Vitis AI User Guide

UG1414 (v1.0) December 18, 2019

Revision History

The following table shows the revision history for this document.

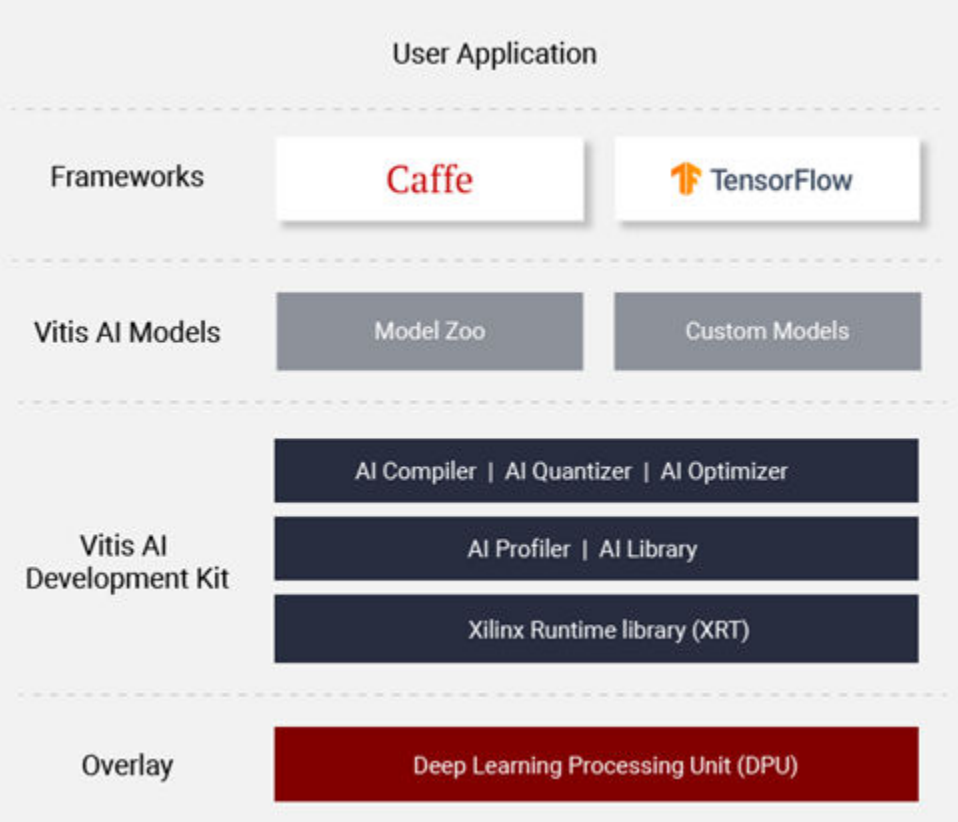
Section	Revision Summary
12/18/2019 Version 1.0	
Entire document	Minor updates
12/02/2019 Version 1.0	
Initial release.	N/A

Table of Contents

Vitis AI Development Kit

The Vitis™ AI development environment consists of the Vitis AI development kit, for the 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, unleashing the full potential of AI acceleration on Xilinx® FPGA and ACAP. It makes simple for users without FPGA knowledge to develop deep-learning inference applications, by abstracting away the intricacies of the underlying FPGA and ACAP devices.

Figure 1: **Vitis AI Stack**



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, we also offer 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.

Components

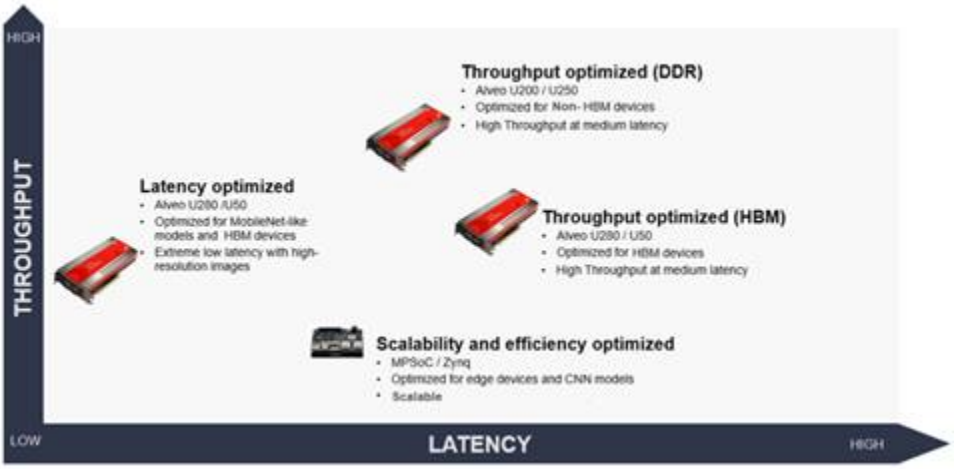
Deep Learning Processor Unit (DPU)

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. The DPU is released with the Vitis AI specialized instruction set, allowing efficient implementation of many deep learning networks.

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.

***Note*:** DPU for U50 and U280 is early access in Vitis AI 1.0 release. It will be public in the future releases.

Figure 2: **DPU Options**

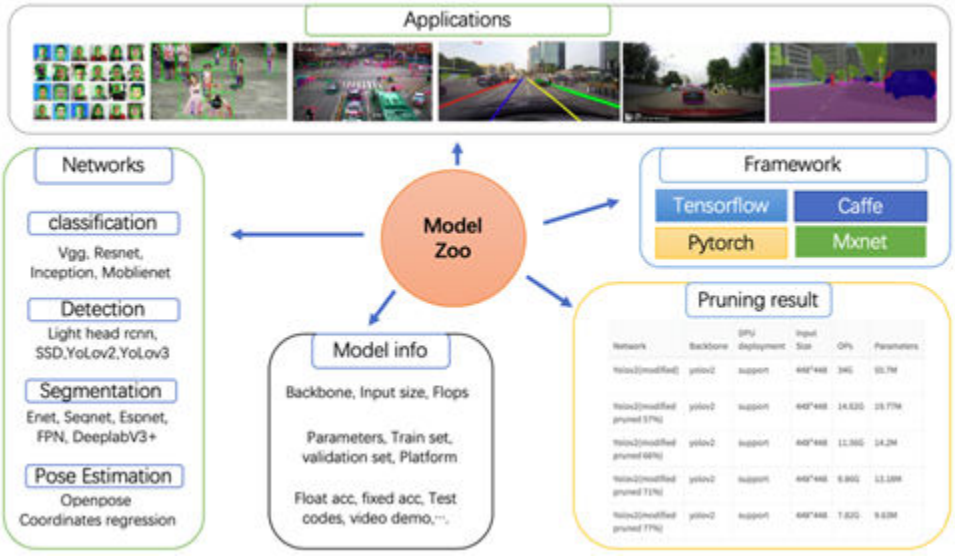


AI Model Zoo

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, data center, etc. You can get started with these pre-trained models to enjoy the benefits of deep learning acceleration.

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

Figure 3: **AI Model Zoo**

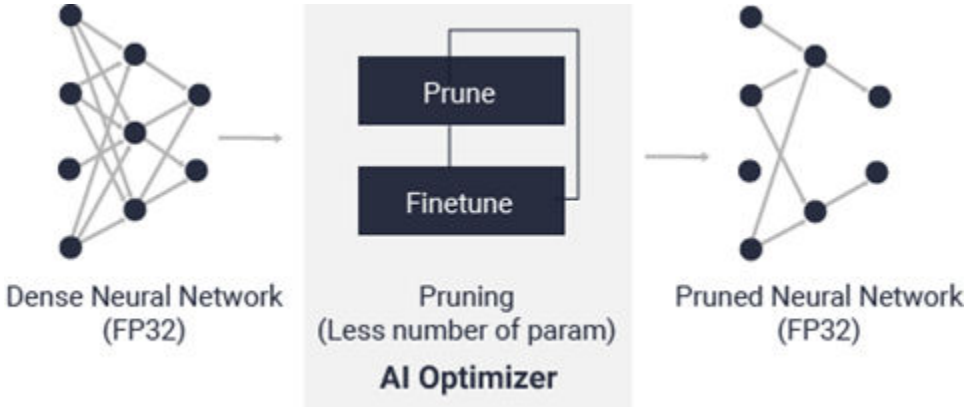


AI Optimizer

With world-leading model compression technology, we can reduce model complexity by 5x to 50x with minimal accuracy impact. Deep Compression takes the performance of your AI inference to the next level.

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

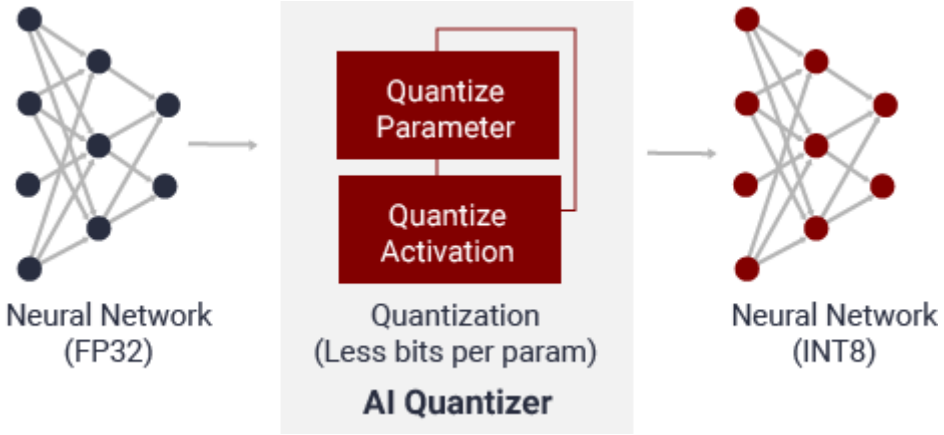
Figure 4: **AI Optimizer**



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.

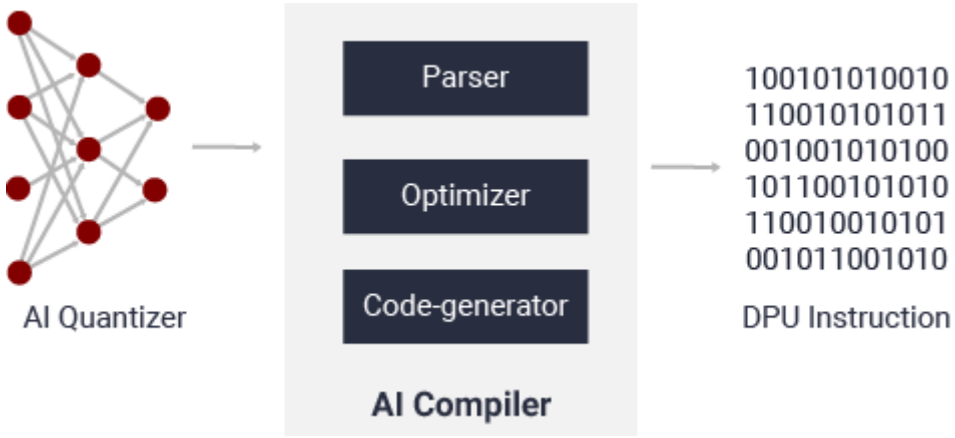
Figure 5: **AI Quantizer**



AI Compiler

The AI Compiler maps the AI model to a high-efficient instruction set and data flow. It also performs sophisticated optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible.

Figure 6: **AI Complier**

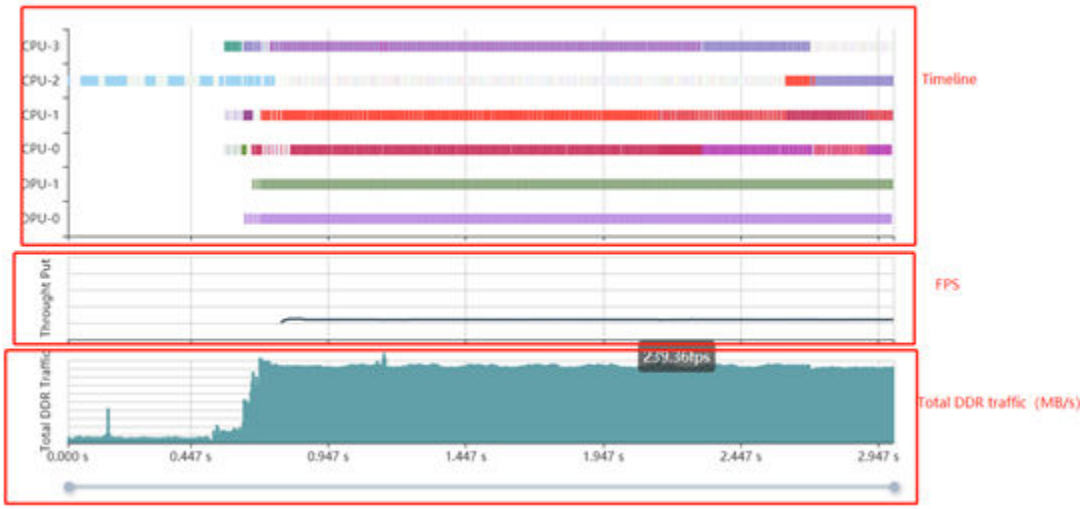


AI Profiler

The Vitis AI profiler can help profiling and visualizing AI applications, to find bottlenecks, and help to allocate computing resources among different devices:

- It is easy to use. There is no change in code nor re-compile the program. It can also trace function calling and time consumption.
- The tool can also collect hardware information, including CPU/DPU/Memory.

Figure 7: **AI Profiler**

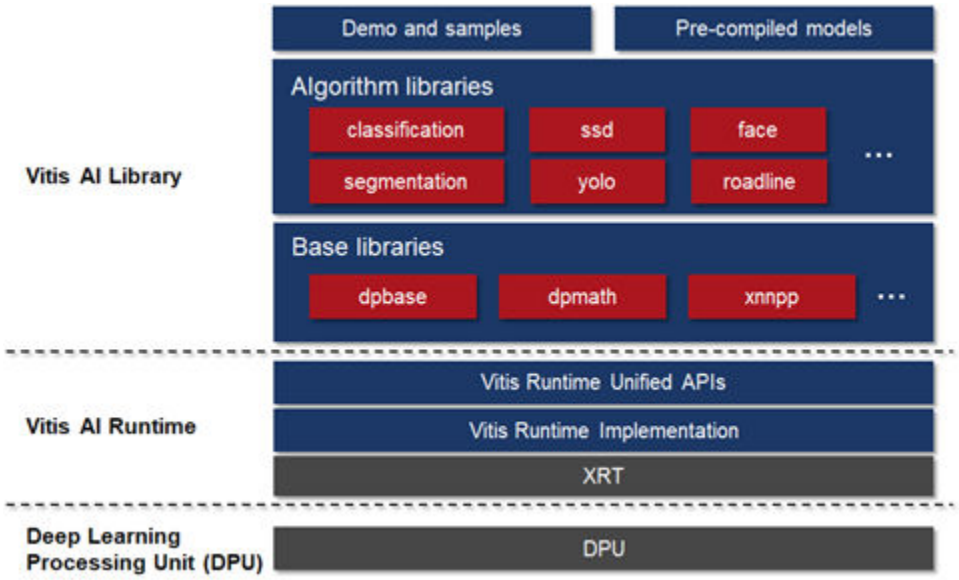


AI Library

The Vitis AI Library is a set of high-level libraries and APIs built for efficient AI inference with Deep-Learning Processor Unit (DPU). It is built based on the Vitis AI Runtime with Vitis Runtime Unified APIs. And It fully supports XRT.

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 the development of their applications, rather than the underlying hardware.

Figure 8: **AI Library**



AI Runtime

Vitis AI Run time enables applications to use the unified high-level runtime API for both cloud and edge. Therefore, making cloud-to-edge deployments seamless and efficient.

The Vitis AI Runtime API features are:

- 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

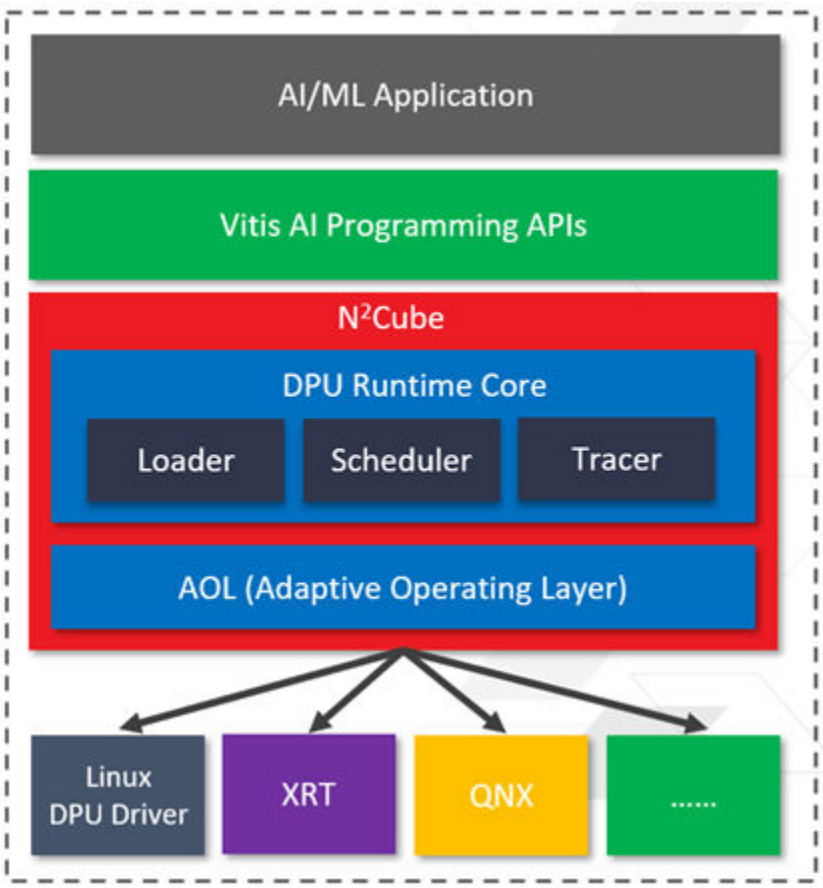
The cloud accelerator has multiple independent Compute Units (CU) that can be programmed to each work on a different AI model, or to work on the same AI model for maximum throughput.

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 Compute Unit or a single FPGA, and the AI resource manager returns a free resource compatible with the user's request. The AI resource manager works with Docker containers, as well as multiple users and multiple tenants on the same host.

For Edge

For edge DPU, the framework of runtime (called as N^{2}Cube) is shown in the following figure. For Vitis AI release, N^{2}Cube is based on the Xilinx Run time (XRT). For legacy Vivado® based DPU, it interacts with the underlying Linux DPU driver (instead of XRT) for DPU scheduling and resource management.

Figure 9: **MPSoc Runtime Stack**



N^{2}Cube runtime offers a comprehensive set of advanced C++/Python programming interface to flexibly meet the diverse requirements for edge scenarios. Refer to Chapter 9 for more details about edge DPU advanced programming. The highlights for N2Cube are listed as follows:

- Support multi-threading and multi-process DPU application deployment.
- Support 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 cores affinity over DPU tasks at run-time.
- Priority based DPU tasks scheduling while adhering to DPU cores affinity.

- Optimized memory usage via DPU code and parameter sharing within multi-threading DPU application.
- Easily adaptive to any POSIX-compliant OS or RTOS (Real-Time Operating System) environment, such as QNX, VxWorks, Integrity.
- Ease-of-use capabilities for DPU debugging and performance profiling.

Currently, N^{2}Cube officially supports three operating environments, including Linux, Xilinx XRT, and BlackBerry QNX RTOS. You can contact theXilinx representatives to acquire Vitis AI package for QNX or to port N^{2}Cube to other third party RTOSs.

Deep-Learning Processor Unit

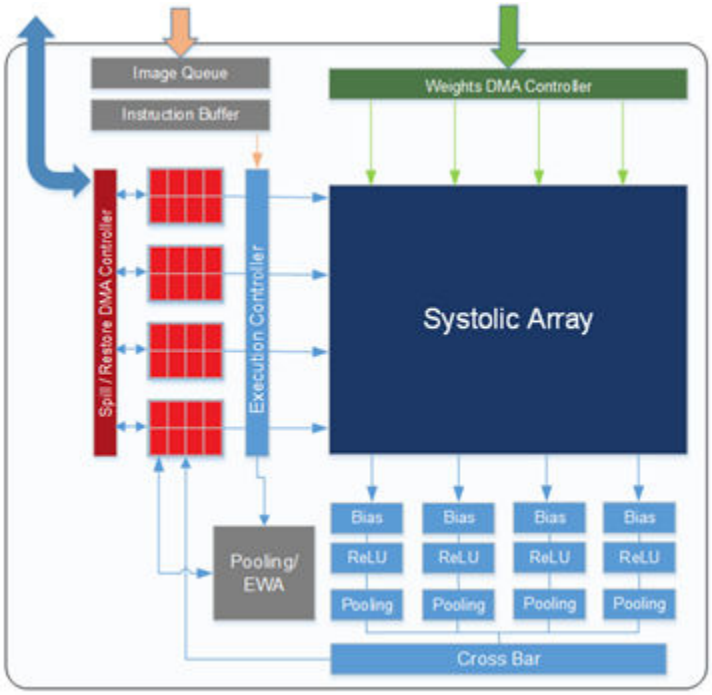
The DPU is designed to accelerate the computing workloads of deep learning inference algorithms widely adopted in various computer vision applications, such as image/video classification, semantic segmentation, and object detection/tracking.

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, and Zynq® UltraScale+™ MPSoCs from edge to cloud to meet the requirements of many diverse applications.

DPU-V1 for Cloud

DPU-V1 (previously known as xDNN) IP cores are high performance general CNN processing engines (PE).

Figure 10: **DPU-V1 Architecture**



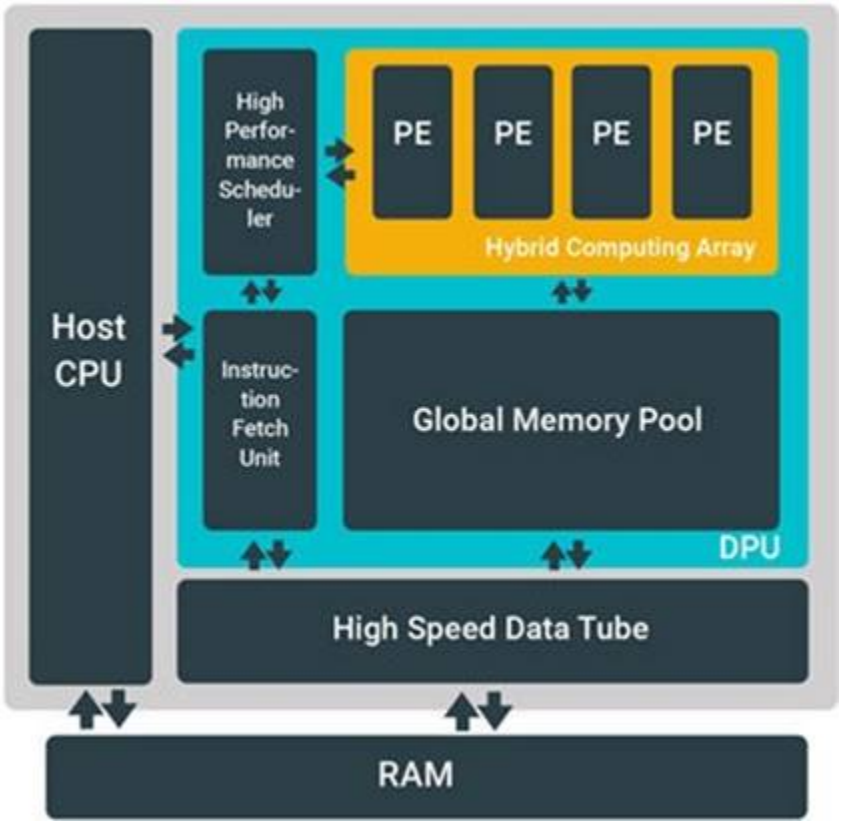
The key features of this engine are:

- 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

DPU-v2 for Edge

The DPU-v2 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 the enabled features. For more information, see the *Zynq DPU v3.1 IP Product Guide* (PG338).

Figure 11: **DPU-V2 Architecture**



Vitis AI Components

Vitis AI 1.0 release uses container technology to distribute the AI software. The release consists of the following components.

- Tools Container
- Run-time container for MPSoC
- Public Github for Examples (https://github.com/Xilinx/Vitis-AI

- Vitis AI Model Zoo (https://github.com/Xilinx/Vitis-AI/tree/master/AI-Model-Zoo)

Tools Container

The tools container consists of the following:

- Containers distributed via dockerhub: https://hub.docker.com/r/xilinx/vitis-ai/tags
- Unified Compiler flow includes
  - Compiler flow for DPUV2 (embedded)
  - Compiler flow for DPUV1 (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 flow
- Alveo Runtime tools

Runtime Container for MPSoC Devices

- Container Path URL: https://www.xilinx.com/bin/public/openDownload?filename=vitis-ai-runtime-1.0.1.tar.gz
- Contents
  - PetaLinux SDK and Cross compiler tool chain
  - VAI board packages based on 2019.2 release
  - Samples for running different network
- Models and overlaybins at https://www.xilinx.com/vitis-ai:
  - All public pre-trained models
  - All Alveo overlay bins
  - This is not visible to users and it is expected examples requiring pre-trained models have scripts for pulling them.

System Requirements

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

Table 1: **System Requirements**

Component	Requirement
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
CPU	Intel® Xeon® Gold 6252 CPU @ 2.10 GHz
GPU (Optional to accelerate quantization)	NVIDIA P100 or V100
CUDA Driver (Optional to accelerate quantization)	nvidia-410
FPGA	Xilinx Alveo U200 or U250
Docker Version	19.03.1

Quick Start

Downloading Vitis AI Development Kit

The Vitis™ AI software is made available via docker hub (https://hub.docker.com/r/xilinx/vitis-ai/tags). Vitis AI consists of the following two docker images:

- xilinx/vitis-ai:tools-1.0.0-cpu
- xilinx/vitis-ai:runtime-1.0.0-cpu

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

The Xilinx FPGA devices and evaluation boards supported by Vitis AI development kit v1.0 release are:

- Cloud: Xilinx Alveo™ cards U200, U250
- Edge: Xilinx MPSoC evaluation boards ZCU102, ZCU104

Setting Up the Host

Use the following steps to set up the host:

1. Clone the Vitis AI repository:

```
git clone https://github.com/xilinx/vitis-ai
```

2. Install the Docker, and add the user to the docker group. Link the user to docker installation instructions from the following docker's website:
   - https://docs.docker.com/install/linux/docker-ce/ubuntu/
   - https://docs.docker.com/install/linux/docker-ce/centos/
   - https://docs.docker.com/install/linux/linux-postinstall/

3. Any GPU instructions will have to be separated from Vitis AI.
4. Set up Vitis AI to target Alveo cards (Only for systems with Alveo cards). To target Alveo cards with Vitis-AI for machine learning workloads, you must install the following software components:
   - Xilinx Runtime (XRT)
   - Alveo Deployment Shells (DSAs)
   - Xilinx Resource Manager (XRM) (xbutler)
   - Xilinx Overlaybins (Accelerators to Dynamically Load – binary programming files)

   While it is possible to install all of these software components individually, a script has been provided to automatically install them at once. To do so:

   a. Run the following commands:

   ```
   cd Vitis-AI/alveo/packages
   sudo su
   ./install.sh
   ```

   b. Power cycle the system.
5. Start the Docker Container.
   a. Change directories to Vitis AI:

   ```
   cd Vitis-AI/
   ```

   b. Run one of the following command sets.
      - For a CPU tools container:

      ```
      ./docker_run.sh xilinx/vitis-ai:1.0.0-cpu
      ```

      - For a GPU-enabled tools container:

      ```
      cd Vitis-AI/docker
        ./docker_build.sh
        cd Vitis-AI
        ./docker_run.sh xilinx/vitis-ai:runtime-1.0.0-gpu
      ```

      - For MPSoC Runtime tools container:

      ```
      ./docker_run.sh xilinx/vitis-ai:runtime:1.0.0-cpu
      ```

   c. Upon starting the container your current working directory will be mounted to: /workspace

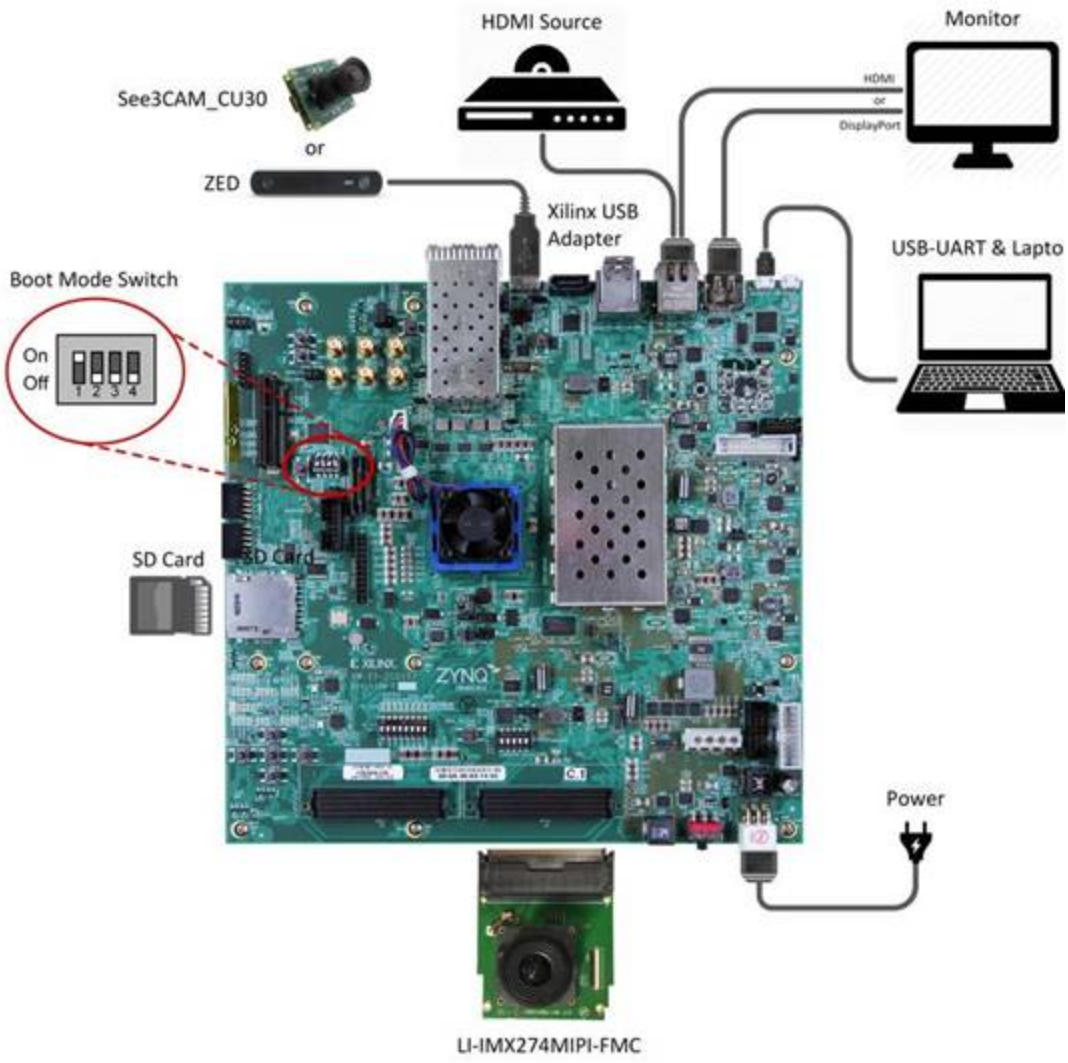
Setting Up the Evaluation Board

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.

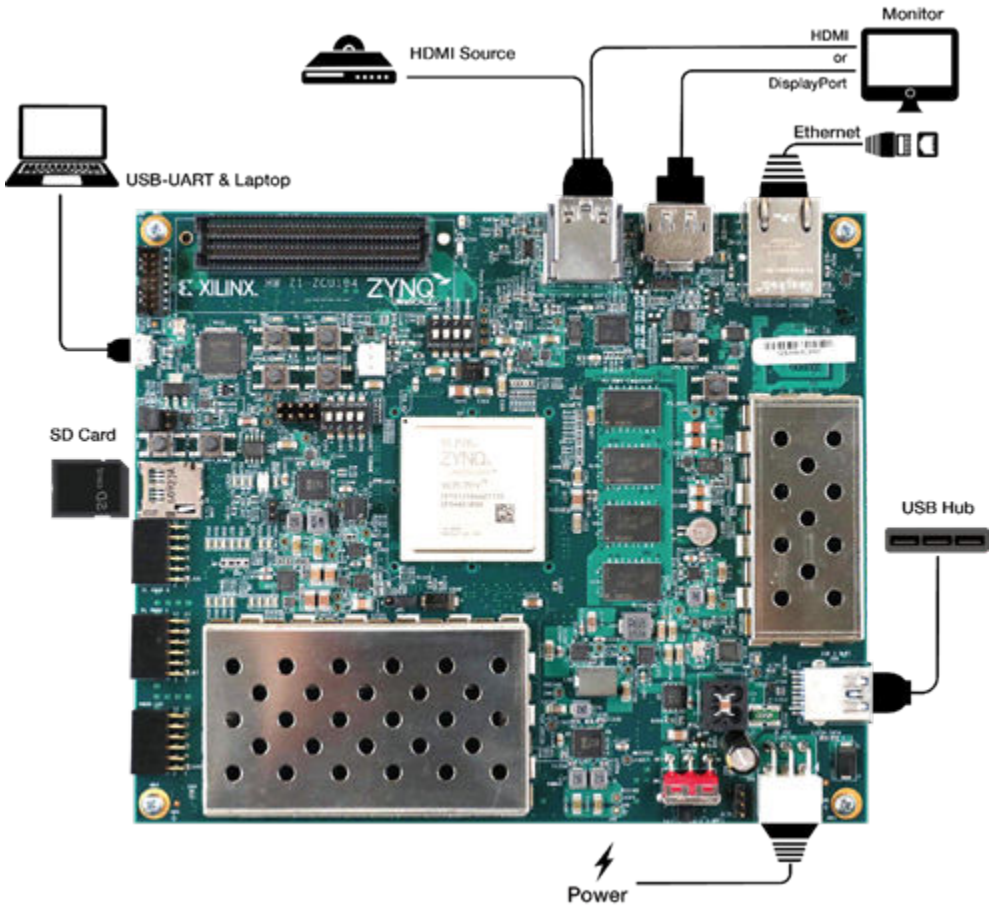
Figure 12: **Xilinx ZCU102 Evaluation Board and Peripheral Connections**



The Xilinx ZCU104 evaluation board uses the mid-range 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 13: **Xilinx ZCU104 Evaluation Board and Peripheral Connections**

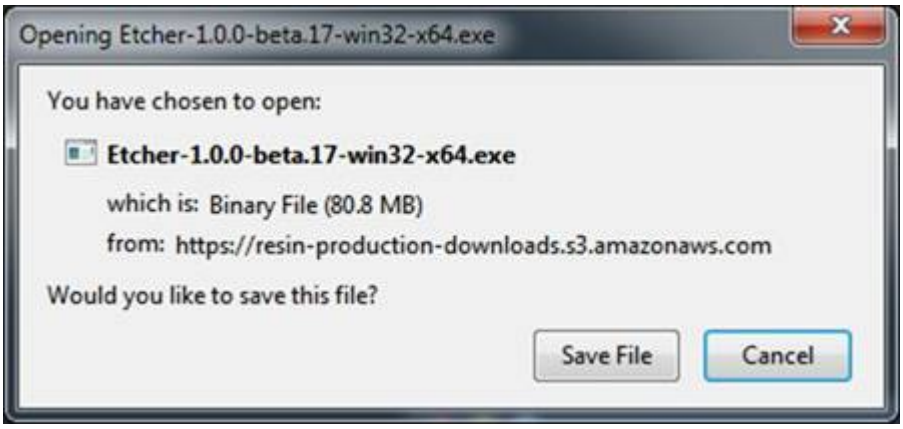


In the following sections, ZCU102 is used as an example to show the steps to setup the Vitis AI running environment on evaluation boards.

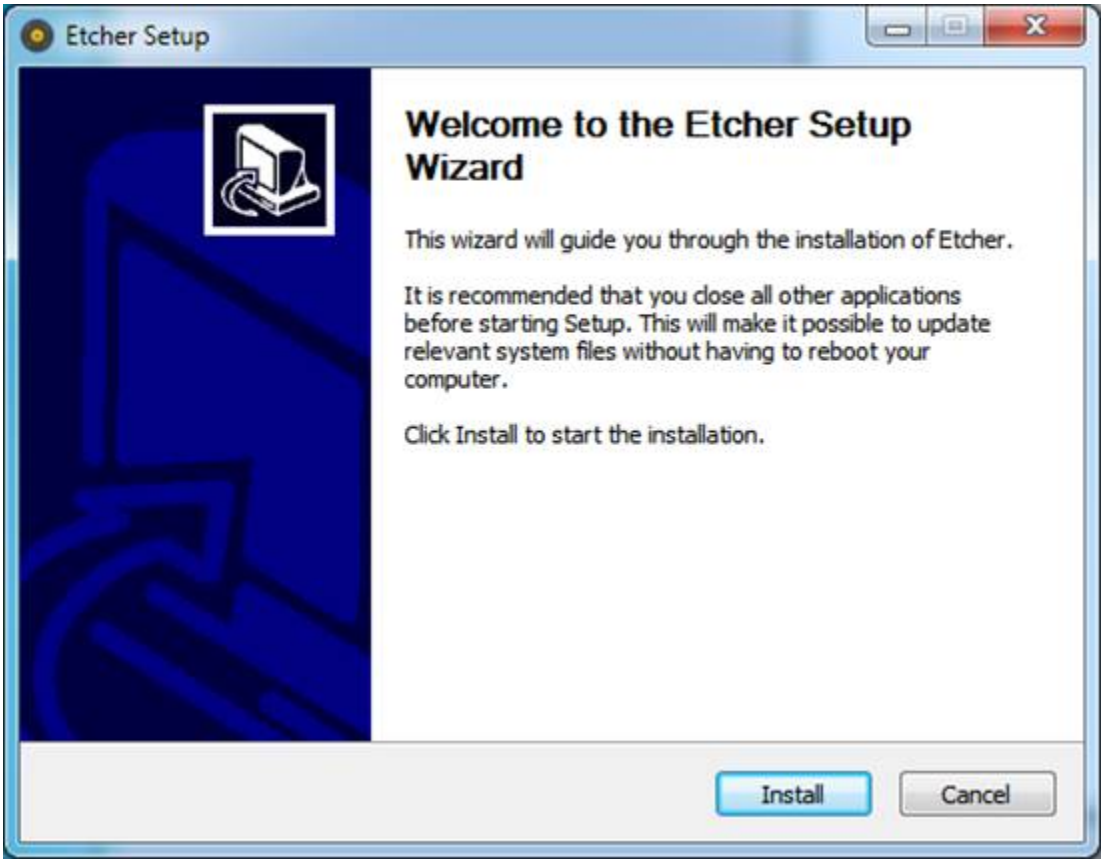
Flashing the OS Image to the SD Card

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.

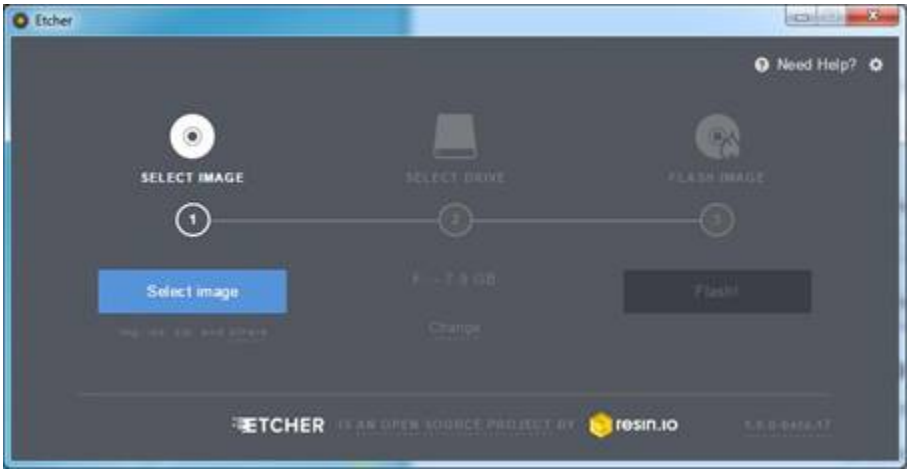


2. Install Etcher, as shown in the following figure.

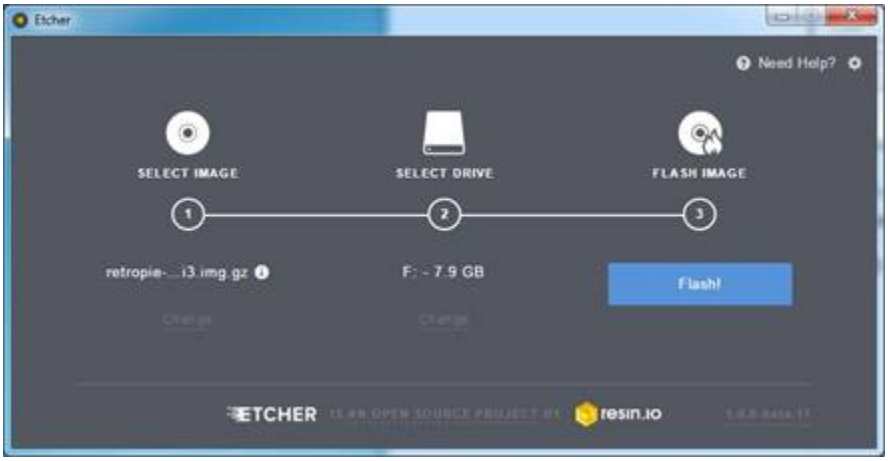


3. Eject any external storage devices such as USB flash drives and backup hard disks. This makes it easier to identify the SD card. Then, insert the SD card into the slot on your computer, or into the reader.
4. Run the Etcher program by double clicking on the Etcher icon shown in the following figure,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

Etcher launches, as shown in the following figure.



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



Booting the Evaluation Board

This example uses a ZCU102 board to illustrate how to boot a Vitis AI evaluation board. Follow 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:

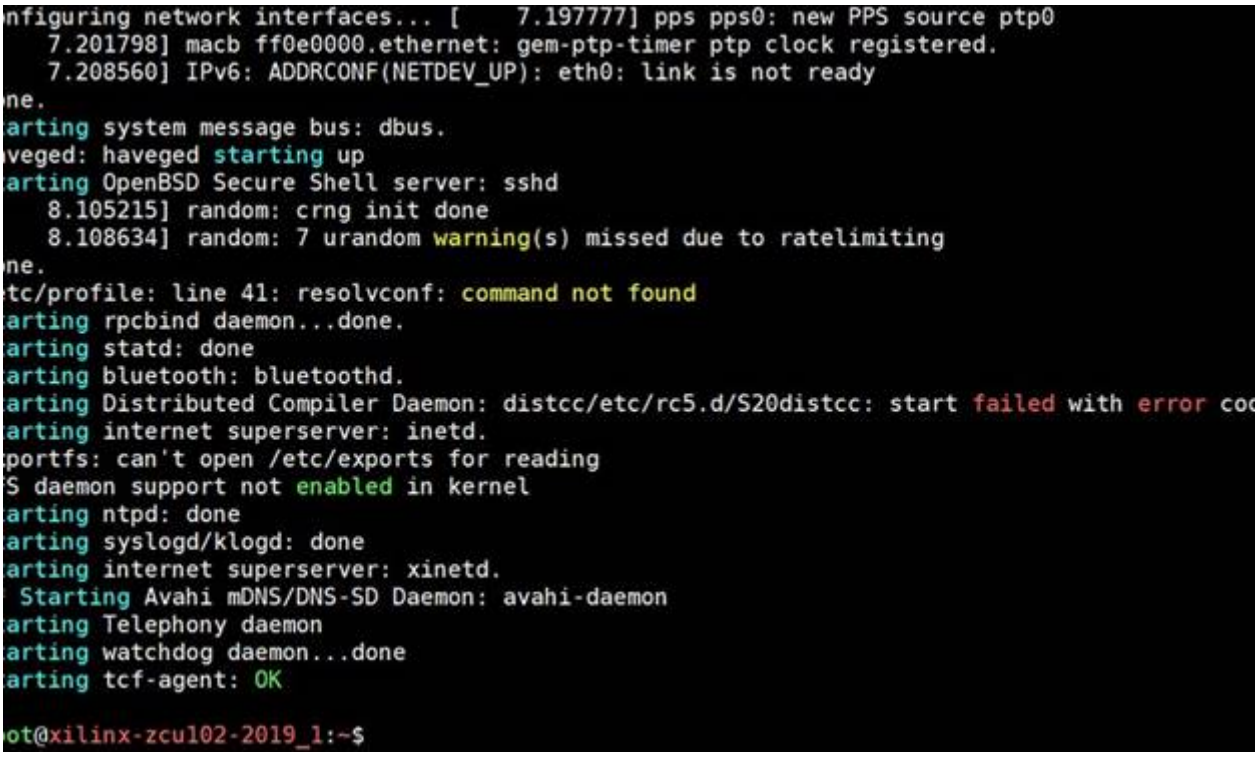
- UART port
- Ethernet connection
- Standalone

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. A screenshot of a sample boot is shown in the following figure. Login into the system with username “root” and password “root”.

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

Figure 14: **Example of Boot Process**



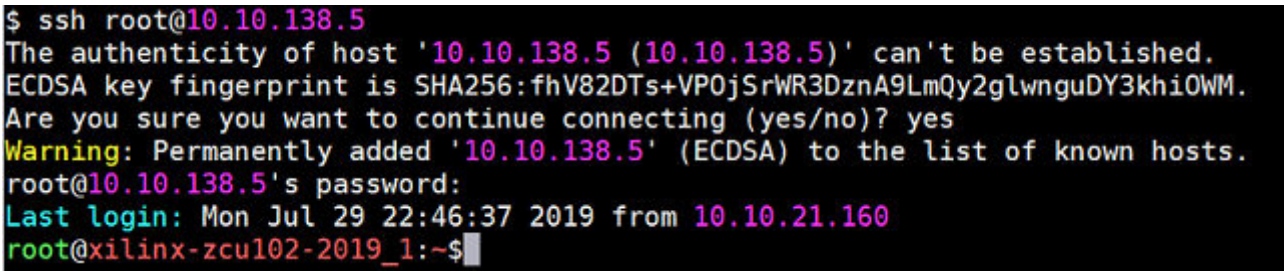
***Note*:** On a Linux system, you can use Minicom to connect to the target board directly; for a Windows system, a USB to UART driver is needed before connecting to the board through a serial port.

Using the Ethernet Interface

There are three ways to access the ZCU102 board: via UART, ethernet, or standalone.

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.

Figure 15: **Logging into the Evaluation Board Using SSH**



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.

Figure 16: **Standalone Linux GUI Screen**

Installing Vitis AI Package on the Evaluation Board

With an Ethernet connection established, you can copy the Vitis AI installation package from docker image `vitis-ai-docker-runtime` to the evaluation board and set up Vitis AI running environment for the ZCU102 board.

Run the following command to remote copy the package from host to ZCU102 board with IP address 192.168.0.10:

```
scp -r /opt/vitis_ai/xilinx_vai_board_package root@192.168.0.10:~/
```

On the ZCU102 board, change to the `~/xilinx_vai_board_package/` directory and run `install.sh`. The Vitis AI runtime and utility tools will be installed into system automatically. You can now copy Vitis AI samples from docker image `vitis-ai-docker-runtime` to the evaluation board for evaluation.

Running Examples

For Vitis AI development kit v1.0 release, there are two kinds of examples distributed together. Vitis AI examples demonstrate the using of Vitis AI unified high-level C++/Python APIs, which are available across cloud-to-edge. And the other kind of examples come from DNNDK and demonstrate the usage of Vitis AI advanced low-level C++/Python APIs, which is only available for edge DPU. These samples can be found at https://github.com/xilinx/vitis-ai. The `/alveo` folder contains the sample for DPU-v1 on Alveo platform, and the folder mpsoc contains the samples for edge DPU on ZCU102 and ZCU104 boards

If you are using Xilinx ZCU102 and ZCU104 boards to run samples, make sure to enable X11 forwarding with 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 since all the examples require Linux windows system to work properly.

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

Vitis AI Examples

Vitis AI provides several C++ and Python examples to demonstrate the use of the unified cloud-edge runtime programming APIs. The following table below describes these Vitis AI examples.

Table 2: **Vitis AI Examples**

ID	Example Name	Models	Framework	Notes
1	resnet50	ResNet50	Caffe	Image classification with Vitis AI unified C++ APIs.
2	resnet50_mt_py	ResNet50	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	YOLO-v3	Caffe	ADAS detection with Vitis AI unified C++ APIs.
7	segmentation	FPN	Caffe	Semantic segmentation with Vitis AI unified C++ APIs.

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

```
auto runners = vitis::ai::DpuRunner::create_dpu_runner("vitis_rundir");
auto runner = runners[0];
// populate input/output tensors
auto job_data = runner->execute_async(inputs, outputs);
runner->wait(job_data.first, -1);
// process outputs
```

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

```
runner = Runner('vitis_rundir')
# populate input/output tensors
            jid = runner.execute_async(fpgaInput, fpgaOutput)
runner.wait(jid)
# process fpgaOutput
```

For Edge

Vitis AI samples can be found in the following locations:

- **ZCU102 board samples:** https://github.com/Xilinx/Vitis-AI/tree/master/mpsoc/vitis_ai_samples_zcu102
- **ZCU104 board samples:** https://github.com/Xilinx/Vitis-AI/tree/master/mpsoc/vitis_ai_samples_zcu104

After downloading the samples, copy them into your `/workspace/sample/` folder within the runtime container. All these samples can be cross-compiled under runtime container on an X86 host machine. There is a folder dpuv2_rundir existing in each sample. It is used while invoking Vitis AI API `vitis::ai::DpuRunner::create_dpu_runner()` to create DpuRunner. Refer to Programming with Vitis AI for more details.

Before running Vitis AI samples of edge DPU, you must prepare several images used by the image classification samples, such as ResNet50 and Inception-v1. The images can be downloaded from the ImageNet dataset. It is better to scale them with the same resolution 640x480, and then place them under the appropriate folders within the runtime container:

- `/workspace/sample/vitis_ai_samples_zcu102/images/`
- `/workspace/sample/vitis_ai_samples_zcu104/images/`

In the following example, let us use ZCU102 board as the reference. After all the samples are built by Arm GCC cross-compilation toolchains within runtime docker container, copying the whole directory `/workspace/sample/vitis_ai_samples_zcu102/` to ZCU102 board directory `/home/root/` is recommend. The users can choose to copy one single DPU hybrid executable from docker container to the evaluation board for running. Pay attention that the dependent image folder `dataset` or video folder `video` should be copied together, and the folder structures should also be kept as expected.

The launching command for each sample is listed in the following table. For Python samples, note that the absolute path for dpuv2_rundir should be specified.

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

ID	Example Name	Command
1	resnet50	./resnet50 dpuv2_rundir
2	resnet50_mt_py	python3 resnet50.py 3 /home/root/vitis_ai_samples_zcu102/resnet50_mt_py/dpuv2_rundir/
3	inception_v1_mt_py	python3 inception_v1.py 3 /home/root/vitis_ai_samples_zcu102/inception_v1_mt_py/dpuv2_rundir/
4	pose_detection	./pose_detection video/pose.mp4 dpuv2_rundir
5	video_analysis	./video_analysis video/structure.mp4 dpuv2_rundir
6	adas_detection	./adas_detection video/adas.avi dpuv2_rundir
7	segmentation	./segmentation video/traffic.mp4 dpuv2_rundir

For Cloud

Vitis AI flow defines common runtime API for edge and cloud DPU. All of the above examples will work for the Alveo flow and the instructions to run them can be found at `/workspace/alveo/examples/vitis_ai_alveo_samples`.

Legacy DNNDK Examples

To keep forward compatibility, Vitis AI still supports the application of DNNDK for deep learning applications development over edge DPU. The legacy DNNDK C++/Python examples can be found at the following locations:

- **ZCU102 examples:** https://github.com/Xilinx/Vitis-AI/tree/master/mpsoc/dnndk_samples_zcu102
- **ZCU104 examples:** https://github.com/Xilinx/Vitis-AI/tree/master/mpsoc/dnndk_samples_zcu104

After downloading the samples, copy them into the `/workspace/sample/` folder within the runtime container. These examples can be built with Arm GCC cross-compilation toolchains.

The following table briefly describes all DNNDK examples.

Table 4: **DNNDK Examples**

ID	Example Name	Models	Framework	Notes
1	resnet50	ResNet50	Caffe	Image classification with Vitis AI advanced C++ APIs.

Table 4: **DNNDK Examples** *(cont'd)*

ID	Example Name	Models	Framework	Notes
2	resnet50_mt	ResNet50	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
3	tf_resnet50	ResNet50	TensorFlow	Image classification with Vitis AI advanced Python APIs.
4	mini_resnet_py	Mini-ResNet	TensorFlow	Image classification with Vitis AI advanced Python APIs.
5	inception_v1	Inception-v1	Caffe	Image classification with Vitis AI advanced C++ APIs.
6	inception_v1_mt	Inception-v1	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
7	inception_v1_mt_py	Inception-v1	Caffe	Multi-threading image classification with Vitis AI advanced Python APIs.
8	mobilenet	MiblieNet	Caffe	Image classification with Vitis AI advanced C++ APIs.
9	mobilenet_mt	MiblieNet	Caffe	Multi-threading image classification with Vitis AI advanced C++ APIs.
10	face_detection	DenseBox	Caffe	Face detetion with Vitis AI advanced C++ APIs.
11	pose_detection	SSD, Pose detection	Caffe	Pose detection with Vitis AI advanced C++ APIs.
12	video_analysis	SSD	Caffe	Traffic detection with Vitis AI advanced C++ APIs.
13	adas_detection	YOLO-v3	Caffe	ADAS detection with Vitis AI advanced C++ APIs.
14	segmentation	FPN	Caffe	Semantic segmentation with Vitis AI advanced C++ APIs.
15	split_io	SSD	TensorFlow	DPU split IO memory model programming with Vitis AI advanced C++ APIs.
16	debugging	Inception-v1	TensorFlow	DPU debugging with Vitis AI advanced C++ APIs.

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

Table 5: **Images Preparation for DNNDK Samples**

ID	Image Directory	Note
1	dnndk_samples_zcu102/dataset/image500_640_480/	Download several images from ImageNet dataset and scale to the same resolution 640*480.
2	dnndk_samples_zcu104/dataset/image500_640_480/	Download several images from ImageNet dataset and scale to the same resolution 640*480.
3	dnndk_samples_zcu102/ image_224_224/	Download one image from ImageNet dataset and scale to resolution 224*224.
4	dnndk_samples_zcu104/ image_224_224/	Download one image from ImageNet dataset and scale to resolution 224*224.
5	dnndk_samples_zcu102/ image_32_32/	Download several images from CIFAR-10 dataset https://www.cs.toronto.edu/~kriz/cifar.html.
6	dnndk_samples_zcu104/ image_32_32/	Download several images from CIFAR-10 dataset https://www.cs.toronto.edu/~kriz/cifar.html
7	dnndk_samples_zcu102/resnet50_mt/image/	Download one image from ImageNet dataset.
8	dnndk_samples_zcu102/ mobilenet_mt/image/	Download one image from ImageNet dataset.
9	dnndk_samples_zcu102/ inception_v1_mt/image/	Download one image from ImageNet dataset.
10	dnndk_samples_zcu102/ debugging/decent_golden/dataset/images/	Download one image from ImageNet dataset.
11	dnndk_samples_zcu104/resnet50_mt/image/	Download one image from ImageNet dataset.
12	dnndk_samples_zcu104/ mobilenet_mt/image/	Download one image from ImageNet dataset.
13	dnndk_samples_zcu104/ inception_v1_mt/image/	Download one image from ImageNet dataset.
14	dnndk_samples_zcu104/ debugging/decent_golden/dataset/images/	Download one image from ImageNet dataset.

This subsequent section illustrates how to run DNDNK examples, using the ZCU102 board as the reference as well. The samples stay under the directory `/workspaces/sample/dnndk_samples_zcu102/`. After all the samples are built by Arm GCC cross-compilation toolchains within runtime container, it is recommended to copy the whole directory `/workspaces/sample/dnndk_samples_zcu102/` to ZCU102 board directory /home/root/. The users can choose to copy one single DPU hybrid executable from docker container to the evaluation board for running. Pay attention that the dependent image folder dataset or video folder video should be copied together, and the folder structures should also be kept as expected.

For the sake of simplicity, the directory of `/home/root/dnndk_samples_zcu102/` 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 ADAS (Advanced Driver Assistance Systems) application using YOLO-v3 network model is located under the directory `$dnndk_sample_base/adas_detection`. 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.mp4 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.

Different from C++ examples, Incpetion-v1 model is compiled to DPU ELF file first and then it is transformed into the DPU shared library `libdpumodelinception_v1.so` with the following command on the evaluation board. dpu_inception_v1_*.elf means to include all DPU ELF files in case that Incpetion-v1 is compiled into several DPU kernels by VAI_C compiler. Refer to the section of DPU Shared Library for more details.

```
aarch64-linux-gnu-gcc -fPIC -shared \
dpu_inception_v1_*.elf -o libdpumodelinception_v1.so
```

Within the Vitis AI runtime container, use the following command for this purpose instead.

```
aarch64-linux-gnu-gcc \
--sysroot=/opt/vitis_ai/petalinux_sdk/sysroots/aarch64-xilinx-linux \
-fPIC -shared dpu_inception_v1_*.elf -o libdpumodelinception_v1.so
```

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

minResNet with Python

`$dnndk_sample_base/mini_resnet_py` contains the image classification example of TensorFlow minResNet 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 CV 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.

Support

You can visit the Vitis AI forum on the Xilinx website https://forums.xilinx.com/t5/ Deephi-DNNDK/bd-p/Deephi and https://forums.xilinx.com/t5/Xilinx-ML-Suite/bd-p/ML for topic discussions, knowledge sharing, FAQs, and requests for technical support.

Model Deployment Overview

There are two stages for developing deep learning applications: training and inference. The training stage is used to design a neural network for a specific task (such as image classification) using a huge amount of training data. The inference stage involves the deployment of the previously designed neural network to handle new input data not seen during the training stage.

The Vitis™ AI toolchain provides an innovative workflow to deploy deep learning inference applications on the DPU with the following four steps, which are described in this chapter using ResNet-50.

1. Quantize the neural network model.
2. Compile the neural network model.
3. Program with Vitis AI programming interface.
4. Run and evaluate the deployed DPU application.

Model Quantization

vai_q_caffe and vai_q_tensorflow are the binary names of our Vitis AI quantizer, where 'q' stands for quantizer and caffe/tensorflow are the framework names. This section helps you to quantize a Resnet-50 model quickly. Refer to Chapter 4: Model Quantization for a full introduction of the VAI quantizer.

Caffe Version

vai_q_caffe takes a floating-point model as an input model and uses a calibration dataset to generate a quantized model. Use the following steps to create and quantize Resnet50 floating-point model.

1. Prepare a floating-point model for Resnet-50. You can download one from Internet or from Xilinx modelzoo (https://github.com/Xilinx/Vitis-AI/tree/master/AI-Model-Zoo).
2. Prepare the calibration dataset used by vai_q_caffe. You can download 100 to 1000 images of ImageNet dataset from http://academictorrents.com/collection/imagenet-2012 or http://www.image-net.org/download.php and then change the settings for the source and root_folder of image_data_param in ResNet-50 prototxt accordingly.

3. Activate the caffe running environment:

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

4. Start quantization:

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

This invokes the vai_q_caffe tool to perform quantization with the appropriate parameters. The running time of this command varies from a few seconds to several minutes, depending on hardware and the size of the neural network. Four files are generated in the output directory, including `deploy.prototxt` and `deploy.caffemodel`, which could be fed to VAI compiler for the following compilation process.

TensorFlow Version

Use the following the steps to run vai_q_tensorflow.

1. Prepare floating-point frozen model and dataset.

Table 6: **Input Files for vai_q_tensorflow**

No.	Name	Description
1	frozen_graph	Frozen Resnet-50 model.
2	calib_images	Before launching quantization for ResNet-50, prepare the calibration dataset. You can download 100 to 1000 images of ImageNet dataset from http://academictorrents.com/collection/imagenet-2012 or http://www.image-net.org/ download.php and then change the calibration dataset path in the input_fn.
3	input_fn	A python function to read images in calibration dataset and complete the pre-process instructions.

Input files for vai_q_tensorflow is shown in the above table. The frozen model can be downloaded from Xilinx modelzoo (https://github.com/Xilinx/AI-Model-Zoo). Some quantization and evaluation scripts can also be found in modelzoo. Enter a script to demonstrate how to write input_fn. The following figure shows the `resnet_v1_50_input_fn.py` script. As shown in the figure, the codes `central_crop` and `mean_subtraction` are used for pre-processing.

```
calib_image_dir = "../../calibration_data/images/"
calib_image_list = "../../calibration_data/calib.txt"
calib_batch_size = 50
def calib_input(iter):
  images = []
  line = open(calib_image_list).readlines()
  for index in range(0, calib_batch_size):
    curline = line[iter * calib_batch_size + index]
    calib_image_name = curline.strip()
    image = cv2.imread(calib_image_dir + calib_image_name)
    image = central_crop(image, 224, 224)
    image = mean_image_subtraction(image, MEANS)
    images.append(image)
  return {"input": images}
```

2. Run vai_q_tensorflow to quantize the TensorFlow frozen models.

```
vai_q_tensorflow quantize \
  --input_frozen_graph frozen_resnet_v1_50.pb \
  --input_nodes input \
  --input_shapes ?,224,224,3 \
  --output_nodes resnet_v1_50/predictions/Reshape_1 \
  --input_fn resnet_v1_50_input_fn.calib_input \
  --method 1 \
  --gpu 0 \
  --output_dir ./quantize_results \
```

The script may take several minutes to finish. Once the quantization is done, the quantization summary will be displayed as shown in the following figure:

```
INFO: Done Calibration
INFO: Start Generate Deploy Model
INFO: End Generate Deploy Model
********************* Quantization Summary *********************
INFO: Output:
  quantize_eval_model: ./quantize_results/quantize_eval_model.pb
  deploy_model: ./quantize_results/deploy_model.pb
```

Two files will be generated in `quantize_results` directory. The `deploy_model.pb` could be fed to VAI compile for the following compilation processes. The `quantize_eval_model.pd` can be used for model evaluation and dump.

Model Compilation

The Vitis AI compiler VAI_C supports both Caffe and TensorFlow model compilation. Before applying VAI_C to build Caffe or TensorFlow models, you should run the `conda activate vitis-ai-caffe` command or the `conda activate vitis-ai-tensorflow` to activate the Conda environment for the Vitis AI tools.

The commands for compiling Caffe/TensorFlow ResNet50 with VAI_C for edge DPU of ZCU102 board are:

```
/vai_c_caffe --prototxt ./deploy.prototxt --caffemodel ./deploy.caffemodel
--arch /opt/vitis_ai/compiler/arch/dpuv2/ZCU102/ZCU102.json --output_dir
model --net_name restnet50

/opt/vitis_ai/compiler/vai_c_tensorflow --frozen_pb ./deploy.pb --arch /opt/
vitis_ai/compiler/arch/dpuv2/ZCU102/ZCU102.json --output_dir model --
net_name resnet50_tf
```

The option `--arch` for vai_c_caffe and vai_c_tensorflow indicates the DPU architecture configuration in which the JSON file is used. For ZCU104 board, the architecture configuration JSON file is `/opt/vitis_ai/compiler/arch/dpuv2/ZCU104/ZCU104.json`. For the other specified options, see Chapter 5: Vitis AI Compiler.

Programming with Vitis AI

The Vitis AI Development Kit offers a unified set of high-level C++/Python programming APIs to smooth the machine learning applications development across Xilinx® cloud-to-edge devices, including DPUv1 and DPUv3 for Alveo™, and DPUv2 for Zynq® devices. It brings the benefits to easily port deployed DPU applications from cloud to edge or vice versa. You can refer to Vitis AI samples within docker image of `vitis-ai-docker` and `vitis-ai-docker-runtime` to get familiar with the usage of unified programming APIs.

For edge DPU, Vitis AI additionally provides the advanced low-level C++/Python programming APIs, which originate from DNNDK (Deep Neural Network Development Kit) programming interface. It consists of a comprehensive set of APIs that can flexibly meet the diverse requirements under various edge scenarios. Meanwhile, these advanced APIs bring forward compatibility so that the DNNDK legacy projects can be ported to Vitis platform without any modifications to the existing source code. You can refer to the DNNDK samples within docker image of `vitis-ai-docker-runtime` to get familiar with the usage of advanced programming APIs.

Using Unified APIs

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);
```

1. Submit input tensors for execution, and output tensors to store results. The host pointer is passed via 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 input tensors it expects for its loaded AI model.

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

4. Query the DpuRunner for the shape and name of the output tensors it expects for its loaded AI model.
5. To create a DpuRunner object call the following:

```
DpuRunner::create_dpu_runner(const std::string& model_directory);
```

   which returns

```
std::vector<std::unique_ptr<vitis::ai::DpuRunner>>
```

The input to create_dpu_runner is a model runtime directory generated by the AI compiler. The directory contains a meta.json that distinguishes each directory for each Vitis Runner, along with files needed by the Runner at runtime.

C++ Example

```
auto runners = vitis::ai::DpuRunner::create_dpu_runner("vitis_rundir");
auto runner = runners[0];
// populate input/output tensors
auto job_data = runner->execute_async(inputs, outputs);
runner->wait(job_data.first, -1);
// process outputs
We provide a C wrapper for the C++ DpuRunner class:
void* DpuPyRunnerCreate(char* path);
void DpuPyRunnerGetInputTensors(void* runner, DpuPyTensor** tensors, int*
tensor_cnt);
void DpuPyRunnerGetOutputTensors(void* runner, DpuPyTensor** tensors, int*
tensor_cnt);
int DpuPyRunnerGetTensorFormat(void* runner);
int DpuPyRunnerExecuteAsync(void* runner, void** indata, void** outdata,int
batch_sz, int* status);
int DpuPyRunnerWait(void* runner, int jobId);
void DpuPyRunnerDestroy(void* runner);
```

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

```
runner = Runner('vitis_rundir')
# populate input/output tensors
jid = runner.execute_async(fpgaInput, fpgaOutput)
runner.wait(jid)
# process fpgaOutput
```

Using Advanced APIs

For edge DPU, you can utilize Vitis AI unified APIs to develop deep learning applications. In addition, they have another choice to adapt advanced low-level APIs to flexibly meet various scenarios' requirements. For more details on advanced API usage, see Appendix A: Advanced Programming Interface .

For Vitis AI advanced low-level APIs, you need to use the following operations:

1. Call APIs to manage DPU kernels and tasks.
   - DPU kernel creation and destruction
   - DPU task creation and destruction
   - Manipulate DPU input and output tensors
2. Deploy DPU un-supported layers/operators over the CPU side.
3. Implement pre-processing to feed input data to DPU and implement post-processing to consume output data from DPU.

```
int main(void) {
/* DPU Kernel/Task for running ResNet-50 */
DPUKernel* kernel;
DPUTask* task;

/* Attach to DPU device and prepare for running */
dpuOpen();

/* Create DPU Kernel for ResNet-50 */
```

```
kernel = dpuLoadKernel("resnet50");

/* Create DPU Task for ResNet-50 */
task = dpuCreateTask(kernel, 0);

/* Run DPU Task for ResNet-50 */
runResnet50(task);

/* Destroy DPU Task & release resources */
dpuDestroyTask(task);

/* Destroy DPU Kernel & release resources */
dpuDestroyKernel(kernel);

/* Detach DPU device & release resources */
dpuClose();

return 0;
}
```

Use ResNet50 as an example, the code snippet for manipulating the DPU kernels and tasks are programmed within the`main()` function as follows. The operations inside `main()` include:

- Call `dpuOpen()` to open the DPU device.
- Call `dpuLoadKernel()` to load the DPU resnet50 kernel.
- Call `dpuCreateTask()` to create a task for DPU kernel.
- Call `dpuDestroyKernel()` and `dpuDestroyTask()` to destroy the DPU kernel and task and release resources.
- Call `dpuClose()` to close the DPU device.

The image classification takes place within the `runResnet50()` function, which performs the following operations:

1. Fetch an image using the OpenCV function `imread()` and set it as the input to the DPU kernel resnet50 by calling the `dpuSetInputImage2()` for Caffe model. For TensorFlow model, the users should implement the pre-processing (instead of directly using `dpuSetInputImage2()`) to feed input image into DPU.
2. Call `dpuRunTask()` to run the task for ResNet-50 model.
3. Perform softmax calculation on the Arm® CPU with the output data from DPU.
4. Calculate the top-5 classification category and the corresponding probability.

```
Mat image = imread(baseImagePath + imageName);
dpuSetInputImage2(task, INPUT_NODE, image);
dpuRunTask(task);
/* Get FC result and convert from INT8 to FP32 format */
dpuGetOutputTensorInHWCFP32(task, FC_OUTPUT_NODE, FCResult, channel);
CPUCalcSoftmax(FCResult, channel, softmax);
TopK(softmax, channel, 5, kinds);
```

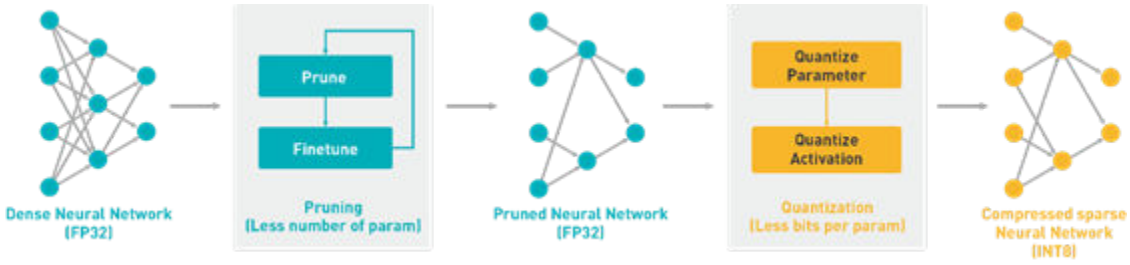
Model Quantization

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 17: **Pruning and Quantization Flow**



Generally, 32-bit floating-point weights and activation values are used when training neural networks. By converting the 32-bit floating-point weights and activations to 8-bit integer (INT8) format, the Vitis AI quantizer can reduce computing complexity without losing prediction accuracy. The fixed-point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model. The Vitis AI quantizer supports common layers in neural networks, such as convolution, pooling, fully connected, and batchnorm.

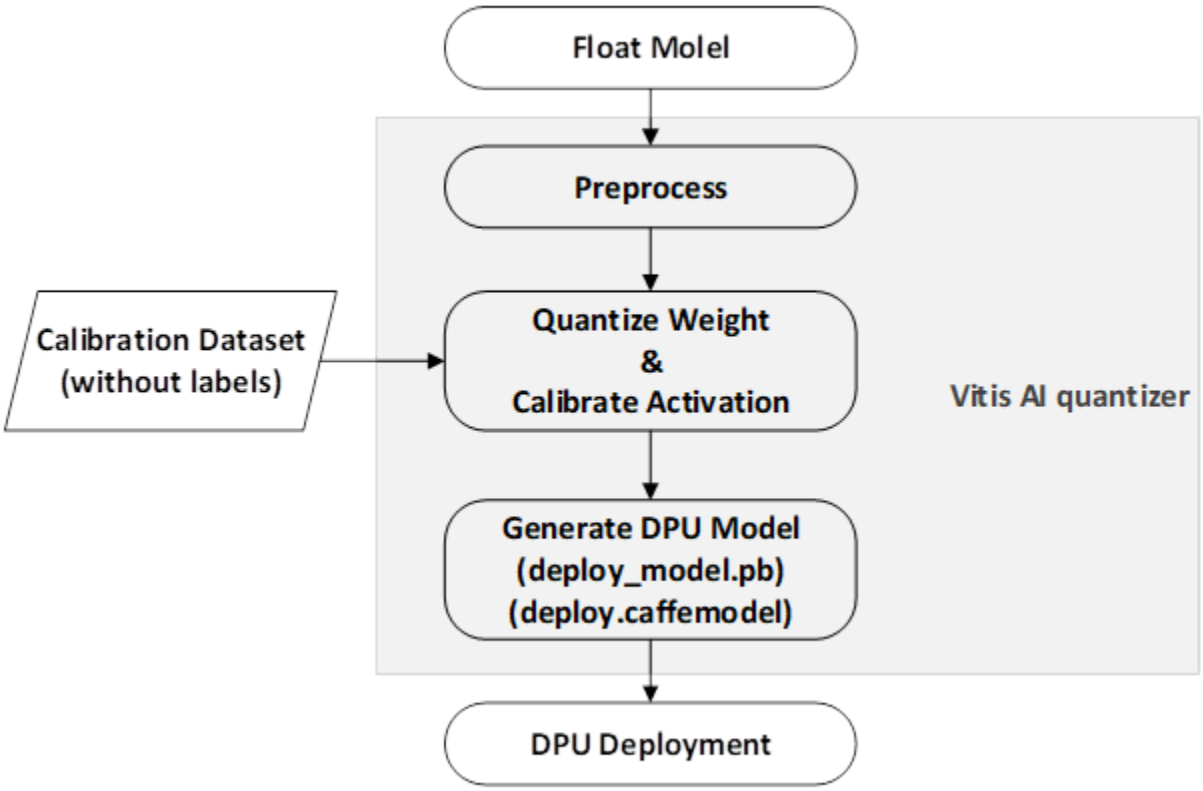
The Vitis AI quantizer now supports TensorFlow and Caffe (the quantizer names are vai_q_tensorflow and vai_q_caffe respectively). The vai_q_tensorflow quantizer is based on Tensorflow 1.12. The vai_q_caffe quantizer supports the quantize finetuning feature, but vai_q_tensorflow does not. The Pytorch version is currently under development.

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 a little decline in accuracy after quantization. However, for some networks such as Mobilenets, 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 train dataset. According to experiments, several epochs of finetuning are needed and the finetune time varies from several minutes to several hours.

Vitis AI Quantizer Flow

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

Figure 18: **VAI Quantizer Workflow**



The Vitis AI quantizer takes a floating-point model as input (prototxt and caffemodel for the Caffe version, and frozen GraphDef file for the TensorFlow version), 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 needs to 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. This is 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 or 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 vision Caffe or TensorFlow framework.

Steps to Run vai_q_caffe

1. Prepare the Neural Network Model

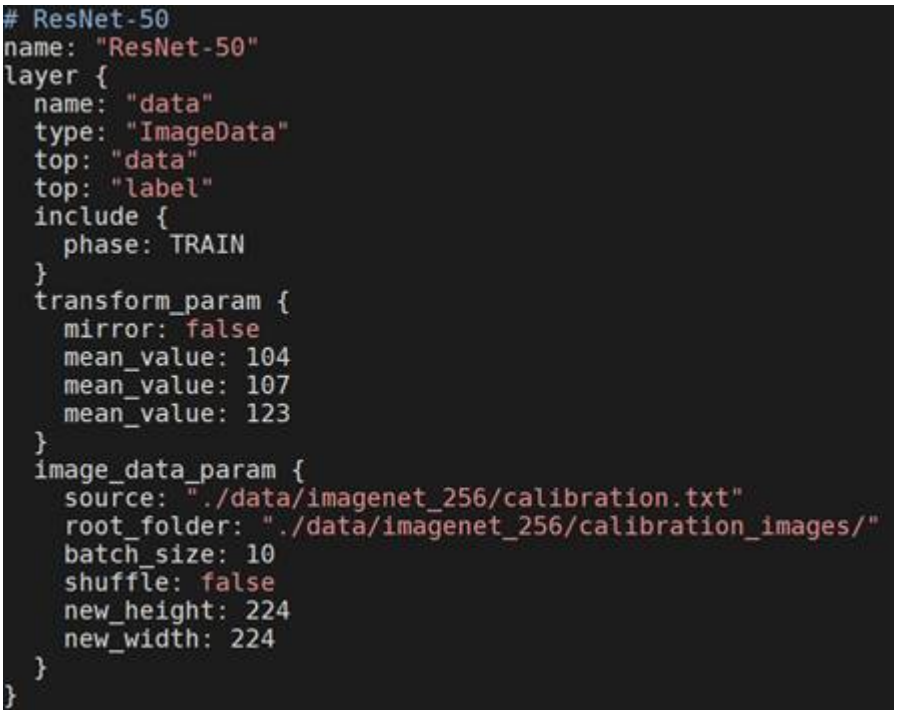
Table 7: **vai_q_caffe Input Files**

No.	Name	Description
1	float.prototxt	Floating-point model for ResNet-50. Thedata 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.

Figure 19: **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*:** Only the 3-mean-value format is supported by vai_q_caffe. Convert to the 3-mean-value format as required.

2. Run vai_q_caffe

   Run vai_q_caffe to generate a fixed-point model:

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

3. Output

   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 8: **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).

Table 8: **vai_q_caffe Output Files** *(cont'd)*

No.	Name	Description
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).

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 9: **vai_q_caffe Options List**

Name	Type	Optional	Default	Description
model	String	Required	-	Floating-point prototxt file (such as float.prototxt).
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	Bool	Optional	FALSE	Run test after calibration, test dataset required.
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.

Table 9: **vai_q_caffe Options List** *(cont'd)*

Name	Type	Optional	Default	Description
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

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 this situation, 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`. Follow the steps below to start finetuning with the `fix_train_test.prototxt` and caffemodel.

1. Assign the training dataset to the input layer of `fix_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_tensorflow Usage

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

Table 10: **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.
--input_meta_graph	String	TensorFlow MetaGraphDef file of the floating-point model, used for quantize finetuning.
--input_checkpoint	String	TensorFlow checkpoint file or directory of the floating-point model, used for quantize finetuning.
--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 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.

Table 10: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--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
--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
--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

Table 10: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--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 ofDPU 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
--output_dir	String	The directory in which to save the quantization results. Default: "./quantize_results"
Dump Configuration		
--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
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.

TensorFlow Version (vai_q_tensorflow)

Use the following steps to Run vai_q_tensorflow.

1. Prepare Float Model: 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 11: **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 preprocessing and augmentation.

For more information, see Getting the Frozen Inference Graph, Getting the Calibration Dataset and Input Function, and Custom Input Function.

2. Run vai_q_tensorflow: 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]
```

For more information, see Setting the --input_nodes and --output_nodes and Setting the Options.

3. After successful execution of the above 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. You need to run import tensorflow.contrib.decent_q explicitly to register the custom quantize operation, because tensorflow.contrib is now lazily loaded.
   - `deploy_model.pb` is used to compile the DPU codes and deploy on it, which can be used as the input files to the Vitis AI compiler.

Table 12: **vai_q_tensorflow Output Files**

No.	Name	Description
1	deploy_model.pb	Quantized model for VAI compiler (extended Tensorflow format)
2	quantize_eval_model.pb	Quantized model for evaluation

4. After deployment of the quantized model, sometimes it is necessary to compare the simulation results on the CPU/GPU and the output values on the DPU. vai_q_tensorflow supports dumping the simulation results with the quantize_eval_model.pb generated in step 3.

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_reuslts \
```

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 successful execution of the above command, 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 13:

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

Getting the Frozen Inference Graph

In most situations, training a model with TensorFlow gives you 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. Some operations behave differently in the training and inference, such as dropout and batchnorm; ensure that they are in 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.

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: 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 the quantization part if the graph has in-graph pre- and postprocessing, because some operations in these 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.

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 preprocessing subgraph is saved into the frozen graph, the input_fn only needs to read the images from dataset and return a feed_dict.

Custom Input Function

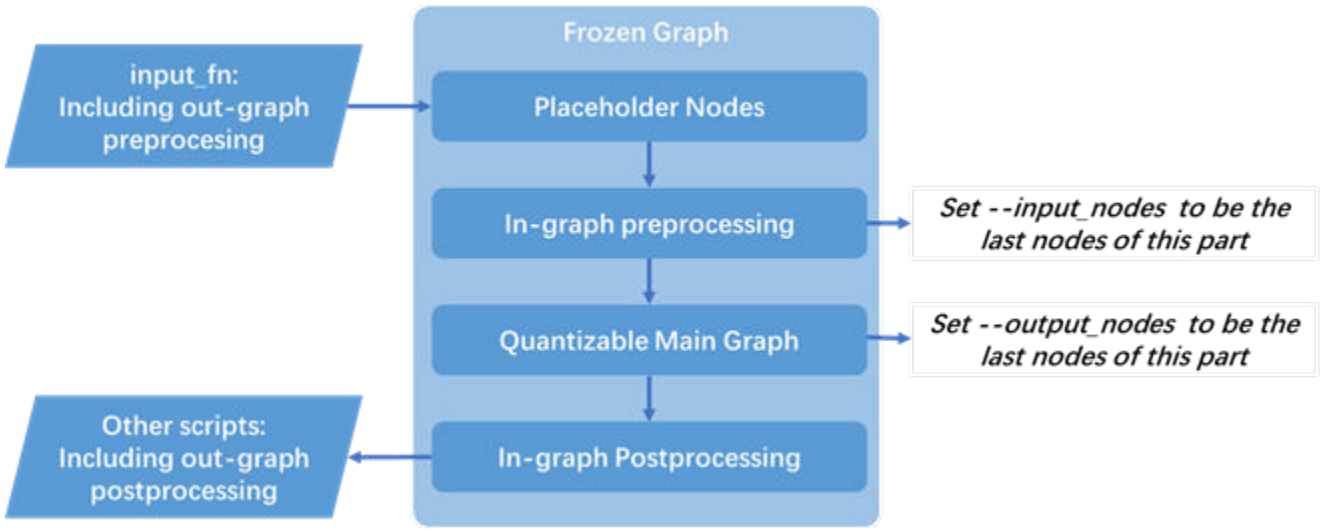
The function input format is module_name.input_fn_name, (for example, my_input_fn.calib_input). The input_fn takes an int object as input, indicating the calibration step number, and returns a dict`(placeholder_name, numpy.Array)` object for each call, which is fed into the placeholder nodes of the model 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}
```

Setting the --input_nodes and --output_nodes

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

Figure 20: **Quantization Flow for TensorFlow**



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

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

The input_fn should be consistent with the placeholder nodes.

Setting the Options

In the command line, [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 easily 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.

vai_q_tensorflow Supported Operations and APIs

Table 14: **Support 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

Table 14: **Support Operations and APIs for vai_q_tensorflow** *(cont'd)*

Type	Operation Type	tf.nn	tf.layers	tf.keras.layers
BatchNorm[#1]	FusedBatchNorm	batch_normalization batch_norm_with_global_normalization fused_batch_norm	BatchNormalization	BatchNormalization
Upsampling	ResizeBilinear ResizeNearestNeighbor	/	/	UpSampling2D
Concat	Concat ConcatV2	/	/	Concatenate
Others	Placeholder Const Pad Squeeze Reshape ExpandDims	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 because the DPU does not support it, and it should be deployed to the CPU.

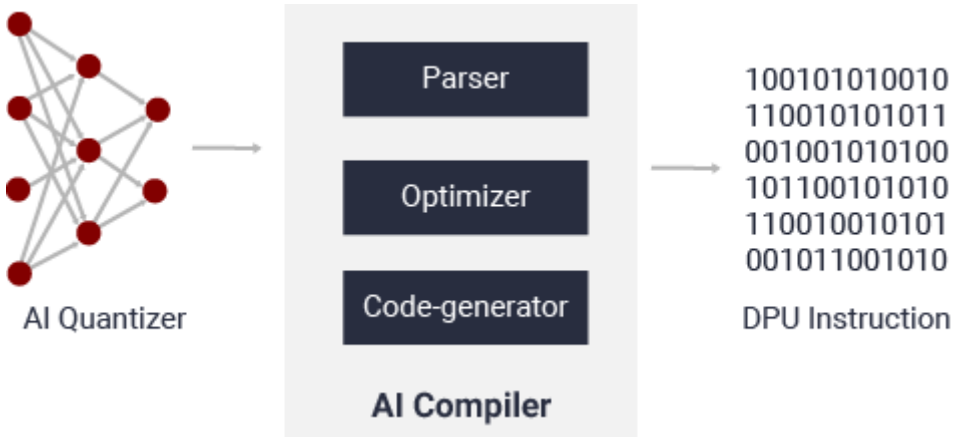
Vitis AI Compiler

Vitis AI Compiler

Vitis™ AI VAI_C is a domain-specific compiler targeting for optimizing neural network computation. It is the key to unleash the computation power of DPU via efficiently mapping the network model into a highly optimized DPU instruction sequence.

The framework of VAI_C is shown in the following figure. After parsing the topology of optimized and quantized input model, VAI_C constructs internal computation graph IR, and corresponding control flow and data flow information. It then performs multiple kinds of compilation optimizations and transforming techniques, including computation nodes fusion, efficient instruction scheduling, full reuse of DPU on-chip data, etc.

Figure 21: **Vitis AI Compiler Framework**



VAI_C Usage

The corresponding Vitis AI compiler for Caffe and TensorFlow framework 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 15: **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.
--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':''}"

Cloud Flows

This section presents a short description of the DPU-V1 (formerly known as xfDNN) front-end compilers. Here, we present a Caffe and a TensorFlow interface, both of which are built on top of a common intermediate representation.

This section describes the steps to take and should be used in combination with examples (refer to software distribution), model quantization, and the proceeding sub-graph.

Only the necessary steps and some of the context are presented to give you 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 DPU-V1 design.

The DPU-V1 compiler is one tool that must be used in combination of a partitioner. We provide 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 run time. This is because the partitioner just broke the computation in a single FPGA subgraph. The Tensor Flow partitioner will allow multiple subgraphs. Refer to the following chapter for more details.

Caffe

For presentation purposes, we assume that we have a MODEL (`model.prototxt`), WEIGHT (`model.caffemodel`), and a QUANT_INFO (i.e., 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 DPU-V1, starting with the minimum inputs.

```
vai_c_caffe.py -p MODEL -c WEIGHT -a vai/dpuv1/tools/compile/arch.json -o
WORK -n cmd -e OPTIONS
```

Specify the MODEL, WEIGHT, and where to write output. Please, specify a name for the code to be generated (i.e., cmd). In turn, this will create four outputs files in the WORK directory.

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

This is the main contract with the run time. 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. At the time of writing this user's guide, the name cmd is necessary, but it is not used by run time.

The main difference with other versions of DPU, we 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, we specify the location of the quantization file that has been computed separately and explained in Chapter 4. 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 DPU-V1 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. A Caffe model to be complete must have both a prototxt and a caffemodel. We postpone the discussion about the `arch.json` file, but it is necessary.

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. Assume we have all needed model and quantization information.

```
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, we now specify the quantization information, but we also specify the shape of the input placeholder. 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. We optimize for latency and we 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 TensorfFlow, but it contains the quantization information. If you run this front end with `[-q, --quant_info extract quant info ]` on, we 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 run time 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.

Edge Flows

For edge DPU, VAI_C is constructed based on DNNC (Deep Neural Network Compiler) compiler to keep forward compatibility for the legacy DNNDK users. This section describes the usage of VAI_C for edge DPU.

Extra Options

In addition to the common options, VAI_C supports some extra options available only for edge DPU, which are specified by '--options' option of VAI_C compiler and described in the following table.

Table 16: **VAI_C Extra Options for Edge DPU**

Parameters	Description
--help	Show all available options of VAI_C for edge DPU.
--version	Show VAI_C version information.
--save_kernel	To save kernel description info into file.
--mode	Compilation mode for DPU kernel: debug or normal. By default, network models are compiled into DPU kernels under debug mode. Debug: DPU nodes (or supper layers) of the network model run one by one under the scheduling of runtime N2Cube. With the help of DExplorer, the users can perform debugging or performance profiling for each node of DPU kernel under debug mode. Normal: All layers/operators of the network model are packaged into one single DPU execution unit and there isn't any interruption involved during launching. Compared with debug mode, normal mode DPU kernel delivers better performance and should be used during production release phase.

Table 16: **VAI_C Extra Options for Edge DPU** *(cont'd)*

Parameters	Description
--dump	Dump different types of info to facilitate debugging and use commas as delimiter when multiple types are given: • Graph: Original graph and transformed graph in DOT format. The dump files' names are ended with ".gv" suffix. • Weights: Weights and bias data for different layers. The dump files' names are ended with ".weights" or ".bias" suffix. • ir: Immediate representation for different layer in VAI_C. The dump files' names are ended with ".ir" suffix. • quant_info: Quantization information for different layers. The dump file's name is "quant.info". • dcf: DPU configuration parameters specified during DPU IP block design. The dump file's name is "dpu.dcf.dump". Note: The dpu.dcf.dump file is just used for dump purposes and should not be fed to VAI_C by "--dcf" option for compilation purpose. • log: Other compilation log generated by VAI_C. • fused_graph_info: VAI_C graph IR informaton to describe the relationship between model's original layers/operators and DPU nodes (or super layer). • all: Dump all listed above. ***Note***: All dumped files except for graph, weights, bias, and fushed_graph_info type are decrypted by VAI_C. In case of network compilation errors, these dump files can be delivered to Xilinx AI support team for further analysis.
--split_io_mem	Enable the using of split IO memory model for DPU kernel to be built. For this model, the DPU memory buffer for input/output tensors are separated from the memory buffer for intermediate feature maps. If not specified, the unique memory model is used for DPU kernel by default. About unique memory model and split IO memory model, refer to section Advanced Programming for Edge for more details.

Compiling ResNet50

To illustrate the compilation flow of VAI_C for edge DPU, we will use ResNet-50 as a compilation example. When compiling a network model, the required options should be specified to the VAI_C compiler. Once the compilation is successful, VAI_C will generate ELF object files and kernel information for deployment. These files are located under the folder specified by output_dir. The following two figures are screenshots of the VAI_C output when compiling ResNet-50 model: one is for unique memory model and the other is for split IO memory model.

Figure 22: **VAI_C Output for ResNet-50 under unique memory model**

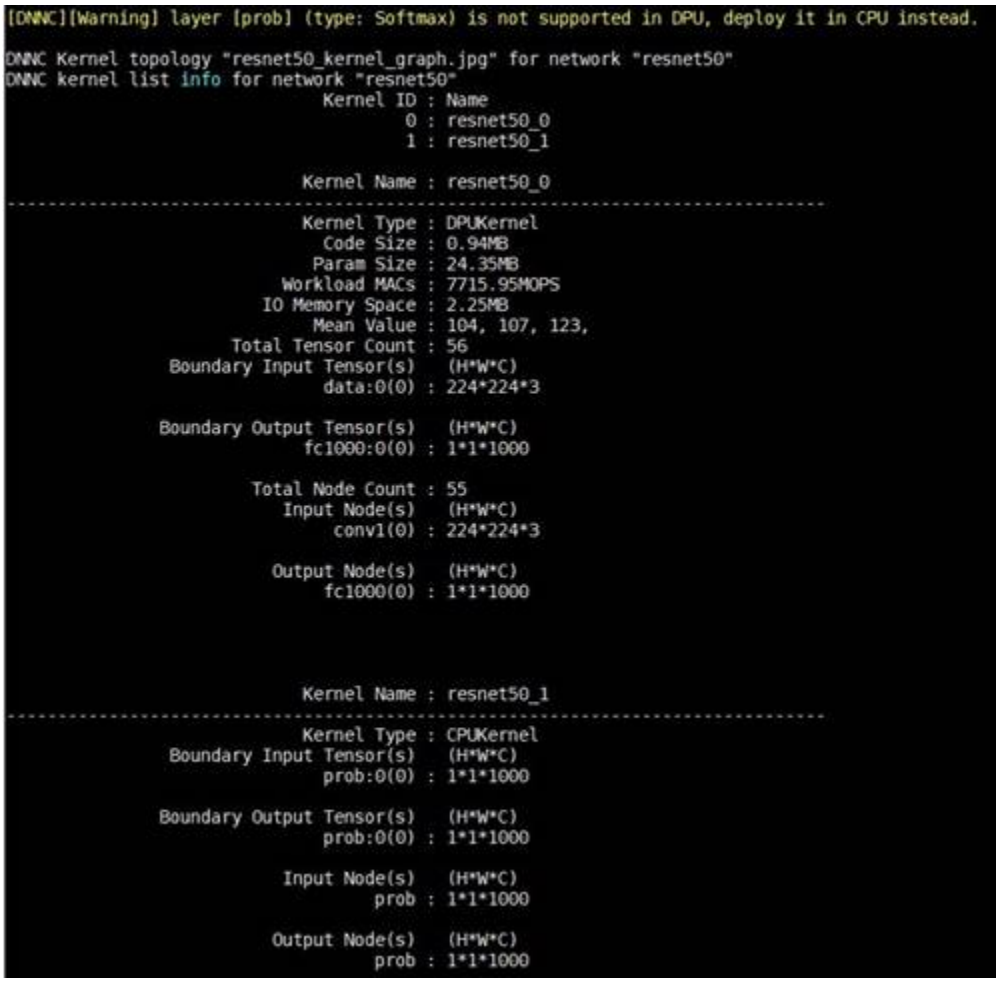
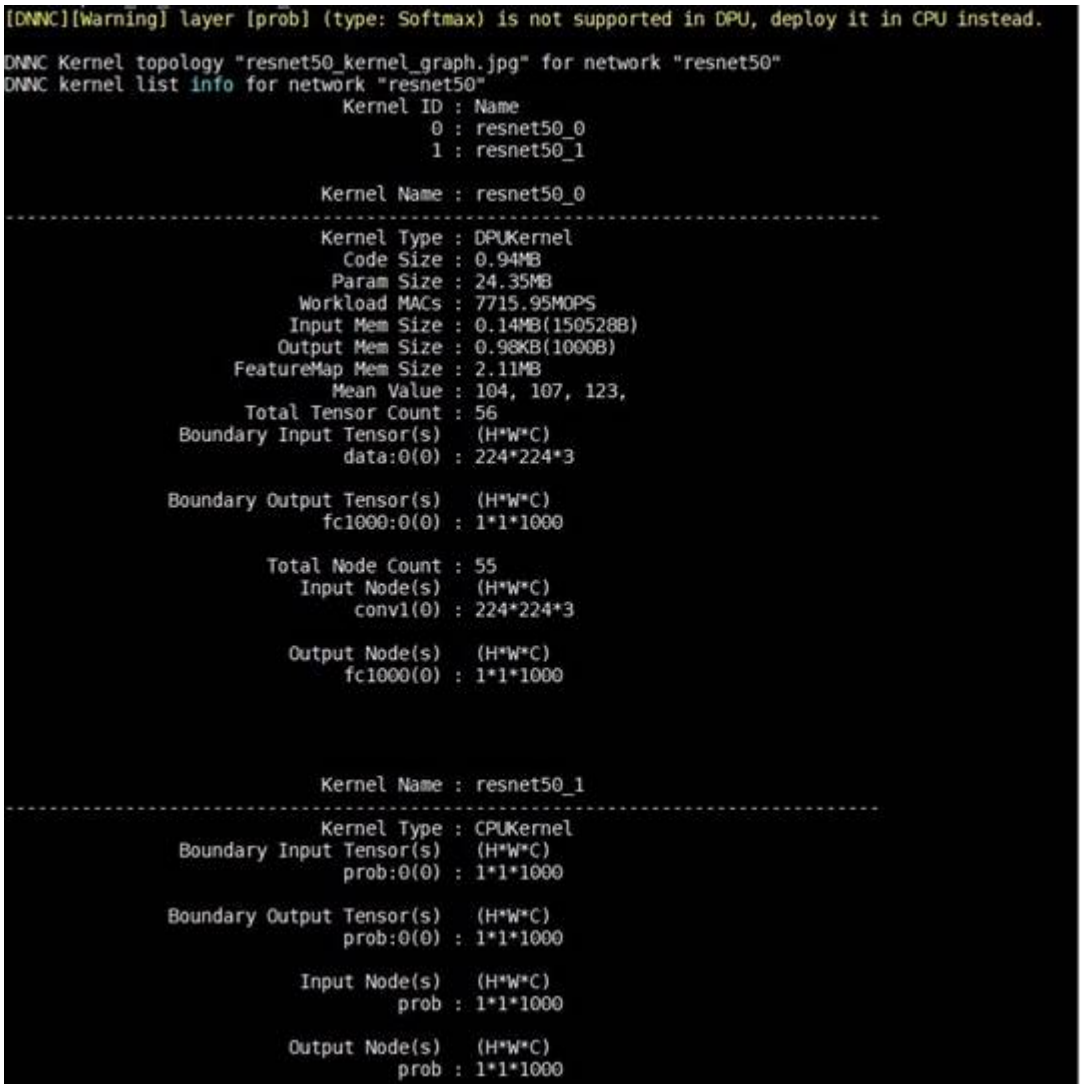


Figure 23: **VAI_C Output for ResNet-50 under split IO memory model**



Due to the limited number of operations supported by the DPU, VAI_C automatically partitions the input network model into serval kernels when there are operations not supported by DPU. The users are responsible for the data transfer and communication between different kernels, using APIs provided by N^{2}Cube that can be used for retrieving input and output address based on the input and output nodes of the kernel.

VAI_C Kernel

The kernel information generated by VAI_C is illustrated as follows. Such information is useful for the users to deploy models over edge DPU.

- **Kernel ID:** The ID of each kernel generated by VAI_C after compilation. Every kernel has a unique id assigned by VAI_C. The neural network model will be compiled to several kernels depending on operators supported by DPU.
- **Kernel Topology:** The kernel topology description file describes the kernels in the kernel graph view when compilation is finished. The `kernel_graph` file is saved in standard JPEG format with file extension `.jpg` in the output directory specified by the `VAI_C --output_dir` option. If graphviz is not installed on the host system, VAI_C will output a DOT (graph description language) format file with extension `.gv` instead. You can convert the `.gv` format file to a JPEG file using the following command:

  ```
  dot -Tjpg -o kernel_graph.jpg kernel_graph.gv
  ```

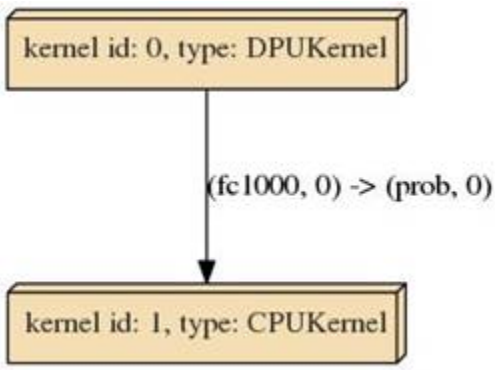
- **Kernel Name:** The name of the current kernel. For each DPU kernel, VAI_C produces one corresponding ELF object file named as `dpu_kernelName.elf`. For example, `dpu_resnet50_0.elf` and `dpu_resnet50_2.elf` are for DPU kernels resnet50_0 and resnet50_2 respectively. The kernel name is expected to be used in the Vitis AI programming, allowing DPU runtime to identify DPU different kernels correctly. As the container for DPU kernel, DPU ELF file encapsulates the DPU instruction codes and parameters for the network model.
- **Kernel Type:** The type of kernel. Three types of kernel are supported by VAI_C, see Table 6 on page 67 for details.
- **Code Size:** DPU instruction code size in the unit of MB, KB, or bytes for the DPU kernel.
- **Param Size:** The size of parameters for this kernel in the unit of MB for the DPU kernel.
- **Workload MACs:** The total computation workload in the unit of MOPS for the DPU kernel.
- **Mean Value:** The mean values for the DPU kernel.
- **IO Memory Space:** Only available for DPU kernel compiled as unique memory model. It is the total size of input tenors, intermediate feature maps, and output tensors in the unit of MB. For split IO memory model, refer to the other three fields: Input Mem Size, Output Mem Size and Feature Map Mem Size, which are described below.
- **Input Mem Size:** The total size of all the input tensors in the unit of MB(B). It is only available for DPU kernel compiled as split IO memory model.
- **Output Mem Size:** The total size of all the outputs tensors in the unit of MB(B). It is only available for DPU kernel compiled as split IO memory model.
- **Feature Map Mem Size:** The total size of the intermediate feature maps in the unit of MB(B). It is only available for DPU kernel compiled as split IO memory model.
- **Total Node Count:** The number of DPU nodes for the DPU kernel.
- **Total Tensor Count:** The number of DPU tensors for the DPU kernel.

- **Boundary Input Tensors:** All input tensors of the kernel are listed out together with their shape information in the format of HWC (height*width*channel). The input tensor name can be used to retrieve DPUTensor via `dpuGetBoundaryIOTensor()` API. For ResNet50, its input tensor is `data:0`.
- **Boundary Output Tensors:** All output tensors of the kernel are listed out together with their shape information in the format of HWC (height*width*channel). The output tensor name can be used to retrieve DPUTensor via `dpuGetBoundaryIOTensor()` API. For ResNet50, its output tensor is `fc1000:0`.
- **Input nodes:** All input nodes of the current DPU kernel and the shape information of each node are listed in the format of height*width*channel. For kernels not supported by the DPU, the user must get the output of the preceding kernel through output nodes and feed them into input nodes of the current node, using APIs provided by N^{2}Cube.
- **Output nodes:** All output nodes of the current DPU kernel and the shape information of each node is listed in the format of height*width*channel. The address and size of output nodes can be extracted using APIs provided by N^{2}Cube.

***Note*:** The fields of Code Size, Param Size, Workload MACs, Mean Value, Node Count and Tensor Count from VAI_C compilation log are only available for DPU kernel.

For ResNet-50, its kernel graph in JPEG format is shown in the following figure. The kernel graph node describes the kernel id and its type, while the edge shows the relationship between different kernels in two tuples. The first item represents the output tensor from the source kernel, while the second item shows the input tensor to the destination kernel. The tuple contains two parts: the name of input/output node binding to the tensor, and the tensor index of the input/output node. Using the node name and index provided in the tuple, users can use the APIs provided by N2Cube to get the input or output tensor address.

Figure 24: **DPU Kernel Graph for ResNet-50**



Regarding the operations supported by edge DPU, the users can refer to the *Zynq DPU v3.1 IP Product Guide* (PG338) for details. After compilation process of VAI_C, network models are normally transformed into the following three kinds of kernels.

- **DPUKernel:** Kernel running on edge DPU

- **CPUKernel:** Kernel running on CPU side. It consists of the DPU un-supported layers/operators, which should be deployed onto the CPU by the user.
- **ParamKernel:** Same as CPU Kernel, but also generates weights and bias parameters for the DPU un-supported layers/operators.

DPU Shared Library

Under some scenarios, DPU ELF files can't be linked together with the host code into the final hybrid executable, such as DPU applications programmed with Python. After Caffe or TensorFlow models are compiled into DPU ELF files, the users can use Arm® GCC toolchain to transform them into DPU shared libraries so that they stay separate with Vitis AI applications and work as expected.

For Vitis AI evaluation boards, a 64-bit Arm GCC toolchain can be used to produce the DPU shared library. The following command links the DPU ELF file for ResNet50 into the shared library.

```
aarch64-linux-gnu-gcc -fPIC -shared \
dpu_resnet50_*.elf -o libdpumodelresnet50.so
```

Within the Vitis AI runtime container, 64-bit Arm GCC cross-compilation toolchain can be used as follows:

```
aarch64-linux-gnu-gcc \
--sysroot=/opt/vitis_ai/petalinux_sdk/sysroots/aarch64-xilinx-linux \
-fPIC -shared dpu_resnet50_*.elf -o libdpumodelresnet50.so
```

With `dpu_resnet50_*.elf`, the DPU ELF file `dpu_resnet50_0.elf` for ResNet50 model is wrapped into `libdpumodelresnet50.so`. For each model, all its DPU ELF files generated by VAI_C should be linked together into one unique DPU shared library in the naming format of `libdpumodelModelName.so`. For ResNet50, ModelName should be replaced with resnet50. If there are more than one network model used in a Vitis AI application, the users must create one DPU shared library for each of them.

***Note*:** DPU shared libraries should be placed in the same folder with DPU applications or folder `/lib/` or `/usr/lib/` or `/usr/local/lib/`. Otherwise, the `dpuLoadKernel()` API reports an error.

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. Following is an example of a Resnet based SSD object detection model. Notice the parts in original graph, Figure 1, in red that is replaced by `fpga_func_0` node in the partitioned graph, Figure 2. The partitioned code is complete and executes on both CPU and FPGA.

Note: This support is currently available for Alveo™ based deep learning solution.

Figure 25: **Original Graph**

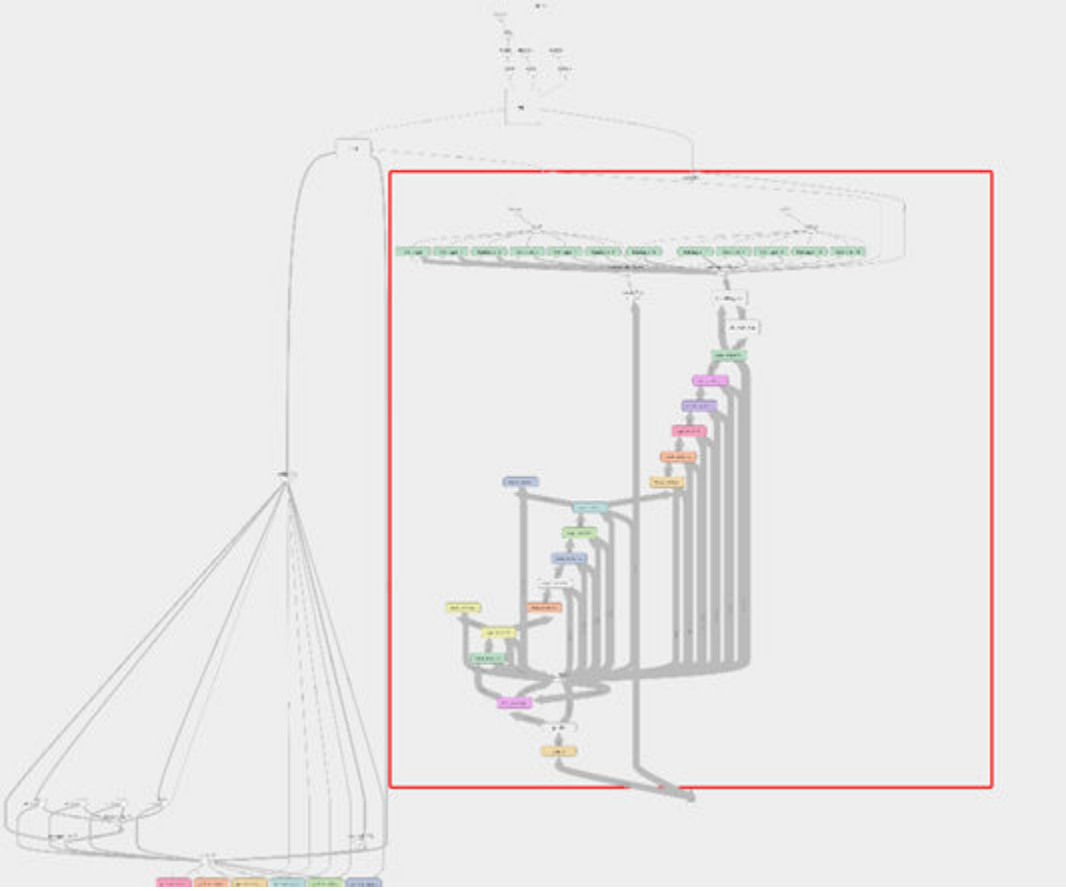
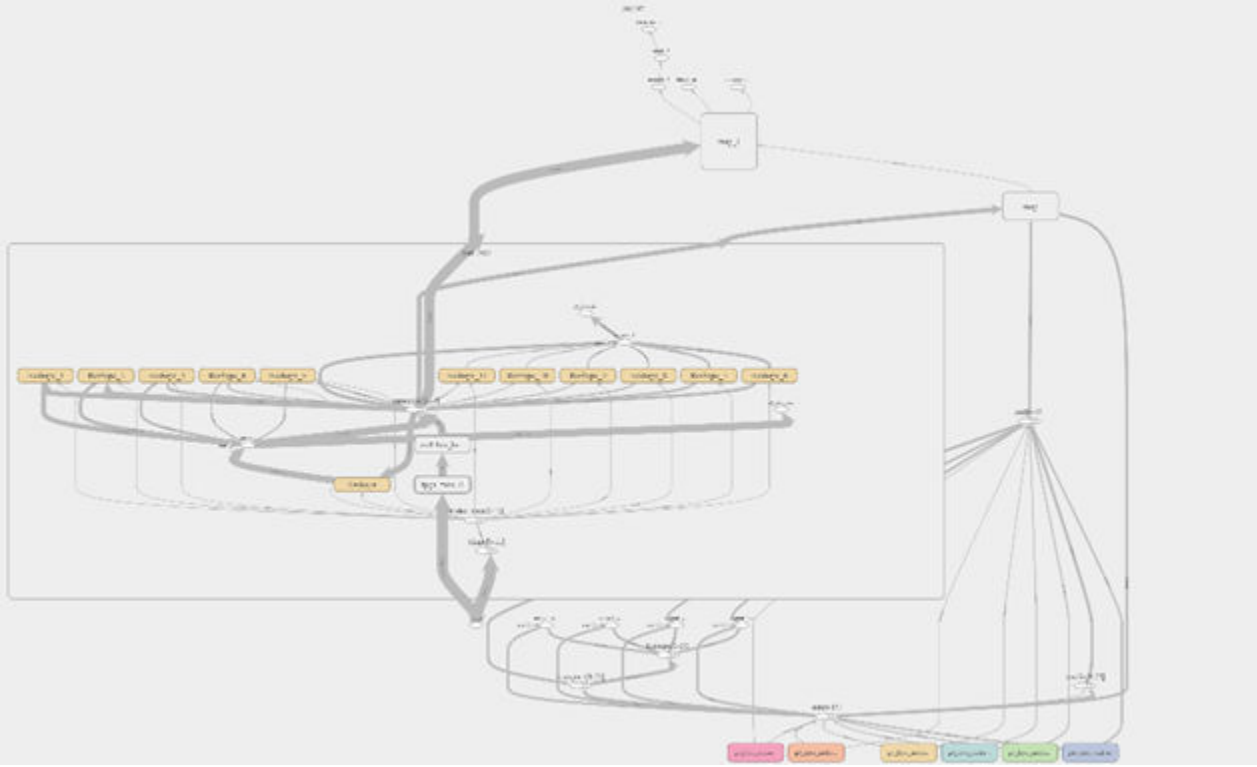


Figure 26: **Partitioned Graph**



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

Partitioner's main input argument (for example, args in item 1 from Partioning usage flow) 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: dpuv1 quantization file
- batch_sz: Inference batch size. (Default 1)
- startnode: List of start nodes for FPGA partition (optional. Defaults to all placehoders)
- 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. Meaning, 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:
- **PREPARE PHASE:** Python

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

  - **prototxt:** Path to model's prototxt
  - **caffemodel:** Path to models caffemodel
  - **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, we need to 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.

Deployment and Runtime

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

Xstream API

A typical end-to-end workflow involves heterogenous 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 ($VAI_PYTHON_DIR`/vai/dpuv1/rt/xstream.py`) provides a standard mechanism for streaming data between multiple processes and controlling execution flow / dependencies.
- Xstream Channel: Channels are defined by an alphanumeric string. Xstream Nodes may publish payloads to channels and subscribe to channels to receive payloads. The default pattern is PUB-SUB, that is, all subscribers of a channel will receive all payloads published to that channel. Payloads are queued up on the subscriber side in FIFO order until the subscriber consumes them off the queue.
- Xstream Payloads contain two items: a blob of binary data and metadata. The binary blob and metadata are transmitted using Redis, as an object store. The binary blob is meant for large data. The metadata is meant for smaller data like IDs, arguments and options. The object IDs are transmitted through ZMQ. ZMQ is used for stream flow control. The id field is required in the metadata. An empty payload is used to signal the end of transmission.
- Xstream Node: Each Xstream Node is a stream processor. It is a separate process that can subscribe to zero or more input channels, and output to zero or more output channels. A node may perform computation on payload received on its input channel(s). The computation can be implemented in CPU, FPGA or GPU. To define a new node, add a new Python file in `vai/dpuv1/rt/xsnodes`. See ping.py as an example. Every node should loop forever upon construction. On each iteration of the loop, it should consume payloads from its input channel(s) and publish payloads to its output channel(s). If an empty payload is received, the node should forward the empty payload to its output channels by calling`xstream.end()` and exit.
- Xstream Graph: Use `$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 will spawn 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")
```

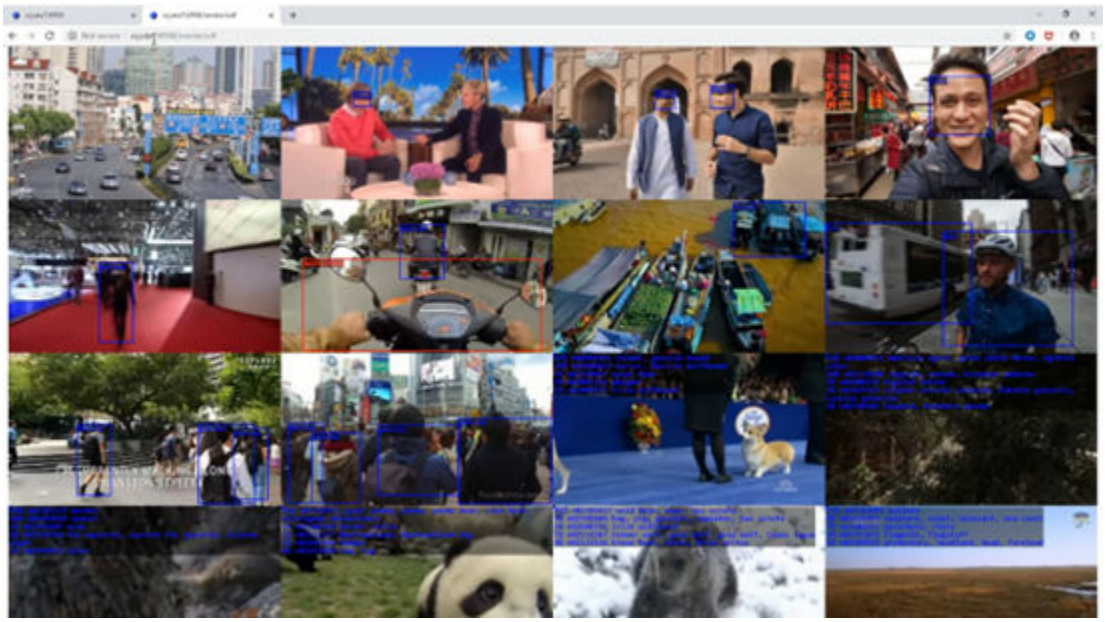
```
graph.edge("DONE", "post", None)

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`

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, we enable time and space multiplexing of FPGA resources. Eventually, with many Neptune servers, we hope to enable a broader scale-out. 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 27: **Multi-stream, Multi-network processing in Alveo**



Advanced Programming for Edge

In addition to the unified high-level programming interface, Vitis AI also offers a comprehensive set of advanced low-level C++/Python programming APIs for deep learning applications' development over Xilinx edge DPU. These low-level APIs originate from the DNNDK (Deep Neural Network Development Kit) programming interface, which ensures future compatibility for end users. Legacy DNNDK projects can be easily ported to the Vitis AI environment without any code changes.

The Vitis AI advanced low-level 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 Xilinx 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's 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

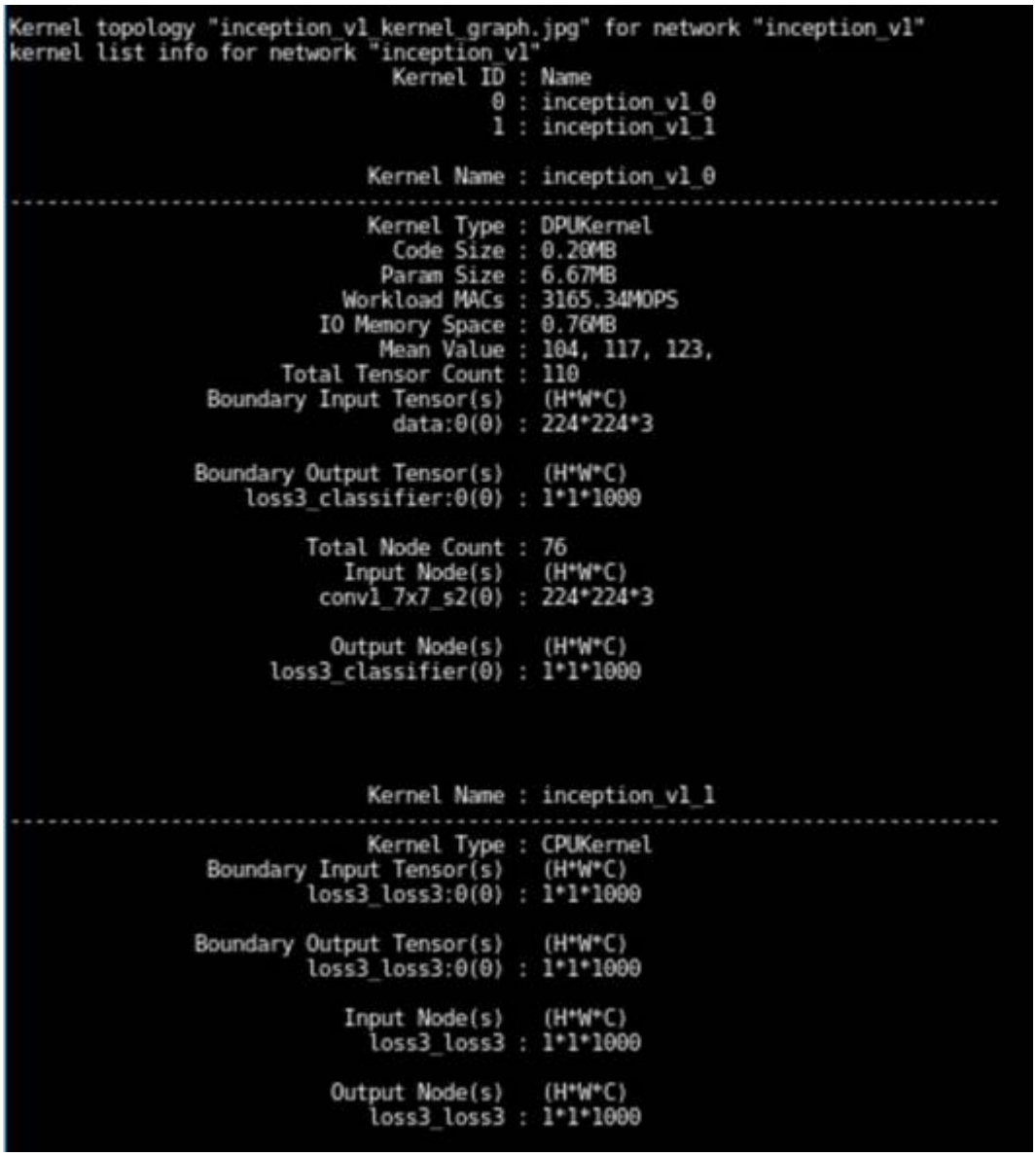
A DPU node is considered a basic element of a network model deployed on the DPU. Each DPU node is associated with input, output, and some parameters. Every DPU node has a unique name to allow APIs exported by Vitis AI to access its information.

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

- 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.
- A boundary output node is a node that does not have any successor nodes in the DPU kernel topology.
- 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 28: **Sample VAI_C Compilation Log**



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: Advanced 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: Advanced 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()` can't be used directly while deploying TensorFlow models. Instead the users need to implement the pre-processing code by themselves.

The following code snippet shows an example to specify image into DPU input tensor for TensorFlow model. Noted that the image color space fed into 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. And 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, the users can reuse those pre-processing and post-processing Python code during training phase. This can help to speed up the workflow of model deployment on DPU for the 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 edge DPU, Vitis™ AI compiler and runtime N2Cube work together to support two different DPU memory models: unique memory model and split IO model, which are described below. The unique memory model is the default when network model is compiled into DPU kernel. To enable a split IO model, you specify the options `--split-io-mem` 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 side. Cache flushing and invalidation is handled by N2Cube. Therefore, you don't 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.

You should copy unique memory model demands, that input data after pre-processing, into the boundary input tensors of DPU task's memory buffer. After this, 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. This buffer which can be accessed by DPU directly. One example is the camera based deep learning application. The pre-processing over 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 DPU input memory buffer again.

Split IO Memory Model

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

DNNDK sample `split_io` provides a programming reference for split IO memory model, and the TensorFlow model SSD is used. There is one input tensor image:0, and two output tensors `ssd300_concat:0` and `ssd300_concat_1:0` for SSD model. From compiler building log, you can see that the size of DPU input memory buffer (for tensor image:0) is 270000, and the size of DPU output memory buffer (for output tensors `ssd300_concat:0` and `ssd300_concat_1:0`) is 218304. Then `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 DPU input memory buffer, `dpuSyncMemToDev()` is called to flush cache line. When DPU task completes running, `dpuSyncDevToMem()` is called to invalidate the cache line.

Note: The four APIs `dpuAllocMem()`, `dpuFreeMem()`, `dpuSyncMemToDev()` and `dpuSyncDevToMem()` are provided only as demonstration purpose for split IO memory model. They aren't 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

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 the users can participate in DPU cores' assignment and scheduling as required. DPU cores' 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 can't 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 a DPU task's priority to a dedicated value at run-time. 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 DPU cores affinity.

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.

Debugging and Profiling

This chapter describes the utility tools included within the Vitis™ AI Development Kit, currently only available for edge DPU. The kit consists of four 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, users can conduct DPU debugging and performance profiling independently.

- **DSight:** DPU performance profiling.
- **DDump:** Parsing and dumping over DPU ELF file, shared library, and hybrid executable
- **DExplorer:** Runtime mode management and DPU signature checking.
- **DLet:** Parsing DPU Hardware Handoff file and generated DPU configuration file.

Vitis AI 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 29: **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 DPU cores' utilization and scheduling efficiency.

- **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 30: **DExplorer Usage Options**

```
Usage: 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` will display 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 31: **DExplorer Status**

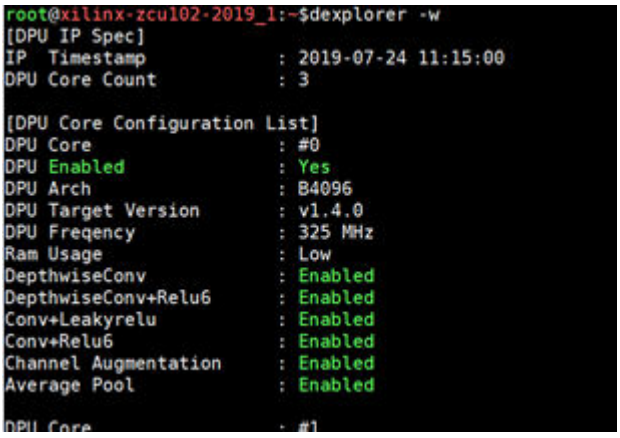
```
root@dp-n1:~# dexplorer -s
[DPU cache]
Enabled

[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 will turn on the profiling switch. When running deep learning applications in profile mode, N2cube will output 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` will be 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.

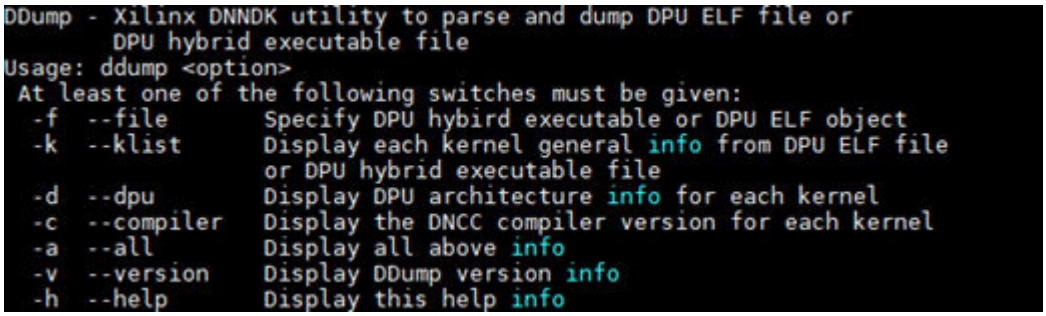
- **Checking DPU Signature:** New DPU cores have been introduced to meet various deep learning acceleration requirements across different Xilinx® FPGA devices. For example, DPU architectures B1024F, B1152F, B1600F, B2304F, and B4096F are available. Each DPU architecture can implement a different version 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 32: **Sample DPU Signature with Configuration Parameters**



- **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.

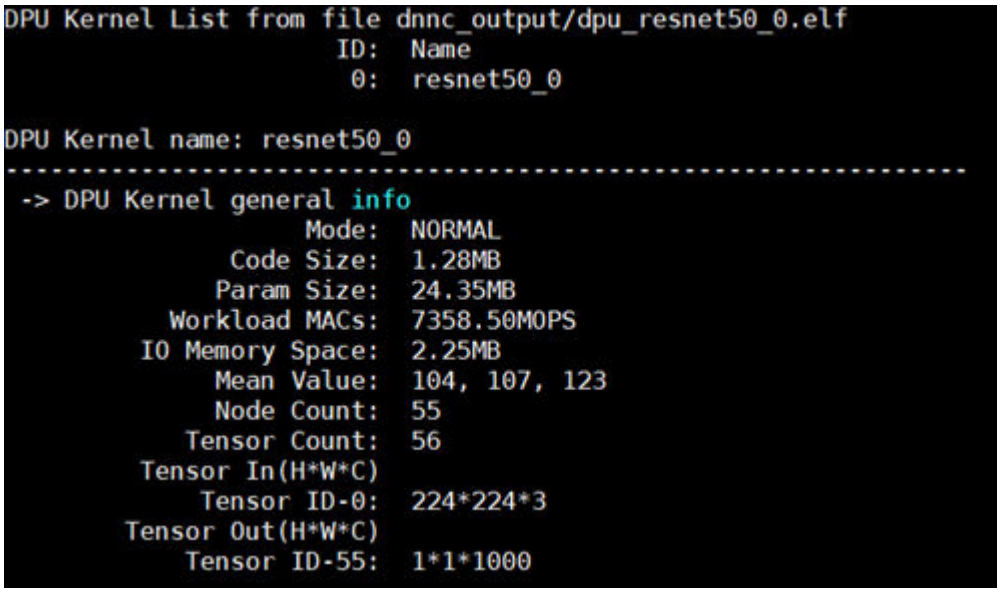
Figure 33: **DDump Usage Options**



- **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.
  - **IO 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 screenshot of DPU kernel information for ResNet50 DPU ELF file `dpu_resnet50_0.elf` with command `ddump -f dpu_resnet50_0.elf -k`.

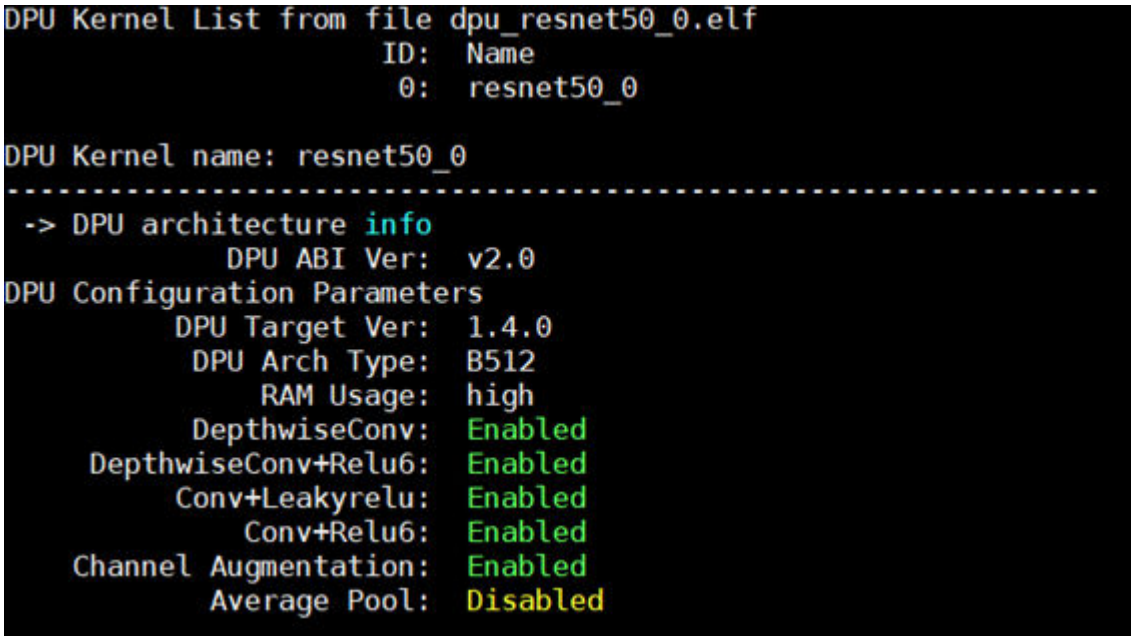
Figure 34: **DDump DPU Kernel Information for ResNet50**



- **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.

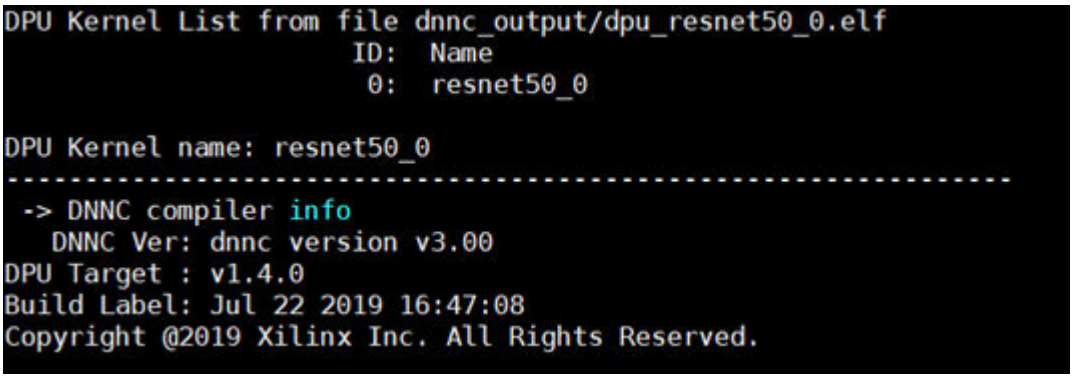
  DPU architecture information may vary with the versions of DPU IP. Running command `ddump -f dpu_resnet50_0.elf -d`, one set of DPU architecture information used by VAI_C to compile ResNet50 model is shown in the following figure.

Figure 35: **DDump DPU Arch Information for ResNet50**



- **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 Xilinx AI support team for debugging purposes.Running command `ddump -f dpu_resnet50_0.elf -c` for ResNet50 model VAI_C information is shown in the following figure.

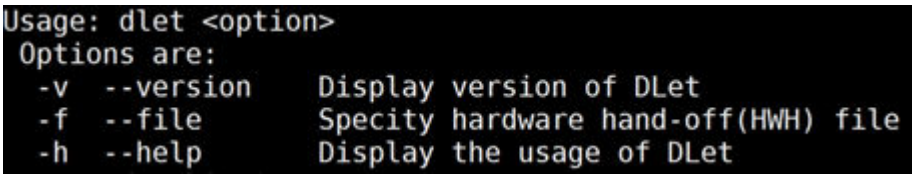
Figure 36: **DDump VAI_C Info for ResNet50**



- **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 ResNet50 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. The following figure shows the usage information of DLet.

Figure 37: **Dlet Usage Options**



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.

Debugging

After the model is deployed on edge DPU, perhaps the running results are not as desired, running into a lower accuracy issue. Under this situation, the users should first check the model's accuracy after quantized by Vitis AI quantizer. If this is fine, then two suspected points are left to be further debugged. One possible point is related to the deployment source code, which should be checked very carefully. The other possible point is related to DPU execution itself. This section focuses on the illustrations about debugging the DPU running result. Normally, it involves the following five steps.

1. Run Vitis AI quantizer to generate the golden baseline from the quantized model.
2. Build the model as debug mode DPU kernel by Vitis AI compiler with option `--dump fused_graph_info specified`.
3. Before launching the running of DPU application, run command `dexplorer -m` debug to switch runtime N2Cube into debug mode, or calling `dpuEnableTaskDebug()` to enable debug mode for the dedicated DPU task only (other tasks will not be affected).
4. Run the DPU application and get the raw dump data for DPU task's each node.
5. Compare DPU raw dump data with the golden baseline from quantizer.

DNNDK sample debugging is delivered within Vitis AI package to demonstrate how to debug the DPU. TensorFlow Inception-v1 model is deployed within this sample and there are two sub-folders: `decent_golden` and `dpu_deployment`. The folder `decent_golden` holds all the required files to generate golden baseline together with the evaluation version model `quantize_eval_model.pb` (deployable version model cannot be used) generated by quantizer. Run script `decent_dump.sh` to dump the golden baseline for the input image `decent_golden/dataset/images/cropped_224x224.jpg` and save into the folder `decent_golden/dump_golden/dump_results_0/`.

For caffe model, the users can apply the following command to generate golden baseline from the quantized model. After completion, the golden results will be dumped into folder `dump_gpu` by default.

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

With option `--dump fused_graph_info` specified to Vitis AI compiler, while compiling Inception-v1 model, one file named `fused_graph_kernel_0.txt` will be produced with DPU kernel `dpu_tf_inception_v1_0`. The folder `dpu_deployment` holds the deployment source code for Inception-v1 and `dpuEnableTaskDump()` is used to enable DPU raw data dumping. After going through the code in source file main.cc, it can be noticed that pre-processing and post-processing for Inception-v1 model are not included, which is helpful for isolating the affections of deployment code during debugging DPU. The file `fused_graph_kernel_0.txt` describes the mapping relationship between DPU node (or super-layer), which may contain several fused layers or operators, and the quantized model's layers or operators, which are divided into two types, in and out. For Caffe model, the layers' names are identical with the original floating-point model. For TensorFlow model, the operators' names are slightly different from the original floating-point model because Vitis AI quantizer performs some operators' fusion. With the name of the quantized model's layer or operator, the users can locate its corresponding dump files from quantizer golden baseline.

For kernel `dpu_tf_inception_v1_0.elf` of TensorFlow Inception-v1 model, the mapping information for its input node `input` and output node `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D` is shown below. For input node `input`, its out operator is input. And for output node `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D`, its out operator is `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D`.

```
input :
{
in(0): null
out(0): input
};
```

```
InceptionV1_Logits_Conv2d_0c_1x1_Conv2D :
{
in(0): InceptionV1_Logits_AvgPool_0a_7x7_AvgPool
out(0): InceptionV1_Logits_Conv2d_0c_1x1_Conv2D
};
```

For out type operator input, its corresponding text format dump file from Vitis AI quantizer is `input_aquant_int8.txt` (`_aquant_int8` is the added suffix), which can be found from `decent_golden/dump_golden/dump_results_0/`. Feed Int8 type input data from `input_aquant_int8.txt` into DPU input node input. After compiling and running this DPU application, raw data for each DPU node will be dumped into a folder like dump_2134 (number 2134 is process ID). For the last DPU node `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D`, locate the DPU Int8 type running result within the file `tf_inception_v1_0_InceptionV1_Logits_Conv2d_0c_1x1_Conv2D_out0.bin` (prefix `tf_inception_v1_0_` is the kernel name. And suffix out0 indicates that it is the first output tensor for this DPU node). For the last DPU node `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D`, use its out type operator `InceptionV1_Logits_Conv2d_0c_1x1_Conv2D` to find the golden output file from quantizer. Quantizer may fuse operators during performing quantization for TensorFlow model. For Inception-v1 model, we can find the similar name dump file `InceptionV1_Logits_Conv2d_0c_1x1_BiasAdd_aquant_int8.bin` (Conv2d and BiasAdd are two adjacent operators within model. _aquant_int8 is the added suffix). Lastly, check to see if DPU output of `tf_inception_v1_0_InceptionV1_Logits_Conv2d_0c_1x1_Conv2D_out0.bin` and quantizer output of `InceptionV1_Logits_Conv2d_0c_1x1_BiasAdd_aquant_int8.bin` are equal or not. If they are the same then it can be confirmed that Inception-v1 runs well over DPU, as expected. Nevertheless, potential issues exist related to DPU execution. Contact Xilinx and report bugs.

Profiling

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 38: **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 with the name `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 isn't available for normal mode kernel.

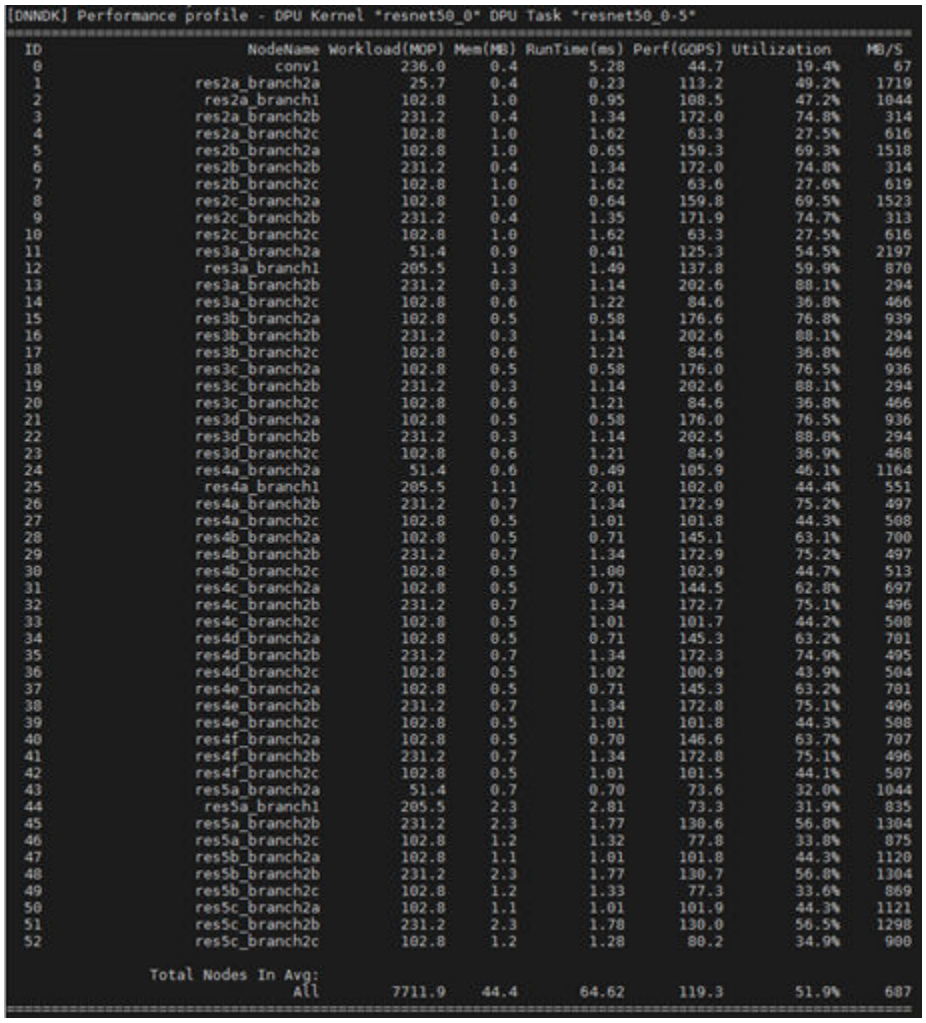
There are two approaches to enable fine-grained profiling for debug mode kernel:

- 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 ResNet50 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 39: **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:

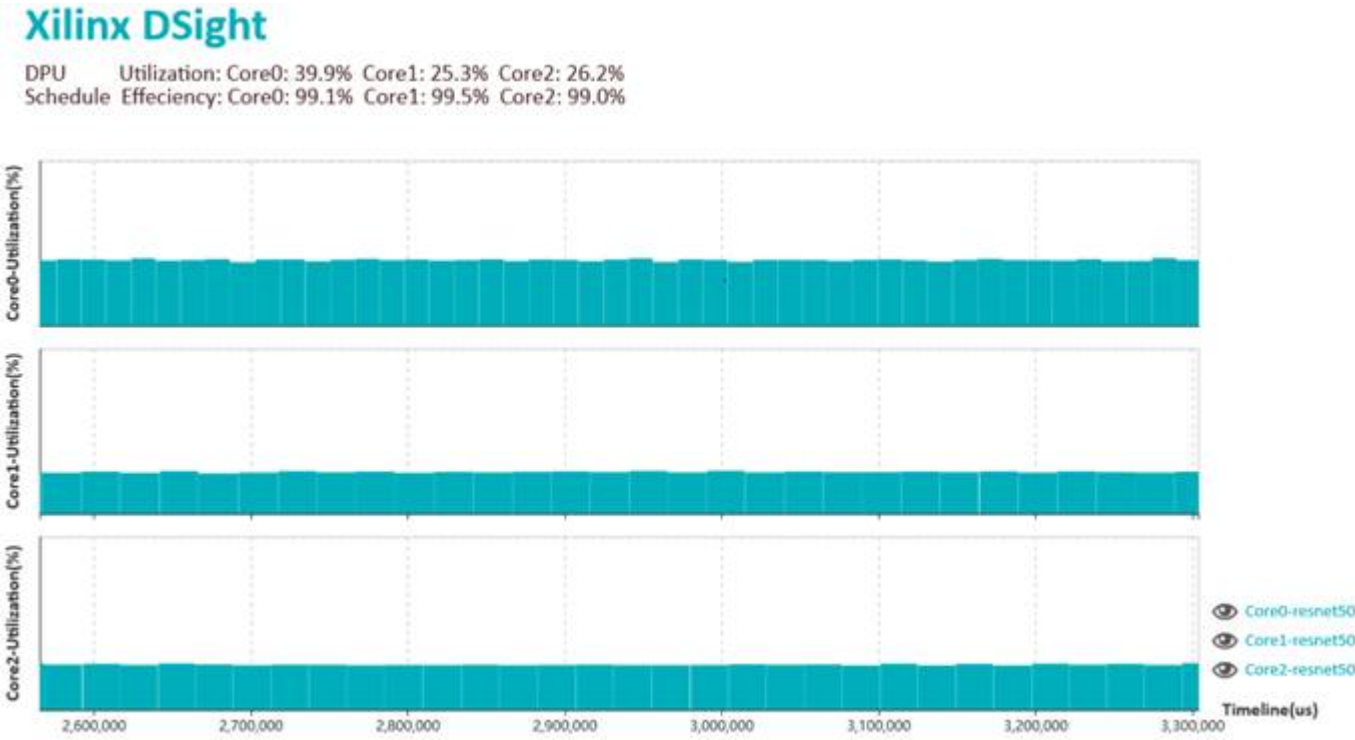
- **ID:** The index ID of DPU node.
- **NodeName:** DPU node name.
- **Workload (MOP):** Computation workload (MAC indicates two operations).
- **Mem (MB):** Memory size for code, parameter, and feature map for this DPU node.
- **Runtime (ms):** The execution time in unit of Millisecond.
- **Perf (GOPS):**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

Panorama-View Profiling

DSight delivers the visual format profiling statistics to let the users 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.



- **DPU Utilization (Y-axis):** List out each DPU core's utilization. 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.

Advanced Programming Interface

For edge DPU, Vitis AI offers the advanced low-level C++/Python programming APIs. 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 the users 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 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, the users can adopt the suited low-level Python APIs in module n2cube, which are equivalent wrappers for those C++ APIs in library `libn2cube`. With Python programming interface, the users 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 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,
- **dpuGetTaskPriority():** Retrieve a DPU Task's priority.
- **dpuSetTaskAffinity():** Dynamically set a DPU Task's affinity over DPU cores at run-time. 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 isn't 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

Retrieve a DPU Task's affinity over DPU cores. If the affinity isn't 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

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

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's.

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:** The virtual address of DPU output memory buffer, which holds all the boundry 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 input memory buffer to DPU task.

***Note*:**

It can only be used for DPU kernel compiled by VAI_C under split IO 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 boundry 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 IO 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

XILINX

Appendix B

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.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright