended  --->
            packagegroup-petalinux-self-hosted  --->
            packagegroup-petalinux-utils  --->
            packagegroup-petalinux-v4lutils  --->
            packagegroup-petalinux-vitisai  --->
            packagegroup-petalinux-weston  --->
            packagegroup-petalinux-x11  --->
            packagegroup-petalinux-xen  --->
            packagegroup-petalinux-xrt  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```



```
packagegroup-petalinux-vitisai
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.
Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
<M> module  < > module capable

        [*] packagegroup-petalinux-vitisai
        [ ] packagegroup-petalinux-vitisai-dbg
        [ ] packagegroup-petalinux-vitisai-dev

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

c. Execute the following command to recompile the system.

```
petalinux-build
```

d. You can also install the Vitis AI dependent libraries online by executing `dnf install packagegroup-petalinux-vitisai` to complete the installation, as shown in the following code:

```
root@xilinx-zcu104-2020_1:/media/sd-mmcblk0p1# dnf install
packagegroup-petalinux-vitisai
Last metadata expiration check: 1 day, 18:12:25 ago on Wed Jun 17
09:35:01 2020.
Package packagegroup-petalinux-vitisai-1.0-r0.noarch is already
installed.
Dependencies resolved.
Nothing to do.
Complete!
```

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

***Note*:** After this step is completed, the 1.2 version of VART is also installed into the system and the system image for Vitis AI is available.

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

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.

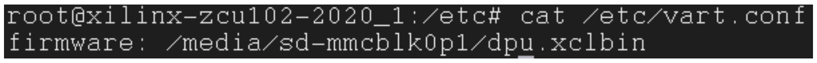
5. Update the Vitis AI Runtime libraries to VAI1.3, if needed.

   After the custom board boots up with the above system image, the 1.2 version of VART is installed. If you want to update them to the 1.3, you have to update the following five library packages, as shown below.

   - 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 under `/etc` to indicate the `dpu.xclbin` file location, as shown below. The Vitis AI examples fetch the `dpu.xclbin` file by reading `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 `vart.conf`.



```
root@xilinx-zcu102-2020_1:/etc# cat /etc/vart.conf
firmware: /media/sd-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 Vitis AI development kit v1.3 release, there are two kinds of examples. They are;

- VART-based examples demonstrating the using of the Vitis AI unified high-level C++/Python APIs (which are available across Cloud-to-Edge).
- DNNDK-based examples demonstrating the usage of the Vitis AI advanced low-level C++/Python APIs (only available for the Edge DPUCZDX8G).

These samples can be found at https://github.com/Xilinx/Vitis-AI/demo. 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.

- Batch mode on Alveo boards need 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 3: **Vitis AI Examples**

ID	Example Name	Models	Framework	Notes
1	resnet50	ResNet-50	Caffe	Image classification with Vitis AI unified C++ APIs.
2	resnet50_mt_py	ResNet-50	TensorFlow	Multi-threading image classification with Vitis AI unified Python APIs.
3	inception_v1_mt_py	Inception-v1	TensorFlow	Multi-threading image classification with Vitis AI unified Python APIs.
4	pose_detection	SSD, Pose detection	Caffe	Pose detection with Vitis AI unified C++ APIs.
5	video_analysis	SSD	Caffe	Traffic detection with Vitis AI unified C++ APIs.
6	adas_detection	YOLOv3	Caffe	ADAS detection with Vitis AI unified C++ APIs.
7	segmentation	FPN	Caffe	Semantic segmentation with Vitis AI unified C++ APIs.
8	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");
//populate input/output tensors
auto job_id = runner->execute_async(inputsPtr, outputsPtr);
runner->wait(job_id.first, -1);
//process outputs
```

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
```

Running Vitis AI Examples on DPUCZD8G and DPUCAHX8H

Before running Vitis™ AI examples on Edge or on Cloud, download the `vitis_ai_runtime_r1.3.0_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 precompiled. You can directly run the example program on the target.

For Edge (DPUCZDX8G)

1. Download the `vitis_ai_runtime_r1.3.0_image_video.tar.gz` from host to the target using `scp` with the following command.

   ```
   scp vitis_ai_runtime_r1.3.0_image_video.tar.gz root@[IP_OF_BOARD]:~/
   ```

2. Unzip the `vitis_ai_runtime_r1.3.0_image_video.tar.gz` package.

   ```
   tar -xzvf vitis_ai_runtime_r1.3.0_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/models/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-r1.3.0.tar.gz -O resnet50-zcu102_zcu104-r1.3.0.tar.gz

   scp resnet50-zcu102_zcu104-r1.3.0.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-r1.3.0.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/demo/VART/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: 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 4: **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_mt_py	`python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
3	inception_v1_mt_py	`python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel`
4	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`
5	video_analysis	`./video_analysis video/structure.webm /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel`
6	adas_detection	`./adas_detection video/adas.webm /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel`
7	segmentation	`./segmentation video/traffic.webm /usr/share/vitis_ai_library/models/fpn/fpn.xmodel`
8	squeezenet_pytorch	`./squeezenet_pytorch /usr/share/vitis_ai_library/models/squeezenet_pt/squeezenet_pt.xmodel`

For Cloud (DPUCAHX8H)

Before running the samples on the cloud, make sure that the Alveo card, such as U50, U50LV, or U280, 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 is located in the path of `/workspace/demo/VART/` in the docker system.

1. Download the `vitis_ai_runtime_r1.3.0_image_video.tar.gz` package and unzip it.

   ```
   tar -xzvf vitis_ai_runtime_r1.3.0_image_video.tar.gz -C /workspace/demo/
   VART
   ```

2. Compile the sample, take `resnet50` as an example.

   ```
   cd /workspace/demo/VART/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 yaml file of the model. You can find the yaml file in `Vitis-AI/models/AI-Model-Zoo` and download the model of the corresponding model. Take `resnet50` as an example:

   ```
   wget https://www.xilinx.com/bin/public/openDownload?filename=resnet50-
   u50-r1.3.0.tar.gz -O resnet50-u50-r1.3.0.tar.gz
   ```

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

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

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

Then install the model package.

```
tar -xzvf resnet50-u50-r1.3.0.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 5: **Launching Commands for Vitis AI Samples on U50/U50LV/U280**

ID	Example Name	Command
1	resnet50	`./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
2	resnet50_mt_py	`/usr/bin/python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel`
3	inception_v1_mt_py	`/usr/bin/python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel`
4	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`
5	video_analysis	`./video_analysis video/structure.mp4 /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel`
6	adas_detection	`./adas_detection video/adas.avi /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel`
7	segmentation	`./segmentation video/traffic.mp4 /usr/share/vitis_ai_library/models/fpn/fpn.xmodel`
8	squeezenet_pytorch	`./squeezenet_pytorch /usr/share/vitis_ai_library/models/squeezenet_pt/squeezenet_pt.xmodel`

Support

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

Understanding the Vitis AI Model Zoo Networks

The Vitis™ AI Model Zoo includes optimized deep learning models to speed up the deployment of deep learning inference applications on Xilinx® platforms. These models cover different application fields, including but not limited to ADAS/AD, video surveillance, robotics, data center, etc. You can get started with these free pre-trained models to enjoy the benefits of deep learning acceleration.

In Vitis 1.3 AI Model Zoo, a variety of Neural Network models with three popular frameworks, Caffe, TensorFlow and PyTorch, are provided. For every model, a .yaml file that provides a description of model name, framework, task type, network backbone, train & validation dataset, float OPS, prune or not, download link, license, and md5 checksum is released. You can browse a model list in Vitis 1.3 AI Model Zoo and select a Neural Network model that you are interested in and get its basic information from a specified .yaml file. With the download link in the .yaml file, you can download the model freely.

For example, if you need a ResNet-50 model used for general image classification on TensorFlow, then find a model named tf_resnetv1_50_imagenet_224_224_6.97G_1.3. According to standard naming rules, models are named using this format: F_M_(D)_H_W_(P)_C_V.

- F specifies training framework: cf is caffe, tf is TensorFlow, dk is Darknet, pt is PyTorch.
- M specifies the model feature.
- D specifies the dataset. It is optional depending on whether the dataset is public or private. Mixed means a mixture of multiple public datasets.
- H specifies the height of input data.
- W specifies the width of input data.
- P specifies the pruning ratio, it means how much computation is reduced. It is optional depending on whether the model is pruned or not.
- C specifies the computation of the model: how many Gops per image.
- V specifies the version of Vitis AI.

As such, tf_resnetv1_50_imagenet_224_224_6.97G_1.3 is a ResNet v1-50 model trained with TensorFlow using the Imagenet dataset, input data size is 224*224, not pruned, the computation per image is 6.97 Gops and Vitis AI version is 1.3.

Then you can choose this model and download it manually using the link provided in tf_resnetv1_50_imagenet_224_224_6.97G_1.3.yaml or through tools that can read .yaml information.

For more information about models list, see https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo/model-list.

Chapter 4

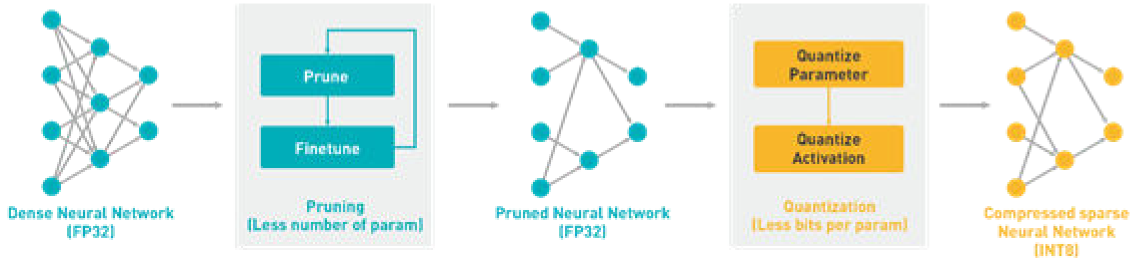
Quantizing the Model

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 20: **Pruning and Quantization Flow**


Dense Neural Network (FP32)
Prune
Finetune
Pruning (Less number of param)
Pruned Neural Network (FP32)
Quantize Parameter
Quantize Activation
Quantization (Less bits per param)
Compressed sparse Neural Network (INT8)

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), PyTorch, and Caffe. The quantizer names are vai_q_tensorflow, vai_q_pytorch, and vai_q_caffe, respectively. The Vitis AI quantizer for TensorFlow 1.x and TensorFlow 2.x are implemented in different ways and are released separately. For TensorFlow 1.x, the Vitis AI quantizer is based on TensorFlow 1.15. After adding quantization features, the Vitis AI quantizer rebuilds and redistributes a standalone package. For TensorFlow 2.x, the Vitis AI quantizer is a Python package with several quantization APIs. You can import this package and the Vitis AI quantizer works like a plugin for TensorFlow.

Table 6: **Vitis AI Quantizer Supported Frameworks and Features**

	Versions	Features		
		Quantize Calibration (post-training quantization)	Quantize Finetuning (quantize-aware training)	Fast Finetuning (Advanced Calibration)
TensorFlow 1.x	Based on 1.15	Yes	Yes	No
TensorFlow 2.x	Supports 2.3	Yes	Yes	No. This feature is currently in development.
PyTorch	Supports 1.2 - 1.4	Yes	Yes	Yes
Caffe	-	Yes	Yes	No

In the quantize calibration process, only a small set of unlabeled images are required 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, quantize finetuning can be used to further improve the accuracy of quantized models. Quantize finetuning requires the original training dataset. According to experiments, 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 quantize finetuning.

For quantize calibration, cross layer equalization [1] algorithm is implemented in Vitis AI 1.3. 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 nothing is that for fast finetuning, each run will get different result. This is similar to quantize finetuning.

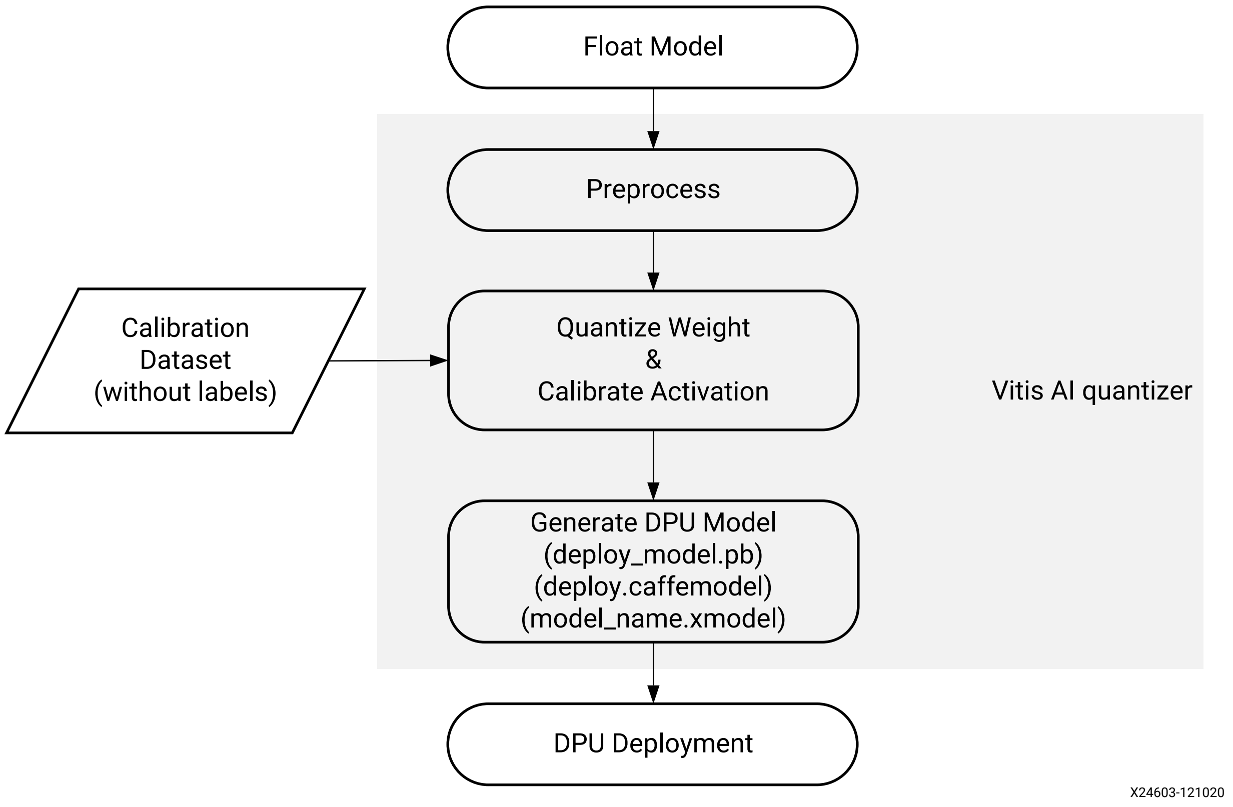
***Note*:**

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 21: **VAI Quantizer Workflow**


Float Model
Preprocess
Calibration Dataset (without labels)
Quantize Weight & Calibrate Activation
Vitis AI quantizer
Generate DPU Model (deploy_model.pb) (deploy.caffemodel) (model_name.xmodel)
DPU Deployment
X24603-121020

The Vitis AI quantizer takes a floating-point model as input and performs pre-processing (folds batchnorms and removes useless nodes), and then quantizes the weights/biases and activations to the given bit width.

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, the un-labeled 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, and `deploy.prototxt` / `deploy.caffemodel` for vai_q_caffe), 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, PyTorch, or Caffe framework.

TensorFlow 1.x Version (vai_q_tensorflow)

Installing vai_q_tensorflow

There are two 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
```

Install using Source Code

vai_q_tensorflow is a fork of TensorFlow from branch "r1.15". It is open source in Vitis_AI_Quantizer. vai_q_tensorflow building process is the same as TensorFlow 1.15. See the TensorFlow documentation for more information.

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 7: **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.

Getting the Frozen Inference Graph

In most situations, 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
```

Getting 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 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}
```

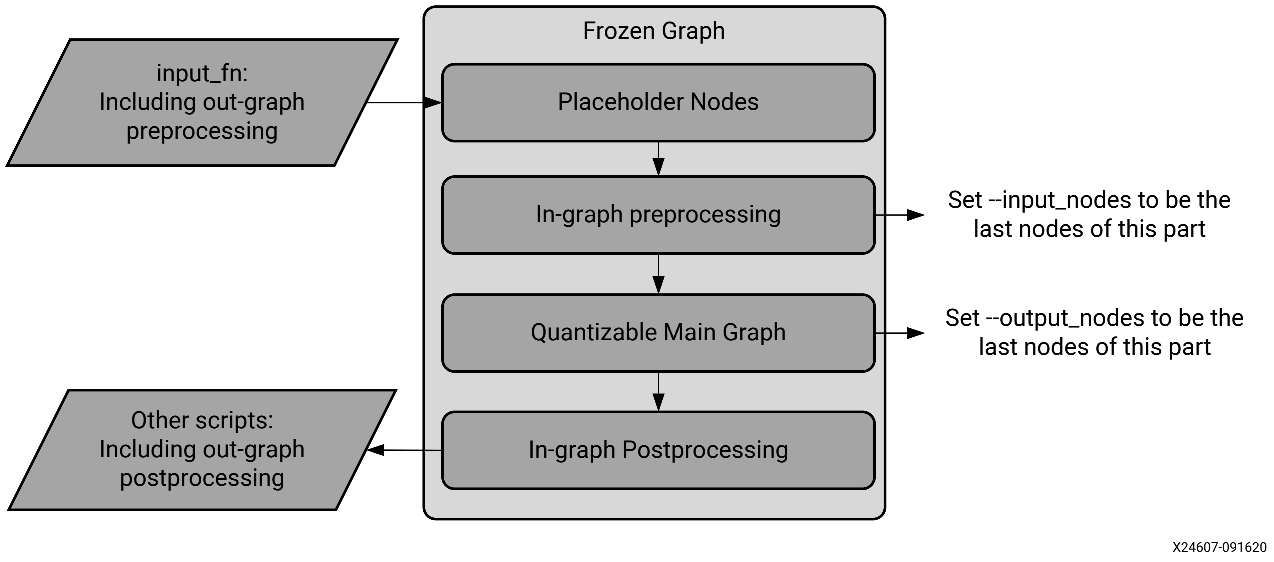
Quantizing the Model using vai_q_tensorflow

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 22: **Quantization Flow for TensorFlow**


Frozen Graph
input_fn: Including out-graph preprocessing
Placeholder Nodes
In-graph preprocessing
Set --input_nodes to be the last nodes of this part
Quantizable Main Graph
Set --output_nodes to be the last nodes of this part
Other scripts: Including out-graph postprocessing
In-graph Postprocessing
X24607-091620

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 post-processing 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 and 1 for min-diffs. 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.

Outputing the Quantized Model

After the successful execution of the vai_q_tensorflow command, two files are generated in `${output_dir}`:

- `quantize_eval_model.pb` is used to evaluate on CPU/GPUs, and can be used to simulate the results on hardware. Run import `tensorflow.contrib.decent_q` explicitly to register the custom quantize operation, because `tensorflow.contrib` is now lazy-loaded.
- `deploy_model.pb` is used to compile the DPU codes and deploy on it. It can be used as the input file for the Vitis AI compiler.

Table 8: **vai_q_tensorflow Output Files**

No.	Name	Description
1	deploy_model.pb	Quantized model for VAI compiler (extended TensorFlow format) for targeting DPUCZDX8G implementations.
2	quantize_eval_model.pb	Quantized model for evaluation (also, VAI compiler input for most DPU architectures, like DPUCAHX8H, DPUCAHX8L, and DPUCADF8H)

(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 `from tensorflow.contrib import decent_q` to register the quantize operation.
- 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

Sometimes it is necessary to compare the simulation results on the CPU/GPU with the output values on the DPU. vai_q_tensorflow supports dumping the simulation results with the `quantize_eval_model.pb` generated by the quantizer.

Run the following commands to dump the quantize simulation results:

```
$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.

After the previous command has executed 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. In the folders, 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 9: **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 Quantize Finetuning

Quantize finetuning is similar to float model finetuning but in the former, 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 finetuning, prepare the following files:

Table 10: **Input Files for vai_q_tensorflow Quantize Finetuning**

No.	Name	Description
1	Checkpoint files	Floating-point checkpoint files to start from. Can be omitted if train from scratch.
2	Dataset	The training dataset with labels.
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 the quantize finetuning.

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
from tensorflow.contrib import decent_q
q_config = decent_q.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
decent_q.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
# ...
```

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 part 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. In such a case, you can add a tf.identity node to make a 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, 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 deploy model: After quantize finetuning, generate the deploy model. Before that, you need to evaluate the quantized graph with a checkpoint file. This can be done by calling the following function after building the float evaluation graph. As the deploy process needs to run based on the quantize evaluation graph, so they are often called together.

```
# eval.py

# ...

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

# *Set the quantize configurations
from tensorflow.contrib import decent_q
q_config = decent_q.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
decent_q.CreateQuantizeEvaluationGraph(config=q_config)
# *Call Vai_q_tensorflow api to freeze the model and generate the deploy
model
decent_q.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 file in the `${output_dir}`:

Table 11: **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. No weights inside, 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. XIR compiler uses it as input.
deploy_model_{suffix}.pb	No	Input to VAI compiler (DPUCZDX8G)	The deploy model, operations and quantize information are fused. DNNC compiler uses it as input.

The suffix contains the iteration information from the checkpoint file and the date information to make it clear to combine it to checkpoints files. For example, if the checkpoint file is "model.ckpt-2000.*" and the date is 20200611, then the suffix is "2000_20200611000000".

Quantize Finetuning APIs for TensorFlow 1.x

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, quantize finetuning can be used to further improve the accuracy of quantized models.

There are three APIs for quantize finetuning in the Python package, `tf.contrib.decent_q`.

tf.contrib.decent_q.CreateQuantizeTrainingGraph(config)

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

Arguments

- `config`: A `tf.contrib.decent_q.QuantizeConfig` object, containing the configurations for quantization.

tf.contrib.decent_q.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 `tf.contrib.decent_q.QuantizeConfig` object, containing the configurations for quantization.

tf.contrib.decent_q.CreateQuantizeDeployGraph(checkpoint, config)

Freeze the checkpoint into the quantize evaluation graph and convert the quantize evaluation graph to deploy graph.

Arguments

- `checkpoint`: A `string` object, the path to checkpoint folder of file.
- `config`: A `tf.contrib.decent_q.QuantizeConfig` object, containing the configurations for quantization.

Tips for Quantize Finetuning

The following are some tips for quantize finetuning.

- **Dropout:** Experiments shows that quantize finetuning works better without dropout ops. This tool does not support quantize finetuning with dropouts at the moment and they should be removed or disabled before running the quantize finetuning. 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-param:** Quantize finetuning is like float finetuning, so the techniques for float finetuning are also needed. The optimizer type and the learning rate curve are some important parameters to tune.

vai_q_tensorflow Supported Operations and APIs

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

Table 12: **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	relu relu6 leaky_relu	/	ReLU LeakyReLU
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	dropout[#2] softmax[#3]	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. There is no need to quantize softmax output and vai_q_tensorflow does not quantize it.

vai_q_tensorflow Usage

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

Table 13: **vai_q_tensorflow Options**

Name	Type	Description
Common Configuration		
--input_frozen_graph	String	TensorFlow frozen inference GraphDef file for the floating-point model, used for quantize calibration.

Table 13: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--input_nodes	String	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. 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 before the post-processing part, because some operations in the pre- and postprocessing parts 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	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. 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 before the post-processing part, because some operations in the pre- and post-processing parts 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	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.

--input_fn	String	This function provides 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 preprocessing 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 preprocessing part which is not in the graph file should be handled in in the input_fn.

Quantize Configuration		
--weight_bit	Int32	Bit width for quantized weight and bias. Default: 8
--activation_bit	Int32	Bit width for quantized activation. Default: 8
--nodes_bit	String	Specify bit width of nodes, nodes name and bit width form a pair of parameter joined by a colon, and parameters are comma separated. When specify conv op name only vai_q_tensorflow will quantize weights of conv op using specified bit width.e.g 'conv1/Relu:16,conv1/weights:8,conv1:16'

Table 13: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--method	Int32	The method for quantization. 0: Non-overflow method. Makes sure that no values are saturated during quantization. Sensitive to outliers. 1: Min-diffs method. Allows saturation for quantization to get a lower quantization difference. Higher tolerance to outliers. Usually ends with narrower ranges than the non-overflow method. Choices: [0, 1] Default: 1
--nodes_method	String	Specify method of nodes, nodes name and method form a pair of parameter joined by a colon, and parameter pairs are comma separated. When specify conv op name only vai_q_tensorflow will quantize weights of conv op using specified method. e.g 'conv1/Relu:1,depthwise_conv1/weights:2,conv1:1'
--calib_iter	Int32	The iterations of calibration. Total number of images for calibration = calib_iter * batch_size. Default: 100
--ignore_nodes	String	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. Choices: [0, 1] Default: 0
--align_concat	Int32	The strategy for the alignment of the input quantizeposition for concat nodes. Set to 0 to align all concat nodes, 1 to align the output concat nodes, and 2 to disable alignment. Choices: [0, 1, 2] Default: 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 for these operations if this flag is set to 1. Choices: [0, 1] Default: 1
--adjust_shift_bias	Int32	The strategy for shift bias check and adjustment for DPU compiler. Set to 0 to disable shift bias check and adjustment, 1 to enable with static constraints, 2 to enable with dynamic constraints. choices: [0, 1, 2] default: 1
--adjust_shift_cut	Int32	The strategy for shift cut check and adjustment for DPU compiler. Set to 0 to disable shift cut check and adjustment, 1 to enable with static constraints. choices: [0, 1] default: 1
--arch_type	String	Specify 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	The directory in which to save the quantization results. Default: "./quantize_results"

Table 13: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--max_dump_batches	Int32	The maximum number of batches for dumping. Default: 1
--dump_float	Int32	If set to 1, the float weights and activations will also be dumped. Choices: [0, 1] Default: 0
--dump_input_tensors	String	Specify input tensor name of Graph when graph entrance is not a placeholder. We will add a placeholder according to the dump_input_tensor, so that input_fn can feed data.
Session Configurations		
--gpu	String	The ID of the GPU device used for quantization, comma separated.
--gpu_memory_fraction	Float	The GPU memory fraction used for quantization, between 0-1. Default: 0.5
Others		
--help		Show all available options of vai_q_tensorflow.
--version		Show vai_q_tensorflow version information.

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.

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
```

Install from Source Code

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
```

Running vai_q_tensorflow2

Use the following steps to run vai_q_tensorflow.

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 14: **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

```
float_model = tf.keras.models.load_model(‘float_model.h5’)
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(float_model)
quantized_model = quantizer.quantize_model(calib_dataset=eval_dataset)
```

Here, `"eval_dataset"` is used as a representative calibration dataset for calibration as an example. You can also use train_dataset or other datasets. The quantizer reads the whole dataset to calibrate it. If you use the `tf.data.Dataset` object, the batch size is controlled by the dataset itself. If you use the `numpy.array` object, the default batch size is 50.

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) 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 quantized model should be loaded to "quantize_scope", for example:

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

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

```
model.compile(    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics= keras.metrics.SparseTopKCategoricalAccuracy())
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
quantizer = vitis_quantize.VitisQuantizer.dump_model(quantized_model,
dump_dataset, dump_output_dir)
```

***Note*:** The batch_size of the dump_dataset should be set to 1 for DPU debugging.

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 *_float.bin and *_float.txt. The / symbol is replaced by _ for simplicity. Examples for dumping results are shown in the following table.

Table 15: **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

vai_q_tensorflow2 Quantize 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, quantize finetuning can be used to further improve the accuracy of quantized models.

Technically, quantize finetuning is similar to float model finetuning. The difference is that quantize finetuning uses the APIs of the vai_q_tensorflow2 to rewrite the float graph to convert it to a quantized model before the training starts. The typical workflow is as follows:

1. Preparation.

   Before finetuning, please prepare the following files:

   Table 16: **Input Files for vai_q_tensorflow2 Quantize Finetuning**

No.	Name	Description
1	Float model file	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

   It is suggested to evaluate the float model first before doing quantize finetuning, which can check the correctness of the scripts and dataset. The accuracy and loss values of the float checkpoint can also be a baseline for the quantize finetuning.

3. Modify the Training Scripts

Use the vai_q_tensorflow2 API, `VitisQuantizer.get_qat_model`, to do the quantization. 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)
model = quantizer.get_qat_model()

# Compile the model
model.compile(
    optimizer= RMSprop(learning_rate=lr_schedule),
loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=keras.metrics.SparseTopKCategoricalAccuracy())

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

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
model.save('trained_model.h5')

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

5. (Optional) Evaluate the Quantized Model

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

   ***Note*:** Quantize finetuning works like float finetuning, so it will be of great help to have some experience on float model training and finetuning. For example, how to choose hyper-parameters like optimizer type and learning rate.

vai_q_tensorflow2 Supported Operations and APIs

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

Table 17: **vai_q_tensorflow2 Supported Layers**

Supported Layers
tf.keras.layers.Conv2D

Table 17: **vai_q_tensorflow2 Supported Layers** *(cont'd)*

Supported Layers
tf.keras.layers.Conv2DTranspose
tf.keras.layers.DepthwiseConv2D
tf.keras.layers.Dense
tf.keras.layers.AveragePooling2D
tf.keras.layers.MaxPooling2D
tf.keras.layers.GlobalAveragePooling
tf.keras.layers.UpSampling2D
tf.keras.layers.BatchNormalization
tf.keras.layers.Concatenate
tf.keras.layers.Zeropadding2D
tf.keras.layers.Flatten
tf.keras.layers.Reshape
tf.keras.layers.ReLU
tf.keras.layers.Activation
tf.keras.layers.Add

vai_q_tensorflow2 Usage

```
vitis_quantize.VitisQuantizer(model, custom_quantize_strategy=None)
```

The construction function of class `VitisQuantizer`.

Arguments:

- **model:** A `tf.keras.Model` object, containing the configurations for quantization.
- **custom_quantize_strategy:** A custom quantize strategy json file.

```
vitis_quantize.VitisQuantizer.quantize_model(
calib_dataset=None,
fold_conv_bn=True,
fold_bn=True,
replace_relu6=True,
include_cle=True,
cle_steps=10)
```

This function quantizes the float model, including model optimization, weights quantization and activation quantize calibration.

Arguments:

- **calib_dataset:** A `tf.data.Dataset` or `np.numpy` object, the calibration dataset.

- **fold_conv_bn:** A `bool` object, whether to fold the batchnorm layers into previous `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` layers
- **fold_bn:** A `bool` object, whether to fold the batchnorm layer's moving_mean and moving_variance values into the gamma and beta values.
- **replace_relu6:** 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.

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

This function dumps the simulation results of the quantized model, including model optimization, weights quantizing and activation quantize calibration.

Arguments:

- **dataset:** A `tf.data.Dataset` 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.

Examples

- **Quantize:**

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

- **Evaluate quantized model:**

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

- **Load 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 quantized model:**

```
from tensorflow_model_optimization.quantization.keras import
vitis_quantize
vitis_quantize.VitisQuantizer.dump_model(quantized_model,
dump_dataset=eval_dataset)
```

PyTorch Version (vai_q_pytorch)

Installing vai_q_pytorch

vai_q_pytorch has GPU and CPU versions. It supports PyTorch version 1.2-1.4 but 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 meet 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.

Install from 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 (1.2-1.4) and torchvision.

The following code takes PyTorch 1.4 and torchvision 0.5.0 as an example. You can find detailed instructions for other versions on the PyTorch website.

```
pip install torch==1.4.0 torchvision==0.5.0
```

3. Install other dependencies.

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

4. Install vai_q_pytorch.

```
cd ./pytorch_binding
python setup.py install (for user)
python setup.py develop (for developer)
```

5. Verify the installation.

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

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

```
import pytorch_nndct
import torch
```

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.

Adding vai_q_pytorch APIs to Float Scripts

If, before quantization, there is a trained float model and some Python scripts to evaluate accuracy/mAP of the model, 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_xmodel())
   ```

Running Quantization and Getting the Result

***Note*:** vai_q_pytorch log messages have special colors and a special keyword, "NNDCT". "NNDCT" is an internal project name and you can change it later. 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 there are loss and accuracy messages displayed in the end, you can ignore them. Note the colorful log messages with the special keyword, "NNDCT".

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, argument "subset_len" controls how many images are used for network forwarding. If the float evaluation script does not have an argument with a similar role, add one; otherwise, it should be changed manually.

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 got. (Keep it for
evaluation of quantized model)
```

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 right accuracy for the quantized model.

3. To generate the xmodel for compilation, 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 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 model
```

Note: XIR is ready in "vitis-ai-pytorch" conda environment in the Vitis-AI docker but if vai_q_pytorch is installed from 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.

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
```

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 get satisfactory results, quantize finetuning can be used to further improve the accuracy of quantized models.

With a small set of unlabeled data, the AdaQuant algorithm[1] not only calibrates the activations but also finetunes the weights. AdaQuant uses a small set of unlabeled data. This is similar to calibration but it finetunes the model. The Vitis AI quantizer implements this algorithm and call it "fast finetuning" or "advanced calibration". Though slightly slower, fast finetuning can achieve better performance than quantize calibration. Similar to quantize finetuning, each run of fast finetuning produces a different result.

Fast finetuning is not real training of the model, and only needs limited number of iterations. For classification models on Imagenet dataset, 1000 images are enough. 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 among fast finetuning. Re-calibration with the original inference code is highly 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=1024,
          train=False,
          batch_size=batch_size,
          sample_method=None,
          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 finetuned quantized model accuracy, run the following command:

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

Note:

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

vai_q_pytorch Quantize Finetuning

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 quantize finetuning 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
quantizer = torch_quantizer(quant_mode = 'calib',
                           module = model,
                           input_args = input,
                           bitwidth = 8,
                           qat_proc = True)
quantized_model = quantizer.quant_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. Convert the trained model to a deployable model.

   After training, dump the quantized model to xmodel. (`batch size=1` is must for compilation of xmodel).

```
# vai_q_pytorch interface function: deploy the trained model and convert
xmodel
  # need at least 1 iteration of inference with batch_size=1
  quantizer.deploy(quantized_model)
  deployable_model = quantizer.deploy_model
  val_dataset2 = torch.utils.data.Subset(val_dataset, list(range(1)))
  val_loader2 = torch.utils.data.DataLoader(
      val_dataset,
      batch_size=1,
      shuffle=False,
      num_workers=workers,
      pin_memory=True)
  validate(val_loader2, deployable_model, criterion, gpu)
  quantizer.export_xmodel()
```

vai_q_pytorch Quantize Finetuning 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 still does not get satisfactory results, quantize finetuning can be used to further improve the accuracy of quantized models.

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

1. All operations to be quantized must be an instance of `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 quantize finetuning. Thus, replace `'+'` with `pytorch_nndct.nn.modules.functional.Add`. A list of operations that need replacement is shown 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 whole complete network or a partial sub-network.

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():
  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"),
               qat_proc: bool = False):
```

Class `torch_quantizer` will create a quantizer object.

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.
- **Qat_proc:** Turn on quantize finetuning, also named quantization-aware-training (QAT).

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

This function exports quantization steps information

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

This function export xmodel and dump operators' output data for detailed data comparison

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 will be dumped to `output_dir/deploy_check_data_int/`.

Caffe Version (vai_q_caffe)

Installing vai_q_caffe

There are two ways to install vai_q_caffe:

Install using Docker Containers

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

```
conda activate vitis-ai-caffe
```

Install from Source Code

vai_q_caffe is an open source in the caffe_xilinx repository. It is a fork of the NVIDIA Caffe from branch "caffe-0.15" maintained by Xilinx. Building process is the same as BVLC Caffe. Refer to installation instructions here.

Running vai_q_caffe

Use the following steps to run vai_q_caffe.

1. Prepare the Neural Network Model

Table 20: **vai_q_caffe Input Files**

No.	Name	Description
1	float.prototxt	Floating-point model for ResNet-50. The data layer in the prototxt should be consistent with the path of the calibration dataset.
2	float.caffemodel	Pre-trained weights file for ResNet-50.
3	calibration dataset	A subset of the training set containing 100 to 1000 images.

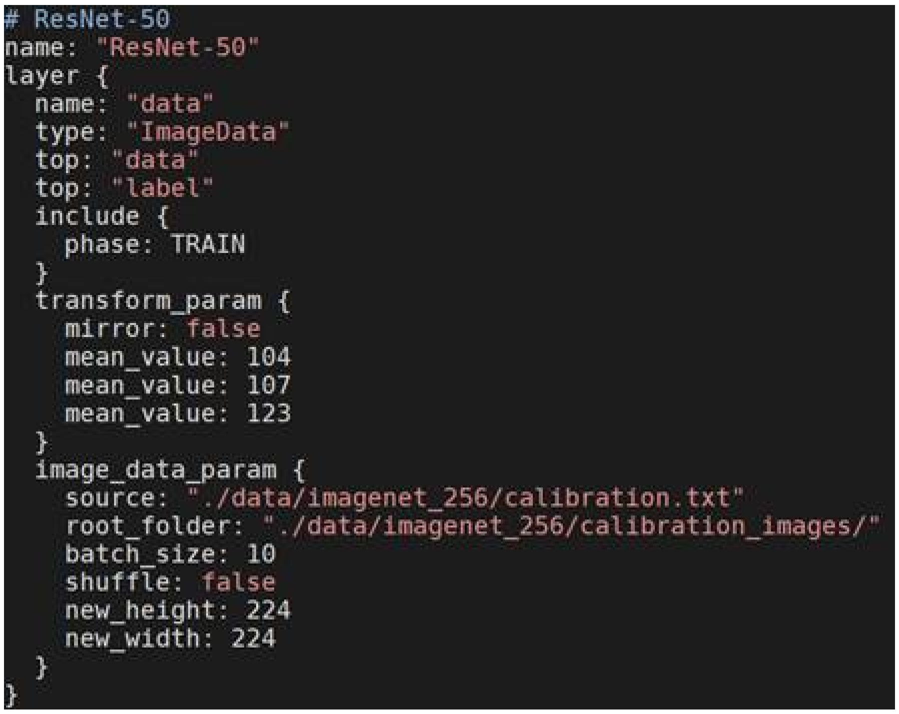
Before running vai_q_caffe, prepare the Caffe model in floating-point format with the calibration data set, including the following:

- Caffe floating-point network model prototxt file.
- Pre-trained Caffe floating-point network model caffemodel file.

- Calibration data set. The calibration set is usually a subset of the training set or actual application images (at least 100 images). Make sure to set the source and root_folder in image_data_param to the actual calibration image list and image folder path. For quantize calibration, calibration data without a label is enough. But due to the implementation, an image list file with two columns is required. Just set the second column to a random value or zero. This is an example of "`calibration.txt`".

```
n01440764_985.JPEG 0
n01443537_9347.JPEG 0
n01484850_8799.JPEG 0
...
```

Figure 23: **Sample Caffe Layer for Quantization**



```
# ResNet-50
name: "ResNet-50"
layer {
  name: "data"
  type: "ImageData"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: false
    mean_value: 104
    mean_value: 107
    mean_value: 123
  }
  image_data_param {
    source: "./data/imagenet_256/calibration.txt"
    root_folder: "./data/imagenet_256/calibration_images/"
    batch_size: 10
    shuffle: false
    new_height: 224
    new_width: 224
  }
}
```

***Note*:** Three mean_value parameters for channels are recommended. If three mean_value parameters are specified, following the order BGR.

2. Run vai_q_caffe to generate a quantized model:

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel
[options]
```

Because there are small differences between hardware platforms, the output formats of `vai_q_caffe` are also different. If the target hardware platform is DPUCAHX8H, the `-keep_fixed_neuron` option should be added to the command.

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -
keep_fixed_neuron[options]
```

3. After successful execution of the above command, four files are generated in the output directory (default directory, `/quantize_results/`). The `deploy.prototxt` and `deploy.caffemodel` files are used as input files to the compiler. The `quantize_train_test.prototxt` and `quantize_train_test.caffemodel` files are used to test the accuracy on the GPU/CPU, and can be used as input files to quantize finetuning.

   Table 21: **vai_q_caffe Output Files**

No.	Name	Description
1	deploy.prototxt	For Vitis AI compiler, quantized network description file.
2	deploy.caffemodel	For Vitis AI compiler, quantized Caffe model parameter file (non-standard Caffe format).
3	quantize_train_test.prototxt	For testing and finetuning, quantized network description file.
4	quantize_train_test.caffemodel	For testing and finetuning, quantized Caffe model parameter file (non-standard Caffe format).

   To evaluate the accuracy of the quantized model, use a command similar to the following, or add an `-auto_test` in step 2. Refer to the next section for vai_q_caffe argument details.

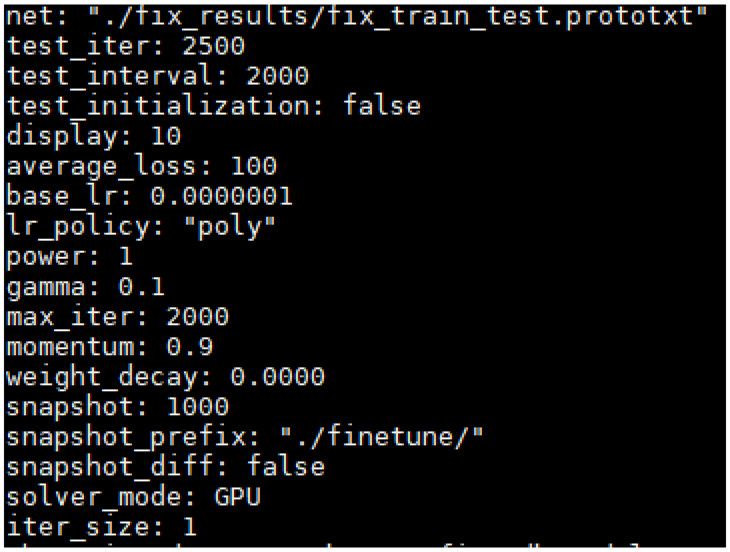
```
vai_q_caffe test -model ./quantize_results/quantize_train_test.prototxt -
weights ./quantize_results/quantize_train_test.caffemodel -gpu 0 -
test_iter 1000
```

vai_q_caffe Quantize Finetuning

Generally, there is a small accuracy loss after quantization, but for some networks such as Mobilenets, the accuracy loss can be large. In such a scenario, quantize finetuning can be used to further improve the accuracy of quantized models.

Finetuning is almost the same as model training, which needs the original training dataset and a `solver.prototxt`. To start finetuning with the `quantize_train_test.prototxt` and Caffe Model, follow these steps:

1. Assign the training dataset to the input layer of `quantize_train_test.prototxt`.
2. Create a `solver.prototxt` file for finetuning. An example of a `solver.prototxt` file is provided below. You can adjust the hyper-parameters to get good results. The most important parameter is base_lr, which is usually much smaller than the one used in training.



```
net: "./fix_results/fix_train_test.prototxt"
test_iter: 2500
test_interval: 2000
test_initialization: false
display: 10
average_loss: 100
base_lr: 0.0000001
lr_policy: "poly"
power: 1
gamma: 0.1
max_iter: 2000
momentum: 0.9
weight_decay: 0.0000
snapshot: 1000
snapshot_prefix: "./finetune/"
snapshot_diff: false
solver_mode: GPU
iter_size: 1
```

3. Run the following command to start finetuning:

   ```
   ./vai_q_caffe finetune -solver solver.prototxt -weights quantize_results/
   quantize_train_test.caffemodel -gpu all
   ```

4. Deploy the finetuned model. The finetuned model is generated in the snapshot_prefix settings of the `solver.prototxt` file, such as `${snapshot_prefix}/finetuned_iter10000.caffemodel`. You can use the test command to test its accuracy.

5. Finally, you can use the deploy command to generate the deploy model (prototxt and caffemodel) for the Vitis AI compiler.

   ```
   ./vai_q_caffe deploy -model quantize_results/
   quantize_train_test.prototxt -weights finetuned_iter10000.caffemodel -
   gpu 0 -output_dir deploy_output
   ```

vai_q_caffe Usage

The vai_q_caffe quantizer takes a floating-point model as an input model and uses a calibration dataset to generate a quantized model. In the following command line, [options] stands for optional parameters.

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel
[options]
```

The options supported by vai_q_caffe are shown in the following table. The three most commonly used options are weights_bit, data_bit, and method.

Table 22: **vai_q_caffe Options List**

Name	Type	Optional	Default	Description
model	String	Required	-	Floating-point prototxt file (such as float.prototxt).

Table 22: **vai_q_caffe Options List** *(cont'd)*

Name	Type	Optional	Default	Description
weights	String	Required	-	The pre-trained floating-point weights (such as float.caffemodel).
weights_bit	Int32	Optional	8	Bit width for quantized weight and bias.
data_bit	Int32	Optional	8	Bit width for quantized activation.
method	Int32	Optional	1	Quantization methods, including 0 for non-overflow and 1 for min-diffs. The non-overflow method ensures that no values are saturated during quantization. It is sensitive to 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.
calib_iter	Int32	Optional	100	Maximum iterations for calibration.
auto_test	-	Optional	Absent	Adding this option will perform testing after calibration using a test dataset specified in the prototxt file. To turn on this option, the floating-point prototxt file must be a workable prototxt for accuracy calculation both in TRAIN and TEST mode.
test_iter	Int32	Optional	50	Maximum iterations for testing.
output_dir	String	Optional	quantize_results	Output directory for the quantized results.
gpu	String	Optional	0	GPU device ID for calibration and test.
ignore_layers	String	Optional	none	List of layers to ignore during quantization.
ignore_layers_file	String	Optional	none	Protobuf file which defines the layers to ignore during quantization, starting with ignore_layers
sigmoided_layers	String	Optional	none	List of layers before sigmoid operation, to be quantized with optimization for sigmoid accuracy
input_blob	String	Optional	data	Name of input data blob
keep_fixed_neuron	Bool	Optional	FALSE	Retain FixedNeuron layers in the deployed model. Set this flag if your targeting hardware platform is DPUCAHX8H

Examples:

- **Quantize:**

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu
0
```

- **Quantize with Auto-Test:**

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu
0 -auto_test -test_iter 50
```

- **Quantize with the Non-Overflow Method:**

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu
0 -method 0
```

- **Finetune the Quantized Model:**

```
vai_q_caffe finetune -solver solver.prototxt -weights quantize_results/
float_train_test.caffemodel -gpu 0
```

- **Deploy Quantized Model:**

```
vai_q_caffe deploy -model quantize_results/quantize_train_test.prototxt -
weights quantize_results/float_train_test.caffemodel -gpu 0
```

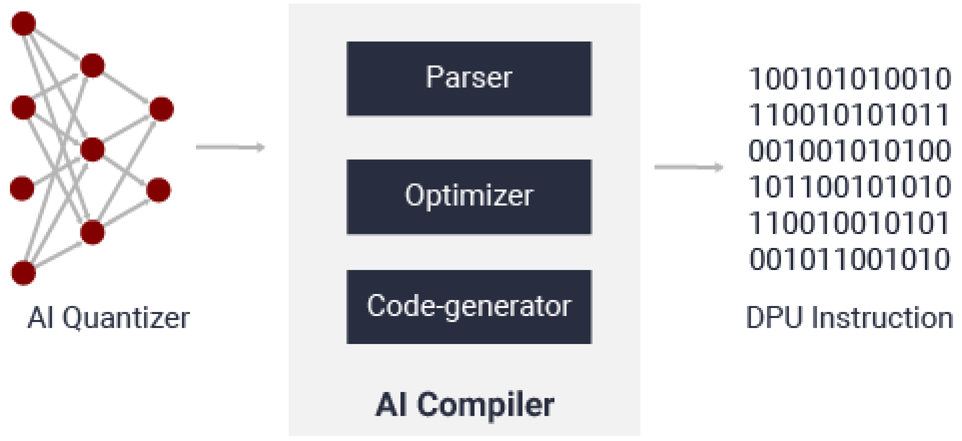
Compiling the Model

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 24: **Vitis AI Compiler Framework**



The Vitis AI Compiler generates the compiled model based on the DPU microarchitecture. There are a number of different DPUs supported in Vitis AI for different platforms and applications. It is important to understand the relations between available compilers and associate DPUs. See DPU Naming for the DPU naming scheme.

To better understand how to map the compilers with DPUs, please refer to the following table.

Table 23: **Mapping Compilers with DPUs**

DPU Name	Compiler	Hardware platform
DPUCZDX8G	XCompiler	Zynq UltraScale+ MPSoC, Zynq-7000 devices
DPUCAHX8H		U50, U280
DPUCAHX8L		U50, U280
DPUCADF8H		U200, U250
DPUCVDX8G		VCK190, Versal AI Core Series
DPUCVDX8H		VCK5000
DPUCADX8G	xfDNN Compiler	U200, U250

XCompiler stands for XIR based Compiler. It can support DPUCZDX8G, DPUCAHX8H, DPUCAHX8L, DPUCVDX8G and DPUCVDX8H. xfDNN Compiler is the compiler from the legacy ML Suite which supports DPUCADX8G only. This has been retained in the Vitis AI 1.3 release for backward compatibility and will be deprecated in the Vitis AI 1.4 release.

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.

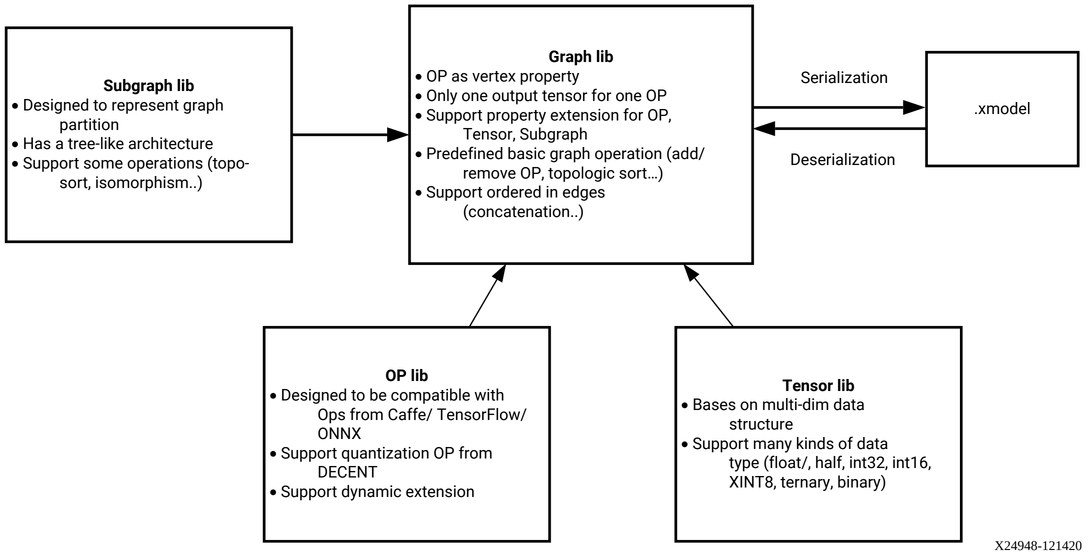
XIR

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, 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.

Figure 25: **XIR Based Flow**


Subgraph lib
• Designed to represent graph partition
• Has a tree-like architecture
• Support some operations (topo-sort, isomorphism..)
Graph lib
• OP as vertex property
• Only one output tensor for one OP
• Support property extension for OP, Tensor, Subgraph
• Predefined basic graph operation (add/ remove OP, topologic sort…)
• Support ordered in edges (concatenation..)
Serialization
Deserialization
.xmodel
OP lib
• Designed to be compatible with Ops from Caffe/ TensorFlow/ ONNX
• Support quantization OP from DECENT
• Support dynamic extension
Tensor lib
• Bases on multi-dim data structure
• Support many kinds of data type (float/, half, int32, int16, XINT8, ternary, binary)
X24948-121420

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 TensorFlow or Caffe model as the input. First, it transforms the input models into the XIR format as the foundation of 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 on the graph and breaks up the graph into several subgraphs on the basis of whether the op can be executed on the DPU. More 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 ZCU platforms, DPUCAHX8H on the Alveo HBM platform optimized for high-throughput applications, DPUCAHX8L on the Alveo HBM platform optimized for low-latency applications, DPUCVDX8G on the Versal Edge platform, and DPUCVDX8H on the Versal Cloud platform. You can find the`arch.json` files for those 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 vai_quantizer.

Caffe

For Caffe, vai_q_caffe generates a PROTOTXT (deploy.prototxt) and a MODEL (deploy.caffemodel). Ensure that you specify the `-keep_fixed_neuron` option for vai_q_caffe which is essential for XIR-based compiler. Run the following command to get the compiled xmodel.

```
vai_c_caffe -p /PATH/TO/deploy.prototxt -c /PATH/TO/deploy.caffemodel -a /
PATH/TO/arch.json -o /OUTPUTPATH -n netname
```

The compiler creates three files in OUTPUTPATH directory. `netname_org.xmodel` is the pre-compiled xmodel which is generated by compiler frontend. `netname.xmodel` is the compiled xmodel which contains instructions and other necessary information. `meta.json` is for runtime.

TensorFlow

For TensorFlow, vai_q_tensorflow generates a pb file (`quantize_eval_model.pb`). Notice that there are two pb files generated by vai_q_tensorflow. The `quantize_eval_model.pb` file is theinput file for the XIR-based compiler. The compilation command is similar.

```
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, which will fail the compilation. You can specify the input tensor shape with an extra option like `--options '{"input_shape": "1,224,224,3"}'`.

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. Sequential APIs will be supported in future releases.

PyTorch

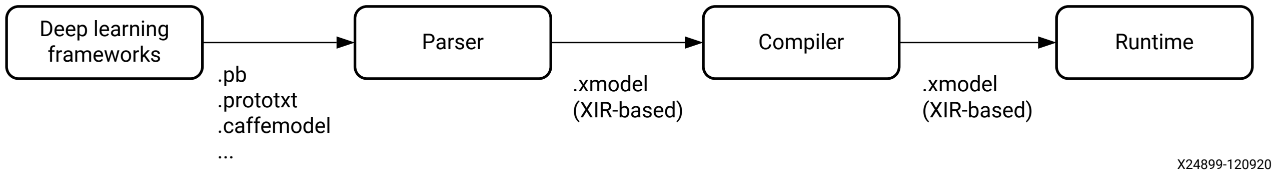
For PyTorch, the quantizer NNDCT outputs the quantized model in the XIR format directly. Use vai_c_xir to compile it.

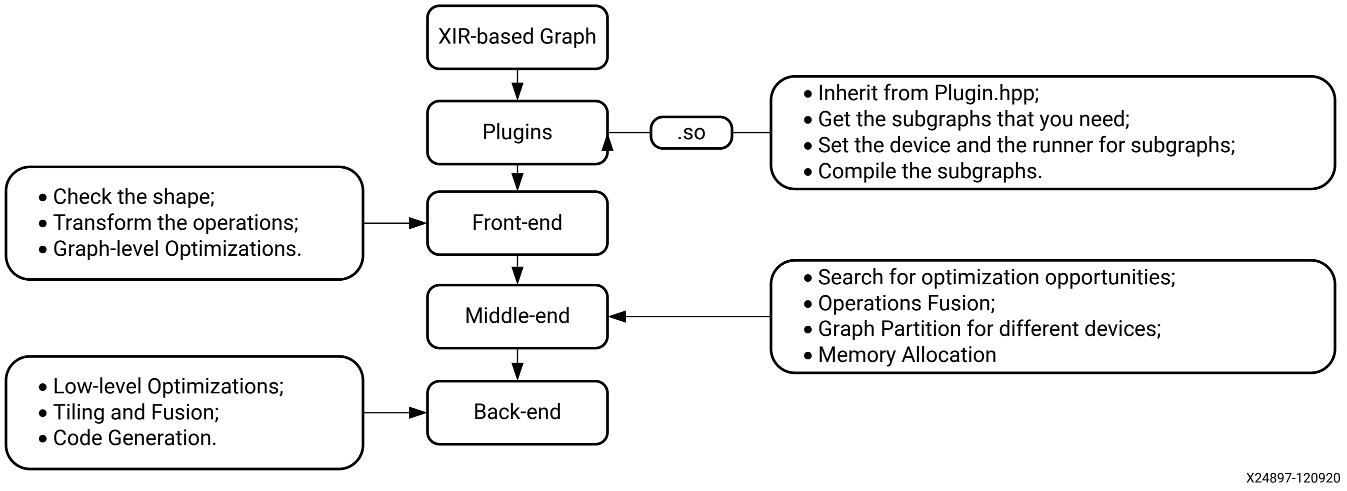
```
vai_c_xir -x /PATH/TO/quantized.xmodel -a /PATH/TO/arch.json -o /OUTPUTPATH
-n netname
```

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 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 26: **Compilation Flow**




XIR-based Graph
Plugins
.so
• Inherit from Plugin.hpp;
• Get the subgraphs that you need;
• Set the device and the runner for subgraphs;
• Compile the subgraphs.
• Check the shape;
• Transform the operations;
• Graph-level Optimizations.
Front-end
Middle-end
• Search for optimization opportunities;
• Operations Fusion;
• Graph Partition for different devices;
• Memory Allocation
• Low-level Optimizations;
• Tiling and Fusion;
• Code Generation.
Back-end
X24897-120920

When the model contains ops 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 ops and get better 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. In the beginning, the child subgraph is created for each operator and the plugin picks the operators which 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.

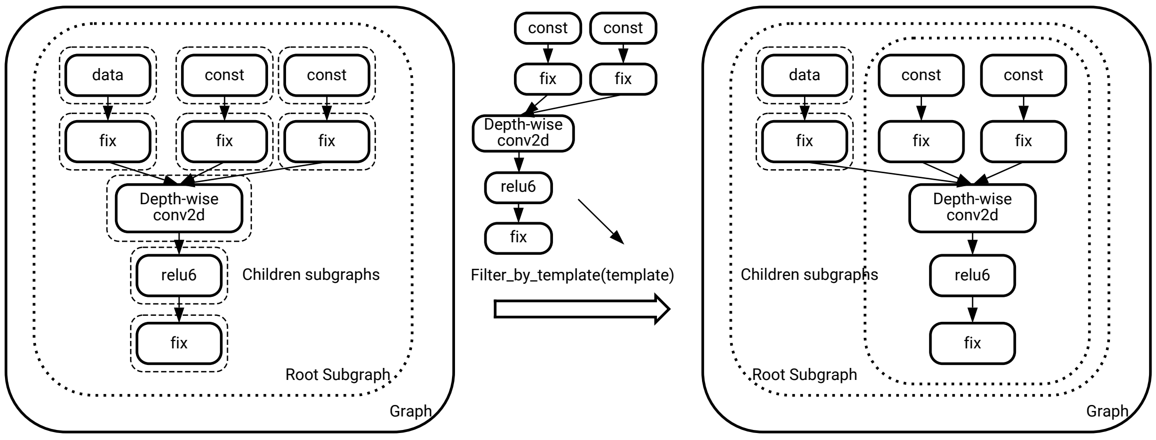
Implementing a Plugin

1. Implement `Plugin::partition()`

   In `std::set<xir::Subgraph*> partition(xir::Graph* graph)`, you should pick the desired ops and merge them into device level subgraphs. You can use 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 27: **Filter by Templates**



X24895-121520

- `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.

If you need to merge the children subgraphs that you get, use the helper function named `merge_subgraph()` to merge the children subgraphs. 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 do whatever you want here and attach information on subgraphs for runtime.

Building the Plugin

You need to 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

You can use `--options '{"plugin": "libplugin0.so,libplugin1.so"}'` in vai_c command line option to pass your plugin library to compiler. While executing your plugin, the compiler will open the library and make an instance of your plugin by loading your extern function named 'get_plugin'. If more than one plugins are specified, they would be executed sequentially in the order defined by command line option. Compilation for DPU and CPU would be executed after all plugins have been implemented.

Supported OPs and DPU Limitations

Currently Supported Operators

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.

In order to make DPU adaptable to a variety of FPGA devices, some kinds of DPU are configurable. You can choose necessary engines, adjust some intrinsic parameters and create your own DPU IP with TRD projects. But that means the limitations can be very different between configurations. You can find more information about how will those options impact on the limitations in PG338. Or it is recommended that you could try compiling the model with your own DPU configuration. The compiler will tell you which operators would be assigned to CPU and why they would be so. The table shows a specific configuration of each DPU architeciture.

Table 24: **Currently Supported Operators**

Typical Operation Type in CNN	**Parameters**	**DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)**	**DPUCAHX8L_ISA0 (U280)**	**DPUCAHX8H_ISA2 (U50LV9E, U50LV10E, U280), DPUCAHX8H_ISA2_ELP2 (U50)**	**DPUCVDX8G_ISA0_B8192C32B3 (VCK190)**	**DPUCVDX8H_ISA0 (VCK5000)**
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 16384**	**channel_parallel: 64** **bank_depth: 256**
conv2d	Kernel size	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16] w * h <= 64	w, h: [1, 16]
	Strides	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 4]	w, h: [1, 4]	w, h: [1, 4]
	Dilation	dilation * input_channel <= 256 * channel_parallel				
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w + 1]				
		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h + 1]				
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth				
	Out Size	output_channel <= 256 * channel_parallel				
	Activation	ReLU, LeakyReLU, ReLU6	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU
	Group* (Caffe)	group==1				
depthwise-conv2d	Kernel size	w, h: [1, 16]	w, h: [3]	Not supported		
	Strides	w, h: [1, 8]	w, h: [1, 2]			
	dilation	dilation * input_channel <= 256 * channel_parallel				
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w + 1]				
		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h + 1]				
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth				
	Out Size	output_channel <= 256 * channel_parallel				
	Activation	ReLU, ReLU6	ReLU, ReLU6			
	Group* (Caffe)	group==input_channel				

Table 24: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	**Parameters**	**DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)**	**DPUCAHX8L_ISA0 (U280)**	**DPUCAHX8H_ISA2 (U50LV9E, U50LV10E, U280), DPUCAHX8H_ISA2_ELP2 (U50)**	**DPUCVDX8G_ISA0_B8192C32B3 (VCK190)**	**DPUCVDX8H_ISA0 (VCK5000)**
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 16384**	**channel_parallel: 64** **bank_depth: 256**
transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]				
	Strides					
	Paddings	pad_left, pad_right: [1, kernel_w-1]				
		pad_top, pad_bottom: [1, kernel_h-1]				
	Out Size	output_channel <= 256 * channel_parallel				
	Activation	ReLU, LeakyReLU, ReLU6	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU
depthwise-transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]	kernel_w/stride_w, kernel_h/stride_h: [3]	Not supported		
	Strides					
	Paddings	pad_left, pad_right: [1, kernel_w-1]				
		pad_top, pad_bottom: [1, kernel_h-1]				
	Out Size	output_channel <= 256 * channel_parallel				
	Activation	ReLU, ReLU6	ReLU, ReLU6			
max-pooling	Kernel size	w, h: [2, 8]	w, h: {2, 3, 5, 7, 8}	w, h: [1, 8]	w, h: [2, 8]	w, h: {1, 2, 3, 7}
	Strides	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]
	Paddings	pad_left, pad_right: [1, kernel_w-1]				
		pad_top, pad_bottom: [1, kernel_h-1]				
	Activation	ReLU	not supported	ReLU	ReLU	not supported

Table 24: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	**Parameters**	**DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)**	**DPUCAHX8L_ISA0 (U280)**	**DPUCAHX8H_ISA2 (U50LV9E, U50LV10E, U280), DPUCAHX8H_ISA2_ELP2 (U50)**	**DPUCVDX8G_ISA0_B8192C32B3 (VCK190)**	**DPUCVDX8H_ISA0 (VCK5000)**
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 16384**	**channel_parallel: 64** **bank_depth: 256**
average-pooling	Kernel size	w, h: [2, 8] w==h	w, h: {2, 3, 5, 7, 8} w==h	w, h: [1, 8] w==h	w, h: [2, 8] w==h	w, h: {1, 2, 3, 7} w==h
	Strides	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]
	Paddings	pad_left, pad_right: [1, kernel_w-1]				
		pad_top, pad_bottom: [1, kernel_h-1]				
	Activation	ReLU	not support	ReLU	ReLU	not support
eltwise-sum	Input Channel	input_channel <= 256 * channel_parallel				
	Activation	ReLU	ReLU	ReLU	ReLU	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 would be fused into adjacent operators during compiler optimization process)				
global pooling	Global pooling will be processed as general pooling with kernel size euqal to input tensor size.					
InnerProduct, Fully Connected, Matmul	These ops will be transformed into conv2d op with kernel size equal to 1x1					

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.

Operators Supported by TensorFlow

Table 25: **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	datashapedata_type	Allocate memory for const data.
conv2d	filter	conv2d	kernel	Convolution Engine
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
	dilations		dilation	
depthwiseconv2dnative	filter	depthwise-conv2d	kernel	Depthwise-Convolution Engine
	strides		stride	
	explicit_paddings		padding	
	padding		pad_mode(SAME or VALID)	
	dilations		dilation	
conv2dbackpropinput / conv2dtranspose*	filter	transposed-conv2d	kernel	Convolution Engine
	strides		stride	
			padding([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 we set have been met.
	padding			
	filter		kernel	
	strides		stride	
	padding		pad_mode(SAME)	
	dilations		dilations	
	block_shape			
	crops			
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	

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

TensorFlow		XIR		DPU Implementations
OP type	Attributes	OP name	Attributes	
maxpool / maxpooling2d*	ksize	maxpool	kernel	Pooling Engine
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
avgpool / averagepooling2d* / globalavgeragepoolin g2d*	ksize	avgpool	kernel	Pooling Engine
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME or VALID)	
			count_include_pad (false)	
			count_include_invalid (true)	
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	leakyrelu	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.
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, we search for opportunities to fuse the add with adjacent operations such as convolutions.

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

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
concatv2 / concatenate*	axis	concat	axis	We reduce the overhead resulting from the concat by special reading or writing strategies and allocating the on-chip memory carefully.
pad / zeropadding2d*	paddings	pad	paddings	"CONSTANT" padding would be fused adjacent operations. "SYMMETRIC" padding would be mapped to DPU instructions. "REFLECT" padding is not supported by DPU yet.
	mode		mode	
shape		shape		The shape operation would be removed.
stridedslice	begin	stridedslice	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		
mul		mul		
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	

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

TensorFlow		XIR		DPU Implementations
OP type	**Attributes**	**OP name**	**Attributes**	
resizebilinear	size/scale	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/scale	resize	size	
	align_corners		align_corners	
	half_pixel_centers		half_pixel_centers	
			mode="NEAREST"	
upsample2d	size/scale	resize	size	
			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.
transpose	perm	transpose	order	
squeeze	axis	squeeze	axis	
exp		exp		They would only be compiled into CPU implementations.
softmax	axis	softmax	axis	
sigmoid		sigmoid		
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 Caffe

Table 26: **Operators Supported by Caffe**

Caffe		XIR		DPU Implementation
OP name	**Attributes**	**OP name**	**Attributes**	
input	shape	data	shape	Allocate memory for input data.
			data_type	

Table 26: **Operators Supported by Caffe** *(cont'd)*

Caffe		XIR		DPU Implementation
OP name	**Attributes**	**OP name**	**Attributes**	
convolution	kernel_size	conv2d (group = 1) / depthwise-conv2d (group = input channel)	kernel	If group == input channel, the convolution would be compiled into Depthwise-Convolution Engine, if group == 1, the convolution would be mapped to Convolution Engine. Otherwise, it would be mapped to CPU.
	stride		stride	
	pad		pad	
			pad_mode (FLOOR)	
	dilation		dilation	
	bias_term			
	num_output			
	group			
deconvolution	kernel_size	transposed-conv2d (group = 1) / depthwise-transposed-conv2d (group = input channel)	kernel	If group == input channel, the deconvolution would be compiled into Depthwise-Convolution Engine, if group == 1, the deconvolution would be mapped to Convolution Engine. Otherwise, it would be mapped to CPU
	stride		stride	
	pad		pad	
			pad_mode (FLOOR)	
	dilation		dilation	
	bias_term			
	num_output			
	group			
innerproduct	bias_term	conv2d / matmul	transpose_a	The inner-product would be transformed to matmul, then the matmul would be transformed to conv2d and compiled to Convolution Engine. If the inner-product fails to be transformed, it would be implemented by CPU.
	num_output		transpose_b	
scale	bias_term	depthwise-conv2d / scale		The scale would be transformed to depthwise-convolution, otherwise, it would be mapped to CPU.
pooling	kernel_size	maxpool2d (pool_method = 0) / avgpool2d (pool_method = 1)	kernel_size	Pooling Engine
	stride		stride	
	global_pooling		global	
	pad		pad	
	pool_method		pad_mode(CEIL)	
			count_include_pad (true)	
			count_include_invalid (false)	

Table 26: **Operators Supported by Caffe** *(cont'd)*

Caffe		XIR		DPU Implementation
OP name	**Attributes**	**OP name**	**Attributes**	
eltwise	coeff = 1	add		Element-wise Add Engine
	operation = SUM			
concat	axis	concat	axis	We reduce the overhead resulting from the concat by special reading or writing strategies and allocate the on-chip memory carefully.
relu	negative_slope	relu / leakyrelu	alpha	Activations would be fused to adjacent operations such as convolution, add, etc.
relu6		relu6		
fixneuron	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	
reshape	shape	reshape	shape	These operations are shape-related operations, they would be removed or transformed into reshape in most cases, which would not affect the on-chip data layout. Otherwise, they would be compiled to CPU.
permute	order	reshape / transpose	order	
flatten	axis	reshape / flatten	start_axis	
	end_axis		end_axis	
reorg	strides	reorg	strides	If the reorg meets the hardware requirements, it would be mapped to DPU implementations.
	reverse		reverse	

Table 26: **Operators Supported by Caffe** *(cont'd)*

Caffe		XIR		DPU Implementation
OP name	**Attributes**	**OP name**	**Attributes**	
deephiresize	scale	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.
	mode		mode	
			align_corners=false	
			half_pixel_centers=false	
gstiling	strides	gstiling	stride	If the strides of gstiling are integers, it may be mapped into special DPU read/write instructions.
	reverse		reverse	
slice	axis	strided_slice	begin	They would only be compiled into CPU implementations.
	slice_point		end	
			strides	
priorbox	min_sizes	priorbox	min_sizes	
	max_sizes		max_sizes	
	aspect_ratio		aspect_ratio	
	flip		flip	
	clip		clip	
	variance		variance	
	step		step	
	offset		offset	
softmax	axis	softmax	axis	

Operators Supported by PyTorch

Table 27: **Operators Supported by PyTorch**

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
Parameter	data	const	data	Allocate memory for input data.
			shape	
			data_type	

Table 27: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
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.
	out_channels			
	kernel_size		kernel	
	stride		stride	
	padding		pad	
	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	
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	

Table 27: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
ReLU		relu		Activations would be fused to adjacent operations such as convolution, add, etc.
LeakyReLU	negative_slope	leakyrelu	alpha	
ReLU6		relu6		
Hardtanh	min_val = 0			
	max_val = 6			
ConstantPad2d / ZeroPad2d	padding	pad	paddings	"CONSTANT" padding would be fused adjacent operations.
	value = 0		mode ("CONSTANT")	
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.
sub / rsub		sub		
mul		mul		
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 27: **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/save instrutions of adjacent operations to reduce the overhead. Otherwise, they would be mapped to CPU.
	dim1			
permute	dims			
view	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 / batchnorm	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		

Notes:

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

Compiling with DPUCADX8G

This section briefly describes the DPUCADX8G (formerly known as xfDNN) front-end compilers. Here, Caffe and TensorFlow interfaces are presented, both of which are built on top of a common intermediate representation. These interfaces are common to all DPUs.

This section also describes the procedure that is used in combination with examples (refer to software distribution), model quantization, and the proceeding sub-graph. As today, the compilers comes as open source and it provides further insights.

Only the necessary steps and some of the context are presented here to give familiarity with this new environment. It is assumed that your environment is set up and running, and that you are considering a network (such as a classification network) and want to see the instructions for generating it to run on a DPUCADX8G design.

If the final goal is to inspect FPGA codes and to infer a time estimate, the compiler can be used in isolation. If the final goal is to execute the network on an FPGA instance, the DPUCADX8G compiler must be used in combination with a partitioner. There are two tools for this purpose in the following chapters. One is for Caffe and the other is for TensorFlow. For Caffe, the partitioner can directly use the compiler outputs and feed the runtime because the computation is broken by the partitioner in a single FPGA subgraph. The TensorFlow partitioner allows multiple subgraphs.

Caffe

For presentation purposes, assume you have a MODEL (`model.prototxt`), WEIGHT (`model.caffemodel`), and a QUANT_INFO (quantization information file). The basic Caffe compiler interface comes with simplified help:

```
vai_c_caffe -help
**************************************************
 * VITIS_AI Compilation - Xilinx Inc.
 **************************************************
 usage: vai_c_caffe.py [-h] [-p PROTOTXT] [-c CAFFEMODEL] [-a ARCH]
                       [-o OUTPUT_DIR] [-n NET_NAME] [-e OPTIONS]optional
arguments:
   -h, --help            show this help message and exit
   -p PROTOTXT, --prototxt PROTOTXT
                         prototxt
   -c CAFFEMODEL, --caffemodel CAFFEMODEL
                         caffe-model
   -a ARCH, --arch ARCH  json file
   -o OUTPUT_DIR, --output_dir OUTPUT_DIR
                         output directory
   -n NET_NAME, --net_name NET_NAME
                         prefix-name for the outputs
   -e OPTIONS, --options OPTIONS
                         extra options
```

The main goal of this interface is to specify the bare minimum across different designs. The following describes how to run specifically for DPUCADX8G, starting with the minimum inputs.

```
vai_c_caffe.py -p MODEL -c WEIGHT -a vai/DPUCADX8G/tools/compile/arch.json -
o WORK -n cmd -e OPTIONS
```

Specify the MODEL, WEIGHT, a location to write the output, and a name for the code to be generated (i.e., cmd). This creates four outputs files in the WORK directory.

```
compiler.json  quantizer.json  weights.h5 meta.json
```

This is the main contract with the runtime. There are three JSON files: one has the information about the instruction to be executed, the other has information about the quantization (i.e., how to scale and shift). The `meta.json` file is created from the `arch.json` file and it is basically a dictionary that specifies run time information. The name cmd is necessary, but it is not used by the Vitis AI runtime.

The main difference with other versions of DPU is that you need to specify the QUANT_INF0 using the options:

```
-e "{'quant_cfgfile' : '/SOMEWHERE/quantize_info.txt'}"
```

The option field is a string that represents a Python dictionary. In this example, specify the location of the quantization file that has been computed separately and explained in Chapter 4: Quantizing the Model. In context, other DPU versions just build this information in either the model or the weight, therefore, enhanced models are not a vanilla Caffe model and you will need a custom Caffe to run them. The DPUCADX8G uses and executes the native Caffe (and the custom Caffe).

Note**:** Remember that the quantization file must be introduced. The compiler will ask to have one and eventually will crash when it looks for one. For a Caffe model to be complete, it must have both a prototxt and a caffemodel. Postpone the discussion about the `arch.json` file, but it is necessary. While this is the unified Caffe interface using a scripting format, there are Python interfaces that allow more tailored uses and compilations where an expert can optimize a model much further.

TensorFlow

The main difference between Caffe and TensorFlow is that the model is summarized by a single file and quantization information must be retrieved from a GraphDef.

```
**************************************************
* VITIS_AI Compilation - Xilinx Inc.
**************************************************
usage: vai_c_tensorflow.py [-h] [-f FROZEN_PB] [-a ARCH] [-o OUTPUT_DIR]
                           [-n NET_NAME] [-e OPTIONS] [-q]

optional arguments:
  -h, --help            show this help message and exit
  -f FROZEN_PB, --frozen_pb FROZEN_PB
                        prototxt
  -a ARCH, --arch ARCH  json file
  -o OUTPUT_DIR, --output_dir OUTPUT_DIR
                        output directory
  -n NET_NAME, --net_name NET_NAME
                        prefix-name for the outputs
  -e OPTIONS, --options OPTIONS
                        extra options
  -q, --quant_info      extract quant info
```

Now, the interface clearly explains how to specify the frozen graph. Assuming that the model and quantization information is required.

```
vai_c_tensorflow.py --frozen_pb deploy.pb --net_name cmd --options
"{'placeholdershape': {'input_tensor' : [1,224,224,3]},   'quant_cfgfile':
'fix_info.txt'}" --arch arch.json --output_dir work/temp
```

As you can see, the quantization information and the shape of the input placeholder are specified. It is common practice to have placeholder layers specifying the input of the model. It is good practice to specify all dimensions and use the number of batches equal to one. Optimize for latency and accept a batch size 1-4 (but this does not improve latency, it improves very little the throughput, and it is not completely tested for any networks).

There are cases where calibration and fine tuning provide a model that cannot be executed in native TensorFlow, but it contains the quantization information. If you run this front end with `[-q, --quant_info extract quant info ]` on, create quantization information.

The software repository should provide examples where the compiler is called twice. The first one is to create a quantization information file (using a default name and location) and this is used as input for the code generation.

***Note*:** Remember to introduce the output directory and the name of the code generated. The runtime contract is based on where the outputs are written. The main approach to call a different compiler for different architecture is through the `arch.json` file. This file is used as a template for the output description and as an internal feature of the platform/target FPGA design. Furthermore, there is also a Python interface where an expert could exploit custom optimizations.

VAI_C Usage

The corresponding Vitis AI compiler for Caffe and TensorFlow frameworks are `vai_c_caffe` and `vai_c_tensorflow` across cloud-to-edge DPU. The common options for VAI_C are illustrated in the following table.

Table 28: **VAI_C Common Options for Cloud and Edge DPU**

Parameters	Description
--arch	DPU architecture configuration file for VAI_C compiler in JSON format. It contains the dedicated options for cloud and edge DPU during compilation.

Table 28: **VAI_C Common Options for Cloud and Edge DPU** *(cont'd)*

Parameters	Description
--prototxt	Path of Caffe prototxt file for the compiler vai_c_caffe. This option is only required while compiling the quantized Caffe model generated by vai_q_caffe.
--caffemodel	Path of Caffe caffemodel file for the compiler vai_c_caffe. This option is only required while compiling the quantized Caffe model generated by vai_q_caffe.
--frozen_pb	Path of TensorFlow frozen protobuf file for the compiler vai_c_tensorflow. This option is only required the quantized TensorFlow model generated by vai_q_tensorflow.
--output_dir	Path of output directory of 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 for cloud or edge DPU in the format of 'key':'value'. If there are multiple options to be specified, they are separated by ',', and if the extra option has no value, an empty string must be provided. For example: --options "{'cpu_arch':'arm32', 'dcf':'/home/edge-dpu/zynq7020.dcf', 'save_kernel':''}" ***Note*:** For arguments specified with "--options", they have the highest priorities and will override the values specified in other places. For example, specifying 'dcf' with "--options" replaces the value specified in the JSON file.

Chapter 6

Deploying and Running the Model

Deploying and Running Models on Alveo U200/250

Vitis AI provides unified C++ and Python APIs for Edge and Cloud to deploy models on FPGAs.

For more information on the C++ APIs: https://github.com/Xilinx/Vitis-AI/blob/master/docs/DPUCADX8G/Vitis-C%2B%2BAPI.md.

For more information on the Python APIs: https://github.com/Xilinx/Vitis-AI/blob/master/docs/DPUCADX8G/Vitis-PythonAPI.md.

Programming with VART

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

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

Note: For historical reasons, this function is actually a blocking function, not an asynchronous non-blocking function.

1. 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.

   ```
   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.

   ```
   TensorFormat get_tensor_format()
   ```

2. Query the DpuRunner for the tensor format it expects.

Returns DpuRunner::TensorFormat::NCHW or DpuRunner::TensorFormat::NHWC

```
std::vector<Tensor*> get_input_tensors()
```

3. Query the DpuRunner for the shape and name of the output tensors it expects for its loaded Vitis AI model.

```
std::vector<Tensor*> get_output_tensors()
```

4. To create a DpuRunner object call the following:

```
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.*

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
```

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 chapter aims to demonstrate how to verify DPU inference result with VART tools. TensorFlow ResNet50, Caffe 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.

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.

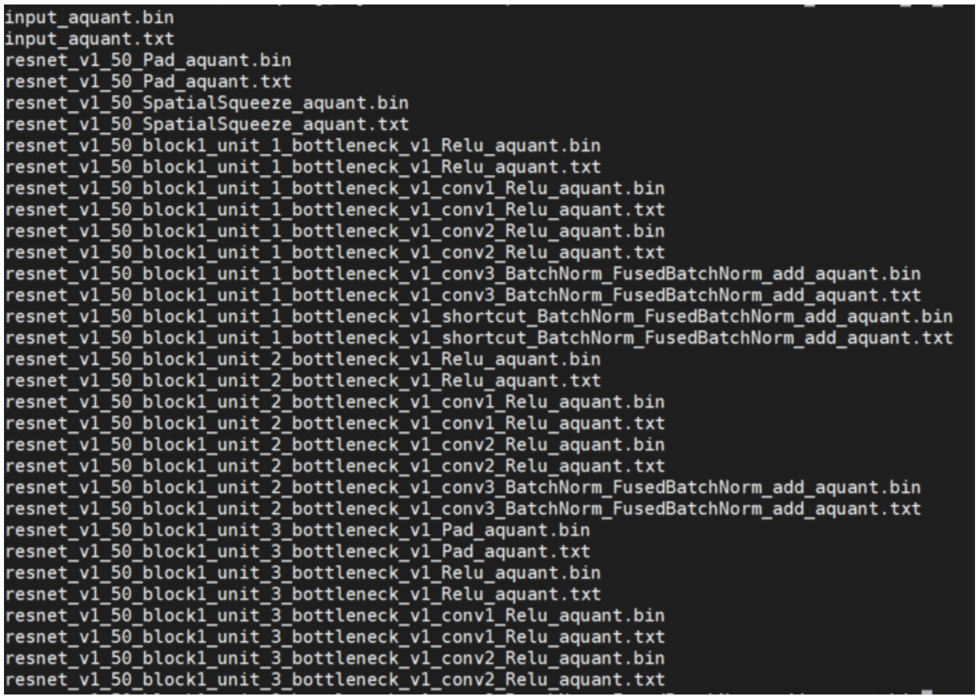
   The quantized model, `quantize_eval_model.pb`, is generated in the `quantize_model` folder.

   ```
   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.

```
vai_c_tensorflow --frozen_pb quantize_model/quantize_eval_model.pb \
  --arch /opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/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 \
   xilinx_test_dpu_runner ./compile_model/resnet_v1_50_tf.xmodel \
   ./dump_gpu/dump_results_0/input_aquant.bin                    \
    2>result.log 1>&2
```

For `xilinx_test_dpu_runner`, the usage is as follow:

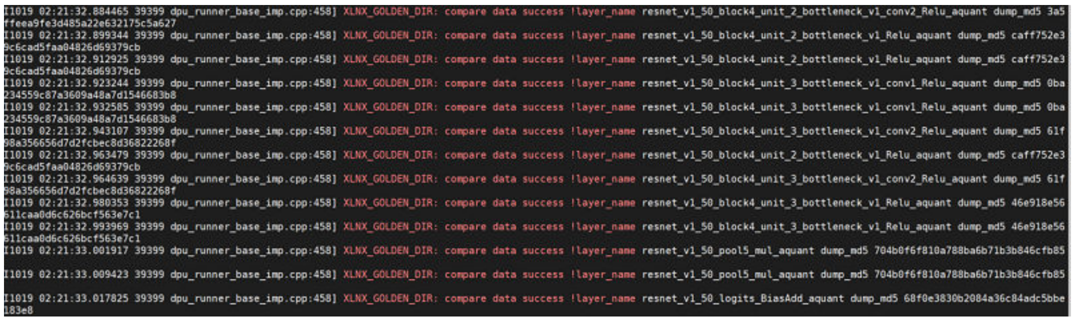
```
xilinx_test_dpu_runner  <model_file> <input_data>
```

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.

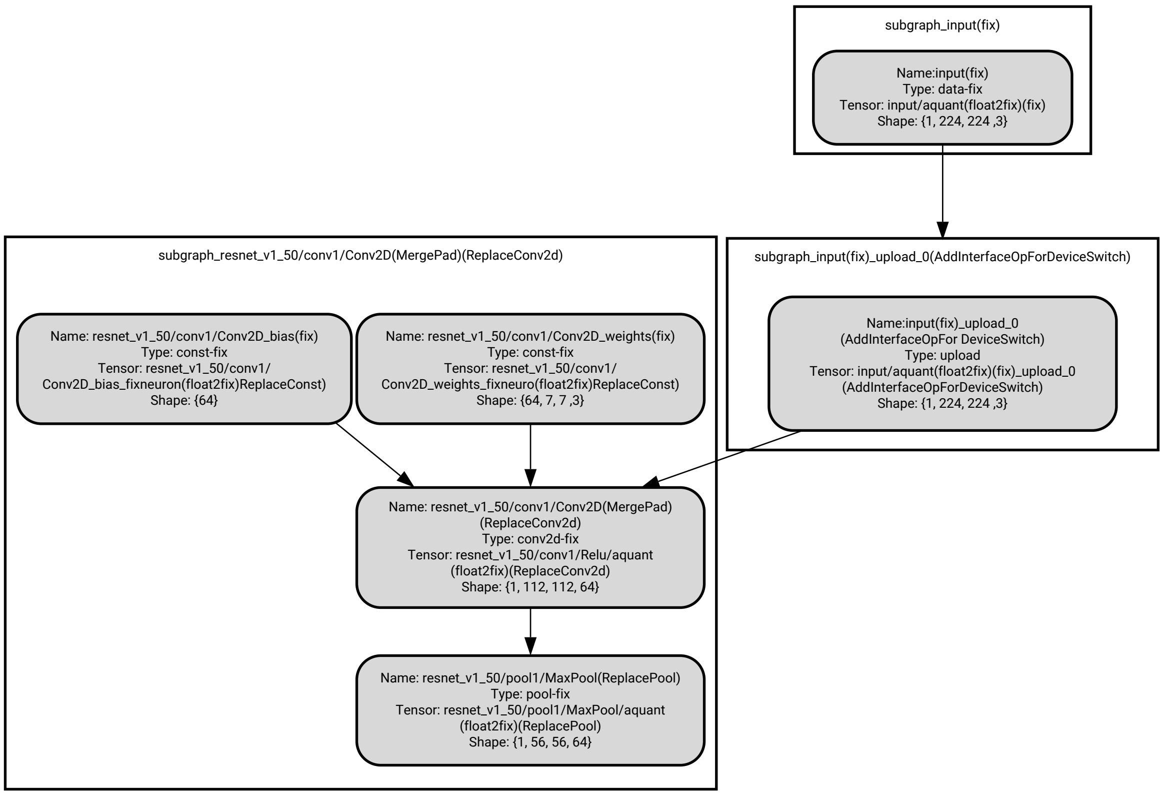
b. Use `xir` tool to generate a picture for displaying the network's structure.

```
Usage: xir svg <xmodel> <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 arround 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.


subgraph_input(fix)
Name:input(fix)
Type: data-fix
Tensor: input/aquant(float2fix)(fix)
Shape: {1, 224, 224 ,3}
subgraph_resnet_v1_50/conv1/Conv2D(MergePad)(ReplaceConv2d)
Name: resnet_v1_50/conv1/Conv2D_bias(fix)
Type: const-fix
Tensor: resnet_v1_50/conv1/
Conv2D_bias_fixneuron(float2fix)ReplaceConst)
Shape: {64}
Name: resnet_v1_50/conv1/Conv2D_weights(fix)
Type: const-fix
Tensor: resnet_v1_50/conv1/
Conv2D_weights_fixneuro(float2fix)ReplaceConst)
Shape: {64, 7, 7 ,3}
subgraph_input(fix)_upload_0(AddInterfaceOpForDeviceSwitch)
Name:input(fix)_upload_0
(AddInterfaceOpFor DeviceSwitch)
Type: upload
Tensor: input/aquant(float2fix)(fix)_upload_0
(AddInterfaceOpForDeviceSwitch)
Shape: {1, 224, 224 ,3}
Name: resnet_v1_50/conv1/Conv2D(MergePad)
(ReplaceConv2d)
Type: conv2d-fix
Tensor: resnet_v1_50/conv1/Relu/aquant
(float2fix)(ReplaceConv2d)
Shape: {1, 112, 112, 64}
Name: resnet_v1_50/pool1/MaxPool(ReplacePool)
Type: pool-fix
Tensor: resnet_v1_50/pool1/MaxPool/aquant
(float2fix)(ReplacePool)
Shape: {1, 56, 56, 64}

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.

Caffe 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.

```
vai_q_caffe quantize -model float/test_quantize.prototxt \
-weights float/trainval.caffemodel                       \
-output_dir quantize_model                               \
-keep_fixed_neuron                                       \
2>&1 | tee ./log/quantize.log
```

The following files are generated in the `quantize_model` folder.

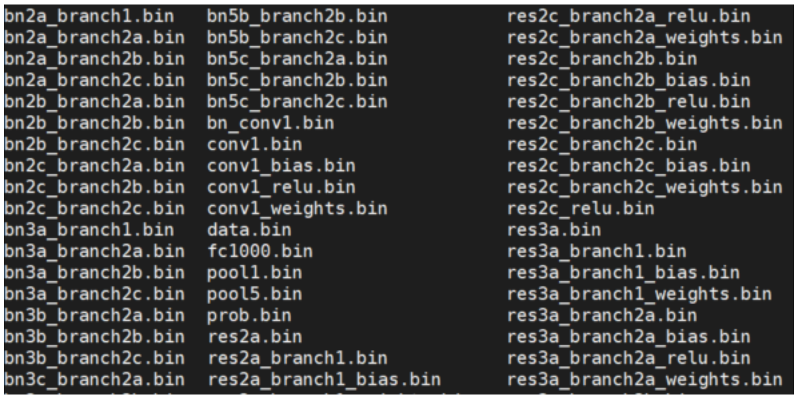
- `deploy.caffemodel`
- `deploy.prototxt`
- `quantize_train_test.caffemodel`

- `quantize_train_test.prototxt`

2. Generate the reference result by running the following command to generate reference data.

```
DECENT_DEBUG=5 vai_q_caffe test -model quantize_model/dump.prototxt \
-weights quantize_model/quantize_train_test.caffemodel              \
-test_iter 1                                                        \
2>&1 | tee ./log/dump.log
```

   This creates the `dump_gpu` folder and files as shown in the following figure.



```
bn2a_branch1.bin   bn5b_branch2b.bin        res2c_branch2a_relu.bin
bn2a_branch2a.bin  bn5b_branch2c.bin        res2c_branch2a_weights.bin
bn2a_branch2b.bin  bn5c_branch2a.bin        res2c_branch2b.bin
bn2a_branch2c.bin  bn5c_branch2b.bin        res2c_branch2b_bias.bin
bn2b_branch2a.bin  bn5c_branch2c.bin        res2c_branch2b_relu.bin
bn2b_branch2b.bin  bn_conv1.bin             res2c_branch2b_weights.bin
bn2b_branch2c.bin  conv1.bin                res2c_branch2c.bin
bn2c_branch2a.bin  conv1_bias.bin           res2c_branch2c_bias.bin
bn2c_branch2b.bin  conv1_relu.bin           res2c_branch2c_weights.bin
bn2c_branch2c.bin  conv1_weights.bin        res2c_relu.bin
bn3a_branch1.bin   data.bin                 res3a.bin
bn3a_branch2a.bin  fc1000.bin               res3a_branch1.bin
bn3a_branch2b.bin  pool1.bin                res3a_branch1_bias.bin
bn3a_branch2c.bin  pool5.bin                res3a_branch1_weights.bin
bn3b_branch2a.bin  prob.bin                 res3a_branch2a.bin
bn3b_branch2b.bin  res2a.bin                res3a_branch2a_bias.bin
bn3b_branch2c.bin  res2a_branch1.bin        res3a_branch2a_relu.bin
bn3c_branch2a.bin  res2a_branch1_bias.bin   res3a_branch2a_weights.bin
```

3. Generate the DPU xmodel by running the following command to generate DPU xmodel file.

```
vai_c_caffe --prototxt quantize_model/deploy.prototxt        \
--caffemodel quantize_model/deploy.caffemodel                \
--arch /opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json   \
--output_dir compile_model                                   \
--net_name resnet50
```

4. Generate the DPU inference result by running the following command to generate the DPU inference result.

```
env XLNX_ENABLE_DUMP=1  XLNX_ENABLE_DEBUG_MODE=1           \
    xilinx_test_dpu_runner ./compile_model/resnet50.xmodel \
    ./dump_gpu/data.bin 2>result.log 1>&2
```

   For `xilinx_test_dpu_runner`, the usage is as follow:

```
xilinx_test_dpu_runner  <model_file> <input_data>
```

   After the above command runs, the DPU inference result and the comparing result `result.log` are generated. The DPU inference results are under `dump` folder.

5. Crosscheck the reference result and the DPU inference result.

   The crosscheck mechanism is to first make sure input(s) to one layer is identical to reference and then the output(s) is identical too. This can be done with commands like `diff`, `vimdiff`, and `cmp`. If two files are identical, `diff` and `cmp` will return nothing in the command line.

   a. Check the input of DPU and GPU, make sure they use the same input data.

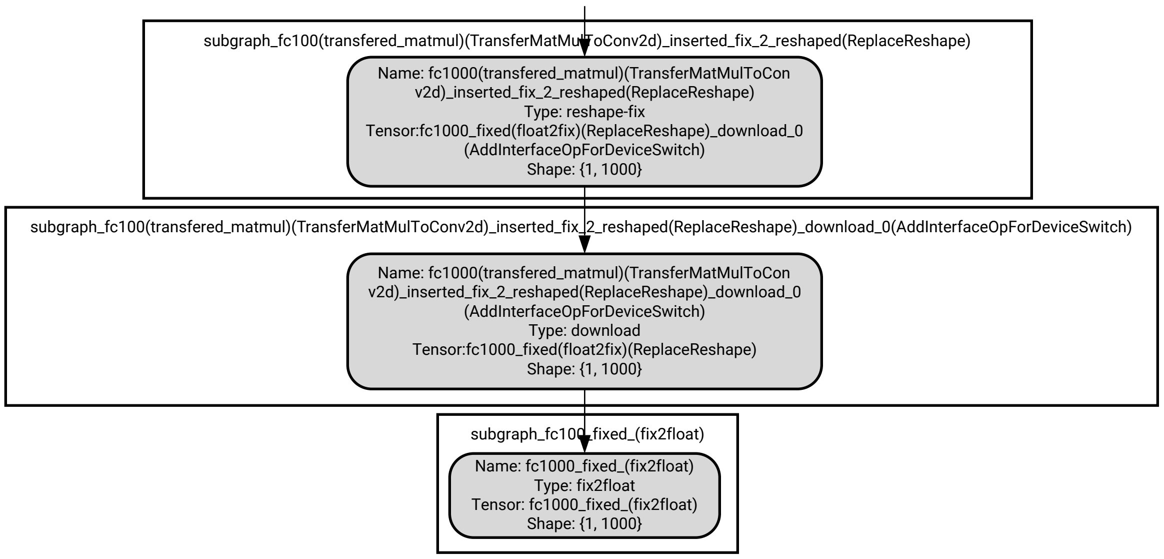
b. Use `xir` tool to generate a picture for displaying the network's structure.

```
Usage: xir svg <xmodel> <svg>
```

***Note*:** In Vitis AI docker environment, execute the following command to install the required library.

```
sudo apt-get install graphviz
```

The following figure is part of the ResNet50 model structure generated by `xir_cat`.


subgraph_fc100(transfered_matmul)(TransferMatMulToConv2d)_inserted_fix_2_reshaped(ReplaceReshape)
Name: fc1000(transfered_matmul)(TransferMatMulToCon v2d)_inserted_fix_2_reshaped(ReplaceReshape) Type: reshape-fix Tensor:fc1000_fixed(float2fix)(ReplaceReshape)_download_0 (AddInterfaceOpForDeviceSwitch) Shape: {1, 1000}
subgraph_fc100(transfered_matmul)(TransferMatMulToConv2d)_inserted_fix_2_reshaped(ReplaceReshape)_download_0(AddInterfaceOpForDeviceSwitch)
Name: fc1000(transfered_matmul)(TransferMatMulToCon v2d)_inserted_fix_2_reshaped(ReplaceReshape)_download_0 (AddInterfaceOpForDeviceSwitch) Type: download Tensor:fc1000_fixed(float2fix)(ReplaceReshape) Shape: {1, 1000}
subgraph_fc100_fixed_(fix2float)
Name: fc1000_fixed_(fix2float) Type: fix2float Tensor: fc1000_fixed_(fix2float) Shape: {1, 1000}
X24896-120920

c. View the xmodel structure image and find out the last layer name of the model.

RECOMMENDED: *Check the last layer first. If the crosscheck of the last layer is successful, then the whole layers' crosscheck will pass and there is no need crosscheck the other layers.*

For this model, the name of the last layer is `subgraph_fc1000_fixed_(fix2float)`.

i. Search the keyword `fc1000` under `dump_gpu` and `dump`. You will find the reference result file `fc1000.bin` under `dump_gpu` and DPU inference result `0.fc1000_inserted_fix_2.bin` under `dump/subgraph_fc1000/output/`.

ii. Diff the two files.

If the last layer's crosscheck fails, then you have to do the crosscheck from the first layer until you find the layer where the crosscheck fails.

***Note*:** For the layers that have multiple input or output (e.g., `res2a_branch1`), input correctness should be checked first and then check the output.

d. Submit the files to Xilinx if the DPU cross check fail.

If a certain layer proves to be wrong on the DPU, prepare the following files as one package for further analysis by factory and send it to Xilinx with a detailed description.

- Float model and prototxt file

- Quantized model, such as `deploy.caffemodel`, `deploy.prototxt`, `quantize_train_test.caffemodel`, and `quantize_train_test.prototxt`.

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. Generate the reference result by running the following command to generate reference data.

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

3. 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}
   ```

4. Generate the DPU inference result.

   This step is same as the step in Caffe workflow.

5. Crosscheck the reference result and the DPU inference result.

   This step is same as the step in Caffe workflow.

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.

Xbutler

The Xbutler tool manages and controls Xilinx FPGA resources on a machine. With the Vitis AI 1.0 release, installing Xbutler is mandatory for running a deep-learning solution using Xbutler. Xbutler is implemented as a server-client paradigm. Xbutler is an addon library on top of Xilinx XRT to facilitate multi-FPGA resource management. Xbutler is not a replacement to Xilinx XRT. The feature list for Xbutler is as follows:

- Enables multi-FPGA heterogeneous support
- C++/Python 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

Multi-FPGA, Multi-Graph Deployment with Vitis AI

Vitis AI provides different applications built using the Unified Runner APIs to deploy multiple models on single/multiple FPGAs. Detailed description and examples are available in the Vitis AI GitHub (Multi-Tenant Multi FPGA Deployment).

Xstream API

A typical end-to-end workflow involves heterogeneous compute nodes which include FPGA for accelerated services like ML, video, and database acceleration and CPUs for I/O with outside world and compute not implemented on FPGA. Vitis AI provides a set of APIs and functions to enable composition of streaming applications in Python. Xstream APIs build on top of the features provided by Xbutler. The components of Xstream API are as follows.

- **Xstream:** Xstream ($VAI_PYTHON_DIR`/vai/dpuv1/rt/xstream.py`) provides a standard mechanism for streaming data between multiple processes and controlling execution flow and dependencies.
- **Xstream Channel:** Channels are defined by an alphanumeric string. Xstream Nodes may publish payloads to channels and subscribe to channels to receive payloads. The default pattern is PUB-SUB, that is, all subscribers of a channel will receive all payloads published to that channel. Payloads are queued up on the subscriber side in FIFO order until the subscriber consumes them off the queue.
- **Xstream Payloads:** Payloads contain two items: a blob of binary data and metadata. The binary blob and metadata are transmitted using Redis, as an object store. The binary blob is meant for large data. The metadata is meant for smaller data like IDs, arguments and options. The object IDs are transmitted through ZMQ. ZMQ is used for stream flow control. The ID field is required in the metadata. An empty payload is used to signal the end of transmission.

- **Xstream Node:** Each Xstream Node is a stream processor. It is a separate process that can subscribe to zero or more input channels, and output to zero or more output channels. A node may perform computation on payload received on its input channel(s). The computation can be implemented in CPU, FPGA or GPU. To define a new node, add a new Python file in `vai/dpuv1/rt/xsnodes`. See `ping.py` as an example. Every node should loop forever upon construction. On each iteration of the loop, it should consume payloads from its input channel(s) and publish payloads to its output channel(s). If an empty payload is received, the node should forward the empty payload to its output channels by calling `xstream.end()` and exit.
- **Xstream Graph:** Use `$VAI_PYTHON_DIR/vai/dpuv1/rt/xsnodes/grapher.py` to construct a graph consisting of one or more nodes. When `Graph.serve()` is called, the graph spawns each node as a separate process and connect their input/output channels. The graph manages the life and death of all its nodes. See `neptune/services/ping.py` for a graph example. For example:

```
graph = grapher.Graph("my_graph")
  graph.node("prep", pre.ImagenetPreProcess, args)
  graph.node("fpga", fpga.FpgaProcess, args)
  graph.node("post", post.ImagenetPostProcess, args)

  graph.edge("START", None, "prep")
  graph.edge("fpga", "prep", "fpga")
  graph.edge("post", "fpga", "post")
  graph.edge("DONE", "post", None)

  graph.serve(background=True)
  ...
  graph.stop()
```

- **Xstream Runner:** The runner is a convenience class that pushes a payload to the input channel of a graph. The payload is submitted with a unique ID. The runner then waits for the output payload of the graph matching the submitted ID. The purpose of this runner is to provide the look-and-feel of a blocking function call. A complete standalone example of Xstream is here: `${VAI_ALVEO_ROOT}/ examples/deployment_modes/xs_classify.py`.

AI Kernel Scheduler

Real world deep learning applications involve multi-stage data processing pipelines which include many compute intensive pre-processing operations like data loading from disk, decoding, resizing, color space conversion, scaling, and croping multiple ML networks of different kinds like CNN, and various post-processing operations like NMS.

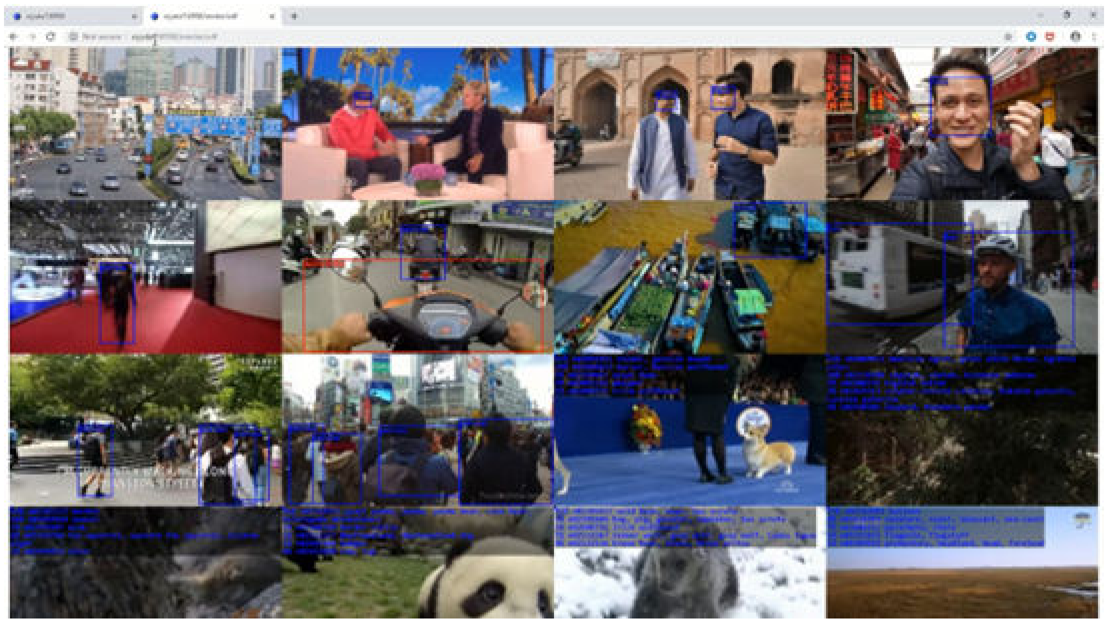
The AI kernel scheduler (AKS) is an application to automatically and efficiently pipeline such graphs without much effort from the users. It provides various kinds of kernels for every stage of the complex graphs which are plug and play and are highly configurable. For example, pre-processing kernels 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).

Neptune

Neptune provides a web server with a modular collection of nodes defined in Python. These nodes can be strung together in a graph to create a service. You can interact with the server to start and stop these services. You can extend Neptune by adding your own nodes and services. Neptune builds on top of the Xstream API. In the following picture, the user is running three different machine learning models on 16 videos from YouTube in real-time. Through a single Neptune server, the time and space multiplexing of the FPGA resources are enabled. Detailed documentation and examples can be found here: `${VAI_ALVEO_ROOT}/neptune`. Neptune is in the early access phase in this Vitis AI release.

Figure 28: **Multi-stream, Multi-network Processing in Alveo**



For more details see, Vitis AI GitHub (Neptune).

Apache TVM and Microsoft ONNX Runtime

In addition to VART and related APIs, Vitis AI has integrated with the Apache TVM and Microsoft ONNX Runtime 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.

TVM is currently supported on the following:

- DPUCADX8G
- DPUCZDX8G

ONNX Runtime is currently supported on the following:

- DPUCADX8G

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/master/external/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 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 ONNXRuntime integration on the Vitis AI GitHub repository: https://github.com/Xilinx/Vitis-AI/tree/master/external/onnxruntime.

Chapter 7

Profiling the Model

This chapter describes the utility tools included within the Vitis™ AI Development Kit, most of them are only available for the Edge DPU, except for the Vitis AI Profiler, which is a set of tools to profile and visualize AI applications based on the VART. The kit consists of five tools, which can be used for DPU execution debugging, performance profiling, DPU runtime mode manipulation, and DPU configuration file generation. With the combined use of these tools, you can conduct DPU debugging and performance profiling independently.

Vitis AI Profiler

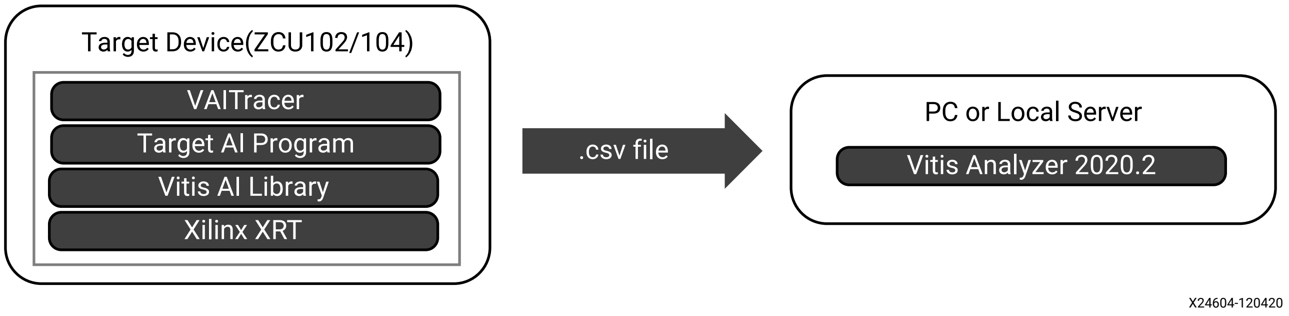
The Vitis AI Profiler is an all-in-one profiling solution for Vitis AI. It is an application level tool to profile and visualize AI applications based on VART. For an AI application, there are components that run on the hardware, for example, neural network computation usually runs on the DPU, and there 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.

- 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/GPU).

Vitis AI Profiler Architecture

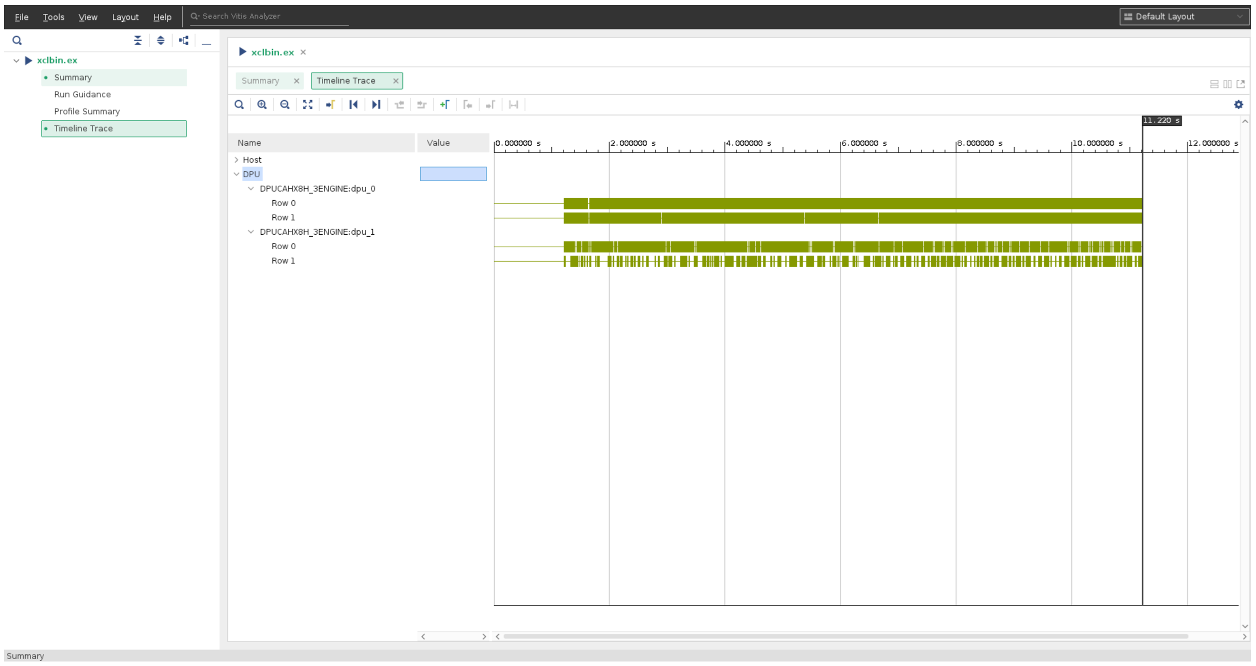
The Vitis AI Profiler architecture is shown in the following figure:

Figure 29: **Vitis AI Profiler Architecture**

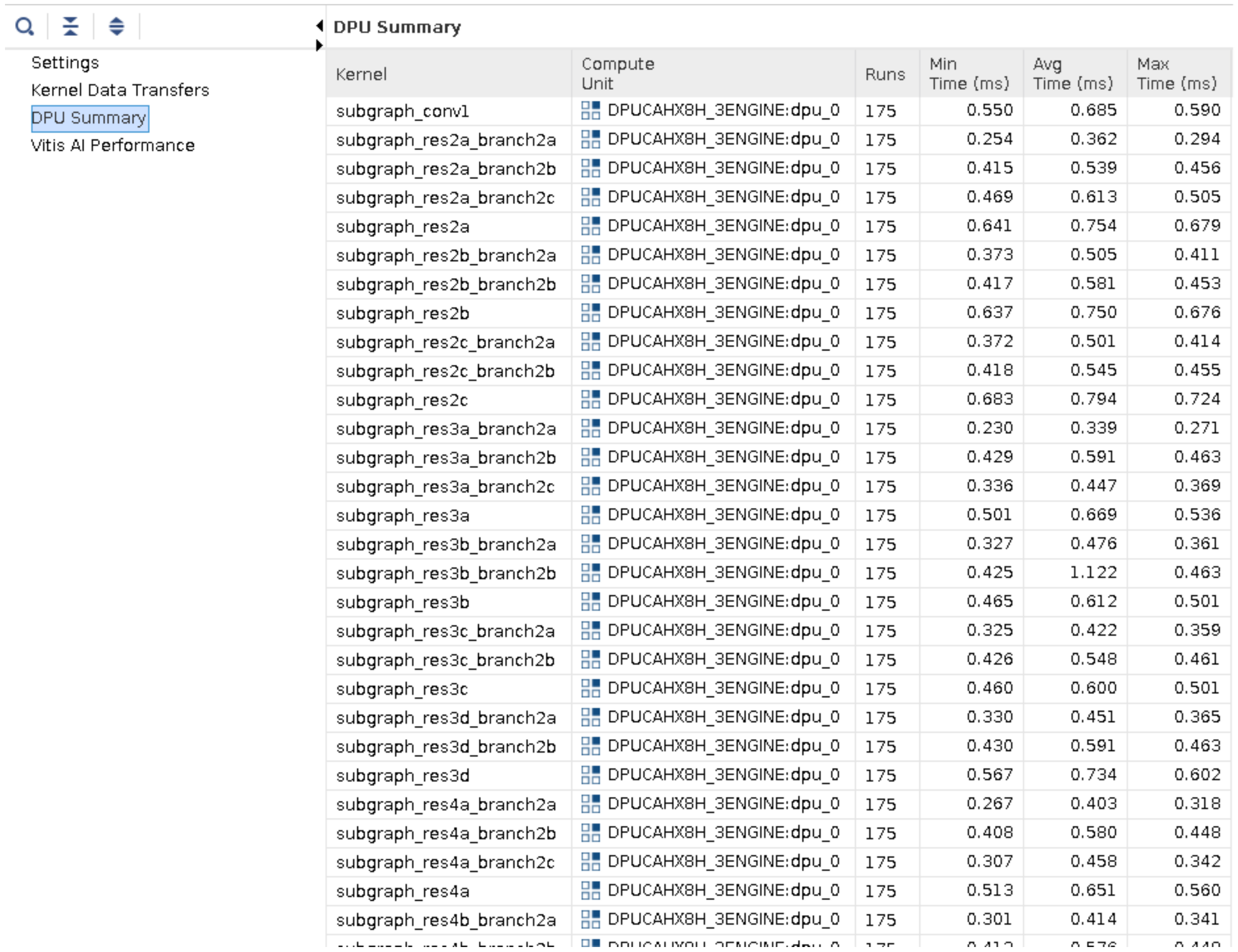


Vitis AI Profiler GUI Overview

Figure 30: **Vitis AI Profiler GUI Overview**



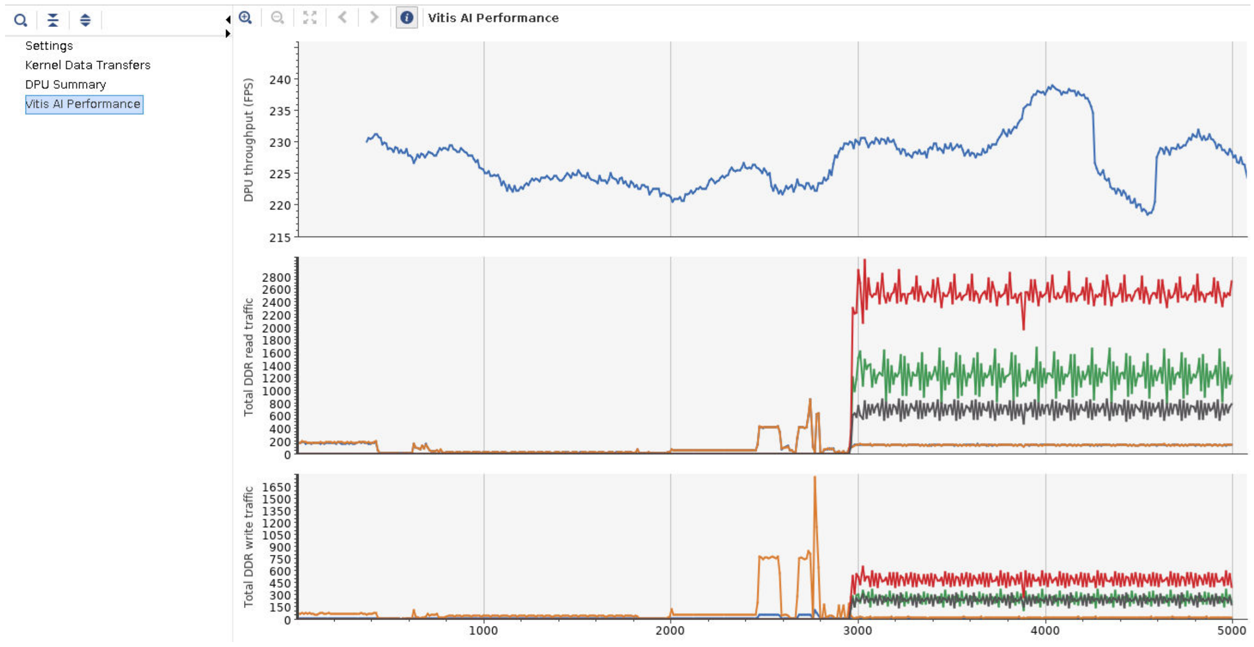
- **DPU Summary:** A table of the number of runs and minimum/average/maximum times (ms) for each kernel.



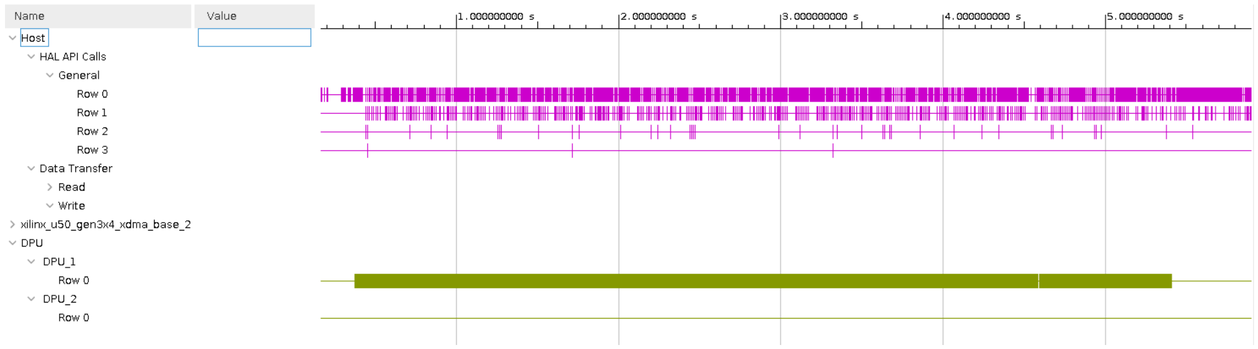
DPU Summary

Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)
subgraph_conv1	DPUCAHX8H_3ENGINE:dpu_0	175	0.550	0.685	0.590
subgraph_res2a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.254	0.362	0.294
subgraph_res2a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.415	0.539	0.456
subgraph_res2a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.469	0.613	0.505
subgraph_res2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.641	0.754	0.679
subgraph_res2b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.373	0.505	0.411
subgraph_res2b_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.417	0.581	0.453
subgraph_res2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.637	0.750	0.676
subgraph_res2c_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.372	0.501	0.414
subgraph_res2c_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.418	0.545	0.455
subgraph_res2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.683	0.794	0.724
subgraph_res3a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.230	0.339	0.271
subgraph_res3a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.429	0.591	0.463
subgraph_res3a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.336	0.447	0.369
subgraph_res3a	DPUCAHX8H_3ENGINE:dpu_0	175	0.501	0.669	0.536
subgraph_res3b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.327	0.476	0.361
subgraph_res3b_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.425	1.122	0.463
subgraph_res3b	DPUCAHX8H_3ENGINE:dpu_0	175	0.465	0.612	0.501
subgraph_res3c_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.325	0.422	0.359
subgraph_res3c_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.426	0.548	0.461
subgraph_res3c	DPUCAHX8H_3ENGINE:dpu_0	175	0.460	0.600	0.501
subgraph_res3d_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.330	0.451	0.365
subgraph_res3d_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.430	0.591	0.463
subgraph_res3d	DPUCAHX8H_3ENGINE:dpu_0	175	0.567	0.734	0.602
subgraph_res4a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.267	0.403	0.318
subgraph_res4a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.408	0.580	0.448
subgraph_res4a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.307	0.458	0.342
subgraph_res4a	DPUCAHX8H_3ENGINE:dpu_0	175	0.513	0.651	0.560
subgraph_res4b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.301	0.414	0.341

- **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*:**

1. The Vitis Analyzer is the default GUI for vaitrace for Vitis AI 1.3 and later releases.
2. The legacy web based Vitis-AI Profiler works for Edge devices (Zynq UltraScale+ MPSoC) in Vitis AI 1.3. For more information, see Vitis-AI Profiler v1.2 README.

Getting Started with Vitis AI Profiler

System Requirements

- **Hardware:**
  - Support Zynq UltraScale+ MPSoC (DPUCZD series)
  - Support Alveo accelerator cards (DPUCAH series)
- **Software:**
  - Support VART v1.3+

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.

   user-packages ---> modules ---> [*] 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/VART/samples/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/VART/samples/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.

VAI Trace Usage

Command Line Usage

```
# vaitrace --help
usage: Xilinx Vitis AI Trace [-h] [-c [CONFIG]] [-d] [-o [TRACESAVETO]] [-t
[TIMEOUT]] [-v]

  cmd                Command to be traced
  -b                   Bypass mode, just run command and by pass vaitrace,
for debug use
  -c [CONFIG]          Specify the configuration file
  -o [TRACESAVETO]     Save trace file to
```

```
  -t [TIMEOUT]        Tracing time limitation, default value is 30 for vitis
analyzer format, and 5 for .xat format
  --va                Generate trace data for Vitis Analyzer
  --xat               Generate trace data in .xat, for the legacy web based
Vitis-AI Profiler, only available for Zynq MPSoC devices
```

Following are some important and frequently-used arguments:

- **cmd:** cmd is your executable program of Vitis AI that want to be traced.
- **-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. It is recommended that trace is about 50~100 images at once because less then 50 may not be enough for some statistic information and more then 100 will slow down the system significantly.
- **-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.

Others arguments are used for debug.

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.

```
{
  "options”: {
      "runmode": "normal",
      "cmd": "/usr/share/vitis-ai-library/sample/classification/
test_jpeg_classification resnet50 sample.jpg",
      "output": "./trace_resnet50.xat",
      "timeout": 3
  },
  "trace": {
      "enable_trace_list": ["vitis-ai-library", "vart", "custom"],
      "trace_custom": []
  }
}
```

Table 29: **Contents of the Configuration File**

Key Name	Value Type	Description
options	object	Vaitrace options

Table 29: **Contents of the Configuration File** *(cont'd)*

Key Name		Value Type	Description
	cmd	string	the same with command line argument cmd
	output	string	the same with command line argument -o
	timeout	integer	the same with command line argument -t
	runmode	string	Xmodel run mode control, can be “debug” or “normal”, if runmode == “debug” VART will control xmodel run in a debug mode by using this, user can achieve fine-grained profiling for xmodel.
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

DPU Profiling Examples

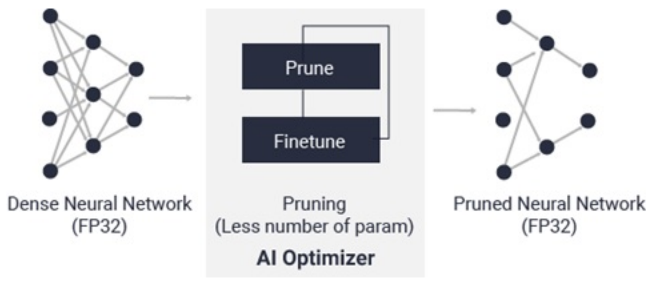
You can find more advanced DPU Profiling examples with Vitis-AI Profiler on the Vitis-AI-Profiler GitHub page.

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 31: **Vitis AI Optimizer**


Prune
Finetune
Dense Neural Network (FP32)
Pruning (Less number of param)
AI Optimizer
Pruned Neural Network (FP32)

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 9

Accelerating Subgraph with ML Frameworks

Partitioning is the process of splitting the inference execution of a model between the FPGA and the host. Partitioning is necessary to execute models that contain layers unsupported by the FPGA. Partitioning can also be useful for debugging and exploring different computation graph partitioning and execution to meet a target objective.

***Note*:** This feature is currently only available for Alveo™ U200/U250 with use of DPUCADX8G.

Partitioning Functional API Call in TensorFlow

Graph partitioning has the following general flow:

1. Create/initialize the partition class:

   ```
   from vai.dpuv1.rt.xdnn_rt_tf import TFxdnnRT
   xdnnTF = TFxdnnRT(args)
   ```

2. Loading the partitioned graph:

   ```
   graph = xdnnTF.load_partitioned_graph()
   ```

3. Apply preprocessing and post processing as if the original graph is loaded.

Partitioner API

The main input argument (for example, args in item 1 from Partitioning usage flow) of the partitioner are as follows:

- **Networkfile:** tf.Graph, tf.GraphDef, or path to the network file
- **loadmode:** Saving protocol of the network file. Supported formats [pb (default), chkpt, txt, savedmodel]
- **quant_cfgfile:** DPUCADX8G quantization file

- **batch_sz:** Inference batch size. The default value for this is one.
- **startnode:** List of start nodes for FPGA partition (optional. Defaults to all placeholders)
- **finalnode:** List of final nodes for FPGA partition (optional. Defaults to all sink nodes)

Partitioning Steps

1. Loading the original graph

   Partitioner can handle frozen tf.Graph, tf.GraphDef, or a path to the network file/folder. If the pb file is provided the graph should be properly frozen. Other options include model stores using tf.train.Saver and tf.saved_model.

2. Partitioning

   In this step the subgraph specified by startnode and finalnode sets is analyzed for FPGA acceleration. This is done in multiple phases.

   a. All graph nodes get partitioned into (FPGA) supported and unsupported sets using one of two method. The default (compilerFunc='SPECULATIVE') method uses rough estimate of the hardware operation tree. The second method (compilerFunc= 'DEFINITIVE') utilizes the hardware compiler. The latter is more accurate and can handle complex optimization schemes based on the specified options, however, it takes considerable more time to conclude the process.

   b. Adjacent supported and unsupported nodes get merged into (fine grained) connected components.

   c. Supported partitions get merged into maximally connected components, while maintaining the DAG property.

   d. Each supported partition gets (re)compiled using hardware compiler to create runtime code, quantization info, and relevant model parameters.

   e. Each supported partition subgraph is stored for visualization and debug purposes.

   f. Each supported subgraph gets replaced by tf.py_func node (with naming convention fpga_func_<partition_id>) that contains all necessary python function calls to accelerate that subgraph over FPGA.

3. Freezing the modified graph

   The modified graph gets frozen and stored with "-fpga" suffix.

4. Run natively in TensorFlow

   The modified graph can be loaded using load_partitioned_graph method of the partitioner class. The modified graph replaces the default TensorFlow graph and can be used similar to the original graph.

Practical Notes

The compiler optimizations can be modified by passing the applicable compiler arguments either through positional argument or options arguments to the Partitioner class TFxdnnRT. If model is not properly frozen, the compiler might fail optimizing some operations such as batchnorm.

startnode, and finalnode sets should be a vertex separators. This means that the removal of startnode or finalnode should separate the graph into two distinct connected components (except when startnode is a subset of graph placeholders).

Wherever possible, do not specify cut nodes between layers that are executed as a single macro layers, e.g., for Conv(x) -> BiasAdd(x), placing Conv(x) in a different FPGA partition than BiasAdd(x) may result in suboptimal performance (throughput, latency, and accuracy).

The partitioner initialization requires quant_cfgfile to exist to be able to create executable code for FPGA. In case FPGA execution is not intended, this requirement can be circumvented by setting `quant_cfgfile=”IGNORE”`.

Partitioning Support in Caffe

Xilinx has enhanced Caffe package to automatically partition a Caffe graph. This function separates the FPGA executable layers in the network and generates a new prototxt, which is used for the inference. The subgraph cutter creates a custom python layer to be accelerated on the FPGA. The following code snippet explains the code:

```
from vai.dpuv1.rt.scripts.framework.caffe.xfdnn_subgraph \
    import CaffeCutter as xfdnnCutter
def Cut(prototxt):

    cutter = xfdnnCutter(
        inproto="quantize_results/deploy.prototxt",
        trainproto=prototxt,
        outproto="xfdnn_auto_cut_deploy.prototxt",
        outtrainproto="xfdnn_auto_cut_train_val.prototxt",
        cutAfter="data",
        xclbin=XCLBIN,
        netcfg="work/compiler.json",
        quantizecfg="work/quantizer.json",
        weights="work/deploy.caffemodel_data.h5"
    )
    cutter.cut()
#cutting and generating a partitioned graph auto_cut_deploy.prototxt
Cut(prototxt)
```

Cut(prototxt)

The `auto_cut_deploy.prototxt` generated in the previous step, has complete information to run inference. For example:

- **Notebook execution:** There are two example notebooks (image detection and image classification) that can be accessed from `$VAI_ALVEO_ROOT/notebooks` to understand these steps in detail.
- **Script execution:** There is a python script that can be used to run the models with default settings. It can be run using the following commands:
  - **PreparePhase:** Python

    ```
    $VAI_ALVEO_ROOT/examples/caffe/run.py --prototxt <example prototxt> --
    caffemodel <example caffemodel> --prepare
    ```

    - **prototxt:** Path to the prototxt of the model
    - **caffemodel:** Path to the caffemodel of the model
    - **output_dir:** Path to save the quantization, compiler and subgraph_cut files
    - **qtest_iter:** Number of iterations to test the quantization
    - **qcalib_iter:** Number of iterations to calibration used for quantization
- **Validate Phase:** Python

  ```
  $VAI_ALVEO_ROOT/examples/caffe/run.py -validate
  ```

  - **output_dir:** If output_dir is given in the prepare phase, give the same argument and value to use the files generated in prepare phase.
  - **numBatches:** Number of batches which can be used to test the inference.

Chapter 10

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 which 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 in which you can configure the DPU with different parameters to meet the performance and resource utilization requirements. For more details, see *Zynq DPU v3.1 IP Product Guide* (PG338) and Vitis DPU 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. 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 Vivado® tools for DPU integration, see the DPU TRD Vivado flow.

If you want to integrate the DPU into Alveo™ accelerator cards and other non-embedded platforms, contact xilinx_ai_prod_mkt@xilinx.com.

Appendix A

Vitis AI Programming Interface

This appendix describes all the programming APIs offered by the VART programming interface.

VART APIs

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 30: **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	

Class 2

The class name is `vart::TensorBuffer`. The following table lists all the functions defined in the `vart::TensorBuffer` class.

Table 31: **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 32: **Quick Function Reference**

Return Type	Name	Arguments
std::vector<vart::TensorBuffer*>	get_inputs	
std::vector<vart::TensorBuffer*>	get_outputs	

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 33: **create_runner Arguments**

Type	Name	Description
const xir::Subgraph*	subgraph	XIR Subgraph

Table 33: **create_runner Arguments** *(cont'd)*

Type	Name	Description
const std::string&	mode	3 mode supported: 'ref' - CPU runner 'sim' - Simulation 'run' - DPU runner

Returns

An instance of CPU/SIM/DPU runner.

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 34: **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 block 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 35: **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. This is a block function.

Prototype

```
int wait(int jobid, int timeout)
```

Parameters

The following table lists the `wait` function arguments.

Table 36: **wait Arguments**

Type	Name	Description
int	jobid	job id, neg for any id, others for specific job id
int	timeout	timeout, neg for block for ever, 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

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.

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.

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 37: **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	

get_tensor

Get Tensor of TensorBuffer.

Prototype

```
const xir::Tensor* get_tensor()
```

Parameters

None.

Returns

A pointer to the Tensor.

data

Get the data address of the index and the left accessible data size.

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 38: **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 left accessible data size in byte unit.

data_phy

Get the data physical address of the index and the left accessible data size.

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 39: **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 left accessible data size in byte unit.

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 40: **sync_for_read Arguments**

Type	Name	Description
uint64_t	offset	The start offset address
size_t	size	The data size

Returns

None.

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 41: **sync_for_write Arguments**

Type	Name	Description
uint64_t	offset	The start offset address
size_t	size	The data size

Returns

None.

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 42: **copy_from_host Arguments**

Type	Name	Description
size_t	batch_idx	The batch index
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.

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 43: **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.

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 44: **copy_tensor_buffer Arguments**

Type	Name	Description
vart::TensorBuffer*	tb_from	The source TensorBuffer
vart::TensorBuffer*	tb_to	The destination TensorBuffer

Returns

None.

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.

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.

Python APIs

Class 1

The class name is `vart.Runner`. The following table lists all the functions defined in the `vart.Runner` class.

Table 45: **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

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 46: **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	

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 47: **create_runner Arguments**

Type	Name	Description
xir.Subgraph	subgraph	XIR Subgraph
String	mode	'run' - DPU runner

Returns

An instance of DPU runner.

execute_async

Executes the runner. This is a block 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 48: **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. This is a block function.

Prototype

```
int wait(tuple[uint32_t, int] jobid)
```

Parameters

The following table lists the `wait` function arguments.

Table 49: **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

All input tensors. A vector of raw pointer to the input tensor.

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.

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.

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.

Legacy DNNDK

The Deep Neural Network Development Kit (DNNDK) is a full-stack deep learning SDK for the Deep-learning Processor Unit (DPU). It provides a unified solution for deep neural network inference applications by providing pruning, quantization, compilation, optimization, and run-time support.

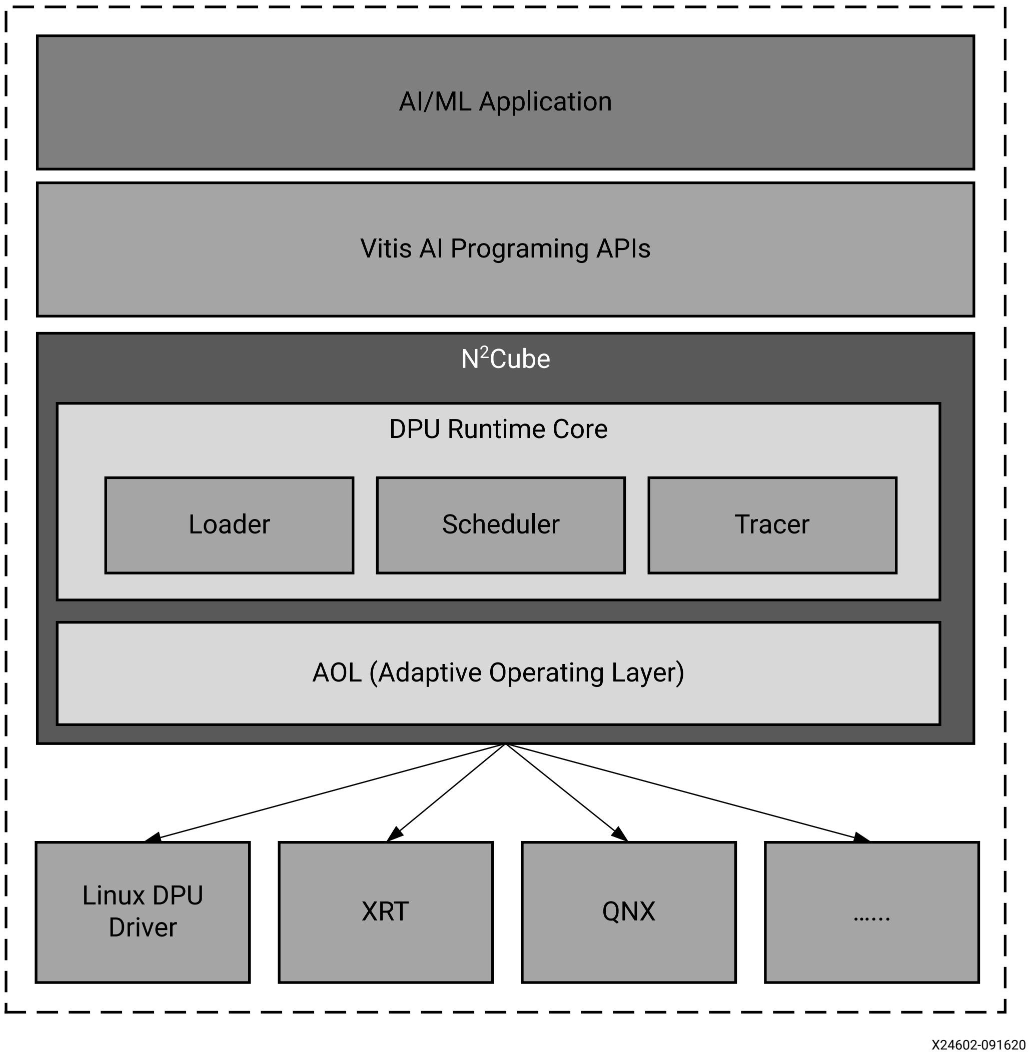
DNNDK N2Cube Runtime

For the Edge DPUCZDX8G, legacy DNNDK runtime framework (called N2Cube) is shown in the following figure. For Vitis AI release, N2Cube is based on the XRT. For legacy Vivado® based DPU, it interacts with the underlying Linux DPU driver (instead of XRT) for DPU scheduling and resource management.

Starting with Vitis AI v1.2 release, N2Cube is now available as open source. More details are available in the `tools/Vitis-AI-Runtime/DNNDK` directory in the Vitis AI repository.

***Note*:** N2Cube will be deprecated in future releases. For new applications or projects, use the VART instead.

Figure 32: **Legacy MPSoC Runtime Stack**



N2Cube offers a comprehensive C++/Python programming interface to flexibly meet the diverse requirements for edge scenarios. Refer to DNNDK Programming APIs for more details about edge DPU advanced programming. The highlights for N2Cube are listed as follows:

- Supports multi-threading and multi-process DPU application deployment.
- Supports multiple models running in parallel and zero-overhead dynamic switching at run time.
- Automated DPU multi-core scheduling for better workload balancing.
- Optional flexibility to dynamically specify DPU core affinity over DPU tasks at run time.
- Priority based DPU task scheduling while adhering to DPU cores affinity.
- Optimized memory usage through DPU code and parameter sharing within multi-threaded DPU application.

- Easily adapts to any POSIX-compliant OS or Real-Time Operating System (RTOS) environment, such as QNX, VxWorks, and Integrity.
- Ease-of-use capabilities for DPU debugging and performance profiling.

Currently, N2Cube officially supports three operating environments, including Linux, Xilinx XRT, and BlackBerry QNX RTOS. You can contact the Xilinx representatives to acquire the Vitis AI package for QNX or to port N2Cube to other third party RTOS. As the source code is accessible, you can freely port N2Cube to any other environment.

DNNDK Examples

To maintain forward compatibility, Vitis AI still supports the application of DNNDK for deep learning applications development over edge DPUCZDX8G. The legacy DNNDK C++/Python examples for ZCU102 and ZCU104 are available in https://github.com/Xilinx/Vitis-AI/tree/master/demo/DNNDK. You can follow the guidelines in https://github.com/Xilinx/Vitis-AI/tree/master/demo/DNNDK/README.md to set up the environment and evaluate these samples.

***Note*:** The DNNDK runtime loads DPU overlay bin from the default directory `/usr/lib/`. Make sure that `dpu.xclbin` exists under `/usr/lib/` as expected before running DNNDK examples. For the downloaded ZCU102 or ZCU104 system images, `dpu.xclbin` is copied to `/usr/lib/` by default. For the customized image, it is up to you to copy `dpu.xclbin` manually.

The following table briefly describes all the available DNNDK examples.

Table 50: **DNNDK Examples**

Example Name	Models	Framework	Notes
resnet50	ResNet50	Caffe	Image classification with Vitis AI advanced C++ APIs.
resnet50_mt	ResNet50	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
tf_resnet50	ResNet50	TensorFlow	Image classification with Vitis AI advanced Python APIs.
mini_resnet_py	Mini-ResNet	TensorFlow	Image classification with Vitis AI advanced Python APIs.
inception_v1	Inception-v1	Caffe	Image classification with Vitis AI advanced C++ APIs.
inception_v1_mt	Inception-v1	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
inception_v1_mt_py	Inception-v1	Caffe	Multi-threading image classification with Vitis AI advanced Python APIs.

Table 50: **DNNDK Examples** *(cont'd)*

Example Name	Models	Framework	Notes
mobilenet	MiblieNet	Caffe	Image classification with Vitis AI advanced C++ APIs.
mobilenet_mt	MobileNet	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
face_detection	DenseBox	Caffe	Face detection with Vitis AI advanced C++ APIs.
pose_detection	SSD, Pose detection	Caffe	Pose detection with Vitis AI advanced C++ APIs.
video_analysis	SSD	Caffe	Traffic detection with Vitis AI advanced C++ APIs.
adas_detection	YOLO-v3	Caffe	ADAS detection with Vitis AI advanced C++ APIs.
segmentation	FPN	Caffe	Semantic segmentation with Vitis AI advanced C++ APIs.
split_io	SSD	TensorFlow	DPU split I/O memory model programming with Vitis AI advanced C++ APIs.
debugging	Inception-v1	TensorFlow	DPU debugging with Vitis AI advanced C++ APIs.
tf_yolov3_voc_py	YOLO-v3	TensorFlow	Object detection with Vitis AI advanced Python APIs.

You must follow the descriptions in the following table to prepare several images before running the samples on the evaluation boards.

Table 51: **Image Preparation for DNNDK Samples**

Image Directory	Note
vitis_ai_dnndk_samples/dataset/image500_640_480/	Download several images from the ImageNet dataset and scale to the same resolution 640*480.
vitis_ai_dnndk_samples2/ image_224_224/	Download one image from the ImageNet dataset and scale to resolution 224*224.
vitis_ai_dnndk_samples/ image_32_32/	Download several images from the CIFAR-10 dataset https://www.cs.toronto.edu/~kriz/cifar.html.
vitis_ai_dnndk_samples/resnet50_mt/image/	Download one image from the ImageNet dataset.
vitis_ai_dnndk_samples/ mobilenet_mt/image/	Download one image from the ImageNet dataset.
vitis_ai_dnndk_samples/ inception_v1_mt/image/	Download one image from the ImageNet dataset.
vitis_ai_dnndk_samples/ debugging/decent_golden/dataset/ images/	Download one image from the ImageNet dataset and save it as cropped_224x224.jpg.
vitis_ai_dnndk_samples/ tf_yolov3_voc_py/image/	Download one image from the VOC dataset http://host.robots.ox.ac.uk/pascal/VOC/ and save it as input.jpg.

The following section illustrates how to run DNDNK examples using the ZCU102 board as the reference. Suppose the samples are located in the `/workspace/mpsoc/vitis_ai_dnndk_samples` directory. After all the samples are built by Arm GCC cross-compilation toolchains by running the `./build.sh zcu102` script in the folder of each sample, it is recommended to copy the whole directory `/workspace/mpsoc/vitis_ai_dnndk_samples` to the ZCU102 board directory, `/home/root/`. You can choose to copy one single DPU hybrid executable from the Docker container to the evaluation board for running. Pay attention that the dependent image folder dataset or video folder video aree copied together, and that the folder structures are kept as expected.

Note: You should run `./build.sh zcu104` for each DNNDK sample for ZCU104 board.

For the sake of simplicity, the directory of `/home/root/vitis_ai_dnndk_samples/` is replaced by `$dnndk_sample_base` in the following descriptions.

ResNet-50

`dnndk_sample_base/resnet50` contains an example of image classification using Caffe ResNet-50 model. It reads the images under the `$dnndk_sample_base/dataset/image500_640_480` directory and outputs the classification result for each input image. You can then launch it with the `./resnet50` command.

Video Analytics

An object detection example is located under the `$dnndk_sample_base/video_analysis` directory. It reads image frames from a video file and annotates detected vehicles and pedestrians in real-time. Launch it with the command `./video_analysis video/structure.mp4` (where `video/structure.mp4` is the input video file).

ADAS Detection

An example of object detection for Advanced Driver Assistance Systems (ADAS) application using the YOLO-v3 network model is located in the directory `$dnndk_sample_base/adas_detection` directory. It reads image frames from a video file and annotates in real-time. Launch it with the `./adas_detection video/adas.avi` command (where `video/adas.avi` is the input video file).

Semantic Segmentation

An example of semantic segmentation in the `$dnndk_sample_base/segmentation` directory. It reads image frames from a video file and annotates in real-time. Launch it with the `./segmentation video/traffic.mp4` command (where `video/traffic.mp4` is the input video file).

Inception-v1 with Python

`dnndk_sample_base/inception_v1_mt_py` contains a multithreaded image classification example of Inception-v1 network developed with the advanced Python APIs. With the command `python3 inception_v1_mt.py 4`, it will run with four threads. The throughput (in fps) will be reported after it completes.

The Inception-v1 model is compiled to DPU xmodel file first and then transformed into the DPU shared library `libdpumodelinception_v1.so` with the following command on the evaluation board. `dpu_inception_v1_*.xmodel` means to include all DPU xmodel files generated by the VAI_C compiler.

```
aarch64-xilinx-linux-gcc -fPIC -shared \
 dpu_inception_v1_*.xmodel -o libdpumodelinception_v1.so
```

Within the Vitis AI cross compilation environment on the host, use the following command instead.

```
source /opt/petalinux/2020.2/environment-setup-aarch64-xilinx-linux

CC -fPIC -shared dpu_inception_v1_*.elf -o libdpumodelinception_v1.so
```

Note: The thread number for best throughput of multithread Inception-v1 example varies among evaluation boards because the DPU computation power and core number are different. Use dexplorer -w to view DPU signature information for each evaluation board.

miniResNet with Python

`dnndk_sample_base/mini_resnet_py` contains the image classification example of TensorFlow miniResNet network developed with Vitis AI advanced Python APIs. With the command `python3 mini_resnet.py`, the results of top-5 labels and corresponding probabilities are displayed. miniResNet is described in the second book Practitioner Bundle of the Deep Learning for Computer Vision with Python series. It is a customization of the original ResNet-50 model and is also well explained in the third book ImageNet Bundle from the same book's series.

YOLO-v3 with Python

`dnndk_sample_base/tf_yolov3_voc_py` contains the object detection example of TensorFlow YOLOv3 network developed with Vitis AI advanced Python APIs. With the command `python3 tf_yolov3_voc.py`, the resulting image after object detection is displayed.

DNNDK Programming for Edge

DNNDK legacy programming interface provides granular manipulations to DPU control at run-time. They implement the functionalities of DPU kernel loading, task instantiation, and encapsulating the calls to invoke the XRT or the DPU driver for DPU task scheduling, monitoring, profiling, and resources management. Using these APIs can flexibly meet the diverse requirements under various edge scenarios across Xilinx® Zynq® UltraScale+™ and Zynq® UltraScale+™ MPSoC devices.

Programming Model

Understanding the DPU programming model makes it easier to develop and deploy deep learning applications over Edge DPU. The related core concepts include DPU Kernel, DPU Task, DPU Node and DPU Tensor.

DPU Kernel

After being compiled by the Vitis AI compiler, the neural network model is transformed into an equivalent DPU assembly file, which is then assembled into one ELF object file by Deep Neural Network Assembler (DNNAS). The `DPU ELF` object file is regarded as DPU kernel, which then becomes one execution unit from the perspective of runtime N2Cube after invoking the API `dpuLoadKernel()`. N2Cube loads the DPU kernel, including the DPU instructions and network parameters, into the DPU dedicated memory space and allocate hardware resources. After that, each DPU kernel can be instantiated into several DPU tasks by calling `dpuCreateTask()` to enable the multithreaded programming.

DPU Task

Each DPU task is a running entity of a DPU kernel. It has its own private memory space so that multithreaded applications can be used to process several tasks in parallel to improve efficiency and system throughput.

DPU Node

A DPU node is considered a basic element of a network model deployed on the DPU. Each DPU node is associated with input, output, and some parameters. Every DPU node has a unique name to allow APIs exported by Vitis AI to access its information.

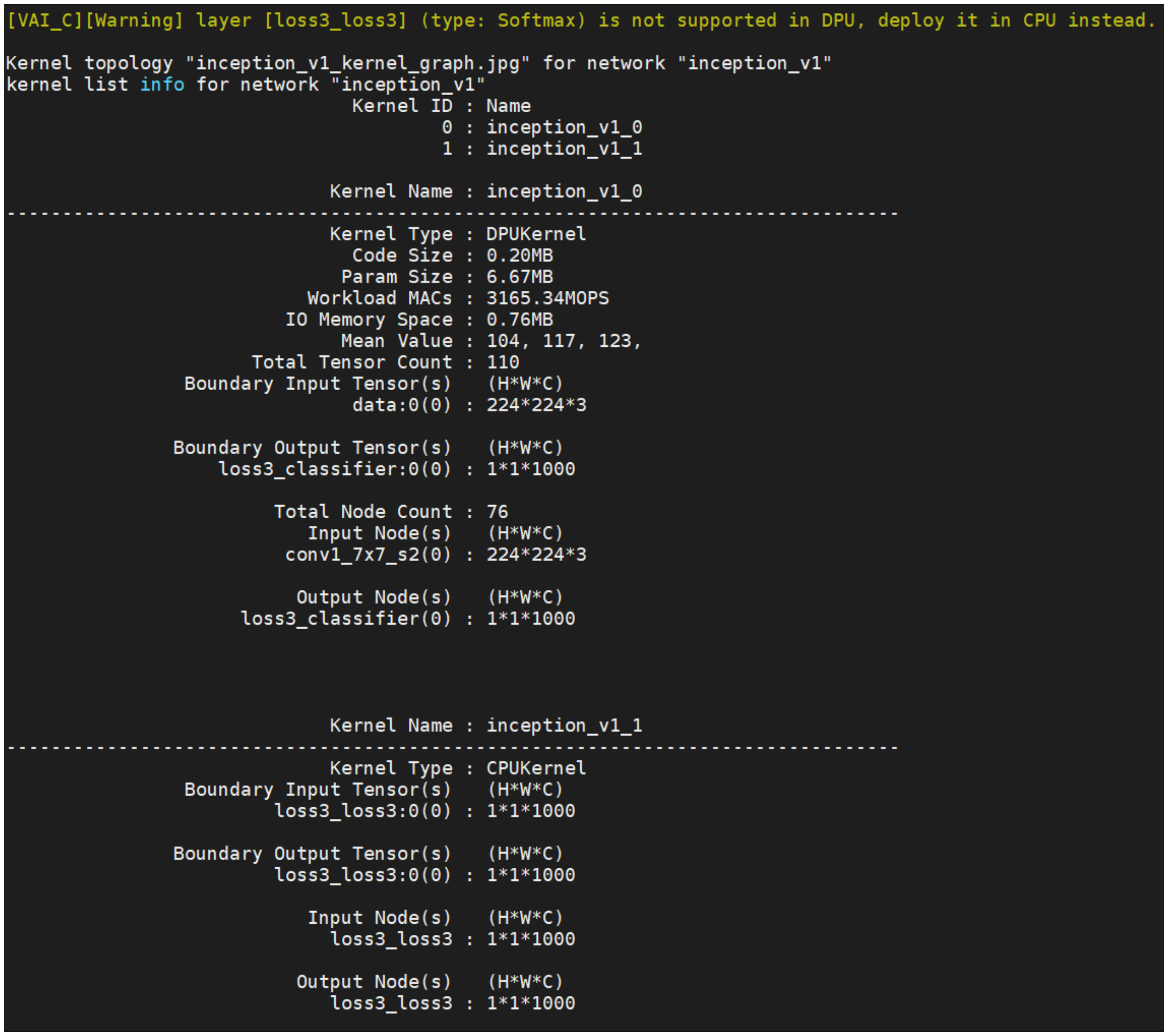
There are three types of nodes: boundary input node, boundary output node, and internal node.

- **Boundary input node:** A boundary input node is a node that does not have any precursor in the DPU kernel topology; it is usually the first node in a kernel. Sometimes there might be multiple boundary input nodes in a kernel.

- **Boundary output node:** A boundary output node is a node that does not have any successor nodes in the DPU kernel topology.
- **Internal node:** All other nodes that are not both boundary input nodes and boundary output nodes are considered as internal nodes.

After compilation, VAI_C gives information about the kernel and its boundary input/output nodes. The following figure shows an example after compiling Inception-v1. For DPU kernel 0, `conv1_7x7_s2` is the boundary input node, and `loss3_classifier` is the boundary output node.

Figure 33: **Sample VAI_C Compilation Log**



```
[VAI_C][Warning] layer [loss3_loss3] (type: Softmax) is not supported in DPU, deploy it in CPU instead.

Kernel topology "inception_v1_kernel_graph.jpg" for network "inception_v1"
kernel list info for network "inception_v1"
                               Kernel ID : Name
                                       0 : inception_v1_0
                                       1 : inception_v1_1

                             Kernel Name : inception_v1_0
--------------------------------------------------------------------------------
                             Kernel Type : DPUKernel
                               Code Size : 0.20MB
                              Param Size : 6.67MB
                           Workload MACs : 3165.34MOPS
                         IO Memory Space : 0.76MB
                              Mean Value : 104, 117, 123,
                      Total Tensor Count : 110
                Boundary Input Tensor(s)   (H*W*C)
                               data:0(0) : 224*224*3

               Boundary Output Tensor(s)   (H*W*C)
                   loss3_classifier:0(0) : 1*1*1000

                        Total Node Count : 76
                           Input Node(s)   (H*W*C)
                         conv1_7x7_s2(0) : 224*224*3

                          Output Node(s)   (H*W*C)
                     loss3_classifier(0) : 1*1*1000



                             Kernel Name : inception_v1_1
--------------------------------------------------------------------------------
                             Kernel Type : CPUKernel
                Boundary Input Tensor(s)   (H*W*C)
                        loss3_loss3:0(0) : 1*1*1000

               Boundary Output Tensor(s)   (H*W*C)
                        loss3_loss3:0(0) : 1*1*1000

                           Input Node(s)   (H*W*C)
                             loss3_loss3 : 1*1*1000

                          Output Node(s)   (H*W*C)
                             loss3_loss3 : 1*1*1000
```

When using `dpuGetInputTensor*/dpuSetInputTensor*`, the `nodeName` parameter is required to specify the boundary input node. When a `nodeName` that does not correspond to a valid boundary input node is used, Vitis AI returns an error message like:

```
[DNNDK] Node "xxx" is not a Boundary Input Node for Kernel inception_v1_0.
[DNNDK] Refer to DNNDK user guide for more info about "Boundary Input Node".
```

Similarly, when using `dpuGetOutputTensor*/dpuSetOutputTensor*`, an error similar to the following is generated when a “nodeName” that does not correspond to a valid boundary output node is used:

```
[DNNDK] Node "xxx" is not a Boundary Output Node for Kernel inception_v1_0.
[DNNDK] Please refer to DNNDK user guide for more info about "Boundary
Output Node".
```

DPU Tensor

The DPU tensor is a collection of multi-dimensional data that is used to store information while running. Tensor properties (such as height, width, channel, and so on) can be obtained using Vitis AI advanced programming APIs.

For the standard image, memory layout for the image volume is normally stored as a contiguous stream of bytes in the format of CHW (Channel*Height*Width). For DPU, memory storage layout for input tensor and output tensor is in the format of HWC (Height*Width*Channel). The data inside DPU tensor is stored as a contiguous stream of signed 8-bit integer values without padding. Therefore, you should pay attention to this layout difference when feeding data into the DPU input tensor or retrieving result data from the DPU output tensor.

Programming Interface

Vitis AI advanced C++/Python APIs are introduced to smoothen the deep learning application development for edge DPU. For detailed description of each API, refer to Appendix A: Vitis AI Programming Interface.

Python programming APIs are available to facilitate the quick network model development by reusing the pre-processing and post-processing Python code developed during the model training phase. Refer to Appendix A: Vitis AI Programming Interface for more information. Exchange of data between CPU and the DPU when programming with Vitis AI for DPU is common. For example, data pre-processed by CPU is fed to DPU for process, and the output produced by DPU might need to be accessed by CPU for further post-processing. To handle this type of operation, Vitis AI provides a set of APIs to make it easy for data exchange between CPU and DPU. Some of them are shown below. The usage of these APIs are identical to deploy network models for Caffe and TensorFlow.

Vitis AI offers the following APIs to set input tensor for the computation layer or node:

- `dpuSetInputTensor()`

- `dpuSetInputTensorInCHWInt8()`
- `dpuSetInputTensorInCHWFP32()`
- `dpuSetInputTensorInHWCInt8()`
- `dpuSetInputTensorInHWCFP32()`

Vitis AI offers the following APIs to get output tensor from the computation layer or node:

- `dpuGetOutputTensor()`
- `dpuGetOutputTensorInCHWInt8()`
- `dpuGetOutputTensorInCHWFP32()`
- `dpuGetOutputTensorInHWCInt8()`
- `dpuGetOutputTensorInHWCFP32()`

Vitis AI provides the following APIs to get the starting address, size, quantization factor, and shape info for DPU input tensor and output tensor. With such information, the users can freely implement pre-processing source code to feed signed 8-bit integer data into DPU or implement post-processing source code to get DPU output data.

- `dpuGetTensorAddress()`
- `dpuGetTensorSize()`
- `dpuGetTensorScale()`
- `dpuGetTensorHeight()`
- `dpuGetTensorWidth()`
- `dpuGetTensorChannel()`

For Caffe Model

For Caffe framework, its pre-processing for model is fixed. Vitis AI offers several pre-optimized routines like `dpuSetInputImage()` and `dpuSetInputImageWithScale()` to perform image pre-processing on CPU side, such as image scaling, normalization and quantization, and then data is fed into DPU for further processing. These routines exist within the package of Vitis AI samples. Refer to the source code of DNNDK sample ResNet-50 for more details about them.

For TensorFlow Model

TensorFlow framework supports very flexible pre-processing during model training, such as using BGR or RGB color space for input images. Therefore, the pre-optimized routines `dpuSetInputImage()` and `dpuSetInputImageWithScale()` cannot be used directly while deploying TensorFlow models. Instead, you need to implement a pre-processing code.

The following code snippet shows an example to load an image into DPU input tensor for TensorFlow model. Note that the image color space fed into the DPU input tensor should be the same with the format used during model training. With `data[j*image.rows*3+k*3+2-i]`, the image is fed into DPU in RGB color space. The process of `image.at<Vec3b>(j,k)[i])/255.0 - 0.5)*2 * scale` is specific to the model being deployed. It should be changed accordingly for the actual model used.

```
void setInputImage(DPUTask *task, const string& inNode, const cv::Mat&
image) {
  DPUTensor* in = dpuGetInputTensor(task, inNode);
  float scale = dpuGetTensorScale(in);
  int width = dpuGetTensorWidth(in);
  int height = dpuGetTensorHeight(in);
  int size = dpuGetTensorSize(in);
  int8_t* data = dpuGetTensorAddress(in);

  for(int i = 0; i < 3; ++i) {
    for(int j = 0; j < image.rows; ++j) {
      for(int k = 0; k < image.cols; ++k) {
           data[j*image.rows*3+k*3+2-i] =
          (float(image.at<Vec3b>(j,k)[i])/255.0 - 0.5)*2 * scale;
      }
     }
   }
}
```

Python is very popularly used for TensorFlow model training. With Vitis AI advanced Python APIs, you can reuse the pre-processing and post-processing Python codes during the training phase. This can help to speed up the workflow of model deployment on the DPU for quick evaluation purpose. After that it can be transformed into C++ code for better performance to meet the production requirements. The DNNDK sample miniResNet provides a reference to deploy TensorFlow miniResNet model with Python.

DPU Memory Model

For the Edge DPU, Vitis AI compiler and runtime N2Cube work together to support two different DPU memory models: unique memory model and split I/O model. The unique memory model is the default model when the network model is compiled into the DPU kernel. To enable a split I/O model, specify the `--split-io-mem` option to the compiler while compiling the network model.

Unique Memory Model

For each DPU task in this mode, all its boundary input tensors and output tensors together with its intermediate feature maps stay within one physical continuous memory buffer, which is allocated automatically while calling `dpuCreateTask()` to instantiate one DPU task from one DPU kernel. This DPU memory buffer can be cached in order to optimize memory access from the Arm® CPU. Because cache flushing and invalidation is handled by N2Cube, you do not need to take care of DPU memory management and cache manipulation. It is very easy to deploy models with unique memory model, which is the case for most of the Vitis™ AI samples.

For the unique memory model, you must copy the Int8 type input data after pre-processing into the boundary input tensors of the memory buffer of the DPU task. Then, you can launch the DPU task for running. This may bring additional overhead as there might be situations where the pre-processed input Int8 data already stays in a physical continuous memory buffer, which can be accessed by the DPU directly. One example is the camera-based deep learning application. The process of pre-processing each input image from the camera sensor can be accelerated by FPGA logic, such as image scaling, model normalization, and Float32-to-Int8 quantization. The log result data is then logged to the physical continuous memory buffer. With a unique memory model, this data must be copied to the DPU input memory buffer again.

Split I/O Memory Model

The split I/O memory model is introduced to resolve the limitation of the unique memory model so that data coming from other physical memory buffer can be consumed by the DPU directly. When the `dpuCreateTask()` function is called to create a DPU task from the DPU kernel compiled with the `-split-io-mem` option, N2Cube only allocates the DPU memory buffer for the intermediate feature maps. It is up to you to allocate the physical continuous memory buffers for boundary input tensors and output tensors individually. The size of the input memory buffer and the output memory buffer can be found from the compiler building log with the field names Input Mem Size and Output Mem Size. You also need to take care of cache coherence, if these memory buffers can be cached.

DNNDK sample `split_io` provides a programming reference for the split I/O memory model. The TensorFlow SSD is used as the reference. There is one input tensor image:0, and two output tensors `ssd300_concat:0` and `ssd300_concat_1:0` for the SSD model. From the compiler building log, you can see that the size of the DPU input memory buffer (for tensor image:0) is 270000, and the size of the DPU output memory buffer (for output tensors `ssd300_concat:0` and `ssd300_concat_1:0`) is 218304. `dpuAllocMem()` is used to allocate memory buffers for them. `dpuBindInputTensorBaseAddress()` and `dpuBindOutputTensorBaseAddress()` are subsequently used to bind the input/output memory buffer address to DPU task before launching its execution. After the input data is fed into the DPU input memory buffer, `dpuSyncMemToDev()` is called to flush the cache line. When the DPU task completes running, `dpuSyncDevToMem()` is called to invalidate the cache line.

Note: The four APIs: `dpuAllocMem()`, `dpuFreeMem()`, `dpuSyncMemToDev()`, and `dpuSyncDevToMem()` are provided only to demonstrate the split IO memory model. They are not expected to be used directly in your production environment. It is up to you whether you want to implement such functionalities to better meet customized requirements.

DPU Core Affinity

The Edge DPU runtime N2Cube support DPU core affinity with the API `dpuSetTaskAffinity()`, which can be used to dynamically assign DPU tasks to desired DPU cores so that you can participate in DPU core assignment and scheduling, as required. DPU core affinity is specified with the second argument `coreMask` to `dpuSetTaskAffinity()`. Each bit of `coreMask` represents one DPU core: the lowest bit is for core 0, second lowest bit is for core 1, and so on. Multiple mask bits can be specified one time but cannot exceed the maximum available DPU cores. For example, the mask value 0x3 indicates that a task can be assigned to DPU core 0 and 1, and it is scheduled right away if either core 0 or 1 is available.

Priority Based DPU Scheduling

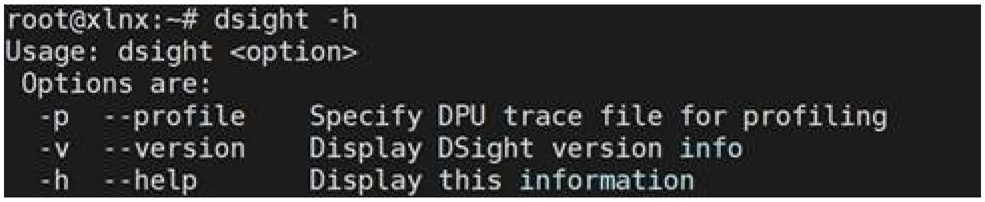
N2Cube enables priority-based DPU task scheduling using the API `dpuSetTaskPriority()`, which can specify the priority of a DPU task to a dedicated value at runtime. The priority ranges from 0 (the highest priority) to 15 (the lowest priority). If not specified, the priority of DPU task is 15 by default. This brings flexibility to meet the diverse requirements under various edge scenarios. You can specify different priorities over the models running simultaneously so that they are scheduled to DPU cores in a different order when they are all in the ready state. When affinity is specified, the N2Cube priority-based scheduling also adheres to the affinity of the DPU cores.

DNNDK samples pose detection demonstrates the feature of DPU priority scheduling. Within this sample, there are two models used: the SSD model for person detection and the pose detection model for body key points detection. The SSD model is compiled into the DPU kernel `ssd_person`. The pose detection model is compiled into two DPU kernels pose_0 and pose_2. Therefore, each input image needs to walk through these three DPU kernels in the order of `ssd_person`, `pose_0` and `pose_2`. During a multi-threading situation, several input images may overlap each other among these three kernels simultaneously. To reach better latency, DPU tasks for `ssd_person`, `pose_0`, and `pose_2` are assigned the priorities 3, 2, and 1 individually so that the DPU task for the latter DPU kernel gets scheduled with a higher priority when they are ready to run.

DNNDK Utilities

- **DSight:** DSight is the Vitis AI performance profiler for Edge DPU and is a visual analysis tool for model performance profiling. The following figure shows its usage.

Figure 34: **DSight Help Info**

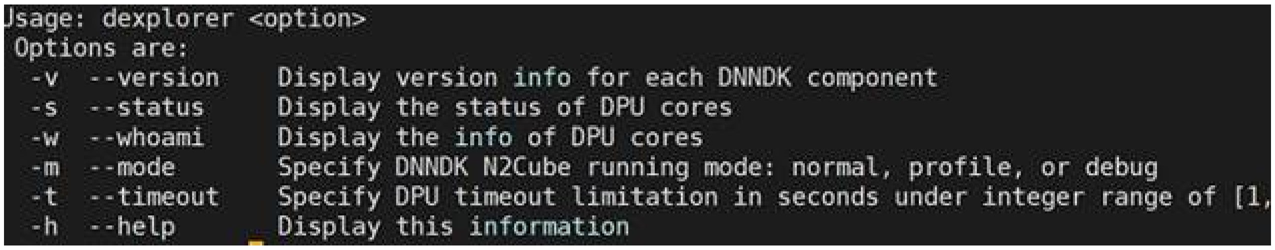


```
root@xlnx:~# dsight -h
Usage: dsight <option>
 Options are:
  -p  --profile    Specify DPU trace file for profiling
  -v  --version    Display DSight version info
  -h  --help       Display this information
```

By processing the log file produced by the runtime N2cube, DSight can generate an HTML web page, providing a visual format chart showing the utilization and scheduling efficiency of the DPU cores.

- **DExplorer:** DExplorer is a utility running on the target board. It provides DPU running mode configuration, DNNDK version checking, DPU status checking, and DPU core signature checking. The following figure shows the help information for the usage of DExplorer.

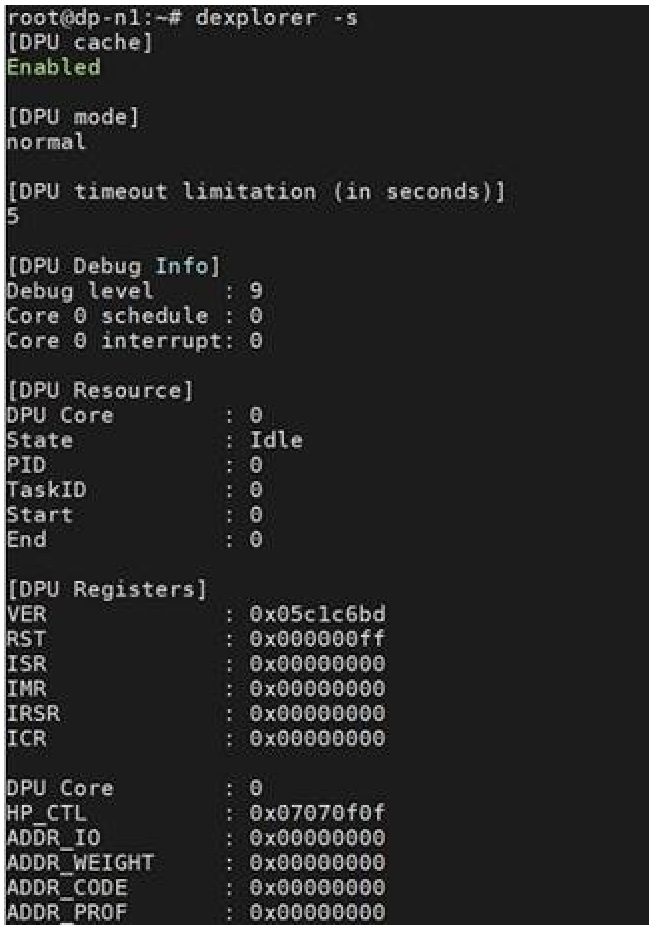
Figure 35: **DExplorer Usage Options**



```
Jsage: dexplorer <option>
 Options are:
  -v  --version    Display version info for each DNNDK component
  -s  --status     Display the status of DPU cores
  -w  --whoami     Display the info of DPU cores
  -m  --mode       Specify DNNDK N2Cube running mode: normal, profile, or debug
  -t  --timeout    Specify DPU timeout limitation in seconds under integer range of [1,
  -h  --help       Display this information
```

- **Check DNNDK Version:** Running `dexplore -v` displays version information for each component in DNNDK, including N2cube, DPU driver, DExplorer, and DSight.
- **Check DPU Status:** DExplorer provides DPU status information, including running mode of N2cube, DPU timeout threshold, DPU debugging level, DPU core status, DPU register information, DPU memory resource, and utilization. The following figure shows a screenshot of DPU status.

Figure 36: **DExplorer Status**



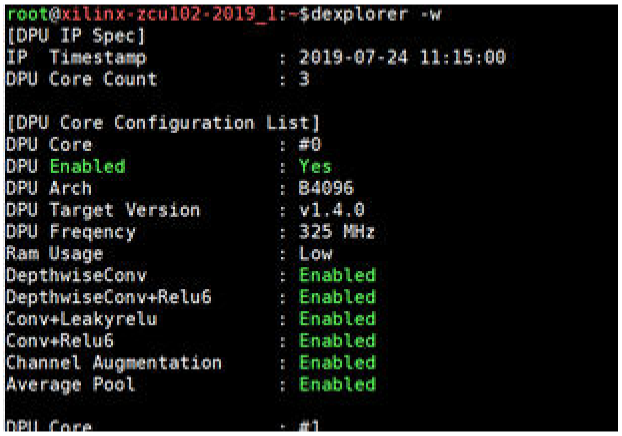
```
root@dp-n1:~# dexplorer -s
[DPU cache]
Enabled

[DPU mode]
normal

[DPU timeout limitation (in seconds)]
5

[DPU Debug Info]
Debug level      : 9
Core 0 schedule : 0
Core 0 interrupt: 0

[DPU Resource]
DPU Core         : 0
State            : Idle
PID              : 0
TaskID           : 0
Start            : 0
End              : 0

[DPU Registers]
VER              : 0x05c1c6bd
RST              : 0x000000ff
ISR              : 0x00000000
IMR              : 0x00000000
IRSR             : 0x00000000
ICR              : 0x00000000

DPU Core         : 0
HP_CTL           : 0x07070f0f
ADDR_IO          : 0x00000000
ADDR_WEIGHT      : 0x00000000
ADDR_CODE        : 0x00000000
ADDR_PROF        : 0x00000000
```

- **Configuring DPU Running Mode:** Edge DPU runtime N2cube supports three kinds of DPU execution modes to help developers to debug and profile Vitis AI applications.
  - **Normal Mode:** In normal mode, the DPU application can get the best performance without any overhead.
  - **Profile Mode:** In profile mode, the DPU turns on the profiling switch. When running deep learning applications in profile mode, N2cube outputs to the console the performance data layer by layer while executing the neural network; at the same time, a profile with the name `dpu_trace_[PID].prof` is produced under the current folder. This file can be used with the DSight tool.
  - **Debug Mode:** In this mode, the DPU dumps raw data for each DPU computation node during execution, including DPU instruction code in binary format, network parameters, DPU input tensor, and output tensor. This makes it possible to debug and locate issues in a DPU application.

***Note*:** Profile mode and debug mode are only available to network models compiled into debug mode DPU ELF objects by the Vitis AI compiler.

- **Checking DPU Signature:** New DPU cores have been introduced to meet various deep learning acceleration requirements across different Xilinx® FPGA devices. For example, DPU architectures B1024F, B1152F, B1600F, B2304F, and B4096F are available. Each DPU architecture can implement a different version of the DPU instruction set (named as a DPU target version) to support the rapid improvements in deep learning algorithms.The DPU signature refers to the specification information of a specific DPU architecture version, covering target version, working frequency, DPU core numbers, harden acceleration modules (such as softmax), etc. The `-w` option can be used to check the DPU signature. The following figure shows a screen capture of a sample run of DExplorer -w.For configurable DPU, DExplorer can help to display all configuration parameters of a DPU signature, as shown in the following figure.

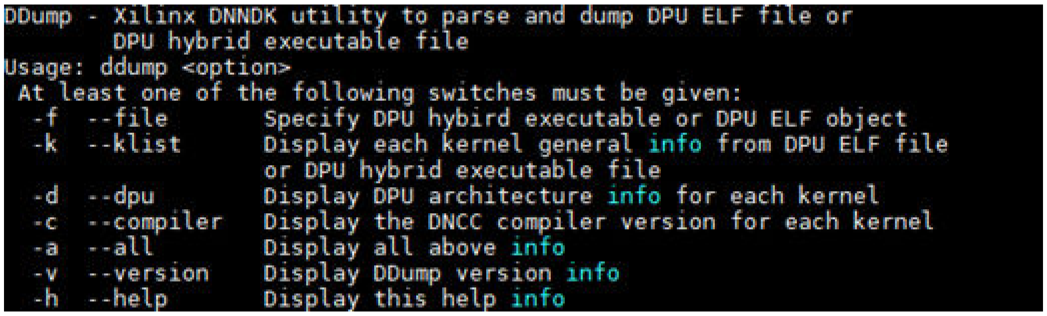
Figure 37: **Sample DPU Signature with Configuration Parameters**



```
root@xilinx-zcu102-2019_1:~$dexplorer -w
[DPU IP Spec]
IP  Timestamp             : 2019-07-24 11:15:00
DPU Core Count            : 3

[DPU Core Configuration List]
DPU Core                  : #0
DPU Enabled               : Yes
DPU Arch                  : B4096
DPU Target Version        : v1.4.0
DPU Freqency              : 325 MHz
Ram Usage                 : Low
DepthwiseConv             : Enabled
DepthwiseConv+Relu6       : Enabled
Conv+Leakyrelu            : Enabled
Conv+Relu6                : Enabled
Channel Augmentation      : Enabled
Average Pool              : Enabled

DPU Core                  : #1
```

- **DDump:** DDump is a utility tool to dump the information encapsulated inside a DPU ELF file, hybrid executable, or DPU shared library and can facilitate users to analyze and debug various issues. Refer to DPU Shared Library for more details. DDump is available on both runtime container vitis-ai-docker-runtime and Vitis AI evaluation boards. Usage information is shown in the figure below. For runtime container, it is accessible from path `/opt/vitis-ai/utility/ddump`. For evaluation boards, it is installed under Linux system path and can be used directly.

Figure 38: **DDump Usage Options**

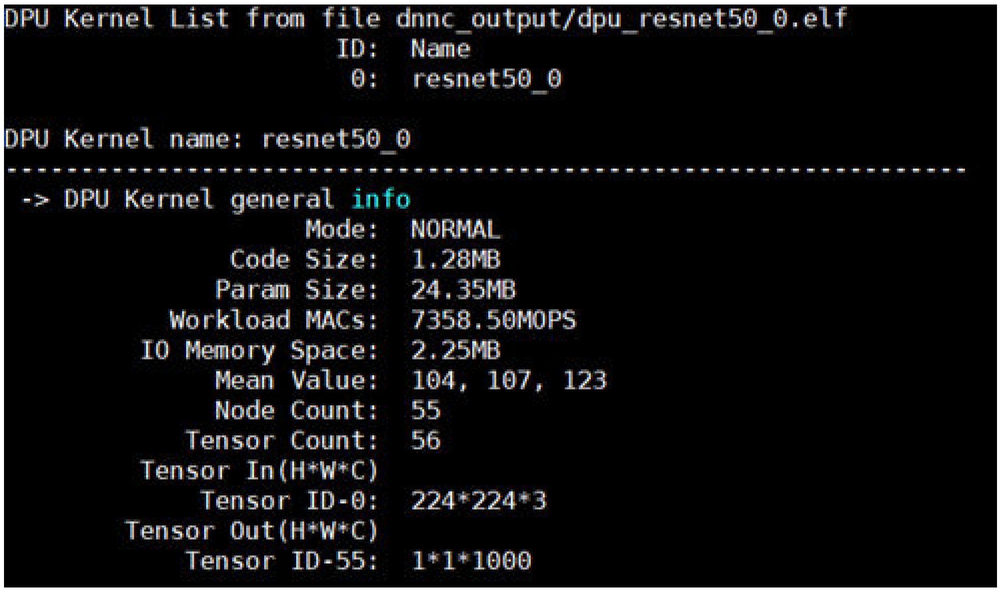


```
DDump - Xilinx DNNDK utility to parse and dump DPU ELF file or
        DPU hybrid executable file
Usage: ddump <option>
 At least one of the following switches must be given:
  -f  --file       Specify DPU hybird executable or DPU ELF object
  -k  --klist      Display each kernel general info from DPU ELF file
                   or DPU hybrid executable file
  -d  --dpu        Display DPU architecture info for each kernel
  -c  --compiler   Display the DNCC compiler version for each kernel
  -a  --all        Display all above info
  -v  --version    Display DDump version info
  -h  --help       Display this help info
```

- **Check DPU Kernel Info:** DDump can dump the following information for each DPU kernel from DPU ELF file, hybrid executable, or DPU shared library.
  - **Mode:** The mode of DPU kernel compiled by VAI_C compiler, NORMAL, or DEBUG.
  - **Code Size:** The DPU instruction code size in the unit of MB, KB, or bytes for DPU kernel.
  - **Param Size:** The Parameter size in the unit of MB, KB, or bytes for DPU kernel, including weight and bias.
  - **Workload MACs:** The computation workload in the unit of MOPS for DPU kernel.
  - **I/O Memory Space:** The required DPU memory space in the unit of MB, KB, or bytes for intermediate feature map. For each created DPU task, N2Cube automatically allocates DPU memory buffer for intermediate feature map.
  - **Mean Value:** The mean values for DPU kernel.
  - **Node Count:** The total number of DPU nodes for DPU kernel.
  - **Tensor Count:** The total number of DPU tensors for DPU kernel.
  - **Tensor In(H*W*C):** The DPU input tensor list and their shape information in the format of height*width*channel.
  - **Tensor Out(H*W*C):** The DPU output tensor list and their shape information in the format of height*width*channel.

The following figure shows the DPU kernel information for ResNet-50 DPU ELF file `dpu_resnet50_0.elf` with command `ddump -f dpu_resnet50_0.elf -k`.

Figure 39: **DDump DPU Kernel Information for ResNet50**



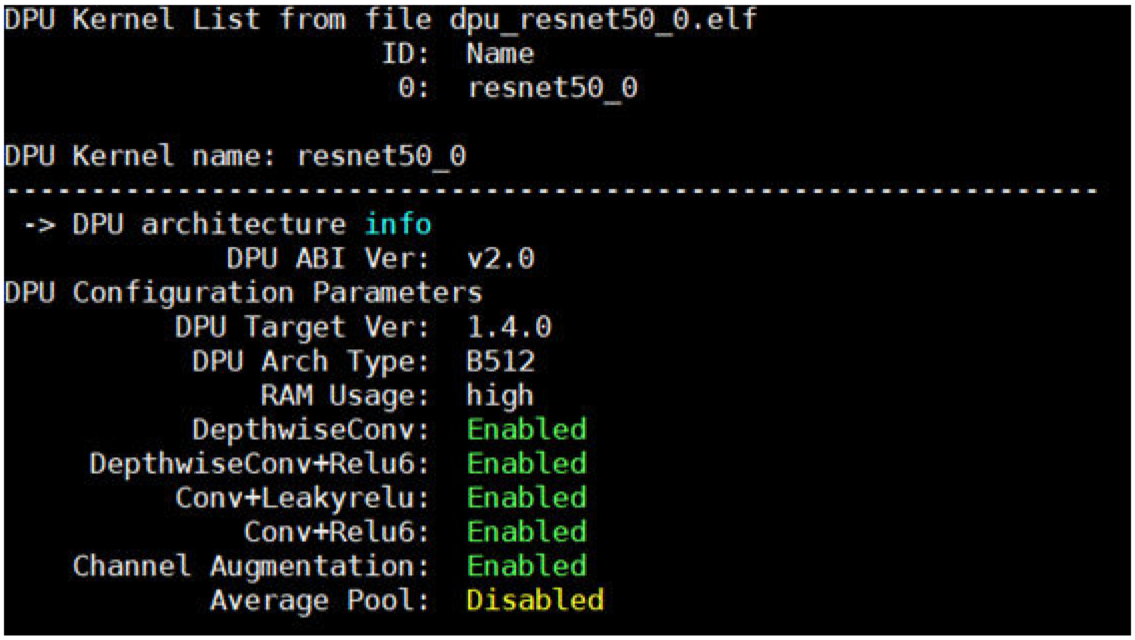
```
DPU Kernel List from file dnnc_output/dpu_resnet50_0.elf
                      ID:  Name
                       0:  resnet50_0

DPU Kernel name: resnet50_0
----------------------------------------------------------------
 -> DPU Kernel general info
                    Mode:  NORMAL
               Code Size:  1.28MB
              Param Size:  24.35MB
           Workload MACs:  7358.50MOPS
         IO Memory Space:  2.25MB
              Mean Value:  104, 107, 123
              Node Count:  55
            Tensor Count:  56
         Tensor In(H*W*C)
             Tensor ID-0:  224*224*3
        Tensor Out(H*W*C)
            Tensor ID-55:  1*1*1000
```

- **Check DPU Arch Info:** DPU configuration information from DPU DCF is automatically wrapped into DPU ELF file by VAI_C compiler for each DPU kernel. VAI_C then generates the appropriate DPU instructions, according to DPU configuration parameters. Refer to *Zynq DPU v3.1 IP Product Guide* (PG338) for more details about configurable DPU descriptions.DDump can dump out the following DPU architecture information:

  - **DPU Target Ver:** The version of DPU instruction set.
  - **DPU Arch Type:** The type of DPU architecture, such as B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096.
  - **RAM Usage:** Low or high RAM usage.
  - **DepthwiseConv:** DepthwiseConv engine enabled or not.
  - **DepthwiseConv+Relu6:** The operator pattern of DepthwiseConv following Relu6, enabled or not.
  - **Conv+Leakyrelu:** The operator pattern of Conv following Leakyrelu, enabled or not.
  - **Conv+Relu6:** The operator pattern of Conv following Relu6, enabled or not.
  - **Channel Augmentation:** An optional feature to improve DPU computation efficiency against channel dimension, especially for those layers whose input channels are much less than DPU channel parallelism.
  - **Average Pool:** The average pool engine, enabled or not.

  DPU architecture information may vary with the versions of DPU IP. Running command `ddump -f dpu_resnet50_0.elf -d`, one set of DPU architecture information used by VAI_C to compile ResNet-50 model is shown in the following figure.

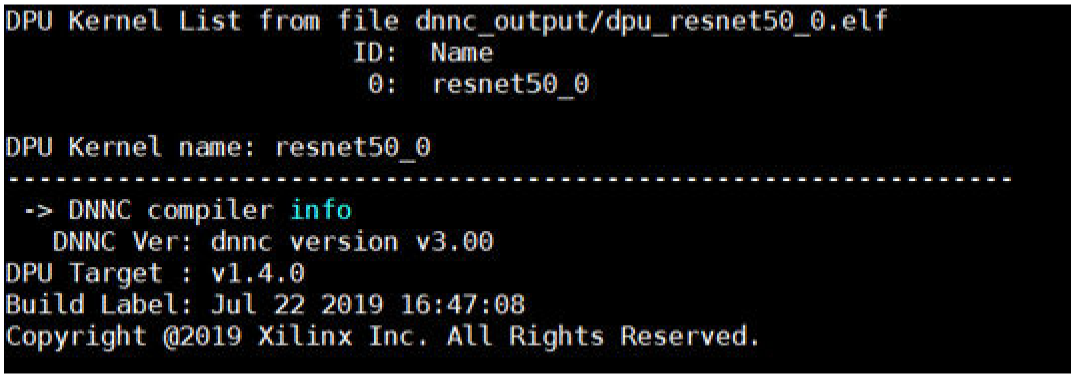
Figure 40: **DDump DPU Arch Information for ResNet50**



```
DPU Kernel List from file dpu_resnet50_0.elf
                      ID:  Name
                       0:  resnet50_0

DPU Kernel name: resnet50_0
----------------------------------------------------------------
 -> DPU architecture info
            DPU ABI Ver:  v2.0
DPU Configuration Parameters
         DPU Target Ver:  1.4.0
          DPU Arch Type:  B512
              RAM Usage:  high
          DepthwiseConv:  Enabled
    DepthwiseConv+Relu6:  Enabled
        Conv+Leakyrelu:  Enabled
            Conv+Relu6:  Enabled
   Channel Augmentation:  Enabled
           Average Pool:  Disabled
```

- **Check VAI_C Info:** VAI_C version information is automatically embedded into DPU ELF file while compiling network model. DDump can help to dump out this VAI_C version information, which users can provide to the Vitis AI support team for debugging purposes.Running command `ddump -f dpu_resnet50_0.elf -c` for ResNet-50 model VAI_C information is shown in the following figure.

Figure 41: **DDump VAI_C Info for ResNet50**

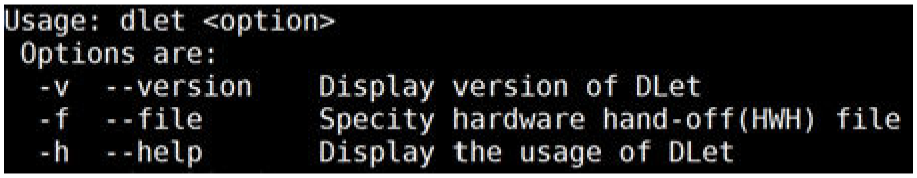


```
DPU Kernel List from file dnnc_output/dpu_resnet50_0.elf
                     ID:  Name
                      0:  resnet50_0

DPU Kernel name: resnet50_0
----------------------------------------------------------------
 -> DNNC compiler info
   DNNC Ver: dnnc version v3.00
DPU Target : v1.4.0
Build Label: Jul 22 2019 16:47:08
Copyright @2019 Xilinx Inc. All Rights Reserved.
```

- **Legacy Support:** DDump also supports dumping the information for legacy DPU ELF file, hybrid executable, and DPU shared library generated. The main difference is that there is no detailed DPU architecture information.An example of dumping all of the information for legacy ResNet-50 DPU ELF file with command `ddump -f dpu_resnet50_0.elf -a` is shown in the following figure.

- **DLet:** DLet is host tool designed to parse and extract various edge DPU configuration parameters from DPU hardware handoff file HWH, generated by Vivado® Design Suite. The following figure shows the usage information of DLet.

Figure 42: **Dlet Usage Options**



```
Usage: dlet <option>
 Options are:
  -v  --version    Display version of DLet
  -f  --file       Specity hardware hand-off(HWH) file
  -h  --help       Display the usage of DLet
```

For Vivado project, DPU HWH is located under the following directory by default. `<prj_name>` is Vivado project name, and `<bd_name>` is Vivado block design name.

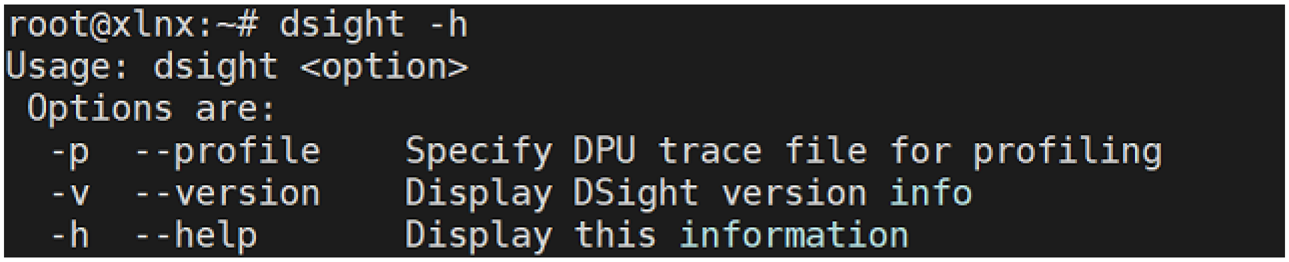
```
 <prj_name>/<prj_name>.srcs/sources_1/bd/<bd_name>/hw_handoff/
<bd_name>.hwh
```

Running command `dlet -f <bd_name>.hwh`, DLet outputs the DPU configuration file DCF, named in the format of `dpu-dd-mm-yyyy-hh-mm.dcf. dd-mm-yyyy-hh-mm` is the timestamp of when the DPU HWH is created. With the specified DCF file, VAI_C compiler automatically produces DPU code instructions suited for the DPU configuration parameters.

Profiling Using the DNNDK Profiler

DSight is the DNNDK performance profiling tool. It is a visual performance analysis tool for neural network model profiling. The following figure shows its usage.

Figure 43: **DSight Help Info**



```
root@xlnx:~# dsight -h
Usage: dsight <option>
 Options are:
  -p  --profile    Specify DPU trace file for profiling
  -v  --version    Display DSight version info
  -h  --help       Display this information
```

By processing the log file produced by the N2cube tracer, DSight can generate an HTML file, which provides a visual analysis interface for the neural network model. The steps below describe how to use the profiler:

1. Set N2Cube to profile mode using the command `dexplorer -m profile`.
2. Run the deep learning application. When finished, a profile file with the name `dpu_trace_[PID].prof` is generated for further checking and analysis (`PID` is the process ID of the deep learning application).

3. Generate the HTML file with the DSight tool using the command: `dsight -p dpu_trace_[PID].prof`. An HTML file named `dpu_trace_[PID].html` is generated.
4. Open the generated HTML file with web browser.

Fine-Grained Profiling

After the models are compiled and deployed over Edge DPU, the utility DExplorer can be used to perform fined-grained profiling to check layer-by-layer execution time and DDR memory bandwidth. This is very useful for the model's performance bottleneck analysis.

Note: The model should be compiled by Vitis AI compiler into debug mode kernel; fine-grained profiling is not available for normal mode kernel.

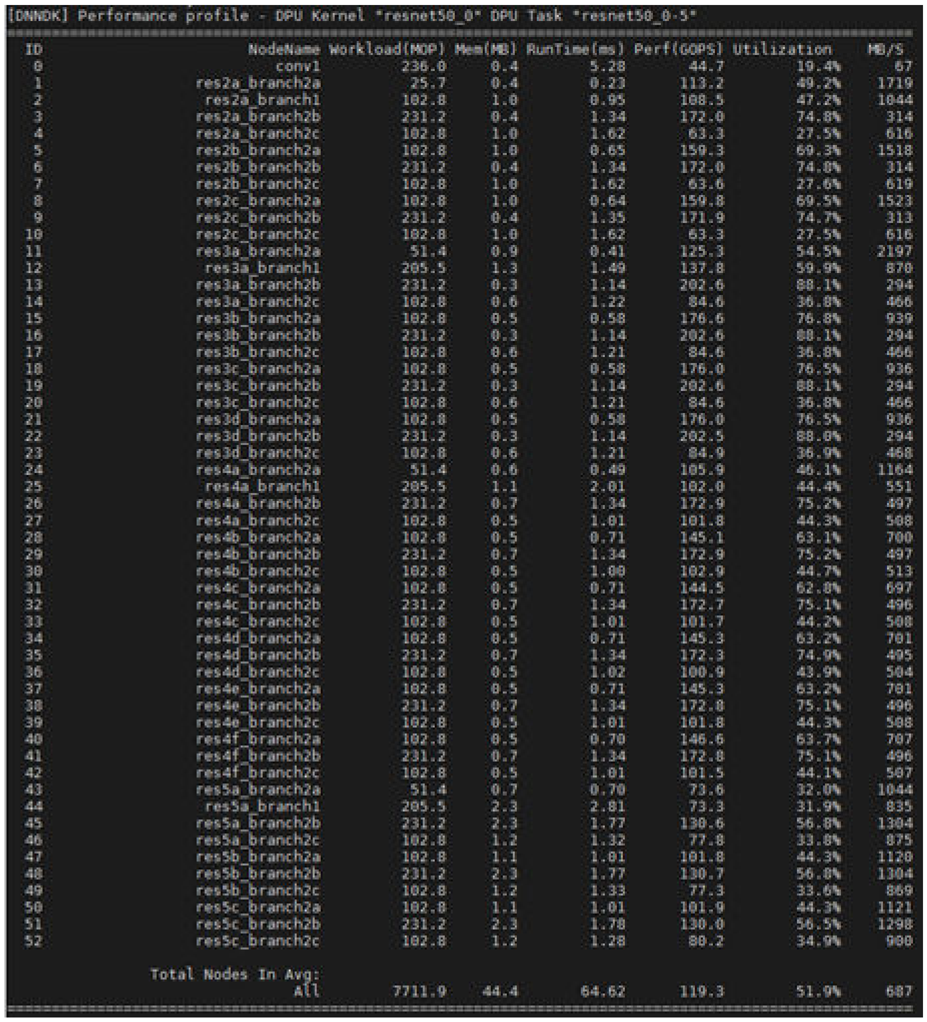
There are two approaches to enable fine-grained profiling for debug mode kernel:

- Run `dexplorer -m profile` before launch the running of DPU application. This will change N2Cube global running mode and all the DPU tasks (debug mode) will run under the profiling mode.
- Use `dpuCreateTask()` with `flag T_MODE_PROF` or `dpuEnableTaskProfile()` to enable profiling mode for the dedicated DPU task only. Other tasks will not be affected.

The following figure shows a profiling screen capture over ResNet50 model. The profiling information for each DPU layer (or node) over ResNet-50 kernel is listed out.

Note: For each DPU node, it may include several layers or operators from original Caffe or TensorFlow models because Vitis AI compiler performs layer/operator fusion to optimize execution performance and DDR memory access.

Figure 44: **Fine-Grained Profiling for ResNet50**



```
[DNNDK] Performance profile - DPU Kernel "resnet50_0" DPU Task "resnet50_0-5"
=====================================================================================================
  ID                      NodeName Workload(MOP) Mem(MB) RunTime(ms) Perf(GOPS) Utilization    MB/S
   0                         conv1         236.0     0.4        5.28       44.7       19.4%      67
   1                res2a_branch2a          25.7     0.4        0.23      113.2       49.2%    1719
   2                 res2a_branch1         102.8     1.0        0.95      108.5       47.2%    1044
   3                res2a_branch2b         231.2     0.4        1.34      172.0       74.8%     314
   4                res2a_branch2c         102.8     1.0        1.62       63.3       27.5%     616
   5                res2b_branch2a         102.8     1.0        0.65      159.3       69.3%    1518
   6                res2b_branch2b         231.2     0.4        1.34      172.0       74.8%     314
   7                res2b_branch2c         102.8     1.0        1.62       63.6       27.6%     619
   8                res2c_branch2a         102.8     1.0        0.64      159.8       69.5%    1523
   9                res2c_branch2b         231.2     0.4        1.35      171.9       74.7%     313
  10                res2c_branch2c         102.8     1.0        1.62       63.3       27.5%     616
  11                res3a_branch2a          51.4     0.9        0.41      125.3       54.5%    2197
  12                 res3a_branch1         205.5     1.3        1.49      137.8       59.9%     870
  13                res3a_branch2b         231.2     0.3        1.14      202.6       88.1%     294
  14                res3a_branch2c         102.8     0.6        1.22       84.6       36.8%     466
  15                res3b_branch2a         102.8     0.5        0.58      176.6       76.8%     939
  16                res3b_branch2b         231.2     0.3        1.14      202.6       88.1%     294
  17                res3b_branch2c         102.8     0.6        1.21       84.6       36.8%     466
  18                res3c_branch2a         102.8     0.5        0.58      176.0       76.5%     936
  19                res3c_branch2b         231.2     0.3        1.14      202.6       88.1%     294
  20                res3c_branch2c         102.8     0.6        1.21       84.6       36.8%     466
  21                res3d_branch2a         102.8     0.5        0.58      176.0       76.5%     936
  22                res3d_branch2b         231.2     0.3        1.14      202.5       88.0%     294
  23                res3d_branch2c         102.8     0.6        1.21       84.9       36.9%     468
  24                res4a_branch2a          51.4     0.6        0.49      105.9       46.1%    1164
  25                 res4a_branch1         205.5     1.1        2.01      102.0       44.4%     551
  26                res4a_branch2b         231.2     0.7        1.34      172.9       75.2%     497
  27                res4a_branch2c         102.8     0.5        1.01      101.8       44.3%     508
  28                res4b_branch2a         102.8     0.5        0.71      145.1       63.1%     700
  29                res4b_branch2b         231.2     0.7        1.34      172.9       75.2%     497
  30                res4b_branch2c         102.8     0.5        1.00      102.9       44.7%     513
  31                res4c_branch2a         102.8     0.5        0.71      144.5       62.8%     697
  32                res4c_branch2b         231.2     0.7        1.34      172.7       75.1%     496
  33                res4c_branch2c         102.8     0.5        1.01      101.7       44.2%     508
  34                res4d_branch2a         102.8     0.5        0.71      145.3       63.2%     701
  35                res4d_branch2b         231.2     0.7        1.34      172.3       74.9%     495
  36                res4d_branch2c         102.8     0.5        1.02      100.9       43.9%     504
  37                res4e_branch2a         102.8     0.5        0.71      145.3       63.2%     701
  38                res4e_branch2b         231.2     0.7        1.34      172.8       75.1%     496
  39                res4e_branch2c         102.8     0.5        1.01      101.8       44.3%     508
  40                res4f_branch2a         102.8     0.5        0.70      146.6       63.7%     707
  41                res4f_branch2b         231.2     0.7        1.34      172.8       75.1%     496
  42                res4f_branch2c         102.8     0.5        1.01      101.5       44.1%     507
  43                res5a_branch2a          51.4     0.7        0.70       73.6       32.0%    1044
  44                 res5a_branch1         205.5     2.3        2.81       73.3       31.9%     835
  45                res5a_branch2b         231.2     2.3        1.77      130.6       56.8%    1304
  46                res5a_branch2c         102.8     1.2        1.32       77.8       33.8%     875
  47                res5b_branch2a         102.8     1.1        1.01      101.8       44.3%    1120
  48                res5b_branch2b         231.2     2.3        1.77      130.7       56.8%    1304
  49                res5b_branch2c         102.8     1.2        1.33       77.3       33.6%     869
  50                res5c_branch2a         102.8     1.1        1.01      101.9       44.3%    1121
  51                res5c_branch2b         231.2     2.3        1.78      130.0       56.5%    1298
  52                res5c_branch2c         102.8     1.2        1.28       80.2       34.9%     900

                   Total Nodes In Avg:
                                   All        7711.9    44.4       64.62      119.3       51.9%     687
=====================================================================================================
```

The following fields are included:

- **ID:** The index ID of DPU node.
- **NodeName:** DPU node name.
- **Workload (MOP):** Computation workload (MAC indicates two operations).
- **Mem (MB):** Memory size for code, parameter, and feature map for this DPU node.
- **Runtime (ms):** The execution time in unit of Millisecond.
- **Perf (GOPS):** The DPU performance in unit of GOP per second.
- **Utilization (%):** The DPU utilization in percent.
- **MB/S:** The average DDR memory access bandwidth.

With the fine-grained profiling result over one specific model by DSight if you are not satisfied with the performance delivered by the DPU core, you can try to modify DPU configuration so as to obtain better performance. For example, you can apply more advanced DPU arch from B1152 to B4096, or use high on-chip RAM. For more information, see the https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD. Otherwise, if the DPU core offers enough performance, you can try to change the DPU configuration with lower logic resource requirements.

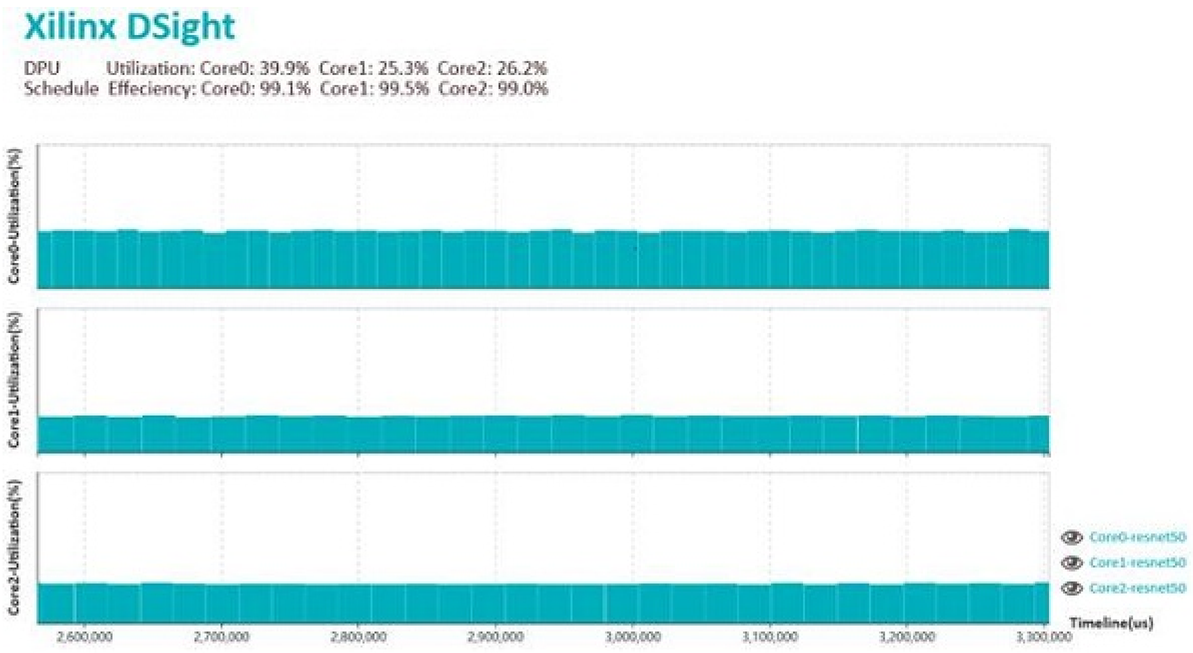
Panorama-View Profiling

DSight delivers the visual format profiling statistics to let you have a panorama view over DPU cores utilization so that they can locate the application's bottleneck and further optimize performance. Ideally, the models should be compiled by VAI_C into normal mode DPU kernels before performing panorama view profiling.

The following steps describe how to conduct profiling with DSight:

- Switch N2Cube into profile mode using the command `dexplorer -m` profile.
- Run the DPU application and stop the process after it stays under the typical performance situation for several seconds A profile file with the name `dpu_trace_[PID].prof` is generated within the application's directory for further processing. (PID is the process ID of the launched DPU application).
- Launch the DSight tool with the command `dsight -p dpu_trace_[PID].prof`. An HTML file with the name `dpu_trace_[PID].html` is generated by DSight
- Open this generated HTML web page with any web browser and visual charts will be shown. One profiling example for multi-threading ResNet-50 over triple DPU cores is shown in the following figure.

Figure 45: **Profiling Example**



- **DPU Utilization (Y-axis):** List out the utilization of each DPU core. A higher percentage means DPU computing power is fully utilized to accelerate the model's execution. For lower percentage, the users can try to change the DPU configuration to reduce its required logic resources or try to re-design the algorithm so that DPU computing resources match the algorithm's requirement better.
- **Schedule Efficiency (X-axis):** Indicate what percentages of each DPU core are scheduled by runtime N2Cube. If the percentage number is lower, the users can try to improve the application's thread number so that DPU cores have more chances to be triggered. To further improve DPU cores' schedule efficiency, the users should try to optimize the other parts of computation workloads running on Arm CPU side, such as using NEON intrinsic, assembly instructions, or using Vitis accelerated libraries (e.g., xfOpenCV). Typically, such non-DPU parts workloads include pre-processing, post-processing, or DPU unsupported deep learning operators.

DNNDK Programming APIs

This section describes the legacy DNNDK programming interface available only for the Edge DPU.

***Note*:** These APIs are deprecated in future releases. For new applications or projects, use the VART APIs.

DNNDK legacy APIs are regarded as Vitis AI advanced low-level C++/Python programming interface. It consists of a comprehensive set of APIs that can flexibly meet the diverse requirements under various edge scenarios. For example, low-level API `dpuSetTaskPriority()` can be used to specify the scheduling priority of DPU tasks so that different models can be scheduled under the dedicated priorities. `dpuSetTaskAffinity()` can be used to dynamically assign DPU tasks to desired DPU cores so that you can participate in DPU cores' assignment and scheduling as required. Meanwhile, such advanced APIs bring forward compatibility so that DNNDK legacy projects can be ported to Vitis platform without any modifications to the existing source code.

Vitis AI advanced low-level C++ APIs are implemented within runtime library `libn2cube` for the Edge DPU and are exported within header file `n2cube.h`, which represents in header file `dnndk.h`. Hence the users only need to include `dnndk.h` at the source code.

In the meantime, you can adopt the suited low-level Python APIs in module n2cube, which are equivalent wrappers for those C++ APIs in library `libn2cube`. With the Python programming interface, you can reuse the Python code developed during model training phase and quickly deploy the models on edge DPU for evaluation purpose.

C++ APIs

The following Vitis AI advanced low-level C++ programming APIs are briefly summarized.

Name

libn2cube.so

Description

DPU runtime library

Routines

- **dpuOpen():** Open & initialize the usage of DPU device
- **dpuClose():** Close & finalize the usage of DPU device
- **dpuLoadKernel():** Load a DPU Kernel and allocate DPU memory space for its Code/Weight/Bias segments
- **dpuDestroyKernel():** Destroy a DPU Kernel and release its associated resources
- **dpuCreateTask():** Instantiate a DPU Task from one DPU Kernel, allocate its private working memory buffer and prepare for its execution context
- **dpuRunTask():** Launch the running of DPU Task

- **dpuDestroyTask():** Remove a DPU Task, release its working memory buffer and destroy associated execution context
- **dpuSetTaskPriority():** Dynamically set a DPU Task's priority to a specified value at runtime. Priorities range from 0 (the highest priority) to 15 (the lowest priority). If not specified, the priority of a DPU Task is 15 by default.
- **dpuGetTaskPriority():** Retrieve a DPU Task's priority.
- **dpuSetTaskAffinity():** Dynamically set a DPU Task's affinity over DPU cores at runtime. If not specified, a DPU Task can run over all the available DPU cores by default.
- **dpuGetTaskAffinity():** Retrieve a DPU Task's affinity over DPU cores.
- **dpuEnableTaskDebug():** Enable dump facility of DPU Task while running for debugging purpose
- **dpuEnableTaskProfile():** Enable profiling facility of DPU Task while running to get its performance metrics
- **dpuGetTaskProfile():** Get the execution time of DPU Task
- **dpuGetNodeProfile():** Get the execution time of DPU Node
- **dpuGetInputTensorCnt():** Get total number of input Tensor of one DPU Task
- **dpuGetInputTensor():** Get input Tensor of one DPU Task
- **dpuGetInputTensorAddress():** Get the start address of one DPU Task's input Tensor
- **dpuGetInputTensorSize():** Get the size (in byte) of one DPU Task's input Tensor
- **dpuGetInputTensorScale():** Get the scale value of one DPU Task's input Tensor
- **dpuGetInputTensorHeight():** Get the height dimension of one DPU Task's input Tensor
- **dpuGetInputTensorWidth():** Get the width dimension of one DPU Task's input Tensor
- **dpuGetInputTensorChannel() :** Get the channel dimension of one DPU Task's input Tensor
- **dpuGetOutputTensorCnt():** Get total number of output Tensor of one DPU Task
- **dpuGetOutputTensor():** Get output Tensor of one DPU Task
- **dpuGetOutputTensorAddress():** Get the start address of one DPU Task's output Tensor
- **dpuGetOutputTensorSize():** Get the size in byte of one DPU Task's output Tensor
- **dpuGetOutputTensorScale():** Get the scale value of one DPU Task's output Tensor
- **dpuGetOutputTensorHeight():** Get the height dimension of one DPU Task's output Tensor
- **dpuGetOutputTensorWidth():** Get the width dimension of one DPU Task's output Tensor

- **dpuGetOutputTensorChannel():** Get the channel dimension of one DPU Task's output Tensor
- **dpuGetTensorSize():** Get the size of one DPU Tensor
- **dpuGetTensorAddress():** Get the start address of one DPU Tensor
- **dpuGetTensorScale():** Get the scale value of one DPU Tensor
- **dpuGetTensorHeight():** Get the height dimension of one DPU Tensor
- **dpuGetTensorWidth():** Get the width dimension of one DPU Tensor
- **dpuGetTensorChannel():** Get the channel dimension of one DPU Tensor
- **dpuSetInputTensorInCHWInt8():** Set DPU Task's input Tensor with data stored under Caffe order (channel/height/width) in INT8 format
- **dpuSetInputTensorInCHWFP32():** Set DPU Task's input Tensor with data stored under Caffe order (channel/height/width) in FP32 format
- **dpuSetInputTensorInHWCInt8():** Set DPU Task's input Tensor with data stored under DPU order (height/width/channel) in INT8 format
- **dpuSetInputTensorInHWCFP32():** Set DPU Task's input Tensor with data stored under DPU order (channel/height/width) in FP32 format
- **dpuGetOutputTensorInCHWInt8():** Get DPU Task's output Tensor and store them under Caffe order (channel/height/width) in INT8 format
- **dpuGetOutputTensorInCHWFP32():** Get DPU Task's output Tensor and store them under Caffe order (channel/height/width) in FP32 format
- **dpuGetOutputTensorInHWCInt8():** Get DPU Task's output Tensor and store them under DPU order (channel/height/width) in INT8 format
- **dpuGetOutputTensorInHWCFP32():** Get DPU Task's output Tensor and store them under DPU order (channel/height/width) in FP32 format
- **dpuRunSoftmax ():** Perform softmax calculation for the input elements and save the results to output memory buffer.
- **dpuSetExceptionMode():** Set the exception handling mode for edge DPU runtime N2Cube.
- **dpuGetExceptionMode():** Get the exception handling mode for runtime N2Cube.
- **dpuGetExceptionMessage():** Get the error message from error code (always negative value) returned by N2Cube APIs.
- **dpuGetInputTotalSize():** Get total size in byte for DPU task's input memory buffer, which includes all the boundary input tensors.
- **dpuGetOutputTotalSize():** Get total size in byte for DPU task's outmemory buffer, which includes all the boundary output tensors.

- **dpuGetBoundaryIOTensor():** Get DPU task's boundary input or output tensor from the specified tensor name. The info of tensor names is listed out by VAI_C compiler after model compilation.
- **dpuBindInputTensorBaseAddress():** Bind the specified base physical and virtual addresses of input memory buffer to DPU task. It can only be used for DPU kernel compiled by VAI_C under split IO mode. Note it can only be used for DPU kernel compiled by VAI_C under split IO mode.
- **dpuBindOutputTensorBaseAddress():** Bind the specified base physical and virtual addresses of output memory buffer to DPU task. Note it can only be used for DPU kernel compiled by VAI_C under split IO mode.

Include File

`n2cube.h`

APIs List

The prototype and parameters for each C++ API within the library libn2cube are described in detail in the subsequent sections.

dpuOpen()

Synopsis

```
int dpuOpen()
```

Arguments

None

Description

Attach and open DPU device file `/dev/dpu` before the utilization of DPU resources.

Returns

0 on success, or negative value in case of failure. Error message "Fail to open DPU device" is reported if any error takes place.

See Also

dpuClose()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuClose()

Synopsis

```
int dpuClose()
```

Arguments

None

Description

Detach and close DPU device file `/dev/dpu` after utilization of DPU resources.

Returns

0 on success, or negative error ID in case of failure. Error message “Fail to close DPU device” is reported if any error takes place

See Also

dpuOpen()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuLoadKernel()

Synopsis

```
DPUKernel *dpuLoadKernel
(
const char *netName
    )
```

Arguments

- **netName:** The pointer to neural network name. Use the names produced by Deep Neural Network Compiler (VAI_C) after the compilation of neural network. For each DL application, perhaps there are many DPU Kernels existing in its hybrid CPU+DPU binary executable. For each DPU Kernel, it has one unique name for differentiation purpose.

Description

Load a DPU Kernel for the specified neural network from hybrid CPU+DPU binary executable into DPU memory space, including Kernel's DPU instructions, weight and bias.

Returns

The pointer to the loaded DPU Kernel on success, or report error in case of any failure.

See Also

dpuDestroyKernel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuDestroyKernel()

Synopsis

```
Dint dpuDestroyKernel
(
DPUKernel *kernel
    )
```

Arguments

- **kernel:** The pointer to DPU kernel to be destroyed.

Description

Destroy a DPU kernel and release its related resources.

Returns

0 on success, or report error in case of any failure.

See Also

dpuLoadKernel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuCreateTask()

Synopsis

```
int dpuCreateTask
(
DPUKernel *kernel,
int mode
    );
```

Arguments

- **kernel:** The pointer to DPU kernel to be destroyed.
- **mode:** The running mode of DPU Task. There are 3 available modes:
  - **T_MODE_NORMAL:** default mode identical to the mode value “0”.
  - **T_MODE_PROF:** generate profiling information layer by layer while running of DPU Task, which is useful for performance analysis.
  - **T_MODE_DEBUG:** dump the raw data for DPU Task's CODE/BIAS/WEIGHT/INPUT/OUTPUT layer by layer for debugging purpose.

Description

Instantiate a DPU Task from DPU Kernel and allocate corresponding DPU memory buffer.

Returns

0 on success, or report error in case of any failure.

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuDestroyTask()

Synopsis

```
int dpuDestroyTask
(
DPUTask *task
    )
```

Arguments

- **task:**

  The pointer to DPU Task to be destroyed.

Description

Destroy a DPU Task and release its related resources.

Returns

0 on success, or report error in case of any failure.

See Also

dpuCreateTask()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuRunTask()

Synopsis

```
int dpuRunTask
(
DPUTask *task
    );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Launch the running of DPU Task.

Returns

0 on success, or negative value in case of any failure.

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetTaskPriority()

Synopsis

```
int dpuSetTaskPriority
(
DPUTask *task,
uint8_t priority
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **priority:** The priority to be specified for the DPU task. It ranges from 0 (the highest priority) to 15 (the lowest priority).

Description

Dynamically set a DPU task's priority to a specified value at run-time. Priorities range from 0 (the highest priority) to 15 (the lowest priority). If not specified, the priority of a DPU Task is 15 by default.

Returns

0 on success, or negative value in case of any failure.

See Also

dpuGetTaskPriority()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTaskPriority()

Synopsis

```
uint8_t
      dpuGetTaskPriority
(
DPUTask *task
    );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Retrieve a DPU Task's priority. The priority is 15 by default.

Returns

The priority of DPU Task on success, or 0xFF in case of any failure.

See Also

dpuSetTaskPriority()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetTaskAffinity()

Synopsis

```
int
      dpuSetTaskAffinity
(
DPUTask *task,
uint32_t coreMask
     );
```

Arguments

- **task:** The pointer to DPU Task.
- **coreMask:** DPU core mask to be specified. Each bit represents one DPU core: the lowest bit is for core 0, second lowest bit is for core 1, and so on. Multiple mask bits can be specified one time but can't exceed the maximum available cores. For example, mask value 0x3 indicates that task can be assigned to DPU core 0 and 1, and it gets scheduled right away if anyone of core 0 or 1 is available.

Description

Dynamically set a DPU task's affinity to DPU cores at run-time. This provides flexibility for the users to intervene in DPU cores' assignment and scheduling to meet specific requirements. If not specified, DPU task can be assigned to any available DPU cores during run-time.

Returns

0 on success, or negative value in case of any failure.

See Also

dpuGetTaskAffinity ()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTaskAffinity ()

Synopsis

```
uint32_t dpuGetTaskAffinity
(
DPUTask *task
     );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Retrieve a DPU Task's affinity over DPU cores. If the affinity is not specified, DPU task can be assigned to all available DPU cores by default. For example, the affinity is 0x7 if the target system holds 3 DPU cores.

Returns

The affinity mask bits over DPU cores on success, or 0 in case of any failure.

See Also

dpuSetTaskAffinity()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuEnableTaskProfile()

Synopsis

```
int dpuEnableTaskProfile
(
DPUTask *task
     );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Set DPU Task in profiling mode. Note that the profiling functionality is available only for DPU Kernel generated by VAI_C in debug mode.

Returns

0 on success, or report error in case of any failure.

See Also

dpuCreateTask()

dpuEnableTaskDebug()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuEnableTaskDebug()

Synopsis

```
int dpuEnableTaskDebug
(
DPUTask *task
     );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Set DPU Task in dump mode. Note that dump functionality is available only for DPU Kernel generated by VAI_C in debug mode.

Returns

0 on success, or report error in case of any failure.

See Also

dpuCreateTask()

dpuEnableTaskProfile()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTaskProfile()

Synopsis

```
int dpuGetTaskProfile
(
DPUTask *task
    );
```

Arguments

- **task:** The pointer to DPU Task.

Description

Get DPU Task's execution time (us) after its running.

Returns

The DPU Task's execution time (us) after its running.

See Also

dpuGetNodeProfile()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetNodeProfile()

Synopsis

```
int dpuGetNodeProfile
(
DPUTask *task,
const char*nodeName
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name

Description

Get DPU Node's execution time (us) after DPU Task completes its running.

Returns

The DPU Node's execution time(us) after DPU Task completes its running. Note that this functionality is available only for DPU Kernel generated by VAI_C in debug mode.

See Also

dpuGetTaskProfile()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetInputTensorCnt()

Synopsis

```
Int dpuGetInputTensorCnt
(
DPUTask *task,
const char*nodeName
    );
```

Arguments

- **task:** The pointer to DPU task.

- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's input node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.

Description

Get total number of input tensors for the specified node of one DPU task.

Returns

The total number of input tensor for specified node.

See Also

dpuGetOutputTensorCnt()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetInputTensor()

Synopsis

```
DPUTensor*dpuGetInputTensor
(
DPUTask *task,
const char*nodeName,
int idx = 0
     );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: The available names of one DPU kernel's or task's input node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.

- **idx:**

  The index of a single input tensor for the node, with default value as 0.

Description

Get DPU Task's input Tensor.

Returns

The pointer to Task's input Tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensor()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetInputTensorAddress()

Synopsis

```
int8_t*
        dpuGetInputTensorAddress
(
DPUTask *task,
const char*nodeName,
int idx = 0
);
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU node's name.

  ***Note*:** The available names of one DPU kernel's or task's input node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.

- **idx:**

  The index of a single input tensor for the node, with default value as 0.

Description

Get the start address of DPU Task's input tensor.

Returns

The start addresses to Task's input Tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorAddress()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetInputTensorSize()

Synopsis

```
int dpuGetInputTensorSize
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: The available names of one DPU kernel's or task's input node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:**

  The index of a single input tensor for the node, with default value as 0.

Description

Get the size (in Byte) of DPU task's input tensor.

Returns

The size of task's input tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorSize()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetInputTensorScale()

Synopsis

```
float dpuGetInputTensorScale
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get the scale value of DPU task's input tensor. For each DPU input tensor, it has one unified scale value indicating its quantization information for reformatting between data types of INT8 and FP32.

Returns

The scale value of task's input tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorScale()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetInputTensorHeight()

Synopsis

```
int dpuGetInputTensorHeight
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get the height dimension of DPU task's input tensor.

Returns

The height dimension of task's input tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorWidth()

dpuGetInputTensorChannel()

dpuGetOutputTensorHeight()

dpuGetOutputTensorWidth()

dpuGetOutputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetInputTensorWidth()

Synopsis

```
int dpuGetInputTensorWidth
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get the width dimension of DPU task's input tensor.

Returns

The width dimension of task's input tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorHeight()

dpuGetInputTensorChannel()

dpuGetOutputTensorHeight()

dpuGetOutputTensorWidth()

dpuGetOutputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetInputTensorChannel()

Synopsis

```
int dpuGetInputTensorChannel
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.

- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get the channel dimension of DPU task's input tensor.

Returns

The channel dimension of task's input tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorHeight()

dpuGetInputTensorWidth()

dpuGetOutputTensorHeight()

dpuGetOutputTensorWidth()

dpuGetOutputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensorCnt()

Synopsis

```
Int dpuGetOutputTensorCnt
(
DPUTask *task,
const char*nodeName
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get total number of output tensors for the specified node of one DPU task's.

Returns

The total number of output tensor for the DPU task.

See Also

dpuGetInputTensorCnt()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensor()

Synopsis

```
DPUTensor*dpuGetOutputTensor
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get DPU task's output tensor.

Returns

The pointer to task's output tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensor()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensorAddress()

Synopsis

```
int8_t* dpuGetOutputTensorAddress
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU task.
- **nodeName:** The pointer to DPU node's name.

  Note: The available names of one DPU kernel's or task's output node are listed out after a neural network is compiled by VAI_C. If invalid node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the node, with default value as 0.

Description

Get the start address of DPU task's output tensor.

Returns

The start addresses to task's output tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorAddress()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensorSize()

Synopsis

```
int dpuGetOutputTensorSize
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: The available names of one DPU Kernel's or Task's output Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the Node, with default value as 0.

Description

Get the size (in Byte) of DPU Task's output Tensor.

Returns

The size of Task's output Tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorSize()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensorScale()

Synopsis

```
float dpuGetOutputTensorScale
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: The available names of one DPU Kernel's or Task's output Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.

- **idx:** The index of a single output tensor for the Node, with default value as 0.

Description

Get the scale value of DPU Task's output Tensor. For each DPU output Tensor, it has one unified scale value indicating its quantization information for reformatting between data types of INT8 and FP32.

Returns

The scale value of Task's output Tensor on success, or report error in case of any failure.

See Also

dpuGetInputTensorScale()

Include File

`n2cube.h`

Availability

Vitis AI v1.0.

dpuGetOutputTensorHeight()

Synopsis

```
int dpuGetOutputTensorHeight
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: The available names of one DPU Kernel's or Task's output Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the Node, with default value as 0.

Description

Get the height dimension of DPU Task's output Tensor.

Returns

The height dimension of Task's output Tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorWidth()

dpuGetOutputTensorChannel()

dpuGetInputTensorHeight()

dpuGetInputTensorWidth()

dpuGetInputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorWidth()

Synopsis

```
int dpuGetOutputTensorWidth
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: the available names of one DPU Kernel's or Task's output Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the Node, with default value as 0.

Description

Get the width dimension of DPU Task's output Tensor.

Returns

The width dimension of Task's output Tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorHeight()

dpuGetOutputTensorChannel()

dpuGetInputTensorHeight()

dpuGetInputTensorWidth()

dpuGetInputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorChannel()

Synopsis

```
int dpuGetOutputTensorChannel
(
DPUTask *task,
const char*nodeName,
int idx = 0
    );
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node's name.

  Note: the available names of one DPU Kernel's or Task's output Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.
- **idx:** The index of a single output tensor for the Node, with default value as 0.

Description

Get the channel dimension of DPU Task's output Tensor.

Returns

The channel dimension of Task's output Tensor on success, or report error in case of any failure.

See Also

dpuGetOutputTensorHeight()

dpuGetOutputTensorWidth()

dpuGetInputTensorHeight()

dpuGetInputTensorWidth()

dpuGetInputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorAddress()

Synopsis

```
int dpuGetTensorAddress
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the start address of DPU Tensor.

Returns

The start address of Tensor, or report error in case of any failure.

See Also

dpuGetInputTensorAddress()

dpuGetOutputTensorAddress()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorSize()

Synopsis

```
int dpuGetTensorSize
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the size (in Byte) of one DPU Tensor.

Returns

The size of Tensor, or report error in case of any failure.

See Also

dpuGetInputTensorSize()

dpuGetOutputTensorSize()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorScale()

Synopsis

```
float dpuGetTensorScale
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the scale value of one DPU Tensor.

Returns

Return the scale value of Tensor, or report error in case of any failure. The users can perform quantization (Float32 to Int8) for DPU input tensor or de-quantization (Int8 to Float32) for DPU output tensor with this scale factor.

See Also

dpuGetInputTensorScale()

dpuGetOutputTensorScale()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorHeight()

Synopsis

```
float dpuGetTensorHeight
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the height dimension of one DPU Tensor.

Returns

The height dimension of Tensor, or report error in case of any failure.

See Also

dpuGetInputTensorHeight()

dpuGetOutputTensorHeight()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorWidth()

Synopsis

```
float dpuGetTensorWidth
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the width dimension of one DPU Tensor.

Returns

The width dimension of Tensor, or report error in case of any failure.

See Also

dpuGetInputTensorWidth()

dpuGetOutputTensorWidth()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetTensorChannel()

Synopsis

```
float dpuGetTensorChannel
(
DPUTensor* tensor
    );
```

Arguments

- **tensor:** The pointer to DPU Tensor.

Description

Get the channel dimension of one DPU Tensor.

Returns

The channel dimension of Tensor, or report error in case of any failure.

See Also

dpuGetInputTensorChannel()

dpuGetOutputTensorChannel()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetInputTensorInCHWInt8()

Synopsis

```
int dpuSetInputTensorInCHWInt8
(
DPUTask *task,
const char *nodeName,
int8_t *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The pointer to the start address of input data.
- **size:** The size (in Byte) of input data to be set.
- **idx:** The index of a single input tensor for the Node, with default value of 0.

Description

Set DPU Task input Tensor with data from a CPU memory block. Data is in type of INT8 and stored in Caffe Blob's order: channel, height and weight.

Returns

0 on success, or report error in case of failure.

See Also

dpuSetInputTensorInCHWFP32()

dpuSetInputTensorInHWCInt8()

dpuSetInputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetInputTensorInCHWFP32()

Synopsis

```
int dpuSetInputTensorInCHWFP32
(
DPUTask *task,
const char *nodeName,
float *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The pointer to the start address of input data.
- **size:** The size (in Byte) of input data to be set.
- **idx:** The index of a single input tensor for the Node, with default value of 0.

Description

Set DPU Task's input Tensor with data from a CPU memory block. Data is in type of 32-bit-float and stored in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuSetInputTensorInCHWInt8()

dpuSetInputTensorInHWCInt8()

dpuSetInputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetInputTensorInHWCInt8()

Synopsis

```
int dpuSetInputTensorInHWCInt8
(
DPUTask *task,
const char *nodeName,
int8_t *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The pointer to the start address of input data.
- **size:** The size (in Byte) of input data to be set.
- **idx:** The index of a single input tensor for the Node, with default value of 0.

Description

Set DPU Task's input Tensor with data from a CPU memory block. Data is in type of 32-bit-float and stored in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuSetInputTensorInCHWInt8()

dpuSetInputTensorInCHWFP32()

dpuSetInputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetInputTensorInHWCFP32()

Synopsis

```
int dpuSetInputTensorInHWCFP32
(
DPUTask *task,
const char *nodeName,
float *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The pointer to the start address of input data.
- **size:** The size (in Byte) of input data to be set.
- **idx:** The index of a single input tensor for the Node, with default value of 0.

Description

Set DPU Task's input Tensor with data from a CPU memory block. Data is in type of 32-bit-float and stored in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuSetInputTensorInCHWInt8()

dpuSetInputTensorInCHWFP32()

dpuSetInputTensorInHWCInt8()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorInCHWInt8()

Synopsis

```
int dpuGetOutputTensorInCHWInt8
(
DPUTask *task,
const char *nodeName,
int8_t *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The start address of CPU memory block for storing output Tensor's data.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its data into a CPU memory block. Data will be stored in type of INT8 and in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCInt8()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorInCHWFP32()

Synopsis

```
int dpuGetOutputTensorInCHWFP32
(
DPUTask *task,
const char *nodeName,
float *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The start address of CPU memory block for storing output Tensor's data.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its data into a CPU memory block. Data will be stored in type of 32-bit-float and in Caffe Blob's order: channel, height and weight.

Returns

0 on success, or report error in case of failure.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInHWCInt8()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorInHWCInt8()

Synopsis

```
int dpuGetOutputTensorInHWCInt8
(
DPUTask *task,
const char *nodeName,
int8_t *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The start address of CPU memory block for storing output Tensor's data.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its data into a CPU memory block. Data will be stored in type of INT8 and in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTensorInHWCFP32()

Synopsis

```
int dpuGetOutputTensorInHWCFP32
(
DPUTask *task,
const char *nodeName,
float *data,
int size,
int idx = 0
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **nodeName:** The pointer to DPU Node name.
- **data:** The start address of CPU memory block for storing output Tensor's data.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its data into a CPU memory block. Data will be stored in type of 32-bit-float and in DPU Tensor's order: height, weight and channel.

Returns

0 on success, or report error in case of failure.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCInt8()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuRunSoftmax()

Synopsis

```
int dpuRunSoftmax
(
int8_t *input,
float *output,
int numClasses,
int batchSize,
float scale
    )
```

Arguments

- **input:** The pointer to store softmax input elements in int8_t type.
- **output:** The pointer to store softmax running results in floating point type. This memory space should be allocated and managed by caller function.
- **numClasses:** The number of classes that softmax calculation operates on.
- **batchSize:** Batch size for the softmax calculation. This parameter should be specified with the division of the element number by inputs by numClasses.
- **scale:** The scale value applied to the input elements before softmax calculation. This parameter typically can be obtained by using API dpuGetRensorScale().

Description

Perform softmax calculation for the input elements and save the results to output memory buffer. This API will leverage DPU core for acceleration if harden softmax module is available. Run “dexplorer -w” to view DPU signature information.

Returns

0 for success.

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuSetExceptionMode()

Synopsis

```
int dpuSetExceptionMode
(
int mode
    )
```

Arguments

- **mode:** The exception handling mode for runtime N2Cube to be specified. Available values include:
  - N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT
  - N2CUBE_EXCEPTION_MODE_RET_ERR_CODE

Description

Set the exception handling mode for edge DPU runtime N2Cube. It will affect all the APIs included in the libn2cube library.

If N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT is specified, the invoked N2Cube APIs will output the error message and terminate the running of DPU application when any error occurs. It is the default mode for N2Cube APIs.

If N2CUBE_EXCEPTION_MODE_RET_ERR_CODE is specified, the invoked N2Cube APIs only return error code in case of errors. The callers need to take charge of the following exception handling process, such as logging the error message with API dpuGetExceptionMessage(), resource release, etc.

Returns

0 on success, or negative value in case of failure.

See Also

dpuGetExceptionMode()

dpuGetExceptionMessage

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetExceptionMode()

Synopsis

```
int dpuGetExceptionMode()
```

Arguments

None.

Description

Get the exception handling mode for runtime N2Cube.

Returns

Current exception handing mode for N2Cube APIs.

Available values include:

- N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT
- N2CUBE_EXCEPTION_MODE_RET_ERR_CODE

See Also

dpuSetExceptionMode()

dpuGetExceptionMessage

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetExceptionMessage

Synopsis

```
const char *dpuGetExceptionMessage
(
int error_code
    )
```

Arguments

- **error code:** The error code returned by N2Cube APIs.

Description

Get the error message from error code (always negative value) returned by N2Cube APIs.

Returns

A pointer to a const string, indicating the error message for error_code.

See Also

dpuSetExceptionMode()

dpuGetExceptionMode()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetInputTotalSize()

Synopsis

```
int dpuGetInputTotalSize
(
DPUTask *task,
    )
```

Arguments

- **task:** The pointer to DPU Task.

Description

Get total size in byte for DPU task's input memory buffer, which holds all the boundary input tensors.

Returns

The total size in byte for DPU task's all the boundary input tensors.

See Also

dpuGetOutputTotalSize()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetOutputTotalSize()

Synopsis

```
int dpuGetOutputTotalSize
(
DPUTask *task,
    )
```

Arguments

- **task:** The pointer to DPU Task.

Description

Get total size in byte for DPU task's output memory buffer, which holds all the boundary output tensors.

Returns

The total size in byte for DPU task's all the boundary output tensors.

See Also

dpuGetInputTotalSize()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuGetBoundaryIOTensor()

Synopsis

```
DPUTensor * dpuGetInputTotalSize
(
DPUTask *task,
Const char
      *tensorName
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **tensorName:** Tensor Name that is listed out by VAI_C compiler after model compilation.

Description

Get DPU task's boundary input or output tensor from the specified tensor name. The info of tensor names is listed out by VAI_C compiler after model compilation.

Returns

Pointer to DPUTensor.

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuBindInputTensorBaseAddress()

Synopsis

```
int dpuBindInputTensorBaseAddress
(
DPUTask *task,
int8_t *addrVirt,
int8_t *addrPhy
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **addrVirt:**
- **addrPhy:** The physical address of DPU output memory buffer, which holds all the boundary output tensors of DPU task. The virtual address of DPU output memory buffer, which holds all the boundary output tensors of DPU task.

Description

Bind the specified base physical and virtual addresses of input memory buffer to DPU task.

Note: It can only be used for DPU kernel compiled by VAI_C under split I/O mode.

Returns

0 on success, or report error in case of any failure.

See Also

dpuBindOutputTensorBaseAddress()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

dpuBindOutputTensorBaseAddress()

Synopsis

```
int dpuBindOutputTensorBaseAddress
(
DPUTask *task,
int8_t *addrVirt,
int8_t *addrPhy
    )
```

Arguments

- **task:** The pointer to DPU Task.
- **addrVirt:** The virtual address of DPU output memory buffer, which holds all the boundary output tensors of DPU task.
- **addrPhy:** The physical address of DPU output memory buffer, which holds all the boundary output tensors of DPU task.

Description

Bind the specified base physical and virtual addresses of output memory buffer to DPU task.

Note: It can only be used for DPU kernel compiled by VAI_C under split I/O mode.

Returns

0 on success, or report error in case of any failure.

See Also

dpuBindInputTensorBaseAddress()

Include File

`n2cube.h`

Availability

Vitis AI v1.0

Python APIs

Most Vitis AI advanced low-level Python APIs in module `n2cube` are equivalent with C++ APIs in library `libn2cube`. The differences between them are listed below, which are also described in the subsequent sections.

- `dpuGetOutputTensorAddress()`: The type of return value different from C++ API.
- `dpuGetTensorAddress()`: The type of return value different from C++ API.
- `dpuGetInputTensorAddress()`: Not available for Python API.
- `dpuGetTensorData()`: Available only for Python API
- `dpuGetOutputTensorInCHWInt8()`: The type of return value different from C++ API.
- `dpuGetOutputTensorInCHWFP32()`: The type of return value different from C++ API.
- `dpuGetOutputTensorInHWCInt8`: The type of return value different from C++ API.
- `dpuGetOutputTensorInHWCFP32()`: The type of return value different from C++ API.
- `dpuRunSoftmax()`: The type of return value different from C++ API.

In addition, the feature of DPU split IO is not available for Python interface. Hence the following two APIs cannot be used by the users to deploy model with Python.

- `dpuBindInputTensorBaseAddress`()
- `dpuBindOutputTensorBaseAddress`()

APIs List

The prototype and parameters for those changed Python APIs of module `n2cube` are described in detail in the subsequent sections.

dpuGetOutputTensorAddress()

Synopsis

```
dpuGetOutputTensorAddress
(
task,
nodeName,
idx = 0
     )
```

Arguments

- **task:** The ctypes pointer to DPU Task.
- **nodeName:** The string DPU Node's name.

  Note: The available names of one DPU Kernel's or Task's input Node are listed out after a neural network is compiled by VAI_C. If invalid Node name specified, failure message is reported.
- **idx:** The index of a single input tensor for the Node, with default value as 0.

Description

Get the ctypes pointer that points to the data of DPU Task's output Tensor.

Note: For C++ API, it returns int8_t type start address of DPU Task's output Tensor.

Returns

Return ctypes pointer that points to the data of DPU Task's output Tensor. Using together with dpuGetTensorData, the users can get output Tensor's data.

See Also

dpuGetTensorData()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetTensorAddress()

Synopsis

```
dpuGetTensorAddress
(
tensor
     )
```

Arguments

- **tensor:** The ctypes pointer to DPU Tensor

Description

Get the ctypes pointer that points to the data of DPU Task's output Tensor.

Note: For C++ API, it returns int8_t type start address of DPU Task's output Tensor.

Returns

Return ctypes pointer that points to the data of DPU Tensor. Using together with dpuGetTensorData, the users can get Tensor's data

See Also

dpuGetTensorData()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetTensorData()

Synopsis

```
dpuGetTensorData
(
tensorAddress,
data,
tensorSize
    )
```

Arguments

- **tensorAddress:** The ctypes pointer to the data of DPU Tensor.
- **data:** The list to store the data of DPU Tensor.
- **tensorSize:** Size of DPU Tensor's data.

Description

Get the DPU Tensor’s data.

Returns

None.

See Also

dpuGetOutputTensorAddress()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetOutputTensorInCHWInt8()

Synopsis

```
dpuGetOutputTensorInCHWInt8
(
task,
nodeName,
int size,
idx = 0
    )
```

Arguments

- **task:** The ctypes pointer to DPU Task.
- **size:** The string DPU Node's name.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its INT8 type data into CPU memory buffer under the layout of CHW (Channel*Height*Width).

Returns

NumPy array to hold the output data. Its size is zero in case of any error.

See Also

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCInt8()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetOutputTensorInCHWFP32()

Synopsis

```
dpuGetOutputTensorInCHWFP32
(
task,
nodeName,
int size,
idx = 0
     )
```

Arguments

- **task:** The ctypes pointer to DPU Task.
- **nodeName:** The string DPU Node's name.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Convert the data of DPU Task's output Tensor from INT8 to float32, and store into CPU memory buffer under the layout of CHW (Channel*Height*Width).

Returns

NumPy array to hold the output data. Its size is zero in case of any error.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInHWCInt8()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetOutputTensorInHWCInt8()

Synopsis

```
dpuGetOutputTensorInHWCInt8
(
task,
nodeName,
int size,
idx = 0
    )
```

Arguments

- **task:** The ctypes pointer to DPU Task.
- **nodeName:** The string DPU Node's name.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Get DPU Task's output Tensor and store its INT8 type data into CPU memory buffer under the layout of HWC (Height*Width*Channel).

Returns

NumPy array to hold the output data. Its size is zero in case of any error.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCFP32()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuGetOutputTensorInHWCFP32()

Synopsis

```
dpuGetOutputTensorInHWCFP32
(
task,
nodeName,
int size,
idx = 0
    )
```

Arguments

- **task:** The ctypes pointer to DPU Task.
- **nodeName:** The string DPU Node's name.
- **size:** The size (in Bytes) of output data to be stored.
- **idx:** The index of a single output tensor for the Node, with default value of 0.

Description

Convert the data of DPU Task's output Tensor from INT8 to float32, and store into CPU memory buffer under the layout of HWC (Height*Width*Channel).

Returns

NumPy array to hold the output data. Its size is zero in case of any error.

See Also

dpuGetOutputTensorInCHWInt8()

dpuGetOutputTensorInCHWFP32()

dpuGetOutputTensorInHWCInt8()

Include File

`n2cube`

Availability

Vitis AI v1.0

dpuRunSoftmax()

Synopsis

```
dpuRunSoftmax
(
int8_t *input,
int numClasses,
int batchSize,
float scale
    )
```

Arguments

- **input:** The pointer to store softmax input elements in int8_t type.
- **numClasses:** The number of classes that softmax calculation operates on.
- **batchSize:** Batch size for the softmax calculation. This parameter should be specified with the division of the element number by inputs by numClasses.
- **scale:** The scale value applied to the input elements before softmax calculation. This parameter typically can be obtained by using API dpuGetRensorScale().

Description

Perform softmax calculation for the input elements and save the results to output memory buffer. This API will leverage DPU core for acceleration if harden softmax module is available. Run “dexplorer -w” to view DPU signature information.

Returns

NumPy array to hold the result of softmax calculation. Its size is zero in case of any error.

Include File

`n2cube.h`

Availability

Vitis AI v1.0

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://github.com/Xilinx/Vitis-AI/blob/master/doc/release-notes/1.x.md

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright