Vitis AI User Guide

UG1414 (v3.5) September 28, 2023

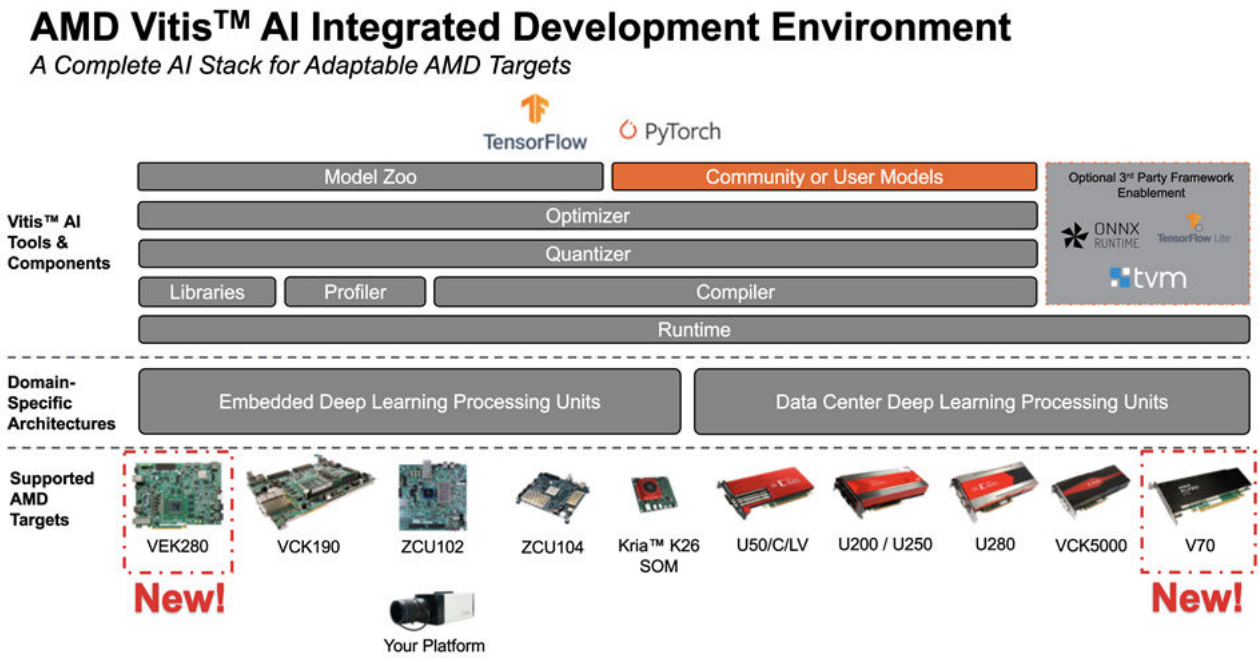
AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce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

Table of Contents

Vitis AI Overview

The AMD Vitis™ AI development environment accelerates AI inference on AMD hardware platforms, including edge devices and AMD Versal™ accelerator cards. This comprehensive framework encompasses optimized IP cores, versatile tools, powerful libraries, diverse models, and illustrative example designs. With a focus on high efficiency and ease of use, Vitis AI succeeds in unlocking the full potential of AI acceleration on AMD SoCs and Adaptive SoCs. The Vitis AI development environment makes it easy for users without extensive FPGA knowledge to develop deep-learning inference applications by abstracting the intricacies of the underlying programmable logic.

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



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

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs.

- **Machine Learning and Data Science:** Importing a machine learning model from a PyTorch, TensorFlow, or other popular framework onto AMD Vitis™ AI, and then optimizing and evaluating its effectiveness. Topics in this document that apply to this design process include:
  - Chapter 2: Optimizing the Model
  - Chapter 3: Quantizing the Model
  - Chapter 4: Compiling the Model
  - Chapter 5: Deploying and Running the Model
  - Chapter 6: Profiling the Model

Features

Vitis AI includes the following features:

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

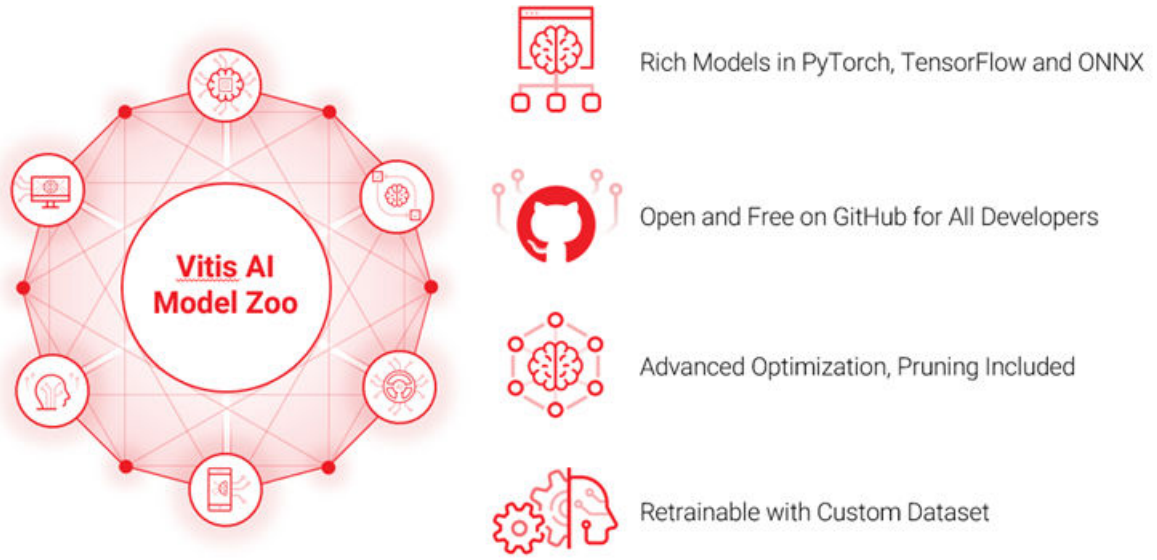
Vitis AI Tools Overview

Vitis AI Model Zoo

The Vitis AI model zoo is a curated collection of finely tuned deep learning models designed to accelerate the deployment of AI inference on AMD platforms. It encompasses diverse applications, such as ADAS/AD, video surveillance, robotics, and data centers. It equips developers with powerful tools and optimized models to unlock the advantages of deep learning acceleration.

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

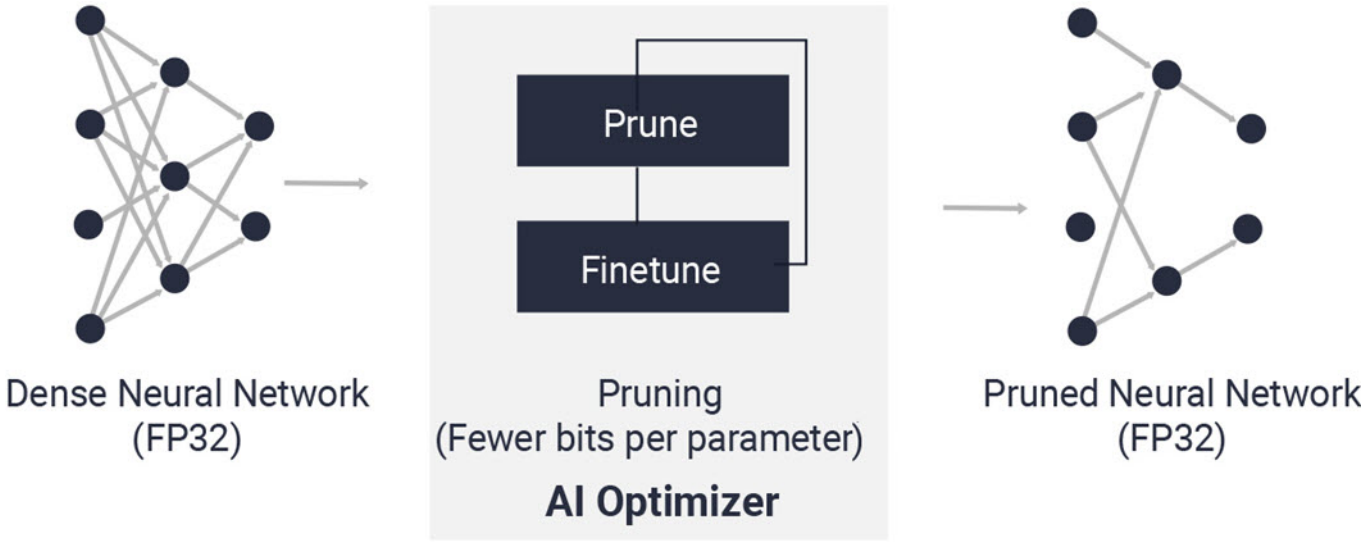
Figure 2: **Vitis AI Model Zoo**



Vitis AI Optimizer

With world-leading model compression technology, you can achieve an impressive reduction in model complexity, ranging from 5x to 50x, while experiencing minimal accuracy degradation.

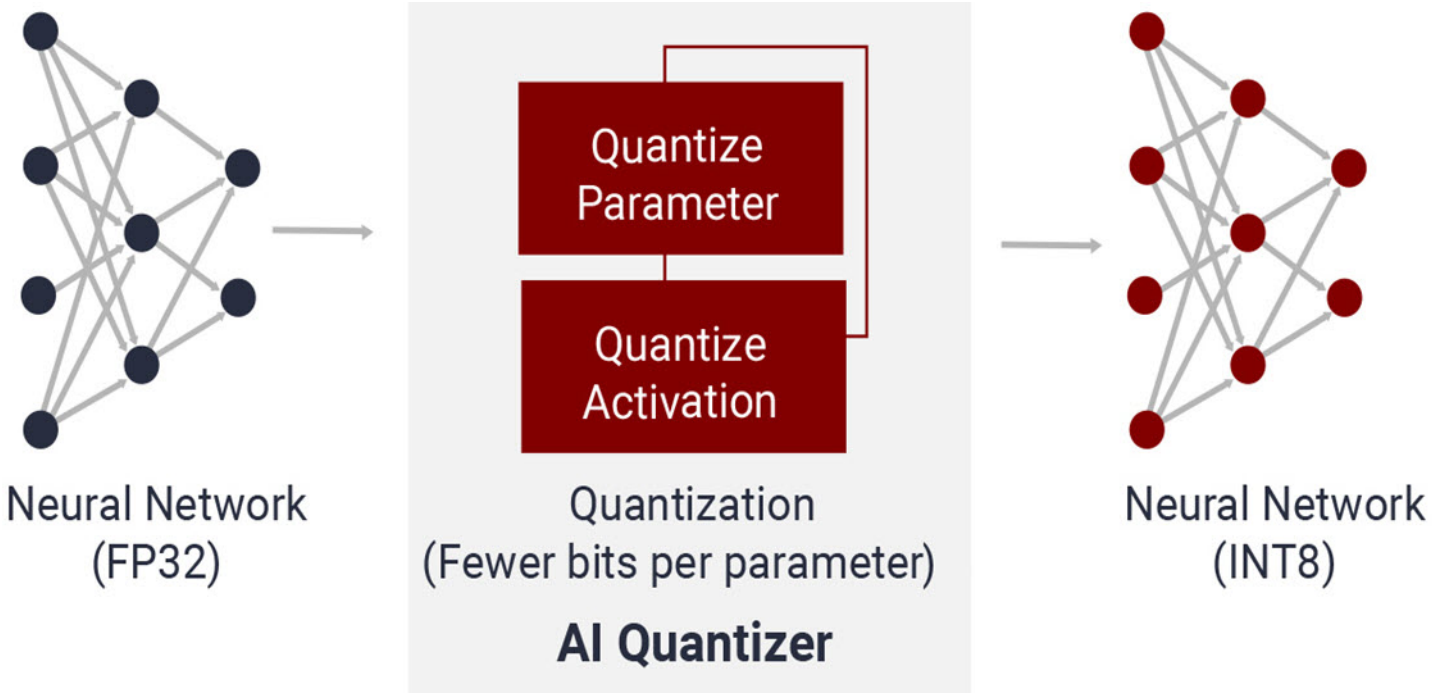
Figure 3: **Vitis AI Optimizer**



Vitis AI Quantizer

The Vitis AI quantizer significantly reduces computational complexity while preserving prediction accuracy by converting the 32-bit floating-point weights and activations to fixed-point formats like INT8. This transformation results in a fixed-point network model that demands less memory bandwidth, leading to faster processing speed and improved power efficiency compared to the floating-point model.

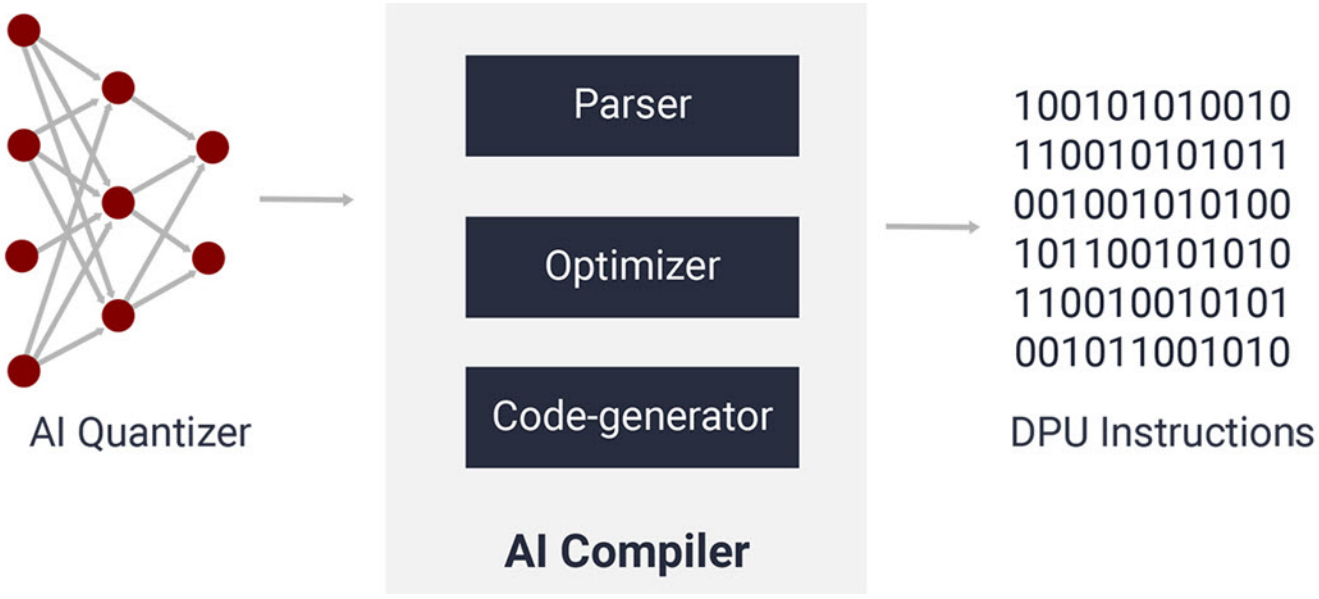
Figure 4: **Vitis AI Quantizer**



Vitis AI Compiler

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

Figure 5: **Vitis AI Complier**

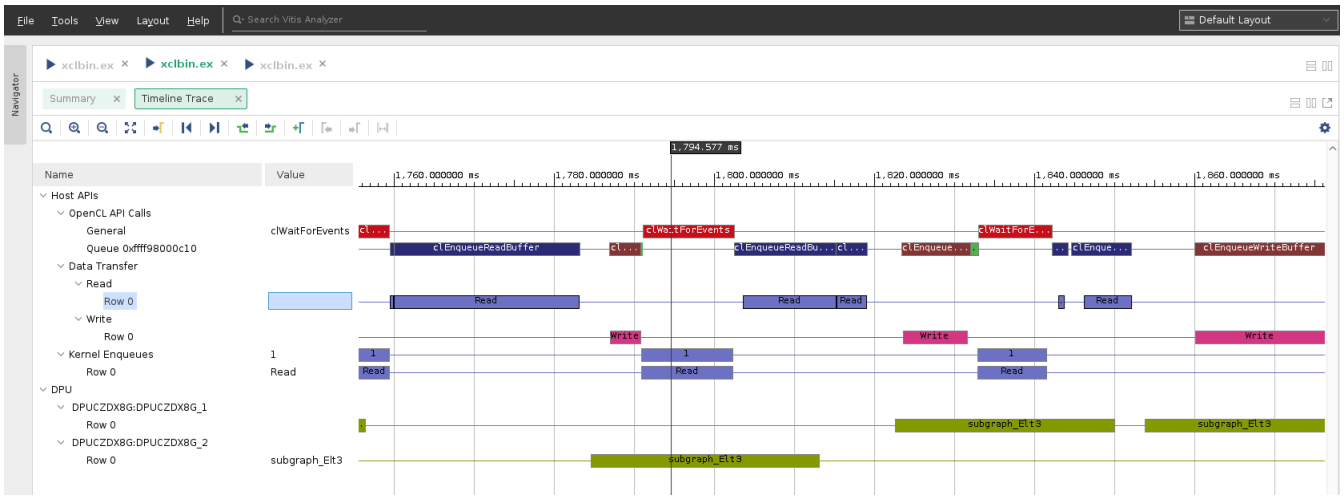


Vitis AI Profiler

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

The Vitis AI profiler is a powerful tool that meticulously analyzes and visualizes AI applications to identify bottlenecks and optimally allocate computing resources across different devices. Its user-friendly interface ensures easy adoption, requiring no code modifications. With the ability to trace function calls and runtime and collect crucial hardware information encompassing CPU, DPU, and memory utilization, the profiler empowers developers with comprehensive insights for efficient performance tuning.

Figure 6: **Vitis AI Profiler**

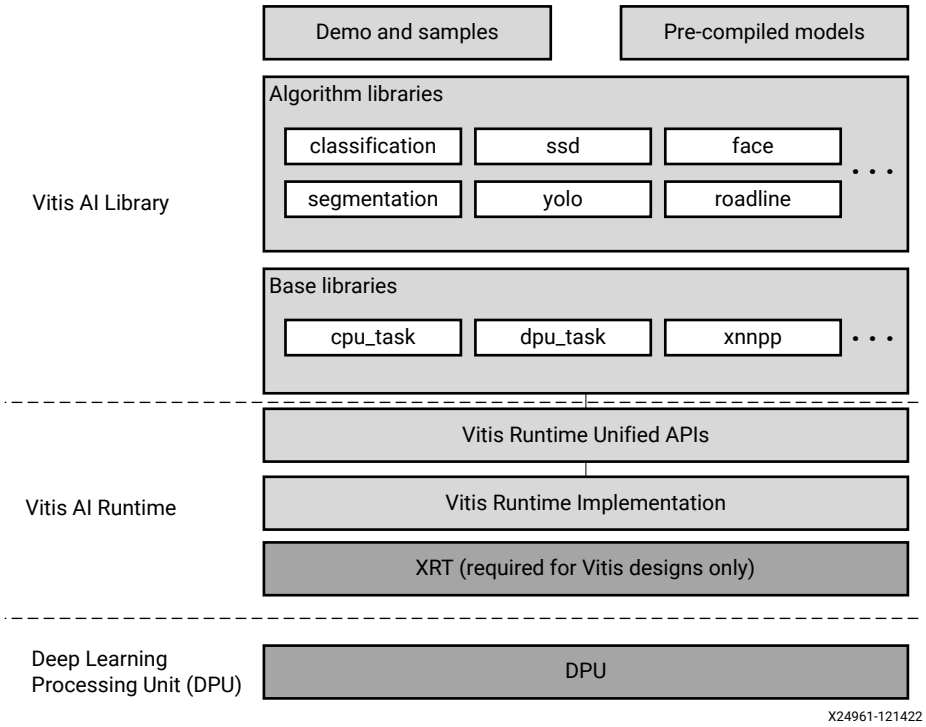


Vitis AI Library

The Vitis AI Library is a collection of high-level libraries and APIs designed for efficient AI inference with DPUs. Seamlessly integrating with the Xilinx Runtime (XRT) and built upon the Vitis AI runtime with Vitis runtime unified APIs, this powerful framework ensures a smooth and unified experience.

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

Figure 7: **Vitis AI Library**



For information on the Vitis AI Library, see *Vitis AI Library User Guide* (UG1354).

Vitis AI Runtime

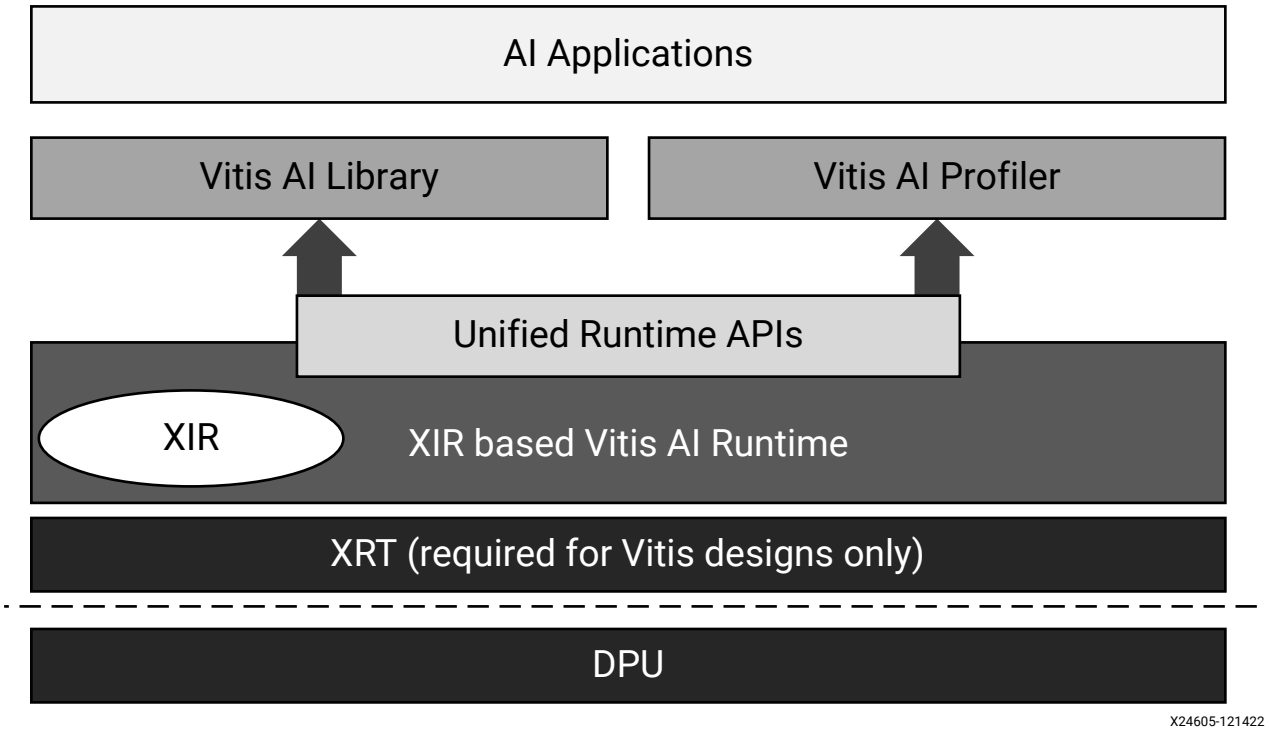
With the Vitis AI runtime, applications access a unified high-level runtime API that facilitates seamless and efficient data center-to-Edge deployments. This cohesive interface streamlines the application deployment process, ensuring a smooth transition between data center and Edge environments.

The following are the features of the AI runtime API:

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

The following figure depicts the VART framework. XIR is the AMD Intermediate Representation, an internal IR used in Vitis AI to represent neural network operators.

Figure 8: **VART Stack**



Vitis AI Containers

The Vitis AI 3.5 release uses containers to distribute the AI software. The release consists of the following components:

- Tools container
- Examples on the public GitHub
- Vitis AI Model Zoo

Tools Container

The tools container consists of the following:

- Containers distributed through Docker Hub
- Unified compiler flow includes:
  - Compiler flow for DPUCZDX8G (Edge)
  - Compiler flow for DPUCVDX8G (Edge)
  - Compiler flow for DPUCV2DX8G (Edge and Data Center)

- Pre-built conda environment to run frameworks:
  - `conda activate vitis-ai-tensorflow` for TensorFlow-based flows
  - `conda activate vitis-ai-tensorflow2` for TensorFlow2-based flows
  - `conda activate vitis-ai-pytorch` for PyTorch-based flows

    ***Note*:** For WeGO workflow in PyTorch, activate the following conda environment:

    ```
    conda activate vitis-ai-wego-torch
    ```
- Versal Runtime tools

Minimum System Requirements

The following URL contains the system requirements required to leverage the Vitis AI Integrated Development Environment.:

https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/system_requirements.html

Optimizing the Model

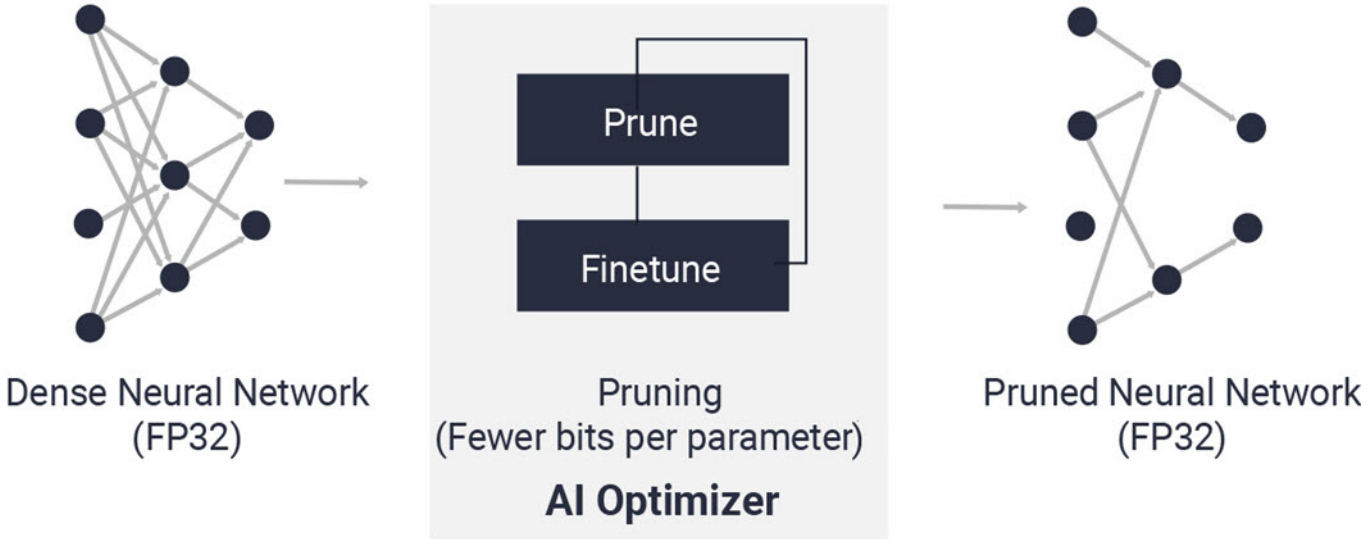
Overview and Installation

Vitis AI Optimizer Overview

AMD Vitis™ AI is an AMD development kit for AI inference on AMD hardware platforms. Machine learning inference is compute-intensive and requires high memory bandwidth to meet low-latency and high-throughput requirements.

The Vitis AI Optimizer provides the ability to optimize neural network models. Currently, the Vitis AI Optimizer includes only one tool called the pruner. The Vitis AI Optimizer removes redundant kernels in neural networks, reducing the overall computational cost for inference. Models pruned with the Vitis AI Optimizer can be quantized and deployed on AMD FPGA, SoC, or adaptive SoC devices.

Figure 9: **VAI Optimizer**



The Vitis AI Optimizer supports TensorFlow and PyTorch. The corresponding tool names (_p_ denotes pruning) are listed in the following table.

Table 1: **Vitis AI Optimizer Frameworks and Tool Names**

Framework	Tool Name
TensorFlow	vai_p_tensorflow (TF1.15), vai_p_tensorflow2 (TF2.x)
PyTorch	vai_p_pytorch

Supported Frameworks and Features

The following table highlights the features and frameworks supported by the Vitis AI Optimizer:

Table 2: **Vitis AI Optimizer Supported Frameworks and Features**

Framework	Versions	Features		
		Iterative	One-step	OFA
PyTorch	Supports 1.4 - 1.13, 2.0	Yes	Yes	Yes
TensorFlow 1.15	Supports 1.15	Yes	No	No
TensorFlow 2.x	Supports 2.4 - 2.12	Yes	No	No

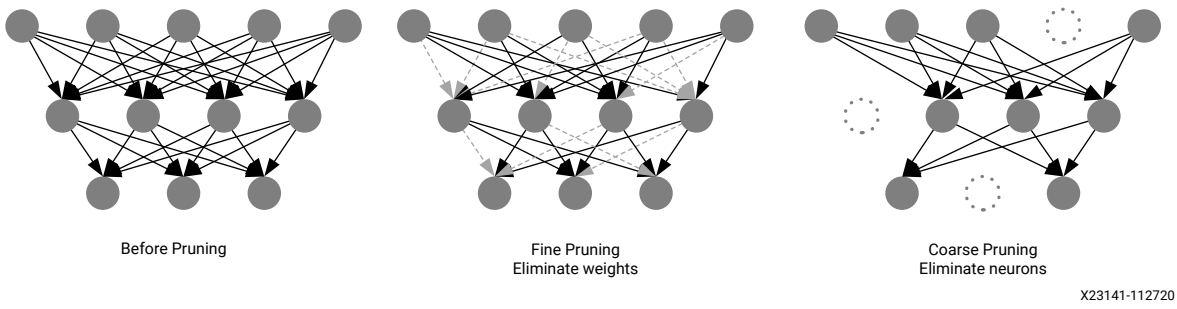
***Note*:** Vitis AI 2.5 and later releases do not support Caffe and Darknet. Use an earlier release for these frameworks.

Pruning

Overview

Neural networks are typically over-parameterized with significant redundancy. Pruning is the process of eliminating redundant weights while keeping the accuracy loss as low as possible.

Figure 10: **Coarse Pruning and Fine Pruning**



Industry research has led to several techniques that reduce the computational cost of neural networks for inference. These techniques include:

- Fine-grained pruning
- Coarse-grained pruning
- Neural Architecture Search (NAS)

Pruning Methods Supported by the DPU

Vitis AI supports the following pruning methods. These methods are applicable on a per-architecture basis as illustrated in the following table:

Table 3: **Pruning Methods Supported by Different DPU Architectures**

DPU IP	Fine-grained	Coarse-grained	NAS
DPUCZ	N	Y	Y
DPUCV	N	Y	Y
DPUCV2	N	Y	Y
DPUCA	N	Y	Y

Fine-Grained Pruning

With fine-grained pruning, weights that have minimal effect on the output are set to zero such that the corresponding computations are skipped or removed from the inference graph. This results in sparse matrices (that is, matrices that have many zero elements). Fine-grained pruning can achieve high compression rates with a modest reduction in accuracy. However, a hardware accelerator capable of implementing fine-grained sparsity must either be a fully customized, pipelined implementation or a more general-purpose “Matrix of Processing Engines” type of accelerator with the addition of specialized hardware and techniques for weight skipping and compression.

The Vitis AI sparsity pruner implements a fine-grained sparse pruning algorithm for multiple N:M sparsity patterns in each contiguous block of M values. The pruning algorithm prunes weight values along input channel dimensions. For each set of M weights, the pruner sets the N weights with the smallest value to zero. Typical values for M can be 4/8/16, and N equals as one half the value of M, which achieves 50% fine-grained sparsity.

The Vitis AI sparsity pruner supports both weight and activation sparsity for convolution and fully connected layers. The sparsity of activations can be 0 or 0.5. When the sparsity of activations is 0, the weights sparsity can be 0, 0.5, or 0.75. When the sparsity of activations is 0.5, the weights sparsity can only be 0.75.

The sparsity pruning steps are as follows:

1. Generate the sparse model
2. Fine-tune the sparse model
3. Export the sparse model

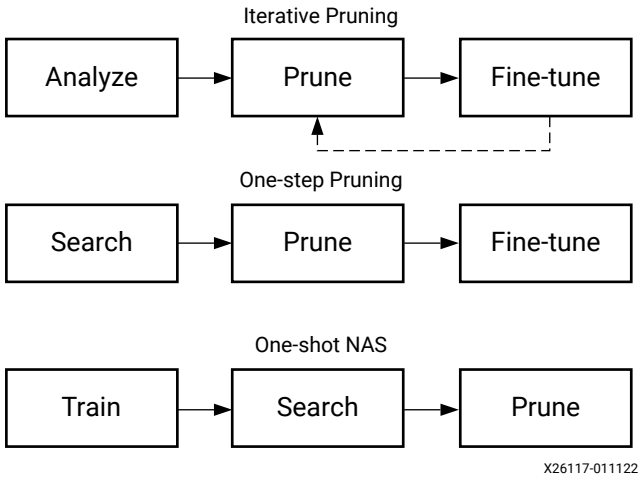
Coarse-Grained Pruning

In coarse-grained pruning, also known as channel pruning, the objective is to prune channels instead of individual weights. The result is a computational graph where the pruning algorithm prunes one or more convolution kernels for a given layer. For instance, a convolution layer with 128 channels prior to pruning might require the computation of only 57 channels post-pruning.

Channel pruning is very friendly to hardware acceleration and can be implemented with any inference architecture. However, the overall pruning ratio achievable is lower than is possible with fine-grained implementations simply because an entire kernel must be pruned always.

Coarse-grained pruning always reduces the accuracy of the original model. Retraining (finetuning) adjusts the remaining weights to recover accuracy. The technique works well on large models that leverage conventional convolutions, for example, ResNet and VGGNet. However, with depthwise convolution models such as MobileNet-v2, the accuracy of the pruned model drops dramatically, even at a low pruning rate.

Figure 11: **Workflow of Three Coarse-grained Pruning Methods**



Comparing Iterative Pruning and One-Step Pruning

Two primary approaches to pruning neural networks are Iterative Pruning and One-Step Pruning, each offering unique strategies for achieving model sparsity while preserving accuracy. Iterative pruning progressively trims model parameters while retaining accuracy, employing multiple iterations to achieve the desired sparsity level. In contrast, One-Step Pruning rapidly identifies and fine-tunes the most promising subnetwork, making it an efficient choice for achieving model sparsity with high potential accuracy in a single step.

A comparison of these two methods is shown in the following table.

Table 4: **Iterative vs. One-step Pruning**

Criteria	Iterative Pruning	One-step Pruning
Prerequisites	-	BatchNormalization in network
Time taken	More than one-step pruning	Less than iterative pruning
Retraining requirement	Required	Required
Code organization	Evaluation function	Evaluation function Calibration function

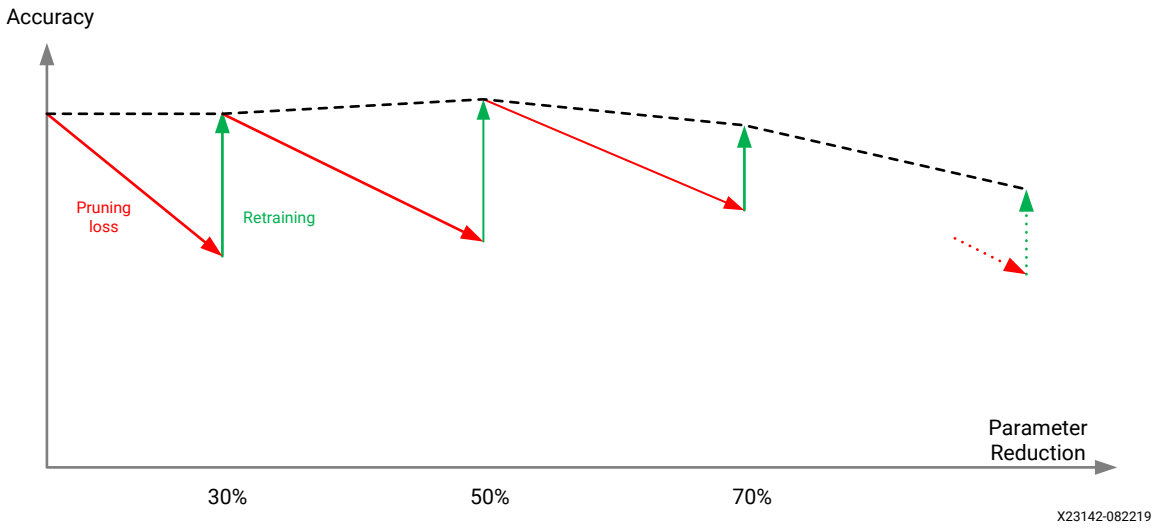
Iterative Pruning

The design of the pruning algorithm is such that it reduces the number of model parameters while minimizing the accuracy loss. The process is iterative, as shown in the following figure. Pruning results in accuracy loss, while fine-tuning of the remaining weights through training recovers accuracy. A trained, unpruned model serves as the input for the first iteration, referred to as the baseline model. This model is pruned and fine-tuned. Next, the fine-tuned model obtained from the previous iteration becomes the new baseline and is again pruned and fine-tuned. This process is repeated through multiple iterations until the desired sparsity is reached. This iterative approach is required because a model cannot be pruned with a high pruning ratio in a single pass while maintaining accuracy. When too many parameters are pruned in a single iteration, the accuracy loss can become too steep, making accuracy recovery through fine-tuning impossible.

IMPORTANT! *The parameters are progressively reduced at each iteration to improve accuracy during the fine-tuning stage.*

Leveraging the process of iterative pruning, higher pruning rates can be achieved without any significant loss of model performance.

Figure 12: **Iterative Pruning**



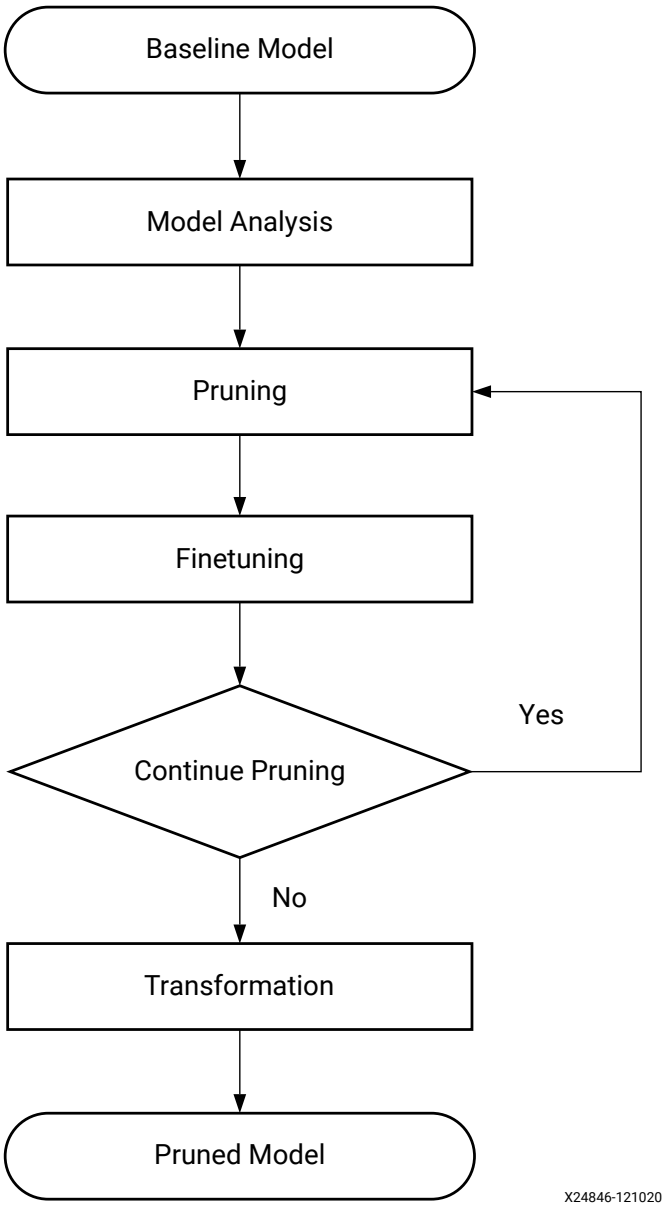
The four primary stages in iterative pruning are as follows:

- **Analysis:** Perform a sensitivity analysis on the model to determine the optimal pruning strategy.
- **Pruning:** Reduce the number of computations in the input model.
- **Fine-tuning:** Retrain the pruned model to recover accuracy.
- **Transformation:** Generate a dense model with fewer weights.

Pruning a Model

Follow these steps to prune a model. The steps are also shown in the following figure.

Figure 13: **Iterative Pruning Workflow**



1. Analyze the original baseline model.
2. Prune the model.
3. Fine-tune the pruned model.
4. Repeat steps 2 and 3 several times until the desired balance between accuracy and sparsity is achieved.
5. Transform the pruned sparse model to a final dense encrypted model for use with the Vitis AI Quantizer.

Recommendations to Achieve Better Pruning Results

The following is a list of suggestions to optimize pruning results. By adhering to these guidelines, developers have observed increased pruning ratios and minimized accuracy loss.

- Use as much data as possible to perform model analysis. Ideally, you should use all the data in the validation dataset, but this can be time-consuming. You can also use partial validation set data to ensure that at least half of the dataset is used.
- During the fine-tuning stage, experiment with a few hyperparameters, including the initial learning rate and the learning rate decay policy. Use the best result as the input for the next iteration.
- The data used in fine-tuning should be a subset of the original dataset used to train the baseline model.
- If the accuracy does not improve sufficiently after several fine-tuning experiments, try reducing the pruning rate and re-run pruning and fine-tuning.

Neural Architecture Search

The concept of Neural Architecture Search (NAS) is that for any given inference task and dataset, several network architectures exist in the potential design space that are both efficient and have high prediction scores. Often, a developer starts with a standard backbone that is familiar to them, such as ResNet50, and trains that network for the best accuracy. However, many cases exist when a network topology with a much lower computational cost can offer similar or better performance. For the developer, the effort to train multiple networks with the same dataset (sometimes going so far as to make this a training hyperparameter) is not an efficient method for selecting the best network topology.

NAS can be flexibly applied for each layer. The number of channels and amount of sparsity are learned by minimizing the loss of the pruned network. NAS successfully balances speed and accuracy but requires extended training times. This method requires a four-step process:

1. Train
2. Search
3. Prune
4. Fine-tune (optional)

Compared with coarse-grained pruning, one-shot NAS implementations assemble multiple candidate "subnetworks" into a single, over-parameterized graph known as a Supernet. The training optimization algorithm attempts to optimize all candidate networks simultaneously using supervised learning. Upon completing this training process, candidate subnetworks are ranked based on computational cost and accuracy. The developer selects the best candidate to meet their requirements. The one-shot NAS method effectively compresses depthwise and conventional convolution models but requires a long training time and a higher skill level.

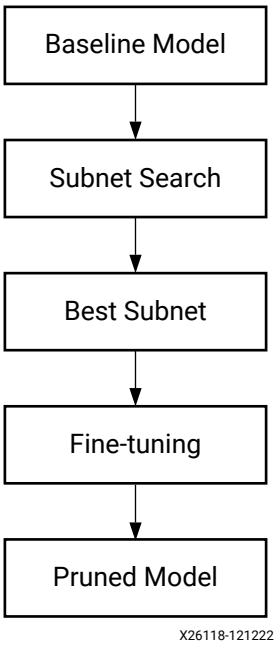
One-Step Pruning

One-step pruning implements the EagleEye#unique_33/unique_33_Connect_42_li_l3s_zvn_dsb algorithm. It introduces a strong positive correlation between pruned models and their corresponding fine-tuned accuracy by a simple yet efficient evaluation component called adaptive batch normalization. It enables you to get the subnetwork with the highest potential accuracy without fine-tuning the models. In short, the one-step pruning method searches for a bunch of subnetworks (that is, generated pruned models) that meet the required model size and selects the most promising one. The selected subnetwork is re-trained to recover accuracy.

The pruning steps are as follows:

1. Search for subnetworks that meet the required pruning ratio.
2. Select a potential network from several subnetworks with an evaluation component.
3. Fine-tune the pruned model.

Figure 14: **One-step Pruning Workflow**



Note: Bailin Li et al., EagleEye: Fast Sub-net Evaluation for Efficient Neural Network Pruning, arXiv:2007.02491

Once-for-All (OFA)

Once-for-All (OFA)[1] is a compression scheme based on One-Shot NAS. You often perform tasks such as compressing and deploying a trained model on one or more devices. Conventional techniques require you to repeat the network design process and retrain the designed network from scratch for each device, which is computationally prohibitive.

OFA introduces a new solution to this challenge: designing a once-for-all network directly deployed under diverse architectural configurations. Therefore, training costs can be amortized. Inference is performed by selecting only a part of the once-for-all network.

OFA reduces the model size across many more dimensions than pruning. It builds a family of models of varying depth, width, kernel size, and image resolution, jointly optimizing all the candidate models. Once trained, the subnetwork with the best balance between accuracy and throughput can be discovered by evolutionary search.

For each layer in the original model, Vitis OFA allows you to use an arbitrary pruning ratio for channels and arbitrary kernel sizes. The original model is split into many child networks with shared weights and becomes a super network. The child networks can do forward passes and update using a part of the convolution weights. All subnetworks should be jointly optimized during training. When all child networks are trained well, you can search for the subnetwork with the optimum balance between accuracy and throughput from the supernetwork.

To get the highest compression, the Vitis OFA pruner can optimize the search space based on the ratio of the number of depthwise convolutions to the number of regular convolutions. If the number of depthwise convolutions exceeds the number of regular convolutions, the Vitis OFA pruner mainly compresses convolution layers where the kernel size is > 1. This would result in a supernetwork with narrow channel widths and reduced accuracy.

OFA uses the original model as a teacher model to guide the training of subnetworks. Knowledge distillation allows the output of teacher models to be used as softened labels to provide more information about intraclasses and interclasses. The Vitis OFA uses adaptive soft KDLoss[2] and the sandwich rule[3] to improve performance and efficiency. Compared with the original OFA, the Vitis OFA reduces the training time by half.

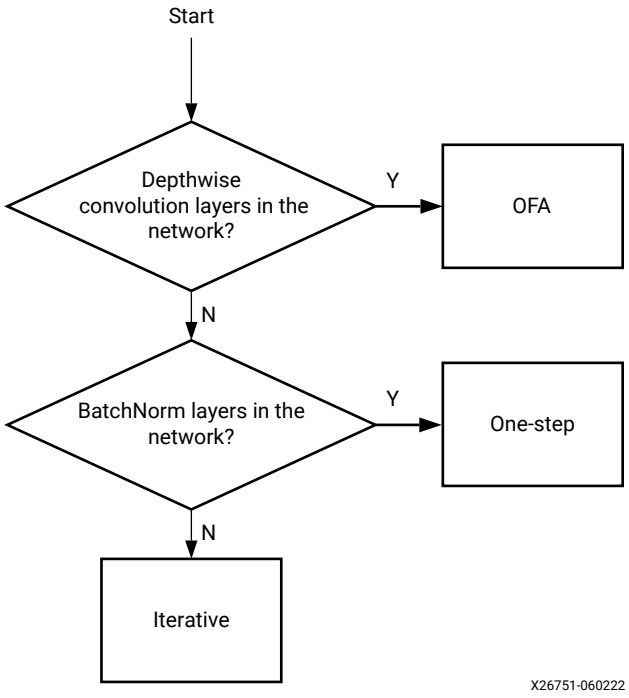
***Note*:**

1. Han Cai et al., Once-for-All: Train One Network and Specialize it for Efficient Deployment, arXiv:1908.09791
2. Dilin Wang et al., AlphaNet: Improved Training of Supernets with Alpha-Divergence, arXiv:2102.07954
3. Jiahui Yu et al., BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models, arXiv:2003.11142

Pruning Method Selection

There are three pruning methods available in Vitis AI Optimizer for PyTorch. Refer to the following decision tree to choose the right method for your network.

Figure 15: **Flowchart for Selecting Pruning Methods in PyTorch**



TensorFlow (1.15) Version - vai_p_tensorflow

You have to create a TensorFlow session that contains a graph and initialized variables (initialized by TensorFlow initializers, checkpoint, SavedModel, and so on) before pruning. Vitis Optimizer TensorFlow prunes the graph in place and provides a method to export frozen pruned graphs.

The pruned graph in memory is sparse, preserving the original weight shapes while removing specific channels and setting them to zero. The exported frozen pruned graph is a slim graph with a smaller size, which means the unnecessary channels are removed.

Preparing a Baseline Model

Here, a simple MNIST convnet is used.

```
model = keras.Sequential([
    layers.InputLayer(input_shape=input_shape),
    layers.Conv2D(16, kernel_size=(3, 3), activation="relu"),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
```

```
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(num_classes),
])
```

Instantiating a TensorFlow Session

```
with tf.Session() as sess:
  model, input_shape = mnist_convnet()
  sess.run(tf.global_variables_initializer()) # initializing variables
```

Creating a Pruning Runner

To get a pruning runner, provide an instance of the TensorFlow session, input specs, and output node names as follows:

```
from tf1_nndct.optimization.pruning import IterativePruningRunner


with tf.Session() as sess:
  model, input_shape = mnist_convnet()
  sess.run(tf.global_variables_initializer())
  input_specs={'input_1:0': tf.TensorSpec(shape=(1, 28, 28, 1),
dtype=tf.dtypes.float32)}
  pruner = IterativePruningRunner("mnist", sess, input_specs, ["dense/
BiasAdd"])
```

Pruning the Model

To prune a model, follow these steps:

1. Define a function of type `Callable[[tf.compat.v1.GraphDef], float]` to evaluate model performance. The only argument is a frozen graph, an intermediate result of the pruning process. The pruning runner performs multiple prunings to find the optimal pruning strategy.

   ```
   def eval_fn(frozen_graph_def: tf.compat.v1.GraphDef) -> float:
     with tf.compat.v1.Session().as_default() as sess:
       tf.import_graph_def(frozen_graph_def, name="")

       # do evaluation here

       return 0.5 # Returning a constant is for demonstration purpose
   ```

2. Use this evaluation function to run model analysis: You can specify devices for model analysis. The default value is ['/GPU:0']. If multiple devices are given, the pruning runner runs a parallel model analysis on each device.

   ```
   pruner.ana(eval_fn, gpu_ids=['/GPU:0', '/GPU:1'])
   ```

3. Determine pruning sparsity. The ratio indicates the reduction in the amount of floating-point computation of the model in the forward pass. `pruned_model's FLOPs = (1 - sparsity) * original_model's FLOPs`. The value of the ratio should be in (0, 1):

   ```
   shape_tensors, masks = pruner.prune(sparsity=0.5)
   ```

   ***Note*:** `sparsity` is only an approximate target value, and the actual pruning ratio cannot exactly equal this value.shape_tensors is a string to NodeDef mapping. The keys are the names of node_defs in graph_def, which need to be updated to get a slim graph. `masks` correspond to variables. It only contains 0s and 1s. After calling this method, the graph within a session is pruned and becomes a sparse graph.

4. Export frozen slim graph. Use `shape_tensors` and `masks` returned by method `prune` to generate a frozen slim graph as follows:

   ```
   slim_graph_def = pruner.get_slim_graph_def(shape_tensors, masks)
   ```

Fine-tuning a Sparse Model

The graph is pruned *in-place*, so that you can fine-tune the pruned graph like the original one. The TensorFlow optimizer applies masks automatically.

vai_p_tensorflow APIs

tf_nndct1.IterativePruningRunner

```
__init__(self,
         model_name: str,
         sess: SessionInterface,
         input_specs: Mapping[str, tf.TensorSpec],
         output_node_names: List[str],
         excludes: List[str]=[])
```

Arguments:

- **model_name:** The name of the model.
- **sess:** An instance of a TensorFlow session containing a graph and initialized variables.
- **input_specs:** The keys of this mapping are input node names of the baseline model.
- **output_node_names:** Target output node names.
- **excludes:** The names of nodes that skip pruning.

Returns: Instance of IterativePruningRunner

```
ana(self,
    eval_fn: Callable[[tf.compat.v1.GraphDef], float],
    gpu_ids: List[str]=['/GPU:0'],
    checkpoint_interval: int = 10) -> None:
```

Arguments

- **eval_fn:** The function is to evaluate the intermediate results of the pruning process. Needs to return a float.
- **gpu_ids:** A list of strings indicating devices to run evaluations.
- **checkpoint_interval:** This method implements a cache mechanism and saves results for every checkpoint_interval evaluation.

Returns: None

```
prune(self,
      sparsity: float=None,
      threshold: float=None,
      max_attemp: int=10) -> Tuple[Mapping[str, TensorProto], Mapping[str,
np.ndarray]]:
```

There are two pruning modes: FLOPs-based and accuracy-based, corresponding to argument sparsity and threshold. These two arguments *must not* be None at the same time. Argument sparsity is more prioritized than the threshold.

Arguments:

- **sparsity:** The ratio indicates the reduction in the amount of floating-point computation of the model in the forward pass.
- **threshold:** Within range [0, 1]. Indicating the maximum acceptable relative difference in accuracy between the pruned graph and the original graph.
- **max_attemp:** Pruning runner finds the optimal pruning strategy iteratively and returns after max_attemp steps anyway.

Returns:

- **shape_tensors:** A string to NodeDef mapping. The keys are the names of node_defs in graph_def, which need to be updated to get a slim graph. The values are target node_def contents masks.
- **masks:** A string-to-array mapping corresponding to variables.

```
get_slim_graph_def(self,
                   shape_tensors: Mapping[str, TensorProto]=None,
                   masks: Mapping[str, np.ndarray]=None) ->
tf.compat.v1.GraphDef:
```

Arguments:

- **shape_tensors:** A string to NodeDef mapping returned from the *prune* method.
- **masks:** A string-to-array mapping corresponding to variables. This object is also obtained from the *prune* method.

Returns: A frozen slim graph_def.

TensorFlow Examples

See the Vitis AI repository on GitHub.

TensorFlow (2.x) Version – vai_p_tensorflow2

vai_p_tensorflow2 only supports Keras models created by the Functional API or the Sequential API. Subclassed models are not supported.

Creating a Model

Here, a simple MNIST convnet from the Keras vision example is used.

```
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(), layers.Dropout(0.5),
    layers.Dense(num_classes, activation="softmax"),
])
```

Creating a Pruning Runner

To create an input specification containing shape and dtype information and subsequently use it for obtaining a pruning runner, use the following command:

```
from tf_nndct.pruning import IterativePruningRunner

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
```

Pruning a Model

To prune a model, follow these steps:

1. Define a function to evaluate model performance. The function must satisfy two requirements:
   - The first argument must be a `keras.Model` instance to be evaluated.

- It must return a Python value to indicate the performance of the model.

```
def evaluate(model):
  model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
  score = model.evaluate(x_test, y_test, verbose=0)
  return score[1]
```

2. Use this evaluation function to run model analysis:

```
runner.ana(evaluate)
```

3. Determine a pruning ratio. The ratio indicates the reduction in the amount of floating-point computation of the model in the forward pass.

   [FLOPs of pruned model] = (1 – ratio) * [FLOPs of original model]

   The value of this ratio should be between zero and one:

```
sparse_model = runner.prune(ratio=0.2)
```

   Note: `ratio` is only an approximate target value, and the actual pruning ratio cannot exactly equal this value.

The returned model from `prune()` is sparse, which means that the weights of the pruned channels are set to zero, and the model size remains unchanged. This sparse model is used in the next iterative pruning round. The sparse model is converted to a pruned dense model only after pruning is completed.

Besides returning a sparse model, the pruning runner generates a specification file in the `.vai` directory that describes how each layer is pruned.

Fine-tuning a Sparse Model

Training a sparse model is no different from training a standard model. There is no need for any additional actions other than adjusting the hyper-parameters.

```
sparse_model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
sparse_model.fit(x_train, y_train, batch_size=128, epochs=15,
validation_split=0.1)
sparse_model.save_weights("model_sparse_0.2", save_format="tf")
```

Note: When calling `save_weights`, use the "tf" format to save the weights.

Performing Iterative Pruning

Load the checkpoint saved from the previous fine-tuning stage. Increase the pruning ratio value to achieve higher levels of sparsity. With each pruning step, fine-tune this sparse model. Repeat this pruning and fine-tuning loop until the sparsity reaches the desired value or until it is observed that the evaluation performance degrades significantly.

```
model.load_weights("model_sparse_0.2")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
sparse_model = runner.prune(ratio=0.5)
```

Getting the Pruned Model

When iterative pruning is completed, a sparse model is generated, with the same number of parameters as the original model but with many of these parameters now set to zero.

Call `get_slim_model()` to remove zeroed parameters from the sparse model and generate the final pruned model:

```
model.load_weights("model_sparse_0.5")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
slim_model = runner.get_slim_model()
```

The runner uses the latest pruning specification to generate the slim model by default. You can see what the latest specification file is with the following command:

```
$ cat .vai/latest_spec
$ ".vai/mnist_ratio_0.5.spec"
```

If this file does not match your sparse model, you can explicitly specify the file path to be used:

```
slim_model = runner.get_slim_model(".vai/mnist_ratio_0.5.spec")
```

You can use keras model saving APIs to save the slim model and reload it for inference or quantization. For example,

```
slim_model.save('/tmp/model')
loaded_model = tf.keras.models.load_model('/tmp/model')
```

vai_p_tensorflow2 APIs

tf_nndct.IterativePruningRunner

Runner for structured pruning of the model in an iterative way. This API has the following methods:

- `__init__(model, input_specs)`

  Creates a new `IterativePruningRunner` object.

  - **model:** A baseline model to prune. The model should be an instance of `keras.Model`.
  - **input_specs:** A single or a list of `tf.TensorSpec` is used to represent model input specifications.

- `ana(eval_fn, excludes=None, forced=False)`

  Performs model analysis. The analysis result is saved in the `'.vai'` directory, and this cached result is used directly in subsequent calls unless `forced` is set to True.

  - **eval_fn:** Callable object that takes a `keras.Model` object as its first argument and returns the evaluation score.
  - **excludes:** A list of layer names or layer instances to be excluded from pruning.
  - **forced:** When set to True, model analysis is run instead of the cached analysis result.

- `prune(ratio=None, threshold=None, spec_path=None, excludes=None, mode='sparse')`

  Prunes the baseline model and returns a sparse model. The degree of pruning can be specified in three ways: ratio, threshold, or pruning specification. The first method is preferred; the latter two are more suitable for experiments with manual tuning.

  - **ratio:** The expected percentage of FLOPs reduction of the baseline model. This is a guidance value, and the actual FLOPs reduction cannot strictly be equal to this value.
  - **threshold:** Relative proportion of model performance loss between the baseline and pruned models.
  - **spec_path:** Pruning specification path is used to prune the model.
  - **excludes:** A list of layer name or layer instance to be excluded from pruning.
  - **mode:** The mode in which the baseline model is pruned to return a sparse model.

- `get_slim_model(spec_path=None)`

  Gets a slim model from a sparse model. By default, you would use the latest pruning specification to do this transformation. A specification path can be provided explicitly if the sparse model was not generated from the latest specification.

  - **spec_path:** The pruning specification path transforms a sparse model into a slim one.

TensorFlow2 Examples

See the Vitis AI repository on GitHub.

PyTorch Version - vai_p_pytorch

The pruning tool on PyTorch is a Python package and not an executable program. vai_p_pytorch provides three methods of model pruning:

- Iterative Pruning
- One-Step Pruning
- Once-for-All (OFA)

Iterative and one-step pruning is suitable for networks with conventional convolution layers but does not work very well on depthwise convolution-based networks such as MobileNet-v2. Convolutional neural networks (CNN) generally contain BatchNormalization layers and one-step pruning is preferred for these networks because it is faster and works better. If no BatchNormalization layers exist in the network, such as VGGNet, iterative pruning should be used.

OFA applies to both depthwise and conventional convolutions. It is important to know that OFA is theoretically the best of these three methods, though it is not easy to get good pruning results. The result depends on how well the supernetwork can be trained; long training time and strong training skills are required.

To summarize, if there are BatchNormalization layers in the network, use one-step pruning. Otherwise, use iterative pruning. If you are unsatisfied with the pruning results, OFA can be an alternative approach.

Fine-grained Pruning

Creating a Model

For simplicity, ResNet50 from `torchvision` is used here.

```
from torchvision.models.resnet import resnet50
model = resnet50(pretrained=True)
```

Creating a Sparse Pruner

The pruner requires two arguments:

- The model to be pruned
- The inputs needed by the model for inference

```
import torch
from pytorch_nndct import SparsePruner

inputs = torch.randn([1, 3, 224, 224], dtype=torch.float32)
pruner = SparsePruner(model, inputs)
```

Generating a Sparse Model

Call `sparse_model()` to get a sparse model. This method finds all the `nn.Conv2d` / `nn.ConvTranspose2d` and `nn.BatchNorm2d` modules replace those modules with `DynamicConv2d` / `DynamicConvTranspose2d` and `DynamicBatchNorm2d`. This method replaces the `nn.Conv2d` / `nn. linear` layers that meet the sparsity condition with `SparseConv2d` / `SparseLinear`.

This method supports `nn.Conv2d` / `nn. linear` weights and activations for simultaneous pruning. The sparsity of activations can be 0 or 0.5. When the sparsity of activations is 0, the sparsity of weights can be 0, 0.5, or 0.75. When the sparsity of activations is 0.5, the sparsity of weights can only be 0.75. block_size is the number of consecutive elements of the input channel. The channel unfolded according to the weights/activation. It is set to 4, 8, or 16. So, the convolution with the weight of the input channel greater than 16 is replaced with sparse convolution.

```
sparse_model =
sparse_pruner.sparse_model(w_sparsity=0.5,a_sparsity=0,block_size=4)
```

Retraining the sparse model is the same as training a baseline model. Knowledge distillation can achieve better accuracy.

Getting a Sparse Model

Call `export_sparse_model()` to get a network converted from sparse networks with sparse weights for inference on hardware for specified sparse computations.

```
model = sparse_pruner.export_sparse_model(sparse_model)
```

Coarse-grained Pruning

The steps for pruning a model according to this method are as follows:

Creating a Model

For simplicity, ResNet18 from `torchvision` is used here.

```
from torchvision.models.resnet import resnet18
model = resnet18(pretrained=True)
```

Creating a Pruning Runner

Import modules and prepare input signature:

```
from pytorch_nndct import get_pruning_runner

# The input signature should have the same shape and dtype as the model
input.
input_signature = torch.randn([1, 3, 224, 224], dtype=torch.float32)
```

Create an iterative pruning runner:

```
runner = get_pruning_runner(model, input_signature, 'iterative')
```

Or, a one-step pruning runner:

```
runner = get_pruning_runner(model, input_signature, 'one_step')
```

Pruning a Model

Iterative Pruning

The method includes two stages: model analysis and pruned model generation. After the model analysis is completed, the analysis result is saved in the file named `.vai/xxx.sens`. You can prune a model iteratively using this file. In iterative pruning, it is necessary to gradually prune the model to the target sparsity. This is accomplished by using an iterative loop comprising both a pruning and a fine-tuning step and a modest pruning ratio per step. Attempting to set the pruning ratio too high results in a steep loss in accuracy that cannot be recoverable.

1. Define an evaluation function. The function must take a model as its first argument and return a score.

   ```
   def eval_fn(model, dataloader):
     top1 = AverageMeter('Acc@1', ':6.2f')
     model.eval()
     with torch.no_grad():
     for i, (images, targets) in enumerate(dataloader):
       images = images.cuda()
       targets = targets.cuda()
       outputs = model(images)
       acc1, _ = accuracy(outputs, targets, topk=(1, 5))
       top1.update(acc1[0], images.size(0))
     return top1.avg
   ```

2. Run model analysis and get a pruned model.

```
runner.ana(eval_fn, args=(val_loader,))

model = pruning_runner.prune(removal_ratio=0.2)
```

The model analysis only needs to be done once. You can prune the model iteratively without re-running analysis because only one pruned model is generated for a specific pruning ratio. The subnetwork obtained by pruning cannot be very good because an approximate algorithm generates this unique pruned model according to the analysis result. The one-step pruning method can generate a better subnetwork.

One-Step Pruning

The method includes two stages: adaptive-BN-based search for pruning strategy and pruned model generation. After searching, a file named `.vai/xxx.search` is generated to store the search result (pruning strategies and corresponding evaluation scores). You can get the final pruned model in one step.

`num_subnet` provides the target number of candidate subnetworks satisfying the sparsity requirement to be identified. The best subnetwork can be selected from these candidates. The higher the value, the longer it takes to search, but the higher the probability of finding a better subnetwork.

```
# Adaptive-BN-based searching for pruning strategy. 'calibration_fn' is a
function for calibrating BN layer's statistics.
runner.search(gpus=['0'], calibration_fn=calibration_fn,
calib_args=(val_loader,), eval_fn=eval_fn, eval_args=(val_loader,),
num_subnet=1000, removal_ratio=0.7)

model = runner.prune(removal_ratio=0.7, index=None)
```

The `eval_fn` is the same as the iterative pruning method. A `calibration_fn` function that implements adaptive-BN is shown in the following example code. You should define your code similarly.

```
def calibration_fn(model, dataloader, number_forward=100):
  model.train()
  with torch.no_grad():
    for index, (images, target) in enumerate(dataloader):
      images = images.cuda()
      model(images)
    if index > number_forward:
      break
```

The one-step pruning method has several advantages over the iterative approach:

- The generated pruned models are typically more accurate. All subnetworks that meet the requirements are evaluated.
- The workflow is simpler because you can obtain the final pruned model in one step without iterations.

- Retraining a slim model is faster than retraining a sparse model.

There are two disadvantages to one-step pruning: One is that the random generation of pruning strategies is unpredictable. The other is that the subnetwork search must be performed once for every pruning ratio.

Retraining a Model

Retraining a model is the same as training a baseline model.

```
optimizer = torch.optim.Adam(model.parameters(), 1e-3, weight_decay=5e-4)
best_acc1 = 0

for epoch in range(args.epoches):
  train(train_loader, model, criterion, optimizer, epoch)
  acc1, acc5 = evaluate(val_loader, model, criterion)

  is_best = acc1 > best_acc1
  best_acc1 = max(acc1, best_acc1)
  if is_best:
    torch.save(model.state_dict(), 'model_pruned.pth')
    # Sparse model has one more special method in iterative pruning.
    if hasattr(model, 'slim_state_dict'):
      torch.save(model.slim_state_dict(), 'model_slim.pth')
```

Generating a Pruned Model

The parameters set to zero in the pruned model are removed from the sparse model. There are two ways to generate a final pruned model.

Using a Pruning API

```
method = 'iterative' # or 'one_step'
runner = get_pruning_runner(model, input_signature, method)
slim_model = runner.prune(removal_ratio=0.2, mode='slim')
slim_model.load_state_dict(torch.load('model_slim.pth'))
```

Without Using a Pruning API

This approach is often used to quantize pruned models as sometimes there can be no way to call the pruning API.

```
from pytorch_nndct.utils import slim

model = create_your_baseline_model()
slim_model = slim.load_state_dict(model, torch.load('model_slim.pth'))
```

Once-for-All (OFA)

Steps for the OFA method are as follows:

Creating a Model

For simplicity, mobilenet_v2 from `torchvision` is used here.

```
from torchvision.models.mobilenet import mobilenet_v2
model = mobilenet_v2(pretrained=True)
```

Creating an OFA Pruner

The pruner requires two arguments:

- The model to be pruned
- The inputs needed by the model for inference

```
import torch
from pytorch_nndct import OFAPruner

inputs = torch.randn([1, 3, 224, 224], dtype=torch.float32)
pruner = OFAPruner(model, inputs)
```

***Note*:** The input does not need to be real data. You can use randomly generated dummy data if it has the same shape and type as the real data.

Generating an OFA Model

Call `ofa_model()` to get an OFA model. This method finds all the `nn.Conv2d` / `nn.ConvTranspose2d` and `nn.BatchNorm2d` modules and replaces those modules with `DynamicConv2d` / `DynamicConvTranspose2d` and `DynamicBatchNorm2d`.

A list of pruning ratios is required to specify a maximum and minimum channel pruning ratio for each layer in the final OFA model.

An arbitrary pruning ratio is used in the output channel for each convolution layer in the OFA model. The maximum and minimum values in this list represent the maximum and minimum compression rates of the model. Other values in the list represent the subnetworks to be optimized. The pruning ratio is set by default to [0.5, 0.75, 1].

For a subnetwork sampled from the OFA model, the output channels of a convolution layer are one of the numbers in the pruning ratio list multiplied by its original number, for example, for a pruning ratio list of [0.5, 0.75, 1] and a convolution layer `nn.Conv2d(16, 32, 5)`, the output channels of this layer in a sampled subnetwork is one of [0.5*32, 0.75*32, 1*32].

Because the first and last layers significantly impact network performance, they are commonly excluded from pruning. By default, this method automatically identifies the first and last convolutions, putting them into the list of excludes. Setting `auto_add_excludes` equals False can cancel this feature.

```
ofa_model = ofa_pruner.ofa_model([0.5, 0.75, 1], excludes = None,
auto_add_excludes=True)
```

Training an OFA Model

This method uses the sandwich rule to jointly optimize all the OFA subnetworks. The `sample_random_subnet()` function is used to get a subnetwork. The dynamic subnetwork is used in both forward and back propagation.

In each training step, given a mini batch of data, the sandwich rule samples a 'max' subnetwork, a 'min' subnetwork, and two random subnetworks. Each subnetwork does a separate forward/backward pass with the given data, and all the subnetworks update their parameters together.

```
# using sandwich rule and sampling subnet.
for i, (images, target) in enumerate(train_loader):

  images = images.cuda(non_blocking=True)
  target = target.cuda(non_blocking=True)

  # total subnets to be sampled
  optimizer.zero_grad()

  teacher_model.train()
  with torch.no_grad():
    soft_logits = teacher_model(images).detach()

  for arch_id in range(4):
    if arch_id == 0:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'max')
    elif arch_id == 1:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'min')
    else:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'random')

    output = model(images)

    loss = kd_loss(output, soft_logits) + cross_entropy_loss(output,
target)
    loss.backward()

  torch.nn.utils.clip_grad_value_(ofa_model.parameters(), 1.0)
  optimizer.step()
  lr_scheduler.step()
```

Searching Constrained Subnetworks

After training is completed, you can conduct an evolutionary search based on the neural-network-twins to get a subnetwork with the best trade-offs between MACs and accuracy using a minimum and maximum MACs range.

```
pareto_global = ofa_pruner.run_evolutionary_search(ofa_model,
calibration_fn, (train_loader,) eval_fn, (val_loader,), 'acc1', 'max',
min_macs=230, max_macs=250)

ofa_pruner.save_subnet_config(pareto_global, 'pareto_global.txt')
```

The evolutionary search result looks like the following:

```
{
"230": {
    "net_id": "net_evo_0_crossover_0",
    "mode": "evaluate",
    "acc1": 69.04999542236328,
    "macs": 228.356192,
    "params": 3.096728,
    "subnet_setting": [...]
}
"240": {
    "net_id": "net_evo_0_mutate_1",
    "mode": "evaluate",
    "acc1": 69.22000122070312,
    "macs": 243.804128,
    "params": 3.114,
    "subnet_setting": [...]
}}
```

Getting a Subnetwork

Call `get_static_subnet()` to get a specific subnetwork. The static_subnet can be used for fine-tuning and quantization.

```
pareto_global = ofa_pruner.load_subnet_config('pareto_global.txt')
static_subnet, static_subnet_config, flops, params = \
ofa_pruner.get_static_subnet(ofa_model, pareto_global['240']
['subnet_setting'])
```

vai_p_pytorch APIs

pytorch_nndct.SparsePruner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.

  - **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as real data.

- `sparse_model(w_sparsity=0.5, a_sparsity=0, block_size=16,excludes=None)`
  - **w_sparsity:** One of ['0', '0.5', '0.75']. A float of weight sparsity of Convolution and Fully Connected layers. By default, the w_sparsity is set to 0.5.
  - **a_sparsity:** One of ['0', '0.5']. A float of sparsity of activations. Here, the activations represent the input feature map of the sparse layer. By default, a_sparsity is set to 0. If a_sparsity is 0.5, w_sparsity must be 0.75.
  - **block_size:** The int number of consecutive elements of the input channel/channel unfolded according to the weights/activations.
  - **excludes:** A list of modules to be excluded from sparsity pruning.
- `export_sparse_model(model)`

  Returns a network converted from sparse networks with sparse weights for inference on hardware for specified sparse computations.
  - **model:** The sparse model.

pytorch_nndct.get_pruning_runner

This API has the following methods:

```
get_pruning_runner(model, inputs, method)
```

- **model:** A `torch.nn.Module` object to prune.
- **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need real data. It can be a randomly generated tensor of the same shape and data type as real data.
- **method:** Either be 'iterative' or 'one_step.'

pytorch_nndct.IterativePruningRunner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as real data.
- `ana(eval_fn, args=(), gpus=None, excludes=None, forced=False)`

- **eval_fn:** Callable object that takes a `torch.nn.Module` object as its first argument and returns the evaluation score.
- **args:** A tuple of arguments that are passed to eval_fn.
- **gpus:** A tuple or list of GPU indices to be used. If not set, the default GPU is used.
- **excludes:** A list of node names or torch modules to be excluded from pruning.
- **forced:** If False, skip model analysis and use cached result.

- `prune(removal_ratio=None, threshold=None, spec_path=None, excludes=None, mode='sparse')`
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **threshold:** Relative proportion of model performance loss that can be tolerated.
  - **spec_path:** Pre-defined pruning specification.
  - **excludes:** A list of node names or torch modules to be excluded from pruning.
  - **mode:** One of ['sparse', 'slim']. Always use 'sparse' in an iterative loop. A slim model is used for quantization-aware training.

pytorch_nndct.OneStepPruningRunner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as real data.
- `search(gpus=['0'], calibration_fn=None, calib_args=(), num_subnet=10, removal_ratio=0.5, excludes=[], eval_fn=None, eval_args=())`
  - **gpus:** A tuple or list of GPU indices to be used. If not set, the default GPU is used.
  - **calibration_fn:** Callable object that takes a `torch.nn.Module` object as its first argument. It is used for calibrating statistics of the BatchNormalization layers.
  - **calib_args:** A tuple of arguments that is passed to calibration_fn.
  - **num_subnet:** Number of subnetworks that satisfy the MACs constraint.
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **excludes:** Modules that need to be excluded from pruning.
  - **eval_fn:** Callable object that takes a `torch.nn.Module` object as its first argument and returns the evaluation score.
  - **eval_args:** A tuple of arguments that is passed to eval_fn.

- `prune(mode='slim', index=None, removal_ratio=None, pruning_info_path=None)`
  - **mode:** One of ['sparse', 'slim']. Should always use 'slim' mode for the one-step method.
  - **index:** Subnetwork index. By default, the optimal subnetwork is selected automatically.
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **pruning_info_path:** A .json file. Save detailed pruning information for the current model. A slim model can be generated with the file and origin model.

pytorch_nndct.OFAPruner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as real data.
- `ofa_model(expand_ratio, channel_divisble=8, excludes=None, auto_add_excludes=True, save_search_space=False)`
  - **expand_ratio:** A list of prune ratio of each convolution layer. An arbitrary pruning ratio can be used for the output channels for each convolution layer in the OFA model.

    The maximum and minimum values in this list represent the maximum and minimum compression rates of the model. Other values represent subnetworks to be optimized. The pruning ratio is set by default to [0.5, 0.75, 0.1].
  - **channel_divisible:** A channel number that is divisible by a given divisor.
  - **excludes:** A list of modules to be excluded from pruning.
  - **auto_add_excludes:** Bool. If True, this method automatically identifies the first and the last convolution and puts them into the list of excludes. If False, skips creating the list of excludes. Defaults to True.
  - **save_search_space:** Bool. If True, save the search space of the model as a file of 'searchspace.config'. You can check the search space for each layer. Defaults to False.
- `sample_subnet(model, mode)`

  Returns a subnetwork and its configuration for a given mode. The subnetwork can do a forward/backward process using a part of the weights from the OFA model and its settings.
  - **model:** The OFA model.
  - **mode:** One of ['random', 'max', 'min'].
- `reset_bn_running_stats_for_calibration(model)`

Resets the running stats of the Batch Normalization layers.

- **model:** The OFA model.

- `run_evolutionary_search(model, calibration_fn, calib_args, eval_fn, eval_args, evaluation_metric, min_or_max_metric, min_macs, max_macs, macs_step=10, parent_popu_size=16, iteration=10, mutate_size=8, mutate_prob=0.2, crossover_size=4)`

  Runs an evolutionary search to find the best subnetwork whose MACs are in a given range.

  - **model:** The OFA model.
  - **calibration_fn:** A BatchNormalization calibration function. All subnetworks share weights in an OFA model, but batch normalization statistics (mean and variance) are not stored when training the OFA model. After the training is completed, the batch normalization statistics must be re-calibrated using the training data for each sampled subnetwork used for evaluation.
  - **calib_args:** The arguments for calibration_fn.
  - **eval_fn:** A function to evaluate the model.
  - **eval_args:** The arguments for eval_fn.
  - **evaluation_metric:** A string of evaluation_metric to record the result.
  - **min_or_max_metric:** One of ['max', 'min']. The maximum or minimum value of the evaluation metric to be recorded in the evolutionary search. For example, when the evaluation metric has an accuracy of top1, record the maximum value of each iteration in the evolutionary search. However, record the minimum value when the evaluation metric is a mean squared error (mse) or mean absolute error (mae).
  - **min_macs:** The minimum MACs of searched subnetworks.
  - **max_macs:** The maximum MACs of searched subnetworks.
  - **macs_step:** The step of MACs for searching. Divide the interval [min_macs, max macs] into segments by macs_step. For each segment, search the best macs-accuracy trade-offs subnetwork.
  - **parent_popu_size:** The number of initial parent population for sampling the given number of random subnetworks whose MACs are in the given range. The larger this number is, the longer the search takes, and the more likely the best results are obtained.
  - **iteration:** The number of iterations for searching or the number of cycles of the whole algorithm.
  - **mutate_size:** The size of mutation. Each value of the subnetwork setting is replaced with another value of the candidate list with a probability of mutate_prob.
  - **mutate_prob:** The probability of mutation.
  - **crossover_size:** The size of the crossover. Sampling two subnetwork settings and swapping any value in the two subnetwork settings randomly.

- `save_subnet_config(setting_config, file_name)`

Saving dynamic/static subnetwork settings with JSON.

- **setting_config:** The configurations for the dynamic subnetwork setting.
- **file_name:** Filepath to save the subnetwork settings.

- `load_subnet_config(file_name)`
  - **file_name:** Filepath to load the subnetwork settings.

PyTorch Examples

See the Vitis AI Optimizer Example on GitHub.

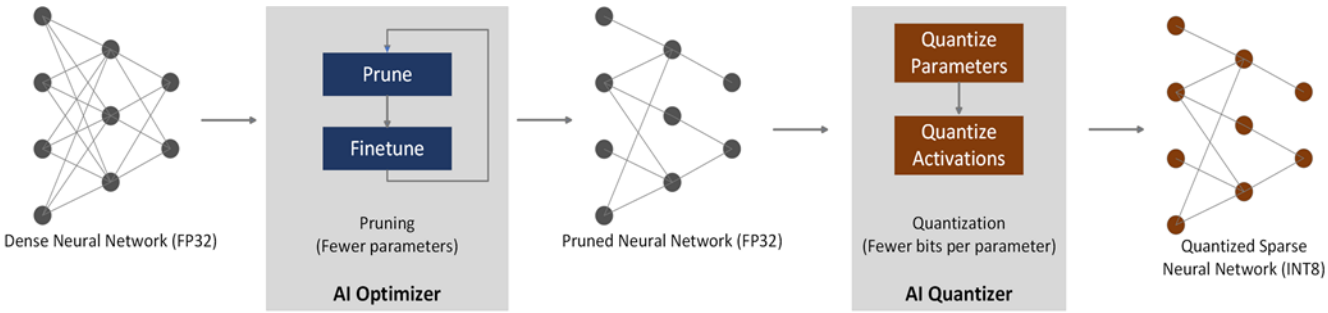
Quantizing the Model

Overview

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

Quantization and channel pruning techniques address these challenges while simultaneously achieving optimal performance and high energy efficiency with minimal degradation in accuracy. Through quantization, integer computing units become viable, and weights and activations can be represented with reduced precision. On the other hand, pruning reduces the overall required operations. The AMD Vitis AI quantizer includes the quantization tool, whereas the pruning tool is integrated into the Vitis AI optimizer.

Figure 16: **Pruning and Quantization Flow**



Generally, 32-bit floating-point weights and activation values are used when training neural networks. The Vitis AI quantizer can reduce computational complexity without losing prediction accuracy by converting the 32-bit floating-point weights and activations to an 8-bit integer (INT8) format. Deployment of the fixed-point network model requires reduced memory bandwidth, thus providing faster speed and higher power efficiency than would be possible with floating-point model. The Vitis AI quantizer supports common layers in neural networks, including, but not limited to, convolution, pooling, fully connected, and batch normalization.

The Vitis AI quantizer supports TensorFlow (1.x and 2.x) and PyTorch. The quantizer names are vai_q_tensorflow and vai_q_pytorch, respectively. In Vitis AI 2.5 and earlier versions, the Vitis AI quantizer for TensorFlow 1.x was based on TensorFlow 1.15 and released as part of the TensorFlow 1.15 package. However, beginning with Vitis AI 3.0, the Vitis AI quantizer is offered as a standalone Python package featuring multiple quantization APIs for both TensorFlow 1.x and TensorFlow 2.x. You can import this package, and once imported, the Vitis AI quantizer functions as a plugin for TensorFlow.

Table 5: **Vitis AI Quantizer Supported Frameworks and Features**

Model	Versions	Features			
		Post Training Quantization (PTQ)	Quantization Aware Training (QAT)	Fast fine-tuning (Advanced Calibration)	Inspector
TensorFlow 1.x	Supports 1.15	Yes	Yes	Yes	No
TensorFlow 2.x	Supports 2.3 - 2.12	Yes	Yes	Yes	Yes
PyTorch	Supports 1.2 - 1.13, 2.0	Yes	Yes	Yes	Yes

Post-training quantization (PTQ) requires only a small set of unlabeled images to analyze the distribution of activations. The run time of post-training quantization varies from a few seconds to several minutes, depending on the neural network size. Generally, there is some tolerable drop in accuracy after quantization. However, the accuracy loss might be considerable for some networks, such as Mobilenet, and this excessive loss might not be tolerable. In such cases, quantization-aware training (QAT) can further improve the accuracy of the quantized models. To conduct QAT, the original training dataset is necessary. The process requires several epochs of fine-tuning, with the fine-tuning duration ranging from several minutes to hours. It is recommended to use small learning rates when performing QAT.

***Note*:** Starting from Vitis AI 1.4, the term *quantize calibration* is replaced with *post-training quantization*, and *quantize fine-tuning* is replaced with *quantization aware training*.

***Note*:** Vitis AI only performs *signed* quantization.It is highly recommended to apply standardization, which involves scaling the input pixel values to have a zero mean and unit variance. It ensures that the DPU sees values within the range of [-1.0, +1.0]. Using scaled unsigned inputs, achieved by dividing the raw input by 255.0 to obtain a range of [0.0, 1.0], results in a loss of dynamic range because only half the input range is used. TensorFlow 2.x and PyTorch quantizers provide configurations to perform unsigned quantization for experiments. The results obtained are not currently deployable for DPUs at this time.

***Note*:** When viewing a model with a tool like Netron, a fix_point parameter for some layers indicates the quantization parameters used for that layer. The fix_point parameter refers to the number of fractional bits used. For example, for 8-bit signed quantization with fix_point= 7, the Q-format representation is Q0.7, which means one sign bit, zero integer bits, and seven fractional bits. To convert an integer value in Q-format to a floating-point, multiply the integer value by 2^-fixed_point.

For post-training quantization, the cross-layer equalization [1] algorithm is implemented. Cross-layer equalization can improve calibration performance, especially for networks including depthwise convolution.

With a small set of unlabeled data, the AdaQuant algorithm [2] not only calibrates the activations but also fine-tunes the weights. AdaQuant uses a small set of unlabeled data, similar to post-training quantization, but it changes the model, which is like fine-tuning. Vitis AI quantizer implements this algorithm and calls it "fast fine-tuning" or "advanced calibration." Fast fine-tuning can perform better than post-training quantization but is slightly slower.

***Note*:** For fast fine-tuning, each run fetches a different result. This is similar to fine-tuning.

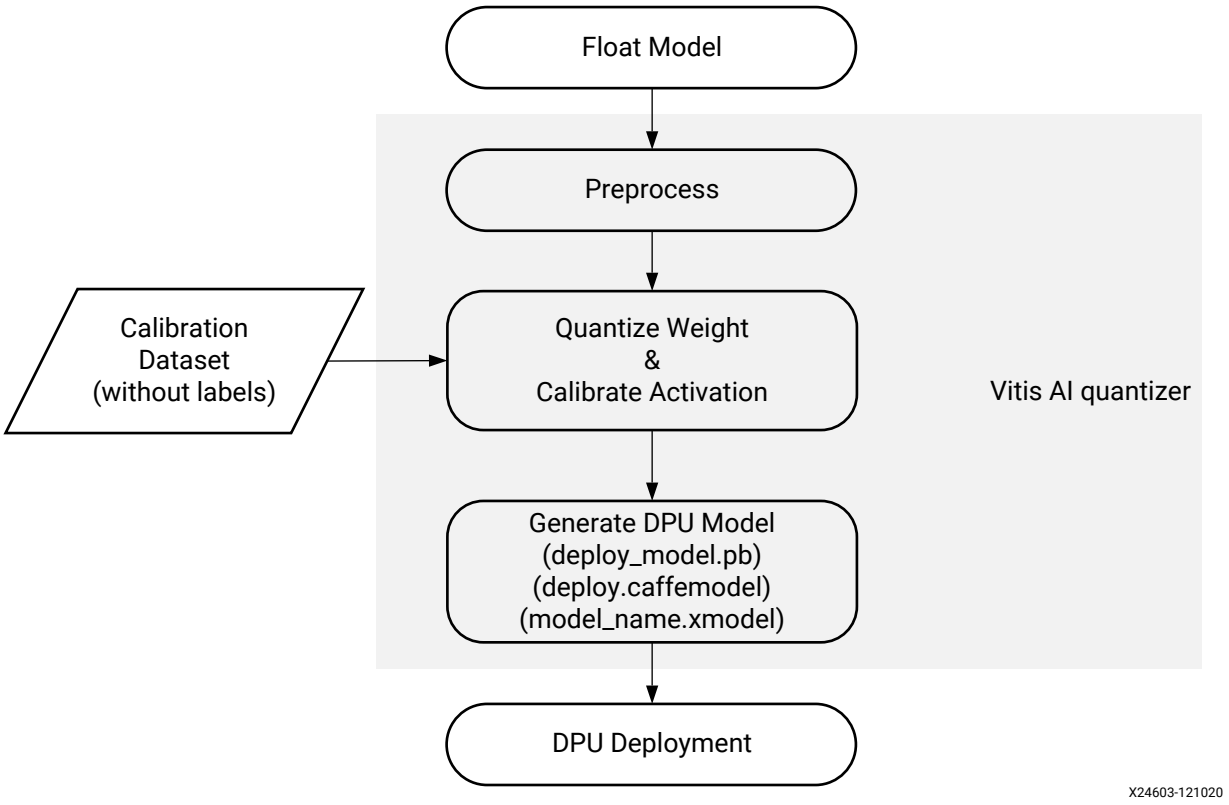
References

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

Vitis AI Quantizer Flow

The following figure depicts the overall model quantization flow.

Figure 17: **VAI Quantizer Workflow**



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

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

Before quantizing the model, it is inspected using the step known as the inspector. The inspector outputs the partition information, indicating which operators can run on which device (DPU/CPU). In general, DPU is faster than CPU, and the idea is to run as many operators as possible on DPU devices. The partition results contain messages explaining why certain operators cannot be executed on the DPU. This helps you to understand DPU's ability better and assists in adapting your model for the DPU.

Vitis AI quantizer requires multiple iterations of inference for calibrating the activations to enhance the accuracy of quantized models and capture activation statistics. This necessitates a calibration image dataset as input. Typically, the quantizer works effectively with 100–1000 calibration images, as backpropagation is unnecessary, and an unlabeled dataset serves the purpose.

After calibration, the quantized model transforms into a DPU deployable format (referred to as deploy_model.pb for vai_q_tensorflow, and model_name.xmodel for vai_q_pytorch), which aligns with the data format of a DPU. The Vitis AI compiler can then compile this model and deploy it to the DPU. However, the standard version of TensorFlow or PyTorch framework cannot directly accept the quantized model.

TensorFlow 1.x Version (vai_q_tensorflow)

Installing vai_q_tensorflow

There are two ways to install the vai_q_tensorflow:

Install Using Docker Containers

Vitis AI provides a Docker container for quantization tools, including vai_q_tensorflow. After running a container, activate the conda environment *Vitis AI-tensorflow*:

```
[docker] $ conda activate vitis-ai-tensorflow
```

Install the Vitis AI-tensorflow patch package inside the Docker container if there is a package-tensorflow patch package inside the Docker container:

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

Install Using Source Code

vai_q_tensorflow is an AMD-maintained plug-in tool for TensorFlow 1.15. It is open source in Vitis_AI_Quantizer. To build vai_q_tensorflow, run the following command:

```
[host] $ sh build.sh
```

Running vai_q_tensorflow

Preparing the Float Model and Related Input Files

Before executing vai_q_tensorflow, ensure that you have the frozen inference TensorFlow model in floating-point format and the calibration set ready, which should include the files listed in the following table.

Table 6: **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 post-training quantization. Usually performs data pre-processing and augmentation.

Generating the Frozen Inference Graph

When using TensorFlow 1.x to train a model, the process creates a folder that includes a GraphDef file (typically with a .pb or .pbtxt extension) and a set of checkpoint files. You need a single GraphDef file that has been frozen or had its variables converted into inline constants for mobile or embedded deployment 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.

The following is an example of command-line usage:

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

The `-input_graph` should be an inference graph other than the training graph. Because the operations of data preprocessing and loss functions are not required for inference and deployment, the `frozen_graph.pb` should only include the essential components of the model. Particularly, the `Input_fn` should take in the data pre-processing operations to generate correct input data for post-training quantization.

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

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

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

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

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

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

```
[docker] $ pip install netron
[docker] $ netron /tmp/inception_v3_inf_graph.pb
```

Preparing the Calibration Dataset and Input Function

The calibration set is typically a subset of the training, validation dataset, or actual application images (consisting of at least 100 for optimal performance). The input function is a Python importable function that handles data pre-processing. This function loads the calibration dataset and performs the necessary data pre-processing steps. The vai_q_tensorflow quantizer can accept an input_fn for pre-processing, which is not saved in the graph. However, if the pre-processing subgraph is saved into the frozen graph, the input_fn only needs to read the images from the dataset and return a feed_dict.

The input function follows the `module_name.input_fn_name` format (for example, `my_input_fn.calib_input`). It accepts an int object representing the calibration step number and returns a dict object with `placeholder_name, numpy.Array` for each call. The object is fed into the model's placeholder nodes during inference. The `placeholder_name` always corresponds to the input node of the frozen graph, which serves as the node for receiving input data.

***Note*:** `placeholder_name` should be replaced with the actual name of the input node receiving the input images. For example, if the input placeholder node is named *the_input_node*, the `placeholder_name` should be replaced with *the_input_node*.

The `input_nodes` in the vai_q_tensorflow options indicate where the quantization starts in the frozen graph. The `placeholder_names` and the `input_nodes` option are sometimes different. When the frozen graph incorporates in-graph pre-processing, the placeholder_name represents the input of the graph. However, setting input_nodes to the last node of the pre-processing steps is advisable. Ensure that the shape of `numpy.the array` is consistent with the corresponding placeholders. Here's a pseudo-code example for reference:

```
$ "my_input_fn.py"
def calib_input(iter):
"""
A function that provides input data for the calibration
  Args:
    iter: A `int` object, indicating the calibration step number
  Returns:
    dict( placeholder_name, numpy.array): a `dict` object, which will be
```

```
fed into the model
"""
  image = load_image(iter)
  preprocessed_image = do_preprocess(image)
  return {"placeholder_name": preprocessed_images}
```

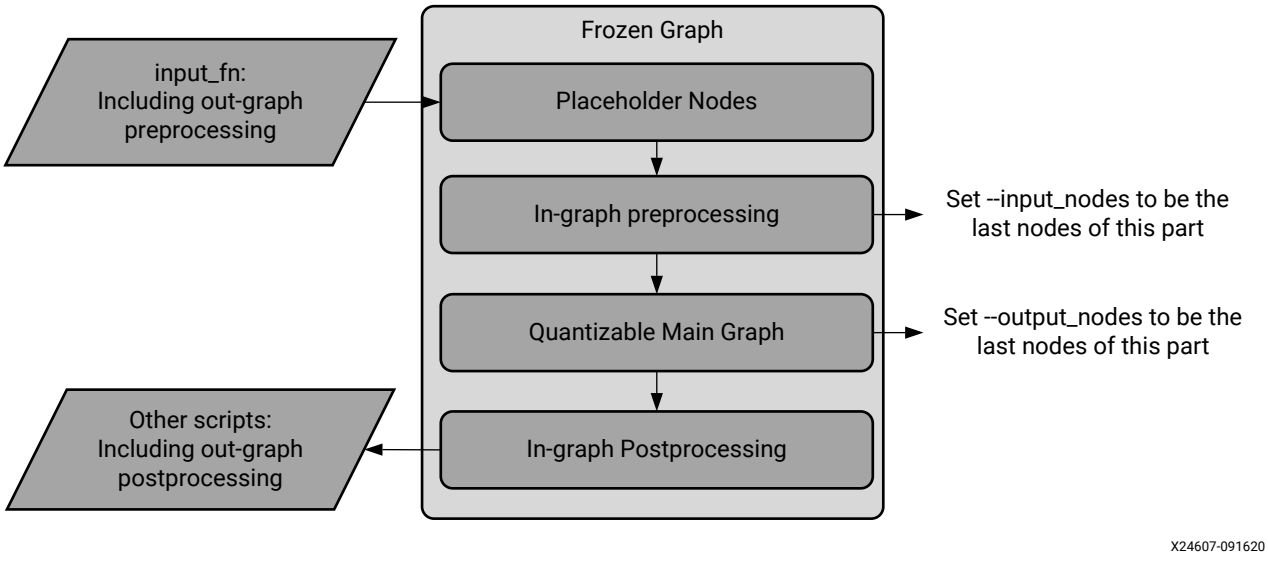
Quantizing the Model Using vai_q_tensorflow

Run the following commands to quantize the model:

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

The `input_nodes` and `output_nodes` arguments are the name list of input nodes of the quantize graph. They serve as 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 18: **Quantization Flow for TensorFlow**



It is recommended to set `-input_nodes` as the last nodes of the pre-processing part and `-output_nodes` as the last nodes of the main graph because some operations in the pre-and-post-processing parts are not quantizable. It might cause errors when the model is compiled by the Vitis AI compiler and deployed to the DPU.

The input nodes might not be the same as the placeholder nodes of the graph. The placeholder nodes should be set as input nodes if the frozen graph does not contain in-graph pre-processing.

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

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

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

Generating the Quantized Model

After the successful execution of the `vai_q_tensorflow` command, an output file is generated in `${output_dir}`. Use `quantize_eval_model.pb` to evaluate the accuracy of the quantized model on CPU and GPUs and to simulate the results on hardware.

Table 7: **vai_q_tensorflow Output Files**

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

(Optional) Exporting the Quantized Model to ONNX

By default, the quantized model is in the TensorFlow Protocol Buffers (Protobuf) format. If you want an ONNX format model, add the `output_format` argument to the `vai_q_tensorflow` command:

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

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

(Optional) Evaluating the Quantized Model

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

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

(Optional) Dumping the Simulation Results

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

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

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

The `input_fn` for dumping is similar to the `input_fn` for post-training quantization, but the batch size is often set to 1 to be consistent with the DPU results.

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

Table 8: **Examples of Dump Results**

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

vai_q_tensorflow Quantization Aware Training

Quantization aware training (QAT) is similar to floating-point model training/finetuning. However, in QAT, the vai_q_tensorflow APIs convert the floating-point graph to a quantized graph before the training starts. Here is the typical workflow:

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

   Table 9: **Input Files for vai_q_tensorflow QAT**

No.	Name	Description
1	Checkpoint files	Floating-point checkpoint files from which to start. Ignore this if you are training the model from scratch.
2	Dataset	The training dataset with labels.
3	Train Scripts	The Python scripts for running the float training/finetuning of the model.

2. Evaluate the floating-point model (optional): Evaluate the float checkpoint files before performing quantize finetuning to check the accuracy of the scripts and dataset. The accuracy and loss values of the float checkpoint can also be a baseline for QAT.
3. Modify the training scripts: To create the quantize training graph, modify the training scripts to call the function after the floating-point graph is built. The following is an example:

```
# train.py

# ...

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

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

vai_q_tensorflow.CreateQuantizeTrainingGraph(config=q_config)

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

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

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

The `QuantizeConfig` contains the configurations for quantization.

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

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

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

  ***Note*:** Operations with multiple output tensors (such as FIFO) are currently unsupported. You can add a tf.identity node to make an alias for the input_tensor to make a single output input node.

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

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

   Note: `vai_q_tensorflow.CreateQuantizeTrainingGraph` and `vai_q_tensorflow.CreateQuantizeEvaluationGraph` functions modify the default graph in TensorFlow. They must be called on different graph phases. `vai_q_tensorflow.CreateQuantizeTrainingGraph` must be called on the float training graph while `vai_q_tensorflow.CreateQuantizeEvaluationGraph` needs to be called on the float evaluation graph. `vai_q_tensorflow.CreateQuantizeEvaluationGraph` cannot be called right after calling the `vai_q_tensorflow.CreateQuantizeTrainingGraph` function because the default graph has been converted to a quantize training graph. The correct approach is to call it after the floating-point model creation function.

```
# eval.py

# ...

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

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

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

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

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

Generated Files

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

Table 10: **Generated File Information**

Name	TensorFlow Compatible	Usage	Description
quantize_train_graph.pb	Yes	Train	The quantize train graph.
quantize_eval_graph_{suffix}.pb	Yes	Evaluation with checkpoint	The quantize evaluation graph with quantize information frozen inside. This file has weights and should be used with the checkpoint file for evaluation.

Table 10: **Generated File Information** *(cont'd)*

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

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

QAT APIs for TensorFlow 1.x

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

vai_q_tensorflow.CreateQuantizeTrainingGraph (config)

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

Arguments

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

vai_q_tensorflow.CreateQuantizeEvaluationGraph(config)

Convert the float evaluation graph to quantize evaluation graph. This is done by in-place rewriting of the default graph.

Arguments

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

vai_q_tensorflow.CreateQuantizeDeployGraph(checkpoint, config)

Freeze the checkpoint into the quantize evaluation graph.

Arguments

- **`checkpoint`:** A `string` object that specifies the path to the checkpoint folder or file.

- **config:** A `vai_q_tensorflow.QuantizeConfig` object that contains the configurations required for quantization.

Tips for QAT

The following are some tips for QAT.

- **Keras Model:**

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

- **Dropout:** Experiments show that QAT works better without dropout ops. This tool does not support finetuning with dropouts, and they should be removed or disabled before running QAT. This can be achieved by setting `is_training=false` when using tf.layers, or by calling `tf.keras.backend.set_learning_phase(0)` when using tf.keras.layers.

- **Hyper-parameter:** QAT is like floating-point model training/finetuning, so the techniques used in those are required in QAT. The optimizer type and the learning rate curve are important parameters to tune.

Converting to Float16 or BFloat16

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

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

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

***Note*:** The BatchNorm operation is folded in advance to implement the conversion.

vai_q_tensorflow Supported Operations and APIs

The following table lists the supported vai_q_tensorflow operations and APIs.

Table 11: **Supported Operations and APIs for vai_q_tensorflow**

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

Table 11: **Supported Operations and APIs for vai_q_tensorflow** *(cont'd)*

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

Notes:

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

***Note*:** The list of operators supported by the Vitis AI quantizer is not the only limiting factor for model deployment, and users should also review operator support for their selected DPU architecture. Operators not supported by the DPU can be executed on the CPU. For more information, see Supported Operators and DPU Limitations.

vai_q_tensorflow Usage

The following table shows the `vai_q_tensorflow` options.

Table 12: **vai_q_tensorflow Options**

Name	Type	Description
Common Configuration		
--input_frozen_graph	String	TensorFlow frozen inference GraphDef file for the floating-point model. It is used for post-training quantization.
--input_nodes	String	Specifies the name list of input nodes of the quantize graph to be used with `-output_nodes`, separated by commas. Input nodes and output nodes are the starting and ending points of quantization. The subgraph between them is quantized if it is quantizable. **RECOMMENDED:** *Set `-input_nodes` as the last nodes for pre-processing and `-output_nodes` as the last nodes for post-processing because some of the operations required for pre-and- and post-processing are not quantizable. They might cause errors when the model is compiled by the Vitis AI compiler deployed to the DPU. The input nodes might not be the same as the placeholder nodes of the graph.*
--output_nodes	String	Specifies the name list of output nodes of the quantize graph, combined with `-input_nodes`, separated by commas. Input nodes and output nodes are the starting and ending points of quantization. The subgraph between them is quantized if it is quantizable. **RECOMMENDED:** *Set `-input_nodes` as the last nodes for pre-processing and `-output_nodes` as the last nodes for post-processing because some of the operations required for pre-and- and post-processing are not quantizable. They might cause errors when the model is compiled by the Vitis AI compiler and deployed to the DPU.*
--input_shapes	String	Specifies the shape list of input nodes. It must be a four-dimensional shape for each node, separated by commas. For example, 1,224,224,3. Supports unknown size for batch_size, for example, 224,224,3. In case of multiple input nodes, assign the shape list of each node separated by, for example,? 224,224,3:? 300,300,1.

Table 12: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description

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

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

Table 12: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--calib_iter	Int32	Specifies the calibration iterations. A total number of images for calibration = calib_iter * batch_size. Default value: 100
--ignore_nodes	String	Specifies the 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 the floating-point model is skipped. Useful when only part of the input model is quantized. Range: [0, 1] Default value: 0
--align_concat	Int32	Specifies the strategy for aligning the input quantize position for concat nodes. 0: Aligns all the concat nodes 1: Aligns the output concat nodes 2: Disables alignment Default value: 0
--align_pool	Int32	Specifies the strategy for aligning the input quantize position for maxpool/avgpool nodes. 0: Aligns all the maxpool/avgpool nodes 1: Aligns the output maxpool/avgpool nodes 2: Disables alignment Default value: 0
--simulate_dpu	Int32	Set to 1 to enable DPU simulation. The behavior of DPU for some operations is different from TensorFlow. For example, the dividing in LeakyRelu and AvgPooling are replaced by bit-shifting, so there might be a slight difference between DPU outputs and CPU/GPU outputs. The vai_q_tensorflow quantizer simulates the behavior of these operations if this flag is set to 1. Range: [0, 1] Default value: 1
--adjust_shift_bias	Int32	Specifies the strategy for shift bias check and adjustment for the DPU compiler. 0: Disables shift bias check and adjustment 1: Enables with static constraints 2: Enables with dynamic constraints Default value: 1
--adjust_shift_cut	Int32	Specifies the shift cut check and adjustment strategy for the DPU compiler. 0: Disables shift cut check and adjustment 1: Enables with static constraints Default value: 1
--arch_type	String	Specifies the arch type for fixed neuron. *DEFAULT* means the quantization range of weights and activations is [-128, 127]. 'DPUCADF8H' means the weights quantization range is [-128, 127] while activation is [-127, 127]

Table 12: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
--output_dir	String	Specifies the directory to save the quantization results. Default value: "./quantize_results"
--max_dump_batches	Int32	Specifies the maximum number of batches for dumping. Default value: 1
--dump_float	Int32	If set to 1, the float weights and activations are dumped. Range: [0, 1] Default value: 0
--dump_input_tensors	String	Specifies the Graph's input tensor name when the graph entrance is not a placeholder. Add a placeholder to the `dump_input_tensor` so that `input_fn` can feed data.
--scale_all_avgpool	Int32	Set to 1 to enable scale output of AvgPooling op to simulate DPU. Only kernel_size <= 64 is scaled. This operation does not affect special cases such as kernel_size=3,5,6,7,14 Default value: 1
--do_cle	Int32	1: Enables implementation of cross-layer equalization to adjust the distribution of the weight 0: Skips cross-layer equalization operation Default value: 0
--replace_relu6	Int32	Available only for do_cle=1 1: Replace ReLU6 with ReLU 0: Skips replacement Default value: 1
--replace_sigmoid	Int32	1: Replace sigmoid with hard-sigmoid 0: Skips replacement Default value: 0
--replace_softmax	Int32	1: Replace softmax with hard-softmax 0: Skips replacement Default value: 0
--convert_datatype	Int32	4: Do BN folding and convert to data type fp32 3: Do BN folding and convert to data type bfloat16 2: Do BN folding and convert to data type double 1: Do BN folding and convert to data type fp16 0: Skips conversion Default value: 0
--output_format	String	Indicates the format to save the quantized model, *pb* for saving tensorflow frozen pb, *and onnx* for saving the ONNX model. Default value: 'pb'
Session Configurations		
--gpu	String	Specifies GPU device IDs used for quantization, separated by commas.
--gpu_memory_fraction	Float	Specifies the GPU memory fraction used for quantization, between 0-1. Default value: 0.5

Table 12: **vai_q_tensorflow Options** *(cont'd)*

Name	Type	Description
Others		
--help		Shows all available vai_q_tensorflow options.
--version		Shows the vai_q_tensorflow version information.

Examples

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

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

vai_q_tensorflow Error Codes

Table 13: **vai_q_tensorflow Error Codes**

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

TensorFlow 2.x Version (vai_q_tensorflow2)

Installing vai_q_tensorflow2

You can install vai_q_tensorflow2 in the following three ways:

Install Using Docker Container

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

```
[docker] $ conda activate vitis-ai-tensorflow2
```

If there is a patch package, install the Vitis AI-tensorflow2 patch package inside the Docker container.

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

Install from Source Code with the Wheel Package

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

```
[host] $ sh build.sh
[host] $ pip install pkgs/*.whl
```

Install from Source Code with the Conda Package

IMPORTANT! *This requires Anaconda.*

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

Inspecting the Float Model

`VitisInspector` is a helper tool that inspects a float model, shows partition results for a given DPU target architecture, and indicates why the layers are not mapped to DPU. Without `target`, you can only show some general, target-independent inspection results. Assign `target` to get more detailed inspection results for it.

***Note*:** This feature is only available for the default `pof2s` quantize strategy because of DPU limitations.

The following code shows how to inspect a model:

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

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

  ***Note*:** Only some output file types can show the hints, such as `.svg`.The default value is False.
- **plot_file:** *string*. The file path of the model image file when plotting the model. The default value is `model.svg`.
- **dump_results:** *bool*. Whether to dump the inspection results and save text to disk. More detailed layer-by-layer results than screen logging are dumped into the text file. The default value is False.

- **dump_results_file:** *string*. The file path of inspect results text file. The default value is `inspect_results.txt`.
- **verbose:** *int*. The logging verbosity level. More detailed logging results are shown for higher verbose values. The default value is 0.

Running vai_q_tensorflow2

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

- **Post-training quantization (PTQ):** PTQ is a technique to convert a pre-trained floating-point model into a quantized model with little degradation in model accuracy. To perform PTQ, a representative dataset is required to run a few batches of inference on the floating-point model, which helps obtain the distributions of the activations. This process is also known as quantize calibration.
- **Quantization aware training (QAT):** QAT models the quantization error in both the forward and backward passes during model quantization. When using QAT, it is recommended to begin with a floating-point pre-trained model that already exhibits good accuracy rather than starting from scratch.

Preparing the Float Model and Calibration Set

Before running vai_q_tensorflow2, ensure you have prepared the floating-point model and calibration set, which include the files listed in the following table.

Table 14: **Input Files for vai_q_tensorflow2**

No.	Name	Description
1	Float model	Floating-point TensorFlow 2 models in either h5 format or saved model format.
2	Calibration dataset	A subset of the training or validation dataset used to represent the input data distribution. Typically, 100 to 1000 images are sufficient for calibration.

Quantizing Using the vai_q_tensorflow2 API

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

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

- **calib_dataset:**

The `calib_dataset` is a representative calibration dataset used during the calibration process. It can be derived from the `eval_dataset`, `train_dataset`, or other datasets in full or part.

- **calib_steps:** `calib_steps` represents the total number of steps for calibration. By default, it is set to None. If `calib_dataset` is a `tf.data dataset`, generator, or `keras.utils.Sequence` instance and steps are None, calibration continues until the dataset is fully exhausted. Array inputs do not support this argument.
- **calib_batch_size:** `calib_batch_size` determines the number of samples per batch used during calibration. If the `calib_dataset` is in the form of a dataset, generator, or `keras.utils.Sequence` instance, the batch size is controlled by the dataset itself. However, if the `calib_dataset` is in the form of a `numpy.array` object, the default batch size is 32.
- **input_shape:** *list(int) or list(list(int)) or tuple(int) or dictionary(int)*. Contains the input shape for each input layer. Use the default shape information from the input layers if not explicitly set. Use list of shapes for multiple inputs, for example input_shape=[1, 224, 224, 3] or input_shape=[[None, 224, 224, 3], [None, 64, 1]]. All dimensions should have concrete values, and the batch_size dimension should be None or int. If the input shape of the model is variable like [None, None, None, 3], you need to specify a shape such as [None, 224, 224, 3] to generate the final quantized model.
- ****kwargs:** `**kwargs` represents a dictionary of user-defined configurations for the quantize strategy. It enables you to override the default built-in quantize strategy. For example, setting `bias_bit=16` quantizes all the biases with 16bit quantizers. For more information on user-defined configurations, refer to the vai_q_tensorflow2 Usage section.

(Optional) vai_q_tensorflow2 Fast Finetuning

Generally, there is a minor accuracy loss after quantization, but for specific networks like MobileNet, the accuracy loss can be significant. To address this, fast fine-tuning uses the AdaQuant algorithm, adjusting weights and quantizing parameters layer-by-layer with the unlabeled calibration dataset to improve accuracy for specific models.

Although fast fine-tuning takes longer than normal PTQ (still significantly shorter than QAT, given the smaller calib_dataset), it is turned off by default. However, you can enable it to enhance performance if you encounter accuracy issues. A recommended workflow is to try PTQ without fast fine-tuning, then attempt quantization with fast fine-tuning if the accuracy is unsatisfactory.

While QAT is another method to improve accuracy, it requires more time and relies on the training dataset. To activate fast fine-tuning during post-training quantization, set `include_fast_ft=True`.

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

Here,

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

Saving the Quantized Model

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

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

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

(Optional) Exporting the Quantized Model to ONNX

The following code shows how to perform post-training quantization and export the quantized model to ONNX with vai_q_tensorflow2 API:

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

- **output_format:** *String*. Indicates the format to save the quantized model. Options are:
  - '' for skip saving
  - *h5* for saving .h5 file
  - *tf* for saving the saved_model file
  - *onnx* for saving the ONNX file

  The default value is ''.
- **onnx_opset_version:** *Int*. The ONNX opset version. It takes effect only when output_format is 'onnx.' The default value is 11.
- **output_dir:** *String*. Indicates the directory to save the quantized model. The default value is './quantize_results.'

(Optional) Evaluating the Quantized Model

If you have scripts to evaluate float models, such as the models in AMD Model Zoo, you can replace the float model file with the quantized model for evaluation. You must import the Vitis _quantize module to ensure support for customized quantize layers. For example:

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

After that, evaluate the quantized model similar to the float model. For example:

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

(Optional) Dumping the Simulation Results

Sometimes, after deploying the quantized model, comparing simulation results on the CPU/GPU with the output values on the DPU becomes essential. You can use the `VitisQuantizer.dump_model` API of vai_q_tensorflow2 to dump the simulation results using the quantized model.

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

Note: Ensure that the `batch_size` of the dump_dataset is set to the same batch_size as the one used on the target device for DPU debugging. For DPU debugging, it is recommended to use CPU simulation results, as GPU results might exhibit non-deterministic behavior and slight differences in float value computation.

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

Table 15: **Example of Dumping Results**

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

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

Quantization Strategy Configuration

This section provides some default quantize strategies, but you must customize quantize configurations to suit different targets or achieve better performance. For instance, specific target devices might require biases to be quantized into 32 bits, while others might require quantization for specific model parts. This section demonstrates how to configure the quantizer to meet such requirements.

Quantize Strategy

The quantize tool enables you to configure three main aspects: the quantize tool pipeline, which parts of the model to quantize, and how to perform the quantization. Define them in `quantize_strategy`, which is a JSON file containing the following configurations:

- **pipeline_config:**

  These configurations control the work pipeline of the quantize tool, including some optimizations during quantization. Examples of such optimizations include deciding whether to fold Conv2D + BatchNorm layers and whether to employ the `Cross-Layer-Equalization` algorithm and other optimizations. It can be further divided into `optimize_pipeline_config`, `quantize_pipeline_config`, `refine_pipeline_config` and `finalize_pipeline_config`.

- **quantize_registry_config:** These configurations control what layer types are quantizable, where to insert the quantize ops, and what kind of quantize op to be inserted. It includes some layer-specific configurations and user-defined global configurations.

The following is an example configuration for the Conv2D layers:

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

Using this quantization configuration, you can apply quantization to the weight, bias, and activations of the Conv2D layer. The weight and bias use Pof2SQuantizer (power-of-2 scale quantizer), while the activation uses FSQuantizer (float scale quantizer). You can employ different quantizers for different objects within a single layer.

***Note*:** `Quantizer` refers to the quantize operation applied to each object in these configurations. It takes a float tensor as input and produces a quantized tensor as output. The quantization process is termed "fake," which means the input is quantized to an integer and then de-quantized back to a float.

Using Built-in Quantize Strategy

You can use dump_quantize_strategy to obtain the JSON file of the current quantize strategy. For simplicity, four types of built-in quantize strategies for common user cases are provided and can be extended or overridden to meet specific requirements. These built-in quantize strategies include:

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

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

You can switch between the built-in quantize strategies by assigning `the quantize_strategy` argument in the construct function of `VitisQuantizer`. The following are two handy ways to configure the quantized strategy:

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

This is an easy way for you to override the default pipeline configurations or make global modifications on the quantize operations. The `kwargs` is a dict object whose keys match the JSON file's quantize configurations. For more information about available keys, see vitis_quantize.VitisQuantizer.quantize_model .

The following example code shows how to use it:

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

The quantizer is set up in this example to quantize only part of the model. Layers preceding `conv2` are neither optimized nor quantized. Also, all the activations and biases are quantized to 32-bit instead of 8-bit, and per-channel quantization is applied to all weights.

Configure by `VitisQuantizer.set_quantize_strategy()`

For advanced users seeking complete control over the quantize tool, this API enables you to set a new quantize strategies JSON file. You can dump the current configurations into a JSON file and make modifications as desired. It empowers you to override default configurations, create more fine-grained quantizer settings, or extend the quantize config to include additional layer types for quantization. After making the modifications, you can set the new JSON file to the quantizer to apply these customized configurations.

The following example code shows how to do it:

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

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

Note: `verbose` is an `int` type argument that controls the verbosity of the dumped JSON file. The greater verbose value dumps a more detailed quantized strategy. Setting verbose to a value greater or equal to two dumps the full quantize strategy.

Quantizing with Float Scale

The quantization for DPU uses power-of-2 scales, symmetry, and per-tensor quantizers and needs some special processes to simulate DPU behaviors. However, for other devices that support floating-point scales, a different quantize strategy is needed, therefore, float-scale quantization is introduced.

- **The `fs` quantize strategy:** Performs quantization for inputs and weights of `Conv2D`, `DepthwiseConv2D`, `Conv2DTranspose`, and `Dense` layers. Conv-BN folding is not performed by default.
- **The `fsx` quantize strategy:** Performs quantization for more layer types than the `fs` quantize strategy, such as `Add`, `MaxPooling2D`, and `AveragePooling2D`. In addition, the quantization process extends to the biases and activations of `Conv2D`, `DepthwiseConv2D`, `Conv2DTranspose` and `Dense` layers. By default, it includes Conv-BN folding.

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

You can switch to using float scale quantization by setting `quantize_strategy` to `fs` or `fsx` in the construct function of `VitisQuantizer`. The following is an example code:

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

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

- **kwargs: `**kwargs` is the dict of the user-defined configurations of quantize strategy. It enables users to override the default built-in quantize strategy. For instance, setting bias_bit=16 enables the tool to quantize all the biases using 16-bit quantizers. For more information on the user-defined configurations, see the vai_q_tensorflow2 Usage section.

Converting to Float16 or BFloat16

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

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

- **convert_datatype:** *String*. Indicates the target data type for the float model. Options are *float16*, *bfloat16*, *float32*, and *float64*. The default value is *float16*.

vai_q_tensorflow2 Quantization Aware Training

Generally, quantization might lead to a slight accuracy loss in the model. However, for specific networks like MobileNets, the accuracy loss can be more significant. To address this, Quantization Aware Training (QAT) offers a solution to enhance the accuracy of quantized models further.

QAT is similar to training/finetuning floating-point models, except that vai_q_tensorflow2 rewrites the float graph to convert it into a quantized model before the training begins. You can find a complete example here.

The typical workflow for QAT is as follows:

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

   Before QAT, prepare the following files:

   Table 16: **Input Files for vai_q_tensorflow2 QAT**

No.	Name	Description
1	Float model	Floating-point model files from which to start. You can ignore this if you are training from the scratch.
2	Dataset	The training dataset with labels.
3	Training Scripts	The Python scripts to run float train/finetuning of the model.

2. (Optional) Evaluate the float model.

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

3. Modify the training scripts and run QAT.

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

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


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

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


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

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

4. Save the model.

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

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

# save the model periodically during fit using callbacks
qat_model.fit(
    train_dataset,
    callbacks = [
                keras.callbacks.ModelCheckpoint(
```

```
                filepath='./quantize_train/'
                save_best_only=True,
                monitor="sparse_categorical_accuracy",
                verbose=1,
      )])
```

5. Convert to a deployable quantized model.

   Modify the trained/finetuned model to meet the compiler requirements. For example, if `train_with_bn` is set to TRUE, the batch normalization layers remain unfolded during training and must be folded before deployment. Some quantizer parameters might vary during training and exceed the compiler limitation ranges. These must be corrected before deployment.

   Use the `get_deploy_model()` function to perform these conversions and generate a deployable model, as shown in the following example:

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

6. (Optional) Evaluate the quantized model

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

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

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

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

Quantizing with Custom Layers

TensorFlow 2 offers a rich set of built-in layers to construct machine learning models, and it also provides straightforward methods to create application-specific layers from scratch or by combining existing ones. The `layer` is a central abstraction in `tf.keras`, and subclassing this class is the recommended approach to developing custom layers. For more detailed information, refer to the TensorFlow user guide.

vai_q_tensorflow2 supports new custom layers through subclassing, which includes the capability to quantize models with custom layers. Also, it provides experimental support for quantizing these custom layers using custom quantize strategies.

Note: Custom model through subclassing `tf.keras.T` is not supported by vai_q_tensorflow2 in this release. Flatten it into layers.

Quantizing models with custom layers

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

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

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


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


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


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


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

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

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

During the quantization, these custom layers are wrapped by `CustomLayerWrapper` and kept unquantized. For a complete example, click here.

Note: When using the `dump_model` API to generate golden results for data checking during deployment, set `dump_float=True` to dump float weights and activations for the custom layers, as these layers are not quantized.

(Experimental) Quantizing custom layers with custom quantize strategy

The default quantize strategy does not quantize custom layers, as they are not included in the list of supported APIs for vai_q_tensorflow2. However, advanced users can create custom quantize strategies using the `custom_quantize_strategy` interface to conduct quantization experiments.

The custom quantize strategy is represented as a Dict object containing the quantize strategy items in JSON format.

The default quantize strategy provides an example of the quantize strategy format, and the custom quantize strategy follows the same structure. However, any item in the custom quantize strategy overrides the corresponding one in the default strategy while new items are added to the quantize strategy.

With this feature, you can quantize the `MyCustomLayer` layer from the previous example using a custom quantize strategy.

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

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

vai_q_tensorflow2 Supported Operations and APIs

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

Table 17: **vai_q_tensorflow2 Supported Layers**

Layer Types	Supported Layers	Description
Core	tf.keras.layers.InputLayer	
Core	tf.keras.layers.Dense	
Core	tf.keras.layers.Activation	If *activation* is *relu* or *linear*, it is quantized. If *activation* is *sigmoid* or *swish*, it is converted to hard-sigmoid or hard-swish and then quantized by default. Otherwise it is not quantized.
Convolution	tf.keras.layers.Conv2D	
Convolution	tf.keras.layers.DepthwiseConv2D	
Convolution	tf.keras.layers.Conv2DTranspose	
Convolution	tf.keras.layers.SeparableConv2D	
Pooling	tf.keras.layers.AveragePooling2D	
Pooling	tf.keras.layers.MaxPooling2D	
Pooling	tf.keras.layers.GlobalAveragePooling	
Normalization	tf.keras.layers.BatchNormalization	By default, BatchNormalization layers are fused with the previous convolution layers. If they cannot be combined, they are converted to depthwise convolutions. In the QAT mode, BatchNormalization layers are pseudo-fused if `train_with_bn` is set to TRUE. They are fused when the `get_deploy_model` function is called.
Regularization	tf.keras.layers.Dropout	By default, the dropout layers are removed. In the QAT mode, dropout layers are retained if remove_dropout is set to FALSE. It is removed when the get_deploy_model function is called.
Reshaping	tf.keras.layers.Reshape	
Reshaping	tf.keras.layers.Flatten	
Reshaping	tf.keras.UpSampling2D	
Reshaping	tf.keras.ZeroPadding2D	
Merging	tf.keras.layers.Concatenate	
Merging	tf.keras.layers.Add	
Merging	tf.keras.layers.Multiply	
Activation	tf.keras.layers.ReLU	
Activation	tf.keras.layers.Softmax	The input for the Softmax layer is quantized. It can run on the standalone Softmax IP for acceleration.

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

Layer Types	Supported Layers	Description
Activation	tf.keras.layers.LeakyReLU	Only LeakyReLU layer with "alpha=0." One can be quantized and mapped to DPU. Internally, the quantizer automatically converts the alpha value to 26/256 to match the DPU implementation. LeakyReLU layer with other values is not quantized and is mapped to the CPU.
Hard_sigmoid	tf.keras.layer.ReLU(6.)(x + 3.) * (1. / 6.)	The supported hard_sigmoid is from Mobilenet_v3. tf.keras.Activation.hard_sigmoid is not supported now and is not quantized.
Activation	tf.keras.layers.PReLU	

***Note*:** The DPU might have limitations on these supported layers. They can be rolled back to the CPU during compilation. For more information, see Supported Operators and DPU Limitations.

vai_q_tensorflow2 Usage

vitis_inspect.VitisInspector

The construction function of the `VitisInspector` class.

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

Arguments

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

vitis_inspect.VitisInspector.inspect_model

This function performs float model inspection:

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

Arguments

- **model:** *tf.keras.Model instance*. The float model to be inspected. The float model should have concrete input shapes. Build the float model with concrete input shapes or call inspect_model with the *input_shape* argument.
- **input_shape:** *list(int) or list(list(int))* Contains the input shape for each input layer. Use default shape information in the input layers if not set. Use the list of shapes for multiple inputs, for example, inspect_model(model, input_shape=[224, 224, 3]) or inspect_model(model, input_shape=[[224, 224, 3], [64, 1]]). All dimensions should have concrete values, and the batch_size dimension should be omitted. The default value is None.
- **dump_model:** *bool*. Indicates whether to dump the inspected model and save the model to disk. The default value is False.
- **dump_model_file:** *String*. The path of the inspected model file. The default value is inspect_model.h5.
- **plot:** *bool*. Indicates whether to plot the model inspect results by `graphviz` and save the image to disk. It is helpful when you need to visualize the model inspection results with some modification hints. Only some output file types can show the hints, such as `.svg`. The default value is False.
- **plot_file:** *String*. The file path of the model image file when plotting the model. The default value is model. svg.
- **dump_results:** *bool*. Indicates whether to dump the inspect results and save text to disk. More detailed layer-by-layer results than screen logging are dumped into the text file. The default value is False.
- **dump_results_file:** *string*. The file path of inspect results text file. The default value is inspect_results.txt.
- **verbose:** *int*. The logging verbosity level; more detailed logging results show for higher verbose value. The default value is 0.

vitis_quantize.VitisQuantizer

The construction function of class `VitisQuantizer`.

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

Arguments

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

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

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

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

- **custom_quantize_strategy:** *String*. The file path of the custom quantize strategy JSON file.
- **custom_objects:** *Dict*. Mapping names (strings) to custom classes or functions.

vitis_quantize.VitisQuantizer.quantize_model

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

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

Arguments

- **calib_dataset:** A `tf.data.Dataset`, `keras.utils.Sequence`, or `np.numpy` object. It is the representative dataset for calibration. You can use the whole or part of eval_dataset, train_dataset, or other datasets as calib_dataset.
- **calib_steps:** *Int*. The total number of steps for calibration. Ignored with the default value of None. If calib_dataset is a `tf.data` dataset, generator, or `keras.utils.Sequence` instance and steps are Nonem, calibration runs until the dataset is exhausted. Array inputs do not support this argument.
- **calib_batch_size:** *Int*. The number of samples per batch for calibration. If the calib_dataset is a dataset, generator, or `keras.utils.Sequence` instances, the batch size is controlled by the dataset itself. If the calib_dataset is in the form of a `numpy.array` object, the default batch size is set to 32.

- **verbose:** *Int*. The verbosity of the logging. Greater verbose value generates more detailed logging. The default value is 0.
- **add_shape_info:** *bool*. Indicates whether to add shape inference information for custom layers. It must be set to True for models with custom layers.
- ****kwargs:** *dict*. The user-defined configurations of the quantize strategy. It overrides the default built-in quantize strategy. Detailed user-defined configurations are listed below.

Arguments in `**kwargs`

`**kwargs` in this API is a dict of the user-defined configurations of quantize strategy. It overrides the default built-in quantize strategy. For example, setting bias_bit=16 lets the tool quantize all the biases with 16-bit quantizers. The following are detailed user-defined configurations:

- **separate_conv_act:** A `bool` object, whether to separate activation functions from the `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` layers. The default value is `True`.
- **fold_conv_bn:** A `bool` object, whether to fold the batch norm layers into previous `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` layers.
- **convert_bn_to_dwconv:** Named `fold_bn` in Vitis AI 2.0 and previous versions.A `bool` object, whether to convert the standalone BatchNormalization layer into DepthwiseConv2D layers.
- **convert_sigmoid_to_hard_sigmoid:** Named `replace_sigmoid` in Vitis AI 2.0 previous versions.A `bool` object, whether to replace the Activation(activation='sigmoid') and Sigmoid layers into hard sigmoid layers and do quantization. If not, the sigmoid layers will be left unquantized and scheduled on the CPU.
- **convert_relu_to_relu6:** Named `replace_relu6` in Vitis AI 2.0 and previous versions.A `bool` object, whether to replace the ReLU6 layers with ReLU layers.
- **include_cle:** A `bool` object, whether to implement Cross-Layer Equalization before quantization.
- **cle_steps:** A `int` object, the iteration steps to do Cross-Layer Equalization.
- **cle_to_relu6:** Named `forced_cle` in Vitis AI 2.0 and previous versions.A `bool` object, whether to do forced Cross-Layer Equalization for ReLU6 layers.
- **include_fast_ft:** A `bool` object determining whether to perform fast fine-tuning or not. Fast fine-tuning adjusts the weights layer by layer with the calibration dataset and can get better accuracy for some models. Fast fine-tuning is turned off by default. It takes longer than normal PTQ (still much shorter than QAT as calib_dataset is much smaller than the training dataset). Turn it on to improve the performance if you meet accuracy issues.
- **fast_ft_epochs:** An `int` object, the iteration epochs to do fast fine-tuning for each layer.

- **output_format:** A string object, indicates what format to save the quantized model. Options are: '' for skip saving, 'h5' for saving .h5 file, 'tf' for saving saved_model file, 'onnx' for saving .onnx file. The default value is ''.
- **onnx_opset_version:** An int object, the ONNX opset version. Take effect only when output_format is 'onnx.' The default value is 11.
- **output_dir:** A string object, indicates the directory to save the quantized model. The default value is './quantize_results.'
- **convert_datatype:** A string object, which indicates the target data type for the float model. Options are 'float16', 'bfloat16', 'float32', and 'float64'. The default value is 'float16'.
- **input_layers:** A `list(string)` object, names of the start layers to be quantized. Layers before these layers in the model will not be optimized or quantized. For example, this argument can skip some pre-processing layers or stop quantizing the first layer. The default value is `[]`.
- **output_layers:** A `list(string)` object, names of the end layers to be quantized. Layers after these layers in the model will not be optimized or quantized. For example, this argument can skip some post-processing layers or stop quantizing the last layer. The default value is `[]`.
- **ignore_layers:** A `List(string)` object, names of the layers to be ignored during quantization. For example, this argument can be used to skip quantizing some sensitive layers to improve accuracy. The default value is `[]`.
- **input_bit:** An `int` object, the bit width of all inputs. The default value is 8.
- **input_method:** An `int` object, the method to calculate scale factors in quantizing all inputs. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, and 3 for `Percentile`. The default value is 0.
- **input_symmetry:** A `bool` object, whether to do symmetry or asymmetry quantization for all inputs. The default value is `True`.
- **input_per_channel:** A `bool` object, whether to do per-channel or per-tensor quantization for all inputs. The default value is `False`.
- **input_round_mode:** An `int` object, the rounding mode used to quantify all inputs. Options are 0 for `HALF_TO_EVEN`, 1 for `HALF_UP`, and 2 for `HALF_AWAY_FROM_ZERO`. The default value is 1.
- **input_unsigned:** A `bool` object, whether to use unsigned integer quantization for all inputs. It is usually used for non-negative numeric inputs (ranging from 0 to 1) when input_unsigned is true. The default value is `False`.
- **weight_bit:** An `int` object, the bit width of all weights. The default value is 8.
- **weight_method:** An `int` object, the method to calculate scale factors in quantizing all weights. Options are: 0 for `Non_Overflow`, 1 for `Min_MSE`, 2 for `Min_KL`, and 3 for `Percentile`. The default value is 1.

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

vitis_quantize.VitisQuantizer.dump_model

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

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

Arguments

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

vitis_quantize.VitisQuantizer.dump_quantize_strategy

This function dumps current quantize strategy configurations to JSON file:

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

Arguments

- **dump_file:** A `string` object. The file path of the dumped quantize strategy JSON file.
- **verbose:** An `int` object. The verbosity of the dumped JSON file. Greater verbose value dumps a more detailed quantize strategy. Setting verbose to a value greater or equal to two dumps the complete quantize strategy. The default value is 0.

vitis_quantize.VitisQuantizer.set_quantize_strategy

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

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

Arguments

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

vitis_quantize.VitisQuantizer.get_qat_model

This function gets the float model for QAT:

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

Arguments

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

vitis_quantize.VitisQuantizer.get_deploy_model

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

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

Arguments

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

Examples

Quantize

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

Evaluate the Quantized Model

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

Load the Quantized Model

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

Dump the Quantized Model

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

vai_q_tensorflow2 Error Codes

Table 18: **vai_q_tensorflow2 Error Codes**

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

Table 18: **vai_q_tensorflow2 Error Codes** *(cont'd)*

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

PyTorch Version (vai_q_pytorch)

Installing vai_q_pytorch

vai_q_pytorch has GPU and CPU versions. It supports PyTorch version 1.2~2.0 but does not support PyTorch data parallelism. There are two ways to install vai_q_pytorch:

Install Using Docker Containers

Vitis AI provides a Docker container for quantization tools, including vai_q_pytorch. After running a GPU or CPU container, activate the Vitis AI-PyTorch conda environment,.

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

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

A new conda environment with a specified PyTorch version (1.2~2.0) can be created using the https://github.com/Xilinx/Vitis-AI/blob/v3.5/docker/common/replace_pytorch.sh script. This script does the following:

- Clones a conda environment from Vitis AI-pytorch.
- Uninstalls the original PyTorch, torchvision, and vai_q_pytorch packages.
- Installs the specified version of PyTorch, torchvision.
- Re-installs vai_q_pytorch from source code.

The following is the command line to create a new conda environment with the script:

```
replace_pytorch.sh new_conda_env_name
```

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

Install from the Source Code

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

1. Add the CUDA_HOME environment variable in .bashrc.

   If the CUDA library is installed in /usr/local/cuda for the GPU version, add the following line into .bashrc. If CUDA is in another directory, change the line accordingly:

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

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

   ```
   unset CUDA_HOME
   ```

2. Install PyTorch (1.2~2.0) and torchvision:

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

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

3. Install other dependencies:

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

4. Install vai_q_pytorch:

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

5. Verify the installation:

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

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

```
import pytorch_nndct
import torch
```

Inspect Float Model Before Quantization

Vai_q_pytorch provides an inspector function to help you diagnose neural network (NN) models under different device architectures. The inspector can predict target device assignments based on hardware constraints, enabling users to generate a report. The generated inspection report can guide you to modify or optimize the NN model, significantly reducing the deployment difficulty and time. It is recommended to inspect float models before proceeding with quantization.

This section uses `resnet18_quant.py` as an example to demonstrate how to modify the model code and apply this feature:

1. Import vai_q_pytorch module:

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

2. Create an inspector with a target name or fingerprint:

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

3. Inspect the float model:

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

4. Run the following command line to inspect the model:

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

Inspector displays special messages on the screen with special color and keyword prefix "VAIQ_*" according to the verbose_level setting.

***Note*:** The messages displayed between "[VAIQ_NOTE]: =>Start to inspect model..." and "[VAIQ_NOTE]: =>Finish inspecting."

If the inspector runs successfully, it typically generates the following three files under the output directory, /quantize_result:

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

***Note*:**

- The inspector relies on the 'xcompiler' package. In the conda environment Vitis AI-Pytorch in Vitis AI docker, xcompiler is ready. But if vai_q_pytorch is installed by source code, it must install xcompiler in advance.

- Visualization of inspection results relies on the dot engine. If you do not install the dot successfully, set `image_format = None` when inspecting.
- For more detailed guidance, see example/jupyter_notebook/inspector/inspector_tutorial.ipynb. Install jupyter notebook in advance. Run the following command:

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

Running vai_q_pytorch

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

Preparing Files for vai_q_pytorch

Prepare the following files for vai_q_pytorch.

Table 19: **Input Files for vai_q_pytorch**

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

Modifying the Model Definition

To enable quantization for a PyTorch model, it is necessary to modify the model definition to ensure it meets the following condition. Vitis AI GitHub provides an example.

1. The model intended for quantization must contain only the forward method. All other functions, which typically serve as pre-processing and post-processing operations, should be moved outside the model or placed in a derived class. Failure to do so might result in unexpected behavior when forwarding the quantized module, as the API removes these functions in the quantized module.
2. The float model should pass the JIT trace test. Set the floating-point module to evaluation status, then use the `torch.jit.trace` function to test the float model. For more details, refer to example/jupyter_notebook/jit_trace_test/jit_trace_test.ipynb.

The most common operators in PyTorch are supported in vai_q_pytorch. For more information, go to doc/support_op.md.

Adding vai_q_pytorch APIs to Float Scripts

Suppose there is a pre-trained float model and Python scripts for evaluating its accuracy/mAP before quantization. In that case, the Quantizer API substitutes the floating-point module with a quantized module. For standard evaluation, the evaluate function promotes the forwarding of the quantized module. By configuring the quant_mode flag to caliber, you can determine the quantization steps of tensors during the evaluation process for post-training quantization. After calibration is complete, evaluate the quantized model by setting quant_mode to test.

1. Import the vai_q_pytorch module:

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

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

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

3. Forward a neural network with the converted model:

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

4. Output the quantization result and deploy the model:

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

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

Running Quantization and Getting the Result

***Note*:** "vai_q_pytorch" log messages are identifiable by special colors and a unique keyword prefix, "VAI_Q_*." The log message types encompass "error," "warning," and "note." Monitoring "vai_q_pytorch" log messages is crucial to verify the flow status.

1. Run the command to quantize the model:

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

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

   Controlling the iteration numbers during quantization and evaluation is crucial. Typically, using 100-1000 images suffices for quantization, while the entire validation set is necessary for evaluation. You can manage the iteration numbers in the data loading part. Here, the `subset_len` argument governs the number of images used for network forwarding. If the float evaluation script lacks a similar argument, it is essential to add one.

Successful execution of the quantization command results in two vital files generated in the output directory, ./quantize_result.

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

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

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

The accuracy shown after the successful execution of the command corresponds to the accuracy of the quantized model.

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

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

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

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

***Note*:** XIR is readily available in the Vitis AI-pytorch conda environment within the Vitis AI Docker. However, if you install vai_q_pytorch from the source code, it is necessary to install XIR beforehand. Failure to install XIR will prevent the generation of the XMODEL file, resulting in an error when executing the command. Nevertheless, you can still verify the accuracy in the output log.

Hardware-Aware Quantization Strategy

The Inspector offers device assignments to operators in the neural network based on the target device, enabling vai_q_pytorch to perform hardware-aware quantization.

Here is the example code in example/resnet18_quant.py:

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

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

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

Command line to test hardware-aware quantized model accuracy:

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

Command line to deploy the quantized model:

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

Module Partial Quantization

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

```
from pytorch_nndct.nn import QuantStub, DeQuantStub

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

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

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

        output1 = self.subm1(output0)

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

Register Custom Operation

To convert a quantized model to an XMODEL, vai_q_pytorch offers a decorator that allows you to register an operation or a group of operations as a custom operation that XIR does not recognize:

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

  attrs_list(Optional[List[str]], optional) -
```

```
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as padding,
dilation, stride and groups.
  The order of names in attrs_list should be consistent with that of the
arguments list.
  Default: None

  """
```

Perform the following steps:

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

```
from pytorch_nndct.utils import register_custom_op

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

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

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

Limitations:

1. Loop operation is not allowed in a custom operation.
2. A custom operation's number of return values can only be one.

vai_q_pytorch Fast fine-tuning

Generally, there might be a slight accuracy loss after quantization, but specific networks, like MobileNets, could experience a more significant accuracy reduction. In such situations, you can first attempt fast finetune to improve the accuracy of the quantized models. If the fast finetune approach still does not produce satisfactory results, consider using Quantization Aware Training (QAT) to enhance the accuracy of the quantized models further. QAT involves training the model with quantization-aware optimizations to achieve better accuracy in the quantized state.

The AdaQuant algorithm[#unique_145/unique_145_Connect_42_note1] uses a small set of unlabeled data for activation calibration and weight fine-tuning. The Vitis AI quantizer incorporates this algorithm under "fast fine-tuning." Although slightly slower, fast fine-tuning can yield better performance than post-training quantization. Similar to Quantization-Aware Training (QAT), each run of fast fine-tuning might produce a different result.

Fast fine-tuning does not involve training the model and requires only a limited number of iterations. For classification models on the Imagenet dataset, 5120 images are sufficient for an experiment. The data used in the fast fine-tuning process does not require annotations. Modifying the model evaluation script is the primary requirement for fast fine-tuning; there is no need to set up the optimizer for training. To use fast fine-tuning, a function for model forwarding iteration is necessary and will be called during the process. Re-calibration with the original inference code is recommended to ensure accuracy.

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

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

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

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

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

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

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

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

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

Quantization Strategy Configuration

Vai_q_pytorch supports the quantization configuration file in JSON format for multiple quantization strategy configurations.

Usage

To activate the customized configuration, you only need to pass the configuration file to the torch_quantizer API:

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

The `./example/` directory contains three examples: `int_config.json`, `bfloat16_config.json`, and `mix_precision_config.json`. To quantize the model, use the configuration files with the `config_xxx_config.json` command:

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

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

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

The tensor_quantize_config specifies the weight configuration using the maxmin calibration method and per_tensor quantization, indicating that weights employ a distinct quantization method from the model configuration:

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

There is a single-layer quantization setup in the layer_quantize_config list. This setup is determined by the type of layer and applies per-channel quantization to torch.nn.Conv2d layers.

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

Configurations

- **convert_relu6_to_relu:** (Global quantizer setting) Whether to convert ReLU6 to ReLU. Options: True or False.
- **include_cle:** (Global quantizer setting) Whether to use cross-layer equalization. Options: True or False.
- **include_bias_corr:** (Global quantizer setting) Whether to use bias correction. Options: True or False
- **target_device:** (Global quantizer setting) Target type for the quantized model. Options: DPU, CPU, GPU
- **quantizable_data_type:** (Global quantizer setting) Tensor types to be quantized in model
- **data_type:** (Tensor quantization setting) Data type used in quantization. Option: int, bfloat16, float16, float32
- **bit_width:** (Tensor quantization setting) Bit width used in quantization. Only applicable when the data type is int.
- **method:** (Tensor quantization setting) Method used to calibrate the quantization scale. Options: Maxmin, Percentile, Entropy, MSE, diffs. Only applicable when the data type is int.
- **round_mode:** (Tensor quantization setting) Rounding method in the quantization process. Options: half_even, half_up, half_down, std_round. Only applicable when the data type is int.
- **symmetry:** (Tensor quantization setting) Whether to use symmetric quantization. Options: True or False. Only applicable when the data type is int.
- **per_channel:** (Tensor quantization setting) Whether to use per_channel quantization. Options: True or False. Only applicable when the data type is int.
- **signed:** (Tensor quantization setting) Whether to use signed quantization. Options: True or False. Only applicable when the data type is int.
- **narrow_range:** (Tensor quantization setting) Whether to use symmetric integer range for signed quantization. Options: True or False. Only applicable when the data type is int.
- **scale_type:** (Tensor quantization setting) Scale type used in the quantization process. Options: Float, poweroftwo. Only applicable when the data type is int.
- **calib_statistic_method:** (Tensor quantization setting) Method for selecting the optimal quantization scale when multiple batch data have different scales. Options: modal, max, mean, median. Only applicable when the data type is int.

Hierarchical Configuration: Quantization configuration follows a hierarchical workflow.

- When the configuration file is not provided in the torch_quantizer API, the default configuration, tailored for the DPU device and using the poweroftwo quantization method, is applied.
- When a configuration file is provided, the model configuration, encompassing global quantizer settings and global tensor quantization settings, overrides default settings.
- When only the model configuration is specified in the file, all tensors within the model will adopt the same configuration.
- You can use the layer configuration to assign specific configuration parameters to specific layers.

Default Configurations:

The following are the details of the default configuration:

```
"convert_relu6_to_relu": false,
"include_cle": true,
"include_bias_corr": true,
"target_device": "DPU",
"quantizable_data_type": [
  "input",
  "weights",
  "bias",
  "activation"],
"datatype": "int",
"bit_width": 8,
"method": "diffs",
"round_mode": "std_round",
"symmetry": true,
"per_channel": false,
"signed": true,
"narrow_range": false,
"scale_type": "poweroftwo",
"calib_statistic_method": "modal"
```

Model Configurations:

In the example configuration file `int_config.json`, all tensors in the model are assigned the same int8 quantization configurations. In such cases, the global quantization parameters must be specified under the `overall_quantize_config` keyword as follows:

```
  "convert_relu6_to_relu": false,
  "include_cle": false,
  "keep_first_last_layer_accuracy": false,
  "keep_add_layer_accuracy": false,
  "include_bias_corr": false,
  "target_device": "CPU",
  "quantizable_data_type": [
    "input",
    "weights",
    "bias",
    "activation"],
"overall_quantize_config": {
    "datatype": "int",
    "bit_width": 8,
    "method": "maxmin",
```

```
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
}
```

Similar to `int_config.json`, all tensors in the model are configured with the same bfloat16 quantization settings in `bfloat16_config.json`. In this case, the only data type specified in the global quantization parameters is as follows:

```
  "convert_relu6_to_relu": false,
  "convert_silu_to_hswish": false,
  "include_cle": false,
  "keep_first_last_layer_accuracy": false,
  "keep_add_layer_accuracy": false,
  "include_bias_corr": false,
  "target_device": "CPU",
  "quantizable_data_type": [
    "input",
    "weights",
    "bias",
    "activation"
  ],
  "overall_quantize_config": {
    "datatype": "bfloat16"
  }
```

Optionally, the quantization configuration of different tensors in the model can be set separately. The configurations must be set in the *tensor_quantize_config* keyword. For example, configuration file `mix_precision_config.json`, the global datatype of quantization is bfloat16, and change the data type of bias to float16. The remaining parameters remain consistent with the global settings:

```
"tensor_quantize_config": {

    "bias": {
        "datatype": "float16",
    }
}
```

Layer Configurations:
Layer quantization configurations must be incorporated into the layer_quantize_config list. The configuration methods for each layer involve two parameters: the layer type and the layer name. There are five notes to consider when performing layer configuration:

- Each layer configuration must be in dictionary format.
- In each layer configuration, the quantizable_data_type and overall_quantize_config parameters are required. In the overall_quantize_config parameter, all quantization parameters for this layer must be included.
- If the setting is based on layer type, the layer_name parameter should be null.

- If the setting is based on the layer name, you need to perform a calibration process for the model. After calibration, you need to pick the required layer name from the Python file generated in the quantized_result directory. Additionally, ensure that the layer_type parameter is null.
- Similar to the model configuration, the quantization configuration for different tensors within a layer can be customized separately. These individual tensor configurations should be specified using the tensor_quantize_config keyword.

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

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

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

```
{
  "layer_type": null,
  "layer_name": "ResNet::ResNet/Conv2d[conv1]/input.2",
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
```

```
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
  },
  "tensor_quantize_config": {
    "activation": {
      "round_mode": "half_up"
    }
  }
}
```

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

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

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

Configuration Restrictions

Due to design constraints related to DPUs, when using integer quantization and deploying quantized models on the DPU, the quantization configuration should meet the following restrictions:

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

For CPU and GPU devices, there are no similar restrictions in place. However, conflicts might arise when employing different configurations. For instance, the calibration methods 'maxmin,' 'percentile,' 'mse,' or 'entropy' do not support the calibration statistic method 'modal.' Furthermore, if the symmetry mode is set to asymmetry, the calibration methods 'mse' and 'entropy' are unsupported. If configuration conflicts occur, the quantization tool provides an error message to notify the user.

Using the New Data Format

vai_q_pytorch introduces a new data format called block floating point (BFP). In BFP, numbers within a block share the same exponent, determined by the largest exponent in the block. Smaller numbers have their mantissa shifted right to accommodate this shared exponent.

Although you can use vai_q_pytorch to assess the quantization results, there is currently no option to deploy the quantized model to hardware.

***Note*:** BFP is a new data format not fully supported in the current version of the Vitis AI toolchain.

Usage

BFP offers various configurations, including bit width, block size, and more. There are two out-of-the-box configuration types ("mx6" and "mx9") that you can use directly without having to set up their configuration items. To quantize the model, follow these steps:

- Preparing the float model and inputs:

```
model = buid_your_model()
batch_size = 32
inputs = torch.randn([batch_size, 3, 224, 224], dtype=torch.float32)
```

- Quantizing the float model:

```
from pytorch_nndct import bfp
quantized_model = bfp.quantize_model(model, inputs, dtype='mx6')
```

- Validating the quantized model: Pass the quantized model to the validation function to evaluate quantization results:

```
validate(quantized_model, data_loader)
```

BFP APIs

```
bfp.quantize_model(model, inputs, dtype='mx6', config_file=None)
```

- **Model:** Float module to be quantized.
- **Inputs:** The input tensor should have the same shape as the actual input of the floating-point module to be quantized, but the values can be random.
- **dtype:** Pre-configured BFP configuration. Available values are *mx6* and *mx9*.

- **config_file:** Configuration file path. This feature is under development. Use the pre-defined dtype.

Inference of Quantized Model

Inference of quantized model in torch script format

You can run the quantized model in TorchScript format, a .pt file, in the PyTorch framework. Before performing inference, import the pytorch_nndct module, as it sets up quantized operators for this model. The following is an example code:

```
import pytorch_nndct

# prepare input data
input = ......

quantized_model = torch.jit.load(quantized_model_path)

# feed input data to quantized model and make inference
output = quantized_model(input)
```

Inference of quantized model in ONNX script format

You can run the quantized model in ONNX format by ONNX Runtime APIs.

For the ONNX model with native Quantize and DeQuantize operators, you can run the model by using ONNX Runtime. The following is an example code:

```
import onnxruntime as ort

# prepare input data
input_data = ......

ort_sess = ort.InferenceSession(quantized_model_path)
input_name = ort_sess.get_inputs()[0].name
ort_output = ort_sess.run(None, {input_name: input_data})
```

You must set up custom operators and then run the model using onnxruntime_extensions for the ONNX model with VAI Quantize and DeQuantize operators. The setting up can be done by function load_vai_ops(), imported from pytorch_nndct. The following is the example code:

```
from onnxruntime_extensions import PyOrtFunction
from pytorch_nndct.apis import load_vai_ops

## Before running the ONNX model, custom ops must be set.
load_vai_ops()

# prepare input data
input = ......

# run using onnxruntime_extensions API
run_ort = PyOrtFunction.from_model(quantized_model_path)
ort_outputs = run_ort(input)
```

Inference of XIR format quantized model

So far, XIR format quantized model cannot be run by any tool.

vai_q_pytorch QAT

Assuming a pre-defined model architecture, follow the steps below to perform quantization aware training (QAT), using ResNet18 from torchvision as an example. The complete model definition is here.

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

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

  def __init__(self,
               inplanes,
               planes,
               stride=1,
               downsample=None,
               groups=1,
               base_width=64,
               dilation=1,
               norm_layer=None):
    super(BasicBlock, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    if groups != 1 or base_width != 64:
      raise ValueError('BasicBlock only supports groups=1 and
base_width=64')
    if dilation > 1:
      raise NotImplementedError("Dilation > 1 not supported in
BasicBlock")
    # Both self.conv1 and self.downsample layers downsample the input
when stride != 1
    self.conv1 = conv3x3(inplanes, planes, stride)
    self.bn1 = norm_layer(planes)
    self.relu1 = nn.ReLU(inplace=True)
    self.conv2 = conv3x3(planes, planes)
    self.bn2 = norm_layer(planes)
    self.downsample = downsample
    self.stride = stride

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

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

  def forward(self, x):
    identity = x
```

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

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

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

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

    return out
```

3. Insert `QuantStub` and `DeQuantStub`.

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

```
class ResNet(nn.Module):

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

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

```
        block, 256, layers[2], stride=2,
dilate=replace_stride_with_dilation[1])
    self.layer4 = self._make_layer(
        block, 512, layers[3], stride=2,
dilate=replace_stride_with_dilation[2])
    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(512 * block.expansion, num_classes)

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

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

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

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

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

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

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

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

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

def resnet18(pretrained=False, progress=True, **kwargs):
  r"""ResNet-18 model from
```

```
    `"Deep Residual Learning for Image Recognition" <https://
arxiv.org/pdf/1512.03385.pdf>'_

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

model = resnet18(pretrained=True)

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

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

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

5. Get the deployable model and test it.

   After completing QAT, convert the quantized model to a deployable model. The accuracy of the deployable model might differ slightly from that of the quantized model:

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

6. Export XMODEL from the deployable model.

   `batch size=1` is mandatory for XMODEL compilation:

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

vai_q_pytorch QAT Requirements

Generally, quantization can cause a slight loss of accuracy, but for certain networks like MobileNets, the accuracy loss can be more significant. In such cases, it is recommended to try fast fine-tuning first. If fast fine-tuning fails to produce satisfactory results, Quantization-Aware Training (QAT) can further enhance the accuracy of the quantized model.

However, specific requirements exist for the model to be trained using QAT APIs. All operations to be quantized must be instances of the `torch.nn.Module` object rather than Torch functions or Python operators. For instance, using `'+'` to add two tensors in PyTorch is common but not supported in QAT. Instead, replace `'+'` with `pytorch_nndct.nn.modules.functional.Add`. The operations that require replacement are listed in the following table.

Table 20: **Operation Replacement Mapping**

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

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

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

Guidelines for Better Training Results

The following are some tips to improve training results:

- If possible, load the pre-trained floating-point weights as initial values to start the quantization aware training. It is possible to train from scratch with random initial values, but it makes training more complex and lengthy.
- If pre-trained floating-point weights are loaded, different initial learning rates and learning rate decrease strategies must be used for the network and quantizer parameters, respectively. In general, the learning rate of network parameters must be set small, while the learning rate of quantizer parameters needs to be larger.

```
model = qat_processor.trainable_model()
param_groups = [{
    'params': model.quantizer_parameters(),
    'lr': 1e-2,
    'name': 'quantizer'
}, {
```

```
    'params': model.non_quantizer_parameters(),
    'lr': 1e-5,
    'name': 'weight'
}]
optimizer = torch.optim.Adam(param_groups)
```

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

vai_q_pytorch Usage

This section introduces the execution tools and APIs used to implement quantization and generate a deployable model for the target hardware. The following are the APIs in the`pytorch_binding/pytorch_nndct/apis/quant_api.py` module:

class torch_quantizer()

The `torch_quantizer` class creates a quantizer object:

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

Arguments

- **Quant_mode:** An integer that indicates which quantization mode the process is using. The value *calib* is used for calibration of quantization, while *test* is used for evaluating the quantized model.
- **Module:** Float module to be quantized.
- **Input_args:** Input tensor with the same shape as the actual input of the floating-point module to be quantized, but the values can be random numbers.
- **State_dict_file:** Float module pretrained parameters file. If the float module has read parameters, the parameter need not be set.
- **Output_dir:** Directory for quantization results and intermediate files. The default value is *quantize_result*.
- **Bitwidth:** Global quantization bit width. The default value is 8.
- **Device:** Run model on GPU or CPU.

- **Quant_config_file:** Location of the JSON file with the quantization strategy configuration.
- **Target:** If the target device is specified, the hardware-aware quantization is on. The default value is None.

def export_quant_config(self)

This function exports information related to the quantization steps:

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

def export_xmodel(self, output_dir, deploy_check)

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

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

Arguments

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

def export_onnx_model(self, output_dir, verbose)-New

Native ONNX models support only INT8 quantization and half-even rounding. When converting models from Vitis AI quantizer to ONNX format, the other quantization bits and more rounding methods, such as half-up or toward zero, cannot be exported. To address this, vai::QuantizeLinear and vai::DequantizeLinear are used to replace the corresponding native ONNX operators when exporting ONNX models. For the DequantizeLinear, the interface between native ONNX and Vitis AI is the same. However, for the QuantizeLinear there are some differences between them, which are outlined in the following points:

- ONNX has an input list (x, y_scale, y_zero_point): Vitis AI has an input list (x, valmin, valmax, scale, zero_point, method), where valmin and valmax are quantization intervals, for example, valmin=-128 and valmax=127 for INT8 symmetric quantization, and method is a rounding way which can be half-even, half-up, down, up, toward zero, away from zero, and so on.
- Obtaining a native Quant-Dequant ONNX model is possible by setting native_onnx=True in the following definition. If is not set, the Quant-Dequant ONNX model is received, with Vitis AI QuantizeLinear and DequantizeLinear operators. The default value is False.

The function exports the quantized model in ONNX format:

```
def export_onnx_model(self, output_dir="quantize_result", verbose=False,
dynamic_batch=False,                    opset_version=None,
native_onnx=True, dump_layers=False, check_model=False, opt_graph=False):
```

Table 21: **Arguments**

Argument	Description
Output_dir	Directory for quantization result and intermediate files. The default value is quantize_result.
Verbose	Flag to control the verbose logging.
Dynamic_batch	A flag to set the batch size of the input shape dynamic or not. The default value is False.
Opset_version	The version of the default (ai.onnx) opset to target. If not set, the latest version that is stable for the current version of PyTorch is valued.
Native_onnx	Export ONNX model with native Quant-Dequant operators or custom Quant-Dequant ones. If set to True, the native Quant-Dequant ONNX model is received. Otherwise, the VAI Quant-Dequant ONNX model is generated. The default is True.
Dump_layers	Dump output of each layer in the ONNX model during runtime. The default value is False.
Check_model	Check the difference in outputs between XMODEL and ONNX models. The default value is False.
Opt_graph	Optimize ONNX graph. The default value is False.

def export_torch_script(self, output_dir, verbose)

This function exports the quantized model in the TorchScript format:

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

Arguments

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

Class Inspector

Class Inspector creates an inspector object as follows:

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

Arguments

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

def inspect(...)

This function inspects a float model:

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

Arguments

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

vai_q_pytorch message

This section contains important messages that can be searched using their message ID. Each message helps users identify issues with their model deployment and provides potential solutions.

VAIQ_WARN

Vai_q_pytorch displays a warning message if an issue could lead to problems or incompleteness in the quantization result (r to the message text for details). The message format is *[VAIQ_WARN] [MESSAGE_ID]: message text*. The quantization process can still proceed to completion despite the warning.

List important warning messages in the following table:

Table 22: **Vai_q_pytorch Warning Message Table**

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

VAIQ_ERROR

Vai_q_PyTorch displays an error message when there is an issue causing problems or incompleteness in the quantization result (refer to the message text for details). The format of this kind of message is "[VAIQ_ERROR][MESSAGE_ID]: message text"

The following table consists of the important error messages:

Table 23: **Vai_q_PyTorch error message table**

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

Table 23: **Vai_q_PyTorch error message table** *(cont'd)*

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

ONNX Runtime Version (vai_q_onnx)

Installing vai_q_onnx

You can install vai_q_onnx as follows:

Install from Source Code with the Wheel Package

To build vai_q_onnx, run the following command:

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

Running vai_q_onnx

Quantization in ONNX Runtime refers to the linear quantization of an ONNX model. vai_q_onnx tool is developed as a plugin for ONNX Runtime to support more post-training quantization (PTQ) functions to quantize a deep learning model.

Post-training quantization (PTQ) is a technique to convert a pretrained float model into a quantized model with little degradation in model accuracy.

A representative dataset is needed to run a few batches of inference on the float model to obtain the distribution of activations at each layer, a process which is called post-training quantization.

Preparing the Float Model and Calibration Set

Before running `vai_q_onnx`, ensure to prepare the float model and calibration set, including the files listed in the following table.

Table 24: **Input Files for vai_q_onnx**

Number	Name	Description
1	float model	Floating-point models in ONNX format.
2	calibration dataset	A subset of the training dataset or validation dataset to represent the input data distribution. Typically, 100 to 1000 images are sufficient for this.

(Optional) Pre-Processing on the Float Model

Pre-processing float32 model transforms and prepares it for quantization. It consists of the following three optional steps:

- Symbolic shape inference: It is best suited for transformer models.
- Model optimization: It uses ONNX Runtime native library to rewrite the computation graph, including merging computation nodes and eliminating redundancies to improve Runtime efficiency.
- ONNX shape inference.

The primary objective of these steps is to enhance quantization quality. The ONNX Runtime quantization tool performs optimally when the tensor's shape is known. Both symbolic shape inference and ONNX shape inference play a crucial role in determining tensor shapes. Symbolic shape inference is particularly effective for transformer-based models, whereas ONNX shape inference works well with other models.

Model optimization performs certain operator fusion, making the quantization tool's job easier. For instance, a Convolution operator followed by BatchNormalization can be fused into one during the optimization, which enables effective quantization.

ONNX Runtime has a known issue: model optimization cannot output a model size greater than 2 GB. As a result, for large models, optimization must be skipped.

The pre-processing API can be found in the onnxruntime.quantization.shape_inference Python module inside the quant_pre_process() function:

```
from onnxruntime.quantization import shape_inference

shape_inference.quant_pre_process(
    input_model_path: str,
    output_model_path: str,
    skip_optimization: bool = False,
    skip_onnx_shape: bool = False,
    skip_symbolic_shape: bool = False,
    auto_merge: bool = False,
    int_max: int = 2**31 - 1,
    guess_output_rank: bool = False,
    verbose: int = 0,
    save_as_external_data: bool = False,
    all_tensors_to_one_file: bool = False,
    external_data_location: str = "./",
    external_data_size_threshold: int = 1024,)
```

- **input_model_path:** Path to the input model file.
- **output_model_path:** Path to the output model file.
- **skip_optimization:** Skip the model optimization step if set to true. This might result in ONNX shape inference failure for some models.
- **skip_onnx_shape:** Skip ONNX shape inference. Symbolic shape inference is most effective with transformer-based models. Skipping all shape inferences might reduce the effectiveness of quantization because a tensor with an unknown shape cannot be quantized.
- **skip_symbolic_shape:** Skip symbolic shape inference. Symbolic shape inference is most effective with transformer-based models. Skipping all shape inferences might reduce the effectiveness of quantization because a tensor with an unknown shape can not be quantized.
- **auto_merge:** For symbolic shape inference. Automatically merge symbolic dims when conflict happens.
- **int_max:** For symbolic shape inference, specify the maximum value for the integer to be treated as boundless for ops like slice.
- **guess_output_rank:** Guess output rank to be the same as input 0 for unknown ops.
- **verbose:** Logs detailed info of inference. Options are 0: turn off, 1: warnings, 3: detailed.
- **save_as_external_data:** Saving an ONNX model to external data.
- **all_tensors_to_one_file:** Saving all the external data to one file.
- **external_data_location:** The file location to save the external file.
- **external_data_size_threshold:** The size threshold for external data.

Quantizing Using the vai_q_onnx API

The static quantization method first runs the model using a set of inputs called calibration data. During these runs, the quantization parameters are computed for each activation. These quantization parameters are written as constants to the quantized model and used for all inputs.

Vai_q_onnx quantization tool has expanded calibration methods to power-of-2 scale/float scale quantization methods. Float scale quantization methods include MinMax, Entropy, and Percentile. Power-of-2 scale quantization methods include MinMax and MinMSE:

```
import vai_q_onnx

vai_q_onnx.quantize_static(
    model_input,
    model_output,
    calibration_data_reader,
    quant_format=vai_q_onnx.VitisQuantFormat.FixNeuron,
    calibrate_method=vai_q_onnx.PowerOfTwoMethod.MinMSE,
    input_nodes=[],
    output_nodes=[],
    extra_options=None,)
```

- **model_input:** File path of the model to quantize.
- **model_output:** File path of the quantized model.
- **calibration_data_reader:** A calibration data reader. It enumerates calibration data and generates inputs for the original model.
- **quant_format:**
  - QOperator: quantizes the model with quantized operators directly.
  - QDQ: quantizes the model by inserting QuantizeLinear/DeQuantizeLinear on the tensor. It only supports 8-bit quantization.
  - VitisQuantFormat.QDQ: quantizes the model by inserting VAIQuantizeLinear/VAIDeQuantizeLinear on the tensor. Supports more bit-width and configurations.
  - VitisQuantFormat.FixNeuron: quantizes the model by inserting FixNeuron (composition of QuantizeLinear and DeQuantizeLinear) on the tensor.
- **calibrate_method:** For DPU devices, set calibrate_method to vai_q_onnx.PowerOfTwoMethod.NonOverflow or vai_q_onnx.PowerOfTwoMethod.MinMSE to apply power-of-2 scale quantization. The PowerOfTwoMethod has two supported methods currently: MinMSE and NonOverflow. The default method is MinMSE.
- **input_nodes:** A *list(string)* object. Names of the start nodes to be quantized. The nodes before these start nodes in the model are not optimized or quantized. For example, this argument can be used to skip some pre-processing nodes or stop quantizing the first node. The default value is [].

- **output_nodes:** A *list(string)* object. Names of the end nodes to be quantized. The nodes after these nodes in the model are not optimized or quantized. For example, this argument can be used to skip some post-processing nodes or stop quantizing the last node. The default value is [].

(Optional) Evaluating the Quantized Model

If you have scripts to evaluate float models, like the models in AMD Model Zoo, you can replace the float model file with the quantized model for evaluation.

To support the customized FixNeuron op, the vai_dquantize module should be imported. The following is an example:

```
import onnxruntime as ort
from vai_q_onnx.operators.vai_ops.qdq_ops import vai_dquantize

so = ort.SessionOptions()
so.register_custom_ops_library(_lib_path())
sess = ort.InferenceSession(dump_model, so)
input_name = sess.get_inputs()[0].name
results_outputs = sess.run(None, {input_name: input_data})
```

After replacing the float model file with the quantized model, evaluate the quantized model as the float model.

(Optional) Dumping the Simulation Results

Sometimes, after deploying the quantized model, it is essential to compare the simulation results on the CPU and GPU with the output values on the DPU.

You can use the dump_model API of vai_q_onnx to dump the simulation results with the quantized_model:

```
import vai_q_onnx
# This function dumps the simulation results of the quantized model,
# including weights and activation results.
vai_q_onnx.dump_model(
    model,
    dump_data_reader=None,
    output_dir='./dump_results',
    dump_float=False)
```

- **model:** File path of the quantized model.
- **dump_data_reader:** A data reader used for the dump. It generates inputs for the original model.
- **output_dir:** *String*. The directory to save the dump results. Dump results are generated in output_dir after the function is successfully executed.
- **dump_float:** *Boolean*. Determines whether to dump the float value of weights and activation results.

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

After successfully executing the command, the dump results are generated in the output_dir. Each quantized node's weights and activation results are saved separately in *.bin and *.txt formats.

In cases where the node output is not quantized, such as the softmax node, the float activation results are saved in *_float.bin and *_float.txt formats if the option "save_float" is set to True.

The following table shows examples of the dump results.

Table 25: **Example of Dumping Results**

Batch Number	Quantized	Node Name	Saved files
1	Yes	resnet_v1_50_conv1	{output_dir}/dump_results/quant_resnet_v1_50_conv1.bin {output_dir}/dump_results/quant_resnet_v1_50_conv1.txt
2	Yes	resnet_v1_50_conv1_weights	{output_dir}/dump_results/quant_resnet_v1_50_conv1_weights.bin {output_dir}/dump_results/quant_resnet_v1_50_conv1_weights.txt
2	No	resnet_v1_50_softmax	{output_dir}/dump_results/quant_resnet_v1_50_softmax_float.bin {output_dir}/dump_results/quant_resnet_v1_50_softmax_float.txt

vai_q_onnx Supported Operations and APIs

The following table lists the quantization-supported operations and APIs for vai_q_onnx. Operations not in this list are skipped during quantization and remain untouched in the output model.

Table 26: **vai_q_onnx Supported Operations**

Operation Type	Description
Conv	
ConvTranspose	
Gemm	
BatchNorm	Will be fused with previous "Conv" op
Add	
Concat	
Relu	
Reshape	
Transpose	
Squeeze	

Table 26: **vai_q_onnx Supported Operations** *(cont'd)*

Operation Type	Description
Resize	
MaxPool	
GlobalAveragePool	
AveragePool	
MatMul	
Mul	
Sigmoid	
Softmax	

vai_q_onnx Usage

```
vai_q_onnx.quantize_static(
    model_input,
    model_output,
    calibration_data_reader,
    quant_format=vai_q_onnx.VitisQuantFormat.FixNeuron,
    calibrate_method=vai_q_onnx.PowerOfTwoMethod.MinMSE,
    input_nodes=[],
    output_nodes=[],
    op_types_to_quantize=None,
    per_channel=False,
    reduce_range=False,
    activation_type=QuantType.QInt8,
    weight_type=QuantType.QInt8,
    nodes_to_quantize=None,
    nodes_to_exclude=None,
    optimize_model=True,
    use_external_data_format=False,
    calibrate_method=CalibrationMethod.MinMax,
    extra_options=None)
```

Arguments

- **model_input:** File path of the model to quantize.
- **model_output:** File path of the quantized model.
- **calibration_data_reader:** A calibration data reader. It enumerates calibration data and generates inputs for the original model. If you want to use random data for a quick test, you can set calibration_data_reader to None.
- **quant_format:**
  - QOperator: quantizes the model with quantized operators directly.
  - QDQ: quantize the model by inserting QuantizeLinear/DeQuantizeLinear on the tensor. Supports only 8-bit quantization.

  - VitisQuantFormat.QDQ: quantizes the model by inserting VAIQuantizeLinear/VAIDeQuantizeLinear on the tensor. Supports more bit-width and configurations.
  - VitisQuantFormat.FixNeuron: quantizes the model by inserting FixNeuron (composition of QuantizeLinear and DeQuantizeLinear) on the tensor.
- **calibrate_method:** For DPU devices, set calibrate_method to 'vai_q_onnx.PowerOfTwoMethod.NonOverflow' or 'vai_q_onnx.PowerOfTwoMethod.MinMSE' to apply power-of-2 scale quantization. The PowerOfTwoMethod has two supported methods currently: MinMSE and NonOverflow. The default method is MinMSE.
- **input_nodes:** A *list(string)* object. Names of the start nodes to be quantized. Nodes before these start nodes in the model are not optimized or quantized. For example, this argument can skip some pre-processing nodes or stop quantizing the first node. The default value is [].
- **output_nodes:** A *list(string)* object. Names of the end nodes to be quantized. Nodes after these nodes in the model are not optimized or quantized. For example, this argument can skip some post-processing nodes or stop quantizing the last node. The default value is [].
- **op_types_to_quantize:** Specifies the types of operators to quantize, such as ['Conv'] to quantize Conv only. It quantizes all supported operators by default.
- **per_channel:** Quantize weights per channel. For DPU, this must be set to False as it currently does not support per-channel.
- **reduce_range:** Quantize weights with 7 bits. For DPU, the reduce_range is not supported, so this must be set to False.
- **weight_type:** Quantization data type of weight. For DPU, this must be set to QuantType.QInt8. For more details on data type selection, refer to https://onnxruntime.ai/docs/performance/quantization.html.
- **nodes_to_quantize:** List of nodes names to quantize. The nodes in this list are quantized only when this list is None.
- **nodes_to_exclude:** List of nodes names to exclude. The nodes in this list are excluded from quantization when it is None.
- **optimize_model:** Optimizes the model before quantization is going to be deprecated soon. It is not recommended because optimization changes the computation graph, making debugging quantization loss difficult.
- **use_external_data_format:** Option used for large size (>2GB) model. The default value is False.
- **extra_options:** Key-value pair dictionary for various options in different cases. Currently used pairs:
  - **ActivationSymmetric:** Symmetrize calibration data for activations (default is False). In PowerOfTwoMethod calibrate_method, it should always set ActivationSymmetric as True.

- **WeightSymmetric:** symmetrize calibration data for weights (The default value is True). In PowerOfTwoMethod calibrate_method, it should always set WeightSymmetric to True.
- **ForceQuantizeNoInputCheck:** By default, some latent operators, such as maxpool and transpose, do not quantize if their input is not quantized already. Setting to True to force such an operator always quantizes input and generates quantized output. Also, the true behavior could be disabled per node using the nodes_to_exclude.
- **MatMulConstBOnly:** The default value is False for static mode. If enabled, only MatMul with const B is quantized.
- **AddQDQPairToWeight:** The default value is False, which quantizes floating-point weight and feeds it to the solely inserted DeQuantizeLinear node. If True, it remains floating-point weight and inserts QuantizeLinear/DeQuantizeLinear nodes to weight. In PowerOfTwoMethod calibrate_method, QDQ should always appear as a pair. Therefore, you need to add qdq pair to weight, and it should always set AddQDQPairToWeight to True.

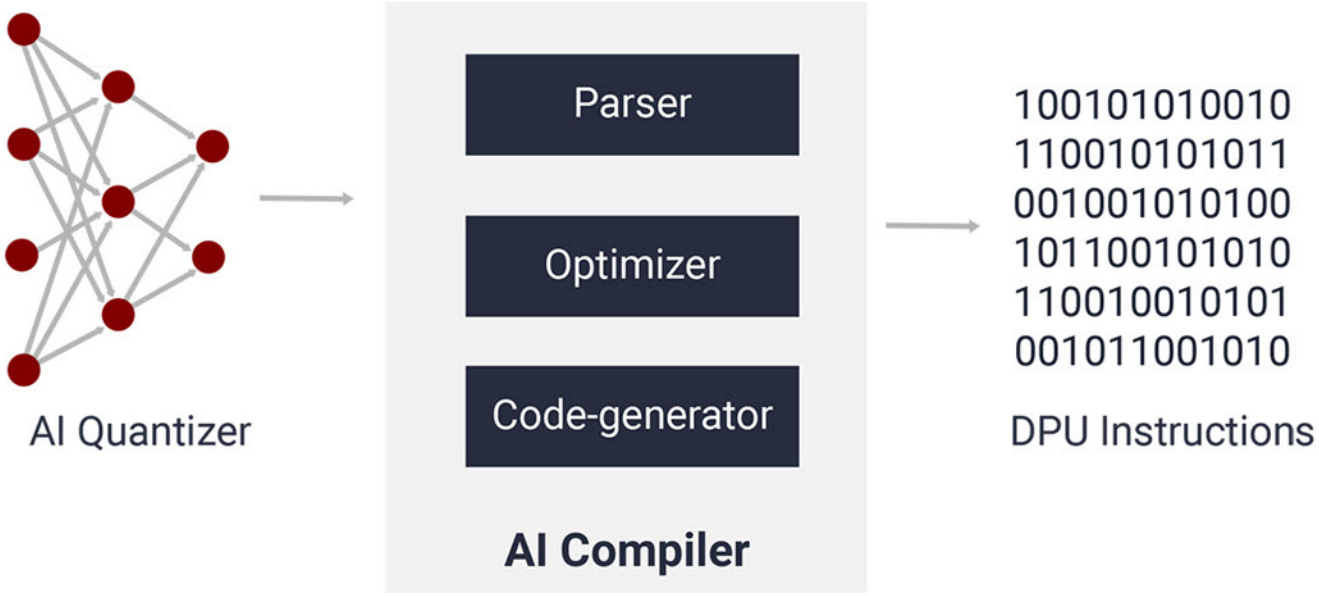
Compiling the Model

Vitis AI Compiler

The AMD Vitis™ AI compiler (VAI_C) serves as a unified interface for a family of compilers to optimize neural-network computations for various Deep Learning Processing Units (DPUs). Each compiler maps a network model to a highly optimized DPU instruction sequence.

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

Figure 19: **Vitis AI Compiler Framework**



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

Table 27: **DPUs on Different Hardware Platforms**

DPU Name	Hardware platform
DPUCZDX8G	AMD Zynq™ UltraScale+™ MPSoC
DPUCVDX8G	AMD Versal™ adaptive SoC VCK190 evaluation board, Versal AI Core Series
DPUCVDX8H	Versal adaptive SoC VCK5000 evaluation kit
DPUCV2DX8G	Versal adaptive SoC VEK280 evaluation board, Versal AI Edge series, Versal adaptive SoC V70 evaluation kit, Alveo V70 Accelerator Card

Compiling with an XIR-based Toolchain

AMD Intermediate Representation (XIR) is a graph-based intermediate representation of the AI algorithms designed for compilation and efficient deployment of the DPU on the FPGA platform. As an advanced user, you can apply whole application acceleration, leveraging the FPGA to its fullest potential by extending the XIR to support customized IPs in the Vitis AI flow. XIR is the foundation for the Vitis AI quantizer, compiler, runtime, and other tools.

XIR

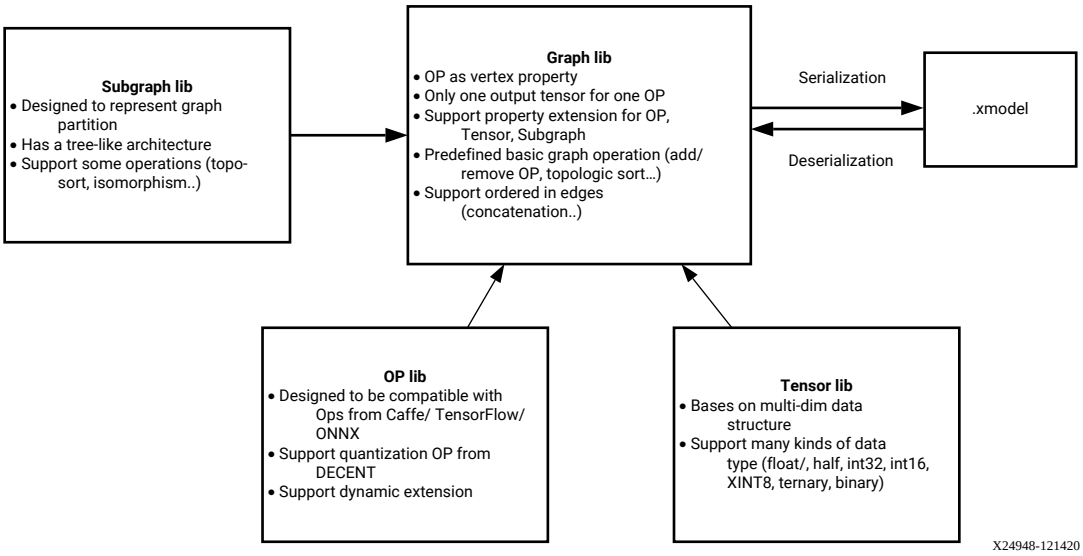
XIR includes the Op, Tensor, Graph, and Subgraph libraries, which provide a clear and flexible representation of the computational graph. XIR has in-memory and file formats for different uses. The in-memory format XIR is a graph object and the file format is an XMODEL. A graph object can be serialized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

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

XIR also offers Python APIs called PyXIR, empowering you to fully access XIR within your Python environment. With PyXIR, you can co-develop and integrate Python projects with the existing XIR-based tools. This integration eliminates the need for extensive effort to bridge the gap between different languages.

[1] Caffe was deprecated with the Vitis AI 2.5 release. For more information, see Vitis AI 2.0 User Guide.

Figure 20: **XIR Based Flow**



xir::Graph

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

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

xir::Op

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

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

xir::Tensor

Tensor is another important class in XIR. Unlike other frameworks' tensor definitions, XIR's Tensor only describes the data block it represents. The actual data block is excluded from the Tensor.

The key attributes of Tensor are the data type and shape.

xir::Subgraph

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

Compiling for DPU

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

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

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

TensorFlow

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

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

The output is the same as the output for Caffe.

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

[2] Caffe was deprecated with the Vitis AI 2.5 release. For information, see Vitis AI 2.0 User Guide.

TensorFlow 2.x

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

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

Currently, vai_c_tensorflow2 only supports Keras functional APIs.

PyTorch

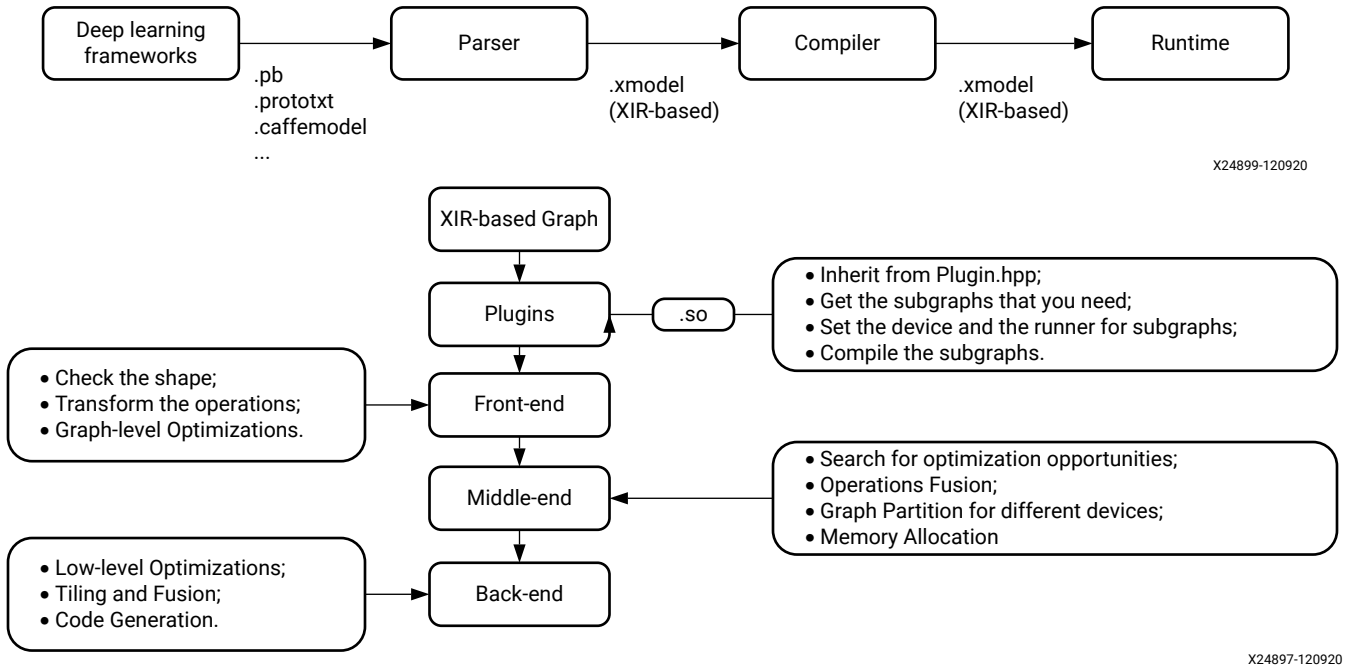
For PyTorch, the quantizer NNDCT outputs the quantized model in the XIR format directly. Use vai_c_xir to compile it.

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

Compiling for Customized Accelerator

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

Figure 21: **Compilation Flow**



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

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

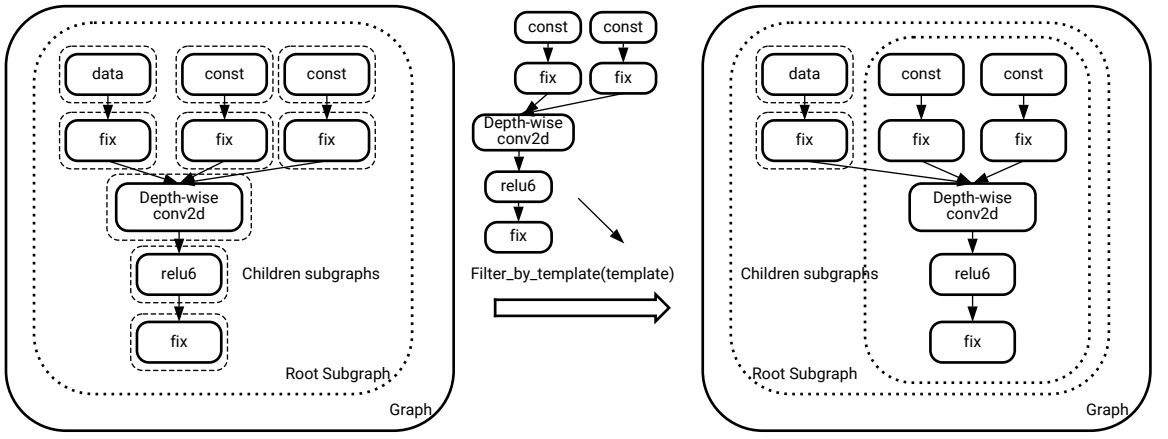
Implementing a Plugin

1. Implement `Plugin::partition()`

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

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

   Figure 22: **Filter by Templates**

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

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

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

Building the Plugin

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

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

Using the Plugin

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

Samples

Check https://github.com/Xilinx/Vitis-AI/tree/v3.5/src/vai_runtime/plugin-samples for samples.

Supported Operators and DPU Limitations

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

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

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

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

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

Currently Supported Operators

Table 28: **Currently Supported Operators**

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
conv2d	Kernel size	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16] w * h * ceil(input_channel/ 2048) <= 64	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16]	w, h: [1, 16] 256 * h * w <= 13760
	Strides	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]
	Dilation	dilation * input_channel <= 256 * channel_parallel						
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]						
		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]						
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth						kernel_w * kernel_h * ceil(input_channel / channel_parallel) * ceil(channel_parallel / 4) + 4 <= bank_depth
		input_channel <= 256 * channel_parallel		input_channel <= 256 * channel_parallel				input_channel <= 256 * channel_parallel
	Out Size	output_channel <= 256 * channel_parallel						
	Activation	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
	Group* (Caffe)	group==1						

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-conv2d	Kernel size	w, h: [1, 256]	w, h: [3]	w, h: [1, 256]	w, h: {1, 2, 3, 5, 7}	Not supported	w, h: [1, 8]	w, h: [1, 256] h * w <= 431
	Strides	w, h: [1, 256]	w, h: [1, 2]	w, h: [1, 256]	w, h: [1, 4]		w, h : [1, 4]	w, h: [1, 256]
	dilation	dilation * input_channel <= 256 * channel_parallel					dilation * input_channel <= 256 * channel_parallel	
	Paddings	pad_left, pad_right: [0, min((kernel_w - 1), 15) * dilation_w]	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]	pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w]	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]		pad_left, pad_right: [0, (kernel_w - 1) * dilation_w]	pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w]
		pad_top, pad_bottom: [0, min((kernel_h - 1), 15) * dilation_h]	pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h]	pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]		pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h]
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth					kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth	(6 * stride_w + kernel_w) * kernel_h + 4 <= 512
	Out Size	output_channel <= 256 * channel_parallel					output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6, LeakyReLU[6], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, ReLU6, LeakyReLU[7], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6		ReLU, ReLU6	ReLU, ReLU6, LeakyReLU, Hard-Swish, Hard-Sigmoid
	Group* (Caffe)	group==input_channel					group==input_channel	

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16 bank_depth: 2048 bank_num: 8**	**channel_parallel: 32 bank_depth: 4096**	**channel_parallel: 16 bank_depth: 8192 bank_num: 8**	**channel_parallel: 16 bank_depth: 2048**	**channel_parallel: 16 bank_depth: 8192**	**channel_parallel: 64 bank_depth: 2048**	**channel_parallel: 32 bank_depth: 65528 bank_num: 1**
transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]						
	Strides							
	Paddings	pad_left, pad_right: [0, kernel_w-1]						
		pad_top, pad_bottom: [0, kernel_h-1]						
	Out Size	output_channel <= 256 * channel_parallel						
	Activation	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
depthwise-transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 256]	kernel_w/stride_w, kernel_h/stride_h: [3]	kernel_w/stride_w, kernel_h/stride_h: [1, 256]	kernel_w/stride_w, kernel_h/stride_h: {1,2, 3, 5, 7}	Not supported	kernel_w/stride_w, kernel_h/stride_h: [1, 8]	kernel_w/stride_w, kernel_h/stride_h: [1, 256]
	Strides							
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1),15)]	pad_left, pad_right: [1, kernel_w-1]		pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1),15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]		pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Out Size	output_channel <= 256 * channel_parallel					output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6, LeakyReLU[6], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6	ReLU, ReLU6, LeakyReLU[7], Hard-Swish, Hard-Sigmoid	ReLU, ReLU6		ReLU, ReLU6	ReLU, ReLU6, LeakyReLU, Hard-Swish, Hard-Sigmoid

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
max-pooling	Kernel size	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: {2, 3, 5, 7, 8}	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: [1, 8]	w, h: [1, 16]	w, h: [1, 128]	w, h: [1, 256] h * w <= bank_depth
	Strides	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 128]	w, h: [1, 256]
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]			pad_left, pad_right: [0, min((kernel_w-1), 15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]			pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Activation	ReLU, ReLU6	not supported	ReLU, ReLU6	not supported	ReLU	not supported	ReLU, ReLU6
average-pooling	Kernel size	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: {2, 3, 5, 7, 8} w==h	w, h: [1, 256] ceil(h/bank_num) * w <= bank_depth	w, h: [1, 8] w==h	w, h: [1, 16]	w, h: [1, 128] w==h	w, h: [1, 256] h * w <= bank_depth
	Strides	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 256]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 128]	w, h: [1, 256]
	Paddings	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]	pad_left, pad_right: [0, min((kernel_w-1), 15)]	pad_left, pad_right: [1, kernel_w-1]			pad_left, pad_right: [0, min((kernel_w-1), 15)]
		pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]	pad_top, pad_bottom: [0, min((kernel_h-1), 15)]	pad_top, pad_bottom: [1, kernel_h-1]			pad_top, pad_bottom: [0, min((kernel_h-1), 15)]
	Activation	ReLU, ReLU6	not supported	ReLU, ReLU6	not supported	ReLU	not supported	ReLU, ReLU6

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
eltwise	type	sum, prod	sum	sum, prod	sum	sum	sum, prod	2-input sum, prod
	Input Channel	input_channel <= 256 * channel_parallel						
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU, Hard-Sigmoid	ReLU
concat		Network-specific limitation, which relates to the size of feature maps, quantization results and compiler optimizations.						
reorg	Strides	reverse==false : stride ^ 2 * input_channel <= 256 * channel_parallel reverse==true : input_channel <= 256 * channel_parallel						
pad	In Size	input_channel <= 256 * channel_parallel						
	Mode	"SYMMETRIC" ("CONSTANT" pad(value=0) would be fused into adjacent operators during compiler optimization process)					"SYMMETRIC", "CONSTANT" (all padding value are identical)	"SYMMETRIC" ("CONSTANT" pad(value=0) would be fused into adjacent operators during compiler optimization process)
global pooling		Global pooling will be processed as general pooling with kernel size equal to input tensor size.						
InnerProduct, Fully Connected, Matmul		These ops will be transformed into conv2d op						

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
resize	scale	NEAREST: ceil(scale/bank_num) * scale * ceil(input_channel/channel_parallel) <= bank_depth BILINEAR: only for 4-D feature maps. This would be transformed into a pad and depthwise-transposed-conv2d. TRILINEAR: only for 5-D feature maps. This would be transformed into a pad and transposed-conv3d.						
	mode	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR, TRILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR	NEAREST, BILINEAR

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
conv3d	kernel size	Not supported	Not supported	w, h, d: [1, 16] w * h * ceil(ceil(input_channel/16) * 16 * d / 2048) <= 64	Not supported	Not supported	Not supported	Not supported
	strides			w, h, d: [1, 8]				
	paddings			pad_left, pad_right: [0, kernel_w-1] pad_top, pad_bottom: [0, kernel_h-1] pad_front, pad_back: [0, kernel_d-1]				
	In size			kernel_w * kernel_h * kernel_d * ceil(input_channel/ channel_parallel) <= bank_depth, input_channel <= 256 * channel_parallel				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid				

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-conv3d	kernel size	Not supported	Not supported	w, h: [1, 256] d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides			w, h: [1, 256] d=1				
	paddings			pad_left, pad_right: [0, min((kernel_w-1), 15)] pad_top, pad_bottom: [0, min((kernel_h-1), 15)] pad_front, pad_back: [0, min((kernel_d-1), 15)]				
	In size			kernel_w * kernel_h * kernel_d * ceil(input_channel/ channel_parallel) <= bank_depth				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, ReLU6				

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
transposed-conv3d	kernel size	Not supported	Not supported	kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides							
	paddings			pad_left, pad_right: [0, kernel_w-1] pad_top, pad_bottom: [0, kernel_h-1] pad_front, pad_back: [0, kernel_d-1]				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid				

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
depthwise-transposed-conv3d	kernel size	Not supported	Not supported	kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16]	Not supported	Not supported	Not supported	Not supported
	strides							
	paddings			pad_left, pad_right: [0, min((kernel_w-1), 15)] pad_top, pad_bottom: [0, min((kernel_h-1), 15)] pad_front, pad_back: [0, min((kernel_d-1), 15)]				
	Out size			output_channel <= 256 * channel_parallel				
	Activation			ReLU, ReLU6				
Strided_slice	Stride	Stride_batch = 1 Stride_channel = 1						
correlation1d_elemwise	input size	input_channel <= 256 * channel_parrallel	Not supported	input_channel <= 256 * channel_parrallel	Not supported	Not supported	Not supported	Not supported
correlation2d_elemwise	input size	input_channel <= 256 * channel_parrallel	Not supported	input_channel <= 256 * channel_parrallel	Not supported	Not supported	Not supported	Not supported

Table 28: **Currently Supported Operators** *(cont'd)*

Typical Operation Type in CNN	Parameters	DPUCZDX8G_ISA1_B4096 [3](ZCU102, ZCU104)	DPUCAHX8L_ISA0 (U50, U50LV, U280)	DPUCVDX8G_ISA3_C32B3 [4](VCK190)	DPUCAHX8H_ISA2_DWC [1](U50, U55C, U50LV, U280)	DPUCADF8H_ISA0 (U200, U250)	DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000)	DPUCV2DX8G_ISA1_C20B1 [5](VEK280/ V70)
Intrinsic Parameter		**channel_parallel: 16** **bank_depth: 2048** **bank_num: 8**	**channel_parallel: 32** **bank_depth: 4096**	**channel_parallel: 16** **bank_depth: 8192** **bank_num: 8**	**channel_parallel: 16** **bank_depth: 2048**	**channel_parallel: 16** **bank_depth: 8192**	**channel_parallel: 64** **bank_depth: 2048**	**channel_parallel: 32** **bank_depth: 65528** **bank_num: 1**
argmax	axis	axis = input_channel	Not supported	axis = input_channel	Not supported	Not supported	Not supported	axis = input_channel
	input size	input_channel <=128		input_channel <=128				input_channel <=128
reduction max	axis	axis = input_channel	Not supported	axis = input_channel	Not supported	Not supported	Not supported	axis = input_channel
	input size	input_channel < 2 ^ 12		input_channel < 2 ^ 12				input_channel < 2 ^ 12
cost_volume	input size	input_channel <= 256 * channel_parallel	Not supported	input_channel <= 256 * channel_parallel	Not supported	Not supported	Not supported	Not supported
transpose								

Notes:

1. For DPUCAHX8H, only list *DPUCAHX8H_ISA2_DWC* here. For more IP configurations, see *DPUCAHX8H for Convolutional Neural Networks Product Guide* (PG367)
2. For DPUCVDX8H, only list *DPUCVDX8H_ISA1_F2W4_4PE* here. For more IP configurations, see *DPUCVDX8H for Convolutional Neural Networks LogiCORE IP* (PG403)
3. For DPUCZDX8G, only list DPUCZDX8G_ISA1_B4096 here. For more IP Configurations, see *DPUCZDX8G for Zynq UltraScale+ MPSoCs* (PG338)
4. For DPUCVDX8G, only list DPUCVDX8G_ISA3_C32B3 here. For more IP Configurations, see *DPUCVDX8G for Versal Adaptive SoCs Product Guide* (PG389)
5. For DPUCV2DX8G, only list DPUCV2DX8G_ISA1_C20B1 here. For more IP Configurations, see *DPUCV2DX8G for Versal Adaptive SoCs Product Guide*(PG425)
6. For DPUCZDX8G, the activation LeakyReLU for depthwise-conv like operators is not enabled by default. About how to enable this activation, please refer to *DPUCZDX8G for Zynq UltraScale+ MPSoCs* (PG338)
7. For DPUCVDX8G, the activation LeakyReLU for depthwise-conv like operators is not enabled by default. About how to enable this activation, please refer to *DPUCVDX8G for Versal Adaptive SoCs Product Guide* (PG389)

Operators Supported by TensorFlow

Table 29: **Operators Supported by TensorFlow**

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

Table 29: **Operators Supported by TensorFlow** *(cont'd)*

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

Table 29: **Operators Supported by TensorFlow** *(cont'd)*

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

Table 29: **Operators Supported by TensorFlow** *(cont'd)*

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

Table 29: **Operators Supported by TensorFlow** *(cont'd)*

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

Notes:

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

Operators Supported by PyTorch

Table 30: **Operators Supported by PyTorch**

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

Table 30: **Operators Supported by PyTorch** *(cont'd)*

PyTorch		XIR		DPU Implementation
API	**Attributes**	**OP name**	**Attributes**	
matmul		conv2d / matmul	transpose_a	The matmul would be transformed to conv2d and compiled to Convolution Engine. If the matmul fails to be transformed, it would be implemented by the CPU.
			transpose_b	
MaxPool2d / AdaptiveMaxPool2d	kernel_size	maxpool2d	kernel	Pooling Engine
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	output_size (adaptive)		global	
AvgPool2d / AdaptiveAvgPool2d	kernel_size	avgpool2d	kernel	Pooling Engine
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	count_include_pad		count_include_pad	
			count_include_invalid (true)	
	output_size (adaptive)		global	
ReLU		relu		Activations would be fused to adjacent operations such as convolution.
LeakyReLU	negative_slope	leakyrelu	alpha	
ReLU6		relu6		
Hardtanh	min_val = 0			
	max_val = 6			
Hardsigmoid		hard-sigmoid		
Hardswish		hardswish		
ConstantPad2d / ZeroPad2d	padding	pad	paddings	First compiler will try to fuse "CONSTANT" padding into adjacent operations, for example, convolution and pooling. If no such operator exists, it can still be mapped to DPU when the padding dimension equals four and meets the hardware requirements.
	value = 0		constant_values	
			mode ("CONSTANT")	

Table 30: **Operators Supported by PyTorch** *(cont'd)*

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

Table 30: **Operators Supported by PyTorch** *(cont'd)*

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

Table 30: **Operators Supported by PyTorch** *(cont'd)*

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

Notes:

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

VAI_C Usage

The corresponding Vitis AI compilers for Caffe and TensorFlow frameworks are `vai_c_caffe`, `vai_c_tensorflow`, `vai_c_tensorflow2`, and `vai_c_xir` across Data Center-to-Edge DPUs. The standard options for VAI_C are shown in the following table.

Table 31: **VAI_C Common Options for Data Center and Edge DPU**

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

Chapter 5

Deploying and Running the Model

Programming with VART

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

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

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

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

   The job ID returned by `execute_async` is passed to `wait()` to block until the job is complete and the results are ready.

3. ```
   TensorFormat get_tensor_format()
   ```

   Query the DpuRunner for the Tensor format it expects.

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

4. ```
   std::vector<Tensor*> get_input_tensors()
   std::vector<Tensor*> get_output_tensors()
   ```

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

5. To create a DpuRunner object, call the following: function

   ```
   create_runner(const xir::Subgraph* subgraph, const std::string& mode =
   "")
   ```

   It returns the following:

   ```
   std::unique_ptr<Runner>
   ```

   The input to create_runner is an XIR subgraph generated by the Vitis AI compiler.

TIP: *To enable multi-threading with VART, create a runner for each thread.*

***Note*:** If the model has multiple subgraphs, refer to

https://github.com/Xilinx/Vitis-AI-Tutorials/tree/3.0/Tutorials/pytorch-subgraphs/.

C++ Example

```
// get dpu subgraph by parsing model file
auto runner = vart::Runner::create_runner(subgraph, "run");
// populate input/output tensors
auto job_data = runner->execute_async(inputs, outputs);
runner->wait(job_data.first, -1);
// process outputs
```

Vitis AI also provides a Python ctypes Runner class that mirrors the C++ class, using the C DpuRunner implementation:

```
class Runner:
def __init__(self, path)
def get_input_tensors(self)
def get_output_tensors(self)
def get_tensor_format(self)
def execute_async(self, inputs, outputs)
# differences from the C++ API:
# 1. inputs and outputs are numpy arrays with C memory layout
#    the numpy arrays should be reused as their internal buffer
#    pointers are passed to the runtime. These buffer pointers
#    may be memory-mapped to the FPGA DDR for performance.
# 2. returns job_id, throws exception on error
def wait(self, job_id)
```

Python Example

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

DPU Debug with VART

This section demonstrates how to verify DPU inference results with VART tools. TensorFlow ResNet50, and PyTorch ResNet50 networks are used as examples. Following are the four steps for debugging the DPU with VART:

1. Generate a quantized inference model and reference result.
2. Generate a DPU XMODEL.
3. Generate a DPU inference result.
4. Crosscheck the reference result and the DPU inference result.

Before you start to debug the DPU result, ensure that you have set up the environment according to the instructions in https://xilinx.github.io/Vitis-AI/docs/quickstart/vek280.html

***Note*:** Caffe was deprecated with AMD Vitis™ AI 2.5. For information on Caffe, see Vitis AI 2.0 User Guide.

TensorFlow Workflow

To generate the quantized inference model and reference result, follow these steps:

1. Generate the quantized inference model by running the following command to quantize the model.

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

   ```
   vai_q_tensorflow quantize                                          \
       --input_frozen_graph ./float/resnet_v1_50_inference.pb         \
       --input_fn input_fn.calib_input                                \
       --output_dir quantize_model                                    \
       --input_nodes input                                            \
       --output_nodes resnet_v1_50/predictions/Reshape_1              \
       --input_shapes    ?,224,224,3                                  \
       --calib_iter    100
   ```

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

   ```
   vai_q_tensorflow dump --input_frozen_graph        \
              quantize_model/quantize_eval_model.pb \
        --input_fn input_fn.dump_input               \
        --output_dir=dump_gpu
   ```

   The following figure shows part of the reference data.

```
input_aquant.bin
input_aquant.txt
resnet_v1_50_Pad_aquant.bin
resnet_v1_50_Pad_aquant.txt
resnet_v1_50_SpatialSqueeze_aquant.bin
resnet_v1_50_SpatialSqueeze_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.txt
```

3. Generate the DPU xmodel by running the following command to generate the DPU xmodel file, for example, V70.

```
vai_c_tensorflow --frozen_pb quantize_model/quantize_eval_model.pb \
  --arch /opt/vitis_ai/compiler/arch/DPUCV2DX8G/V70/arch.json       \
  --output_dir compile_model                                        \
  --net_name resnet50_tf
```

4. Run the following command to generate the DPU inference result and compare the DPU inference result with the reference data automatically.`

```
env XLNX_ENABLE_DUMP=1 XLNX_ENABLE_DEBUG_MODE=1 XLNX_GOLDEN_DIR=./
dump_gpu/dump_results_0 \
   xdputil run ./compile_model/resnet_v1_50_tf.xmodel             \
   ./dump_gpu/dump_results_0/input_aquant.bin                     \
   2>result.log 1>&2
```

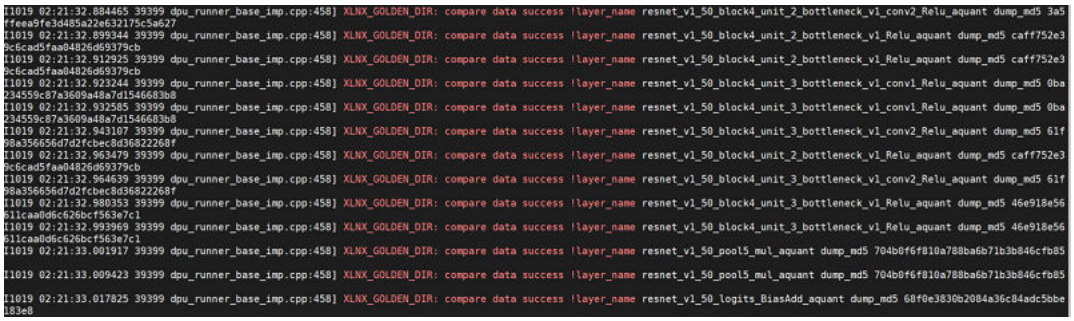
   For `xdputil` more usage, execute `xdputil --help` command.

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

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

   a. View comparison results for all layers.

```
grep --color=always 'XLNX_GOLDEN_DIR.*layer_name' result.log
```



b. View only the failed layers.

```
grep --color=always 'XLNX_GOLDEN_DIR.*fail ! layer_name' result.log
```

If the crosscheck fails, use the following methods to check further from which layer the crosscheck fails.

a. Check the input of DPU and GPU. Ensure that they use the same input data.

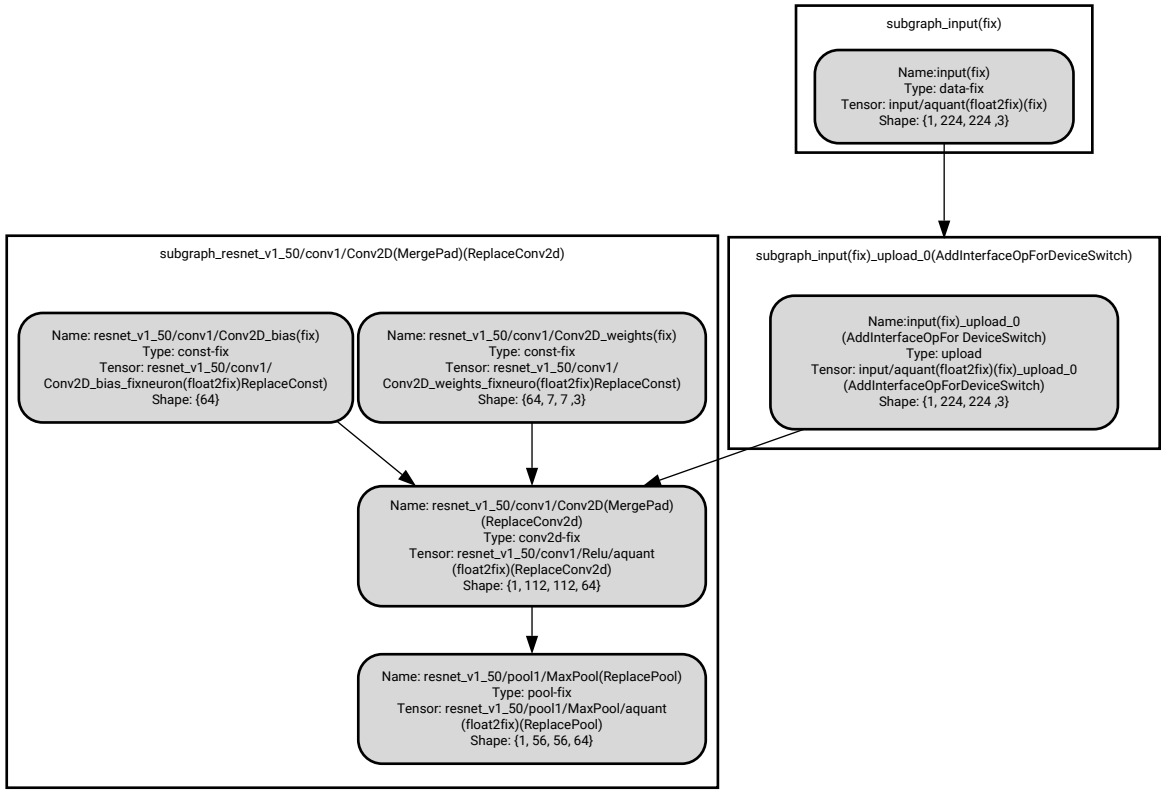
b. Use the `xdputil` tool to generate a picture for displaying the network's structure.

```
Usage: xdputil xmodel <xmodel> -s <svg>
```

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

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

When you open the picture you created, you can see many little boxes around these ops. Each box means a layer on DPU. You can use the last op's name to find its corresponding one in the GPU dump result. The following figure shows parts of the structure.



c. Submit the files to AMD.

   If a particular layer proves wrong on DPU, prepare the quantized model, such as `quantize_eval_model.pb` , as one package for further analysis by the factory and send it to AMD with a detailed description.

PyTorch Workflow

To generate the quantized inference model and reference result, follow these steps:

1. Generate the quantized inference model by running the following command to quantize the model.

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

2. Set `deploy_check` to `True` in `export_xmodel` API.

   ```
   quantizer.export_xmodel(deploy_check=True)
   ```

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

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

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

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

5. Generate the DPU inference result.

   This step is the same as the step in the TensorFlow workflow.

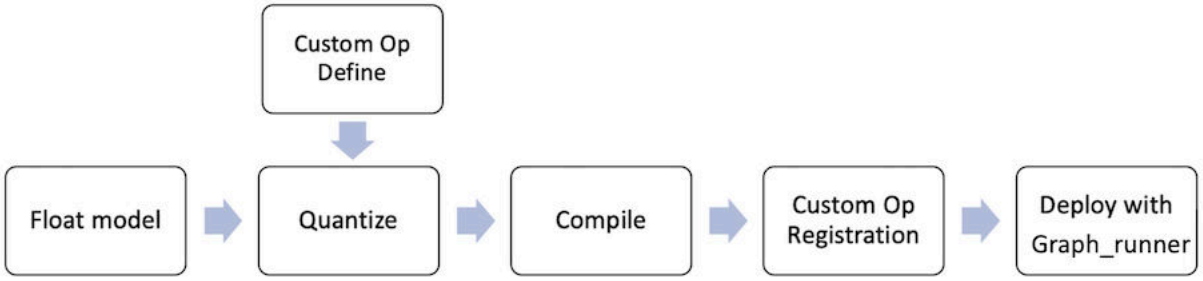
6. Crosscheck the reference result and the DPU inference result.

   This step is the same as the step in the TensorFlow workflow.

Custom OP Workflow

Starting with version 2.5, Vitis AI supports Pytorch and Tensorflow2 models with custom op. The basic workflow for custom op is shown below.

Figure 23: **Custom Op Workflow**



The following are the steps in the workflow:

1. Define the OP as a custom OP unknown to XIR and then quantize the model.
2. Compile the quantized model.
3. Register and implement the custom OP.
4. Deploy the model with `graph_runner` APIs.

Step 3 supports `C++` and `Python` to implement and register the custom OP. There are more than 50 supported common OPs by the Vitis AI library. You can find the source code of the common OPs in https://github.com/Xilinx/Vitis-AI/tree/v3.5/src/vai_library/cpu_task/ops.

***Note*:** If you want to implement an accelerated (PL or AI Engine) function for a custom op, make it a CPU OP, but implement the PL/AI Engine calling codes in this CPU OP's implementation.

For step 4, `graph_runner` APIs support both `C++` and `Python`. When using the Graph_runner API to deploy Custom OP, its runtime has been optimized, including Zero-copy technology between different DPU OPs and CPU OPs. It means address sharing between different layers without data copying.

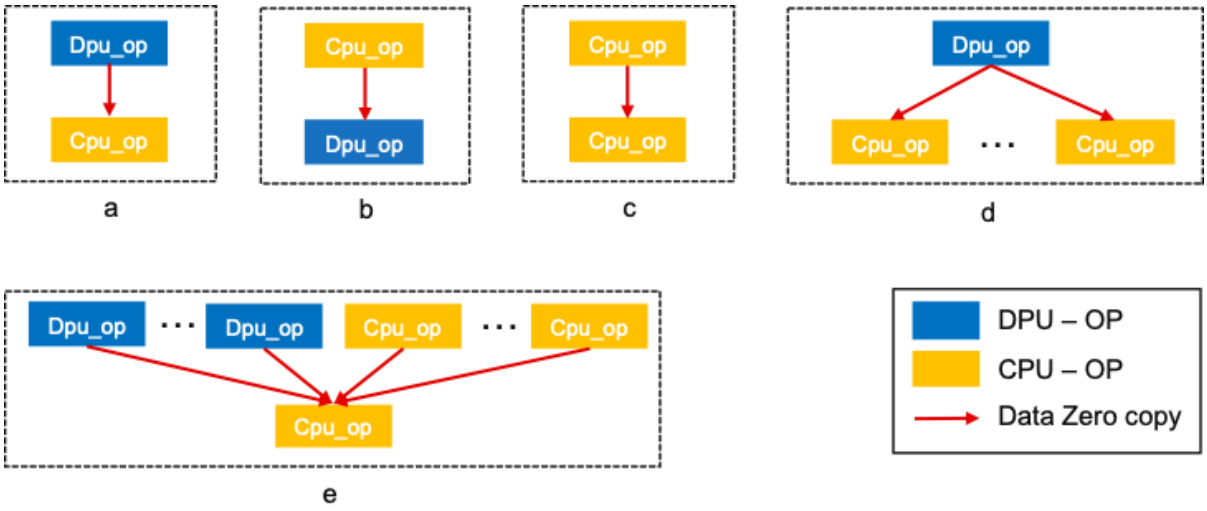
The following model structure is supported by Zero copy.

Table 32: **Model structure supported by Zero copy**

Type	Output of OP	Input of OP	Using Zero copy
a	Single dpu OP	Single cpu OP	Yes
b	Single cpu OP	Single dpu OP	Yes
c	Single cpu OP	Single cpu OP	Yes
d	Single dpu OP	Multiple cpu OP	Yes
e	Multiple cpu OP and multiple dpu OP	Single cpu OP	Yes

***Note*:** Model structure types a-e are shown in the following figure.

Figure 24: **Model Structure Types**



***Note*:** The application of Zero copy for the other model structures depends on the situation.

The following are examples of the two models, respectively.

- MNIST model based on Tensorflow2
- Pointpillars model based on Pytorch

Quick Start Custom OP Workflow

This section uses a TensorFlow2 model with custom OP to show how to quick-start custom OP workflow on the Edge ZCU102 platform.

Quantize

1. Launch the Docker image:

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

2. Download the model source code package tf2_custom_op_demo.tar.gz:

```
[Docker]$ wget https://www.xilinx.com/bin/public/openDownload?
filename=tf2_custom_op_demo.tar.gz -O tf2_custom_op_demo.tar.gz
[Docker]$ tar -xzvf tf2_custom_op_demo.tar.gz
[Docker]$ cd tf2_custom_op_demo
```

3. Quantize:

```
[Docker]$ conda activate vitis-ai-tensorflow2
[Docker]$ bash 1_run_train.sh
[Docker]$ bash 3_run_quantize.sh
```

After quantizing, the quantized model named `quantized.h5` is generated in the `./quantized/` directory.

Compile

```
[Docker]$ vai_c_tensorflow2 -m ./quantized/quantized.h5 -a /opt/vitis_ai/
compiler/arch/DPUCZDX8G/ZCU102/arch.json -o ./ -n tf2_custom_op
```

OP Registration

1. Copy the `tensorflow2_example` folder to the target board:

```
[Host]$ scp -r Vitis-AI/examples/custom_operator/tensorflow2_example
root@[BOARD_IP]:~
```

2. Run the following commands to register the custom OP on the target:

```
[Target]# cd ~/tensorflow2_example/op_registration/cpp
[Target]# bash op_registration.sh
```

Deployment

1. Copy the compiled model to the board:

```
[Host]$ scp tf2_custom_op.xmodel root@[BOARD_IP]:~
```

2. Download the test image sample.jpg and copy it to the board.
3. Compile the application code on the target:

```
[Target]# cd ~/tensorflow2_example/deployment/cpp
[Target]# bash build.sh
```

4. Run the demo:

```
[Target]# ./tf2_custom_op_graph_runner ~/tf2_custom_op.xmodel ~/
sample.jpg
```

Tensorflow2 Custom OP Model Example

Using the Tensorflow2 model as an example, download the code package from here. Refer to `the readme.md` in the package to generate and quantize the model.

Quantizing the Model

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

Vai_q_tensorflow2 provides support for new custom layers through subclassing. This tutorial will demonstrate how to quantize models with custom operations step-by-step.

***Note*:** Custom model via subclassing `tf.keras.T` is not yet supported by vai_q_tensorflow2. Flatten it to layers.

1. Train custom layer model

This example defines the custom layer named `MyLayer` to perform a "PReLU" function in `train_eval.py`. This custom layer performs the function below with a trainable weight `alpha`.

```
f(x) = alpha * x , if x < 0
f(x) = x , if x >= 0
```

where `alpha` is a learned array with the same shape as `x`.

A CNN model is built next to classify the MNIST dataset as an example. Before you train and quantize the model, launch the docker and activate `the vitis-ai-tensorflow2` environment. Run `1_run_train.sh` to train the model, and you will get the float model `my_model.h5`. The accuracy of the model should be >90%.

```
bash 1_run_train.sh
```

```
Epoch 9/10
32/32 [==============================] - 0s 5ms/step - loss: 0.0108 - accuracy: 0.9960
Epoch 10/10
32/32 [==============================] - 0s 4ms/step - loss: 0.0078 - accuracy: 0.9990
313/313 [==============================] - 1s 3ms/step - loss: 0.0530 - accuracy: 0.9237

****************** Summary ******************
Trained float model accuracy:  0.9236999750137329
Trained float model is saved in  ./my_model.h5
```

This float model contains both the model structure and the weights, with a custom layer named `custom_layer`. You can get this information from the printed summary.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 22, 22, 32)        1600
_________________________________________________________________
batch_normalization (BatchNo (None, 22, 22, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 32)        50208
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
custom_layer (Mylayer)       (None, 2, 2, 64)          256
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 10)                2570
=================================================================
Total params: 106,154
Trainable params: 106,026
Non-trainable params: 128
_________________________________________________________________
```

2. (Optional) Evaluate the float model

You can run the script `2_run_eval_float.sh` to test the trained float model.

```
bash 2_run_eval_float.sh
```

3. Quantize the float model

You can quantize the float model with custom layers using the vai_q_tensorflow2 `quantize_model` API. Example code is shown below:

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quant_model = vitis_quantize.VitisQuantizer(loaded_model,
custom_objects={'MyLayer': MyLayer}).quantize_model(calib_dataset=x_test,
add_shape_info=True)
```

The custom_objects argument must be passed into the VitisQuantizer class when quantizing models with custom layers. The `custom_objects` argument is a dict containing the `{"custom_layer_class_name":"custom_layer_class"}`. Multiple custom layers should be separated by a comma. Moreover, `add_shape_info` should also be set to True for models with custom layers to add shape inference information for them.

These custom layers will be kept untouched in the quantized model during quantization. Run `3_run_quantize.sh` to do quantization:

```
bash 3_run_quantize.sh
```

If everything runs correctly, the quantized model named `quantized.h5` will be generated in `the ./quantized/` directory. This model can be used as the input of the xcompiler and then deployed on boards.

```
2022-01-06 12:43:53.428589: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 8s 17ms/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quantize Adjustment...
[VAI INFO] Post-Quantize Adjustment Done.
[VAI INFO] Quantization Finished.
[VAI INFO] Start Getting Shape Information...
[VAI INFO] Getting model layer shape information
[VAI INFO] Getting Shape Information Done.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` wi
ll be empty until you train or evaluate the model.

**************** Summary ****************
Quantized model is saved in  ./quantized/quantized.h5
```

4. (Optional) Evaluate the quantized model

Use `model.evaluate` API to evaluate the quantized model. Remember to recompile the model with correct losses and metrics because this information is ignored during quantization.

```
quantized_model.compile(loss="binary_crossentropy", metrics=["accuracy"])
quantized_model.evaluate(x_test, y_test)
```

Run `4_run_eval_quant` to evaluate the quantized model.

```
bash 4_run_eval_quant.sh
```

It can be seen that the quantized model has close accuracy to the float model.

```
2022-01-06 12:44:56.418449: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimizati
on Passes are enabled (registered 2)
2022-01-06 12:44:57.448941: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 3s 4ms/step - loss: 0.0897 - accuracy: 0.9207

**************** Summary ****************
Quantized model accuracy:  0.9207000136375427
```

5. Dump the golden results

Golden results are used to check the data correctness or debug the deployed models. Vai_q_tensorflow2 provides `dump_model` API to dump the quantized model's weights/biases and intermediate activations with a sample input. Since the DPU dumping results are batch by batch, set the batch_size of the dataset to 1 when dumping golden results.

```
vitis_quantize.VitisQuantizer.dump_model( model=quant_model,
dataset=x_test[0:1], output_dir="./dump_results", dump_float=True)
```

Since the custom layer is not quantized, set `dump_float=True` to dump float weights and activation for them. Run `5_run_dump.sh` to dump the quantized model.

```
bash 5_run_dump.sh
```

You can see the generated golden results in the folder `./dump_results`. `./dump_results/dump_results_weights` are the save weights, while `./dump_results/dump_results_0` are the saved activation, where the number 0 represents the index of the dataset.

Compiling the Model

The following are the commands for TensorFlow2:

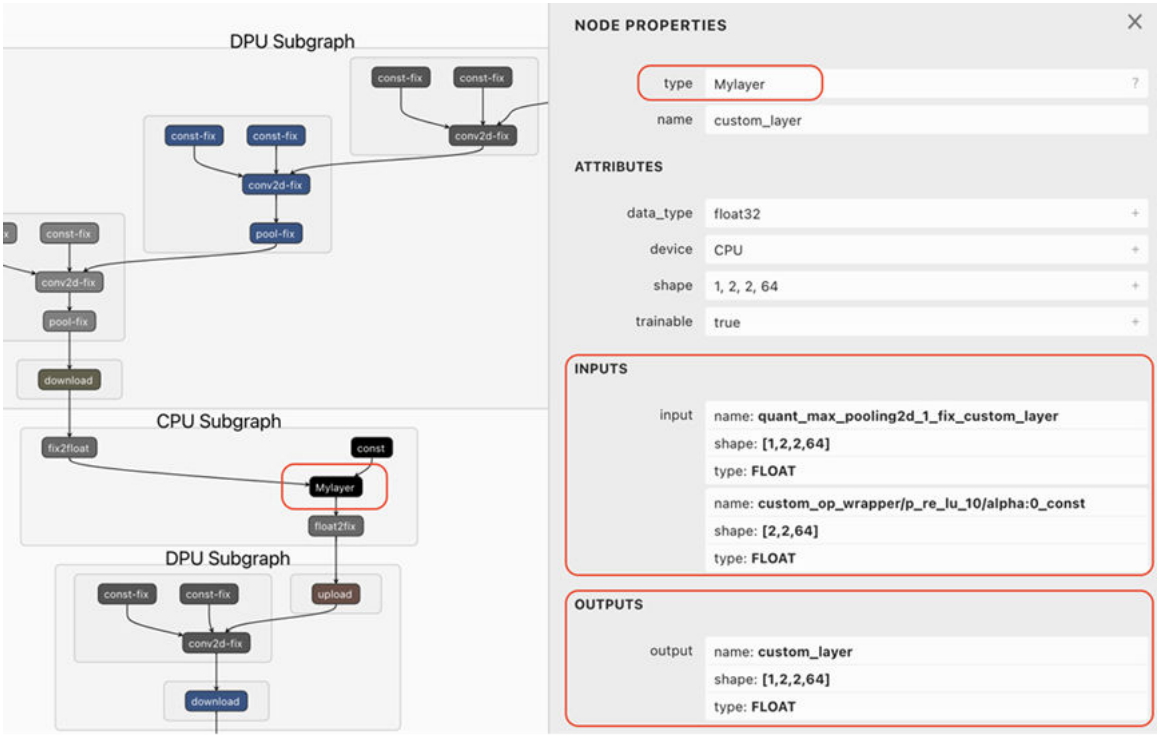
```
conda activate vitis-ai-tensorflow2
cd <path of Vitis-AI>/examples/custom_operator/tensorflow2_example/model/
quantized
vai_c_tensorflow2 -m ./quantized.h5 -a /opt/vitis_ai/compiler/arch/
DPUCV2DX8G/VEK280/arch.json -o ./ -n tf2_custom_op
```

Custom OP Registration

This section describes how to register a custom OP step-by-step.

You can use the `Netron` to check the custom OP properties, such as inputs and outputs.

Figure 25: **Custom OP properties**



You can also use `xdputil` to check the OP's detailed information. Run the following command to check the `custom_layer` OP:

```
xdputil xmodel tf2_custom_op.xmodel --op custom_layer
```

Then, you can create a directory for the new OP and all your coding and building can be done under this new folder. Because `Mylayer` op has been realized in VAI2.0, use `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer` for reference.

Code

Custom OP registration supports both C++ and Python. The following steps describe how to implement the OP in C++. For the OP implementation in Python, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/python/`:

1. Create CPP file for the new OP. It is my_Mylayer_op.cpp in this example.
2. Include the header `<vart/op_imp.h>`.
3. Define a class with a constructor and a function named calculate. The class name is `MylayerOp in this example`.
4. Implement your algorithm in the `calculate` function. `PReLU` is implemented in this example.

5. Register your class by DEF_XIR_OP_IMP(className), for Mylayer, className is `MylayerOp`.

   Note: You can copy the CPP file from my_Mylayer_op.cpp and modify the class name as needed, and then you can focus on step 4 and 5. The details of CPP file for Mylayer OP are as follows:

```
#include <vart/op_imp.h>
class MylayerOp {
 public:
  MylayerOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {}
  int calculate(vart::simple_tensor_buffer_t<float> output,
                std::vector<vart::simple_tensor_buffer_t<float>> inputs) {
    CHECK_EQ(inputs.size(), 2);
    auto input_data_shape = inputs[0].tensor->get_shape();
    auto input_alpha_shape = inputs[1].tensor->get_shape();
    auto output_shape = output.tensor->get_shape();
    auto dims = output_shape.size();

    CHECK_EQ(input_data_shape.size(), 4);
    CHECK_EQ(input_alpha_shape.size(), 3);
    for (auto i = 1u; i < dims; i++)
      CHECK_EQ(input_data_shape[i], input_alpha_shape[i - 1]);

    auto element_num = inputs[0].tensor->get_element_num();
    auto alpha_size = inputs[1].tensor->get_element_num();
    for (auto i = 0; i < element_num; i++) {
      if (inputs[0].data[i] < 0) {
        output.data[i] = inputs[0].data[i] * inputs[1].data[i % alpha_size];
      } else {
        output.data[i] = inputs[0].data[i];
      }
    }

    return 0;
  }

 public:
  const xir::Op* const op;
};

DEF_XIR_OP_IMP(MylayerOp)
```

Build

1. Create a Makefile to build the OP library.

   Refer to `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer/Makefile`.

2. Set the output directory and add the dependency between target (output .so), object (.o), and source (.cpp) files.

   If you use the reference Makefile in step 1, you only need to replace the file names with yours, including `libvart_op_imp_Mylayer.so`, `my_Mylayer_op.o` and `my_Mylayer_op.cpp`.

3. Execute “make” to complete the build of the library.

   After building, `libvart_op_imp_Mylayer.so` library is generated.

The details of Makefile for Mylayer OP are as follows:

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_Mylayer.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_Mylayer_op.o: my_Mylayer_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_Mylayer.so: $(OUTPUT_DIR)/my_Mylayer_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -lglog -
lvitis_ai_library-runner_helper -lvart-runner -lxir
```

***Note*:** The name of the output library must follow the same format as libvart_op_imp_Mylayer.so. In this format, "Mylayer", which is the type of Mylayer OP, must be changed to the type of your custom OP. Otherwise, the library cannot be linked when debugging the OP or running the graph.

Debug

This section introduces how to debug the custom OP after you generate the custom OP library. Before you debug the custom OP, copy the `libvart_op_imp_Mylayer.so` to `/usr/lib` on the target. Then, use `run_op` command in `xdputil` to test the OP on the target. The usage of `run_op` is as follows:

```
xdputil run_op <model_file> <op_name> [-r ref] [-d dump]
```

***Note*:** For Edge, run `xdputil run_op` on the board.

Execute `xdputil run_op -h` to view the valid arguments for `run_op`.

```
root@xilinx-zcu102-2021_2:~# xdputil run_op -h
usage: xdputil.py run_op [-h] [-r REF_DIR] [-d DUMP_DIR] xmodel op_name

positional arguments:
  xmodel                xmodel file name
  op_name               op name, this op_name should be consistent with the name in xmodel

optional arguments:
  -h, --help            show this help message and exit
  -r REF_DIR, --ref_dir REF_DIR
                        reference directory, this directory default as "ref" should contain inputs tensor file like <TENSOR_NAME>.bin
  -d DUMP_DIR, --dump_dir DUMP_DIR
                        dump directory, this directory default as "dump" will be the dump destination of output tensor file
```

In the ref directory, you must provide all the input bin files with the same names as the input tensors of the custom OP. For `Mylayer`, it has two inputs and the input tensor names are `quant_max_pooling2d_1_fix_custom_layer` and `custom_op_wrapper/p_re_lu_10/alpha:0_const`, as shown in the following figure. Thus, the input files stored in the ref directory should be `quant_max_pooling2d_1_fix_custom_layer.bin` and `custom_op_wrapper_p_re_lu_10_alpha:0_const.bin`.

Note: You can dump the golden files when you perform model quantization.

Note: The slash ("/") marks in the name should be replace by underscore ("_").

INPUTS

input	op_name: **quant_max_pooling2d_1_fix_custom_layer**
	tensor_name: **quant_max_pooling2d_1_fix_custom_layer**
	shape: **[1,2,2,64]**
	type: **FLOAT32**
	op_name: **custom_op_wrapper/p_re_lu_10/alpha:0_const**
	tensor_name: **custom_op_wrapper/p_re_lu_10/alpha:0_const**
	shape: **[2,2,64]**
	type: **FLOAT32**

OUTPUTS

output	op_name: **custom_layer**
	tensor_name: **custom_layer**
	shape: **[1,2,2,64]**
	type: **FLOAT32**

If you still do not know what the names should be, you can put your input files in ref and try to execute run_op. Then you can find the file names expected in the resulting error message as shown in the following figure.

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
default directory ./ref has been created, please put the input tensor file in.
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:22:48.982190 16636 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:22:48.987767 16636 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:22:48.987846 16636 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:22:48.987871 16636 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
F1202 04:22:48.988036 16636 test_op_run.cpp:52] Check failed: std::ifstream(filename).read((char*)data.data, data.size).good() fail to
read! filename=./ref/quant_max_pooling2d_1_fix_custom_layer.bin;tensor=quant_max_pooling2d_1_fix_custom_layer
*** Check failure stack trace: ***
/usr/bin/xdputil: line 16: 16636 Aborted                 /usr/bin/python3 -m xdputil $*
```

After the successful execution of `run_op`, as shown in the following figure, you can find the output bin file with the same name as the output tensor in the dump directory. You can compare it with the golden output and debug your code until they are the same. For Mylayer, the output tensor name is `custom_layer`, so the output bin file name should be `custom_layer.bin` as shown in the following figure.

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:17:31.793207 16626 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:17:31.798681 16626 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:17:31.798761 16626 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:17:31.798786 16626 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
I1202 04:17:31.799113 16626 test_op_run.cpp:55] read ./ref/quant_max_pooling2d_1_fix_custom_layer.bin to 0xaaab004831e0 size=1024
I1202 04:17:31.799202 16626 test_op_run.cpp:55] read ./ref/custom_op_wrapper_p_re_lu_10_alpha:0_const.bin to 0xaaab00506a90 size=1024
I1202 04:17:31.799890 16626 test_op_run.cpp:114] graph name:functional_1testing op: {
       {args: input= TensorBuffer{@0xaaab004d2fa0,tensor=xir::Tensor{name = quant_max_pooling2d_1_fix_custom_layer, type = FLOAT32, sh
ape = {1, 2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab004831e0, 1024)]} TensorBuffer{@0xaaab004d2220,tensor=xir::Tensor{name = cust
om_op_wrapper/p_re_lu_10/alpha:0_const, type = FLOAT32, shape = {2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab00506a90, 1024),(Virt=
0xaaab00506c90, 512)]}}
{
I1202 04:17:31.800941 16626 test_op_run.cpp:68] write output to dump/custom_layer.bin from 0xaaab004e11e0 size=1024
test pass
```

Finally, you can compare the `custom_layer.bin` with the golden output file of the custom OP. If they are same, that means the OP library you implemented is correct.

***Note*:** Because the floating point numbers can vary on different platforms, it can cause your dump and golden results in not having an exact match. Thus, the following tool for floating-point number comparison is supplied. If the difference is within a certain threshold, you can consider it consistent:

```
xdputil comp_float <golden_file> <dump_file> [-t threshold] [--verbose]

-t: threshold, the default value is 0.5, in %
```

Deployment

This section will introduce how to deploy the tensorflow2 model with custom op in `graph_runner` APIs. For `graph_runner` APIs, it supports both C++ and Python. For C++ examples, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp` . For Python examples, refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/python`.

TIP: *You can create a new folder for your application, then all your code and build files can be placed under this folder.*

Code

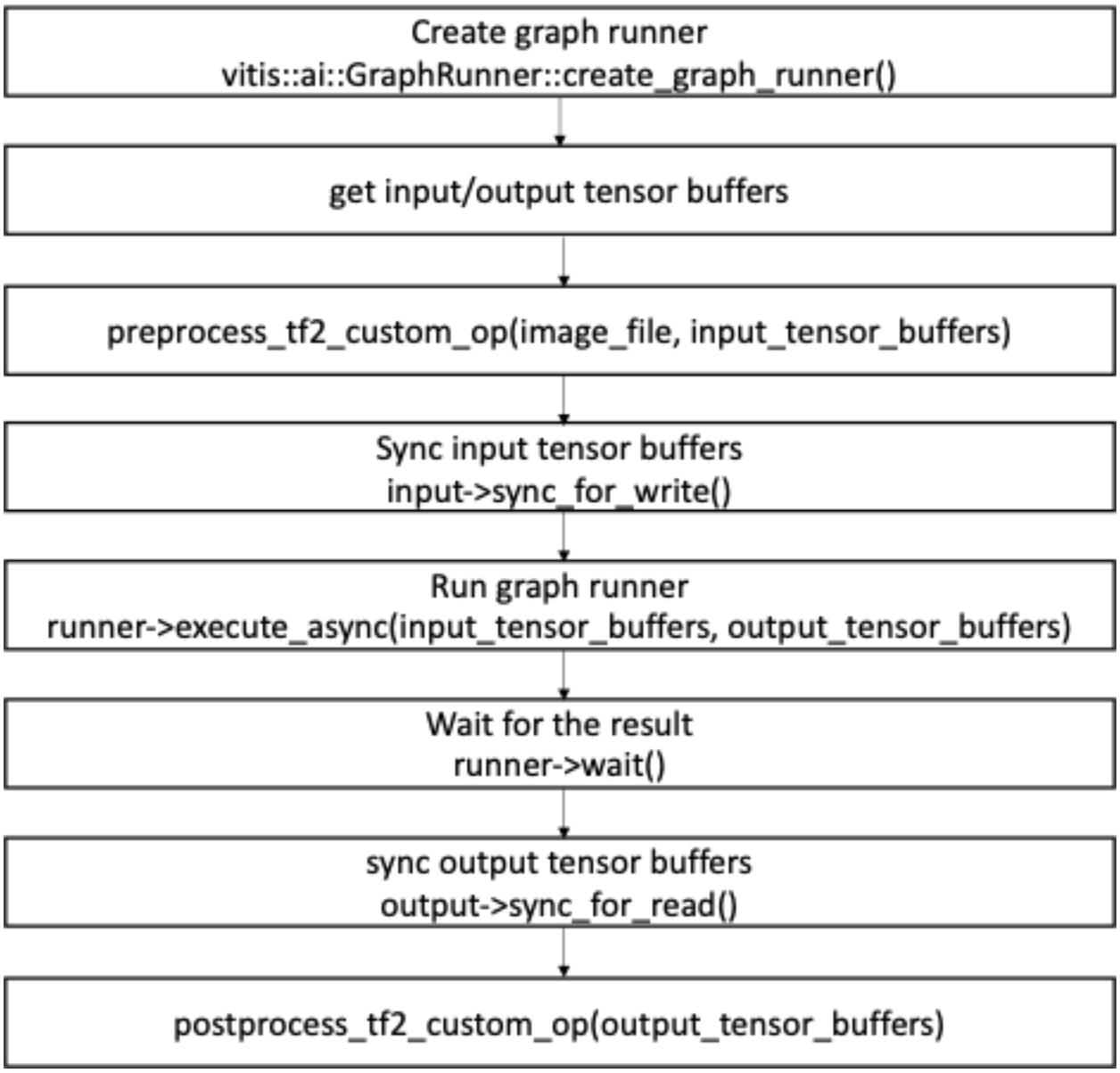
1. Create a cpp source file, such as `tf2_custom_op_graph_runner.cpp`
2. Include the header: `<vitis/ai/graph_runner.hpp>`
3. Implement your model's pre-process, post-process and result-display-process in the following functions.

```
static void preprocess_tf2_custom_op(…)
static void postprocess_tf2_custom_op(…)
static void print_result(…)
```

Tips: You can copy the cpp source file from `tf2_custom_layer_graph_runner.cpp` and focus on step 3 to implement the previous three functions.

The following shows the flow of `tf2_custom_op_graph_runner.cpp`

Figure 26: **The workflow of tf2_custom_op_graph_runner**



Part of the code is shown below.

```
  // create graph runner
  auto graph = xir::Graph::deserialize(xmodel_file);
  auto attrs = xir::Attrs::create();
  auto runner =
      vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
  CHECK(runner != nullptr);
```

```
  // get input/output tensor buffers
  auto input_tensor_buffers = runner->get_inputs();
  auto output_tensor_buffers = runner->get_outputs();

  // preprocess
  preprocess_tf2_custom_op(image_file, input_tensor_buffers);

  // sync input tensor buffers
  for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
                                 input->get_tensor()->get_shape()[0]);
  }

  // run graph runner
  auto v = runner->execute_async(input_tensor_buffers,
output_tensor_buffers);
  auto status = runner->wait((int)v.first, -1);
  CHECK_EQ(status, 0) << "failed to run the graph";

  // sync output tensor buffers
  for (auto output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
                                 output->get_tensor()->get_shape()[0]);
  }

  // postprocess
  postprocess_tf2_custom_op(output_tensor_buffers);
```

Build

1. Create `a build.sh` to build the code using the following instructions. Refer to `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp/build.sh`:

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
    -o tf2_custom_op_graph_runner \
    tf2_custom_op_graph_runner.cpp \
    -lglog \
    -lxir \
    -lvart-runner \
    -lvitis_ai_library-graph_runner \
    ${OPENCV_FLAGS} \
    -lopencv_core \
    -lopencv_imgcodecs \
    -lopencv_imgproc
```

2. Execute `bash build.sh` to build the program on the target, and the executable program `tf2_custom_op_graph_runner` will be generated.

Run

Before you run the demo, ensure that the board's environment has been set up correctly. Also, ensure the following files are generated or ready. Then copy them to the target:

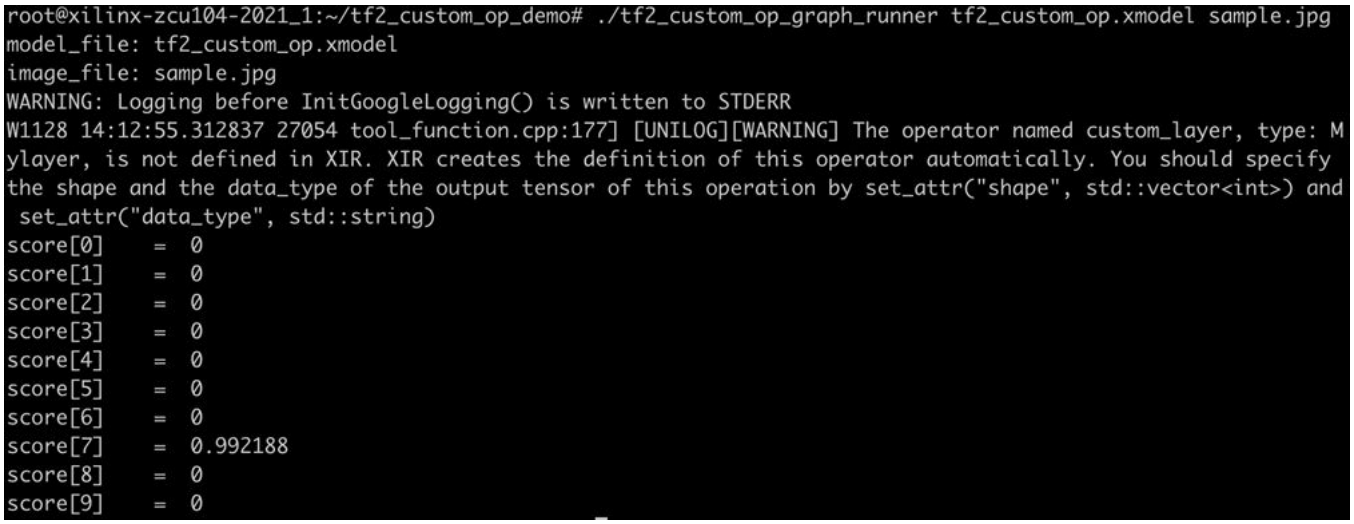
- compiled model, such as tf2_custom_op.xmodel
- custom op library, such as libvart_op_imp_Mylayer.so
- test image from here
- executable program, such as tf2_custom_op_graph_runner

Copy the custom op library to `/usr/lib` on the target. Then, run the following command to test the model on the target.

```
./tf2_custom_op_graph_runner tf2_custom_op.xmodel sample.jpg
```

The following figure shows the result of running `tf2_custom_op_graph_runner`.

Figure 27: **tf2_custom_op_graph_runner example**



Pytorch Custom OP Model Example

Download the float model and code package from here using the Pointpillars model as an example, and then refer to `README.md` in the package to set up the environment.

Quantizing the Model

vai_q_pytorch provides a decorator to register an operation or a group of operations as a custom operation unknown to

XIR.

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

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

"""
```

To quantize a model with custom op, edit the code by following these steps:

1. Move the target code into a function and change its calling accordingly. To the Pointpillar model, replace the PointPillarsScatter model with a PPScatterV2 function. Check related code in `the code/test/models/voxelnet.py` file.
2. Decorate this function with decorator API:

```
from pytorch_nndct.utils import register_custom_op
...

@register_custom_op("PPScatterV2", attrs_list=['ny', 'nx', 'nchannels'])
def PPScatterV2(ctx, voxel_features, coords, ny, nx, nchannels):
    '''
    input:
    voxel_features: B x 64 x 12000 x 1
    coords: B x 12000 x 4, 4 channels: [batch_idx, z_idx, y_idx, x_idx]
    '''
    batch_size = voxel_features.shape[0]
    # batch_canvas will be the final output.
    batch_canvas = []

    for b_idx in range(batch_size):
        # Create the canvas for this sample
        canvas = torch.zeros(nchannels, nx * ny,
dtype=voxel_features.dtype,
                             device=voxel_features.device)
        # Only include non-empty pillars

        batch_mask = coords[b_idx, :, 0] > -1
        this_coords = coords[b_idx, batch_mask, :]
        indices = this_coords[:, 2] * nx + this_coords[:, 3]
        indices = indices.type(torch.long)

        voxels = voxel_features[b_idx, :, batch_mask, 0]

        # Now scatter the blob back to the canvas.
        canvas[:, indices] = voxels
        # Append to a list for later stacking.
        batch_canvas.append(canvas)
```

```
    # Stack to 3-dim tensor (batch-size, nchannels, nrows*ncols)
    batch_canvas = torch.stack(batch_canvas, 0)
    # Undo the column sthe the tacking to final 4-dim tensor
    batch_canvas = batch_canvas.view(batch_size, nchannels, ny, nx)
    return batch_canvas
```

After the target custom op code has been prepared and decorated, add general vai_q_pytorch API functions (check the related code in code/test/test.py)

```
if quant_mode != 'float':
    max_voxel_num = config.eval_input_reader.max_number_of_voxels
    max_point_num_per_voxel =
model_cfg.voxel_generator.max_number_of_points_per_voxel
    aug_voxels = torch.randn((1, 4, max_voxel_num,
max_point_num_per_voxel)).to(device)
    # coors = torch.randn((max_voxel_num, 4)).to(device)
    coors = torch.randn((1, max_voxel_num, 4)).to(device)
    quantizer = torch_quantizer(quant_mode=quant_mode,
                                module=net,
                                input_args=(aug_voxels, coors),
                                output_dir=output_dir,
                                device=device,
                                )
    net = quantizer.quant_model
...
...
for example in iter(eval_dataloader):
...
    if quant_mode == 'test' and args.dump_xmodel:
        quantizer.export_xmodel(output_dir=output_dir, deploy_check=True)
        sys.exit()
...
...
if quant_mode == 'calib':
    quantizer.export_quant_config()
```

After all the changes are ready, run script code/test/run_quant.sh to get quantization result files, including the XMODEL file to the compiler (./quantized/VoxelNet_int.xmodel):

```
sh ./code/test/run_quant.sh
```

Compiling the Model

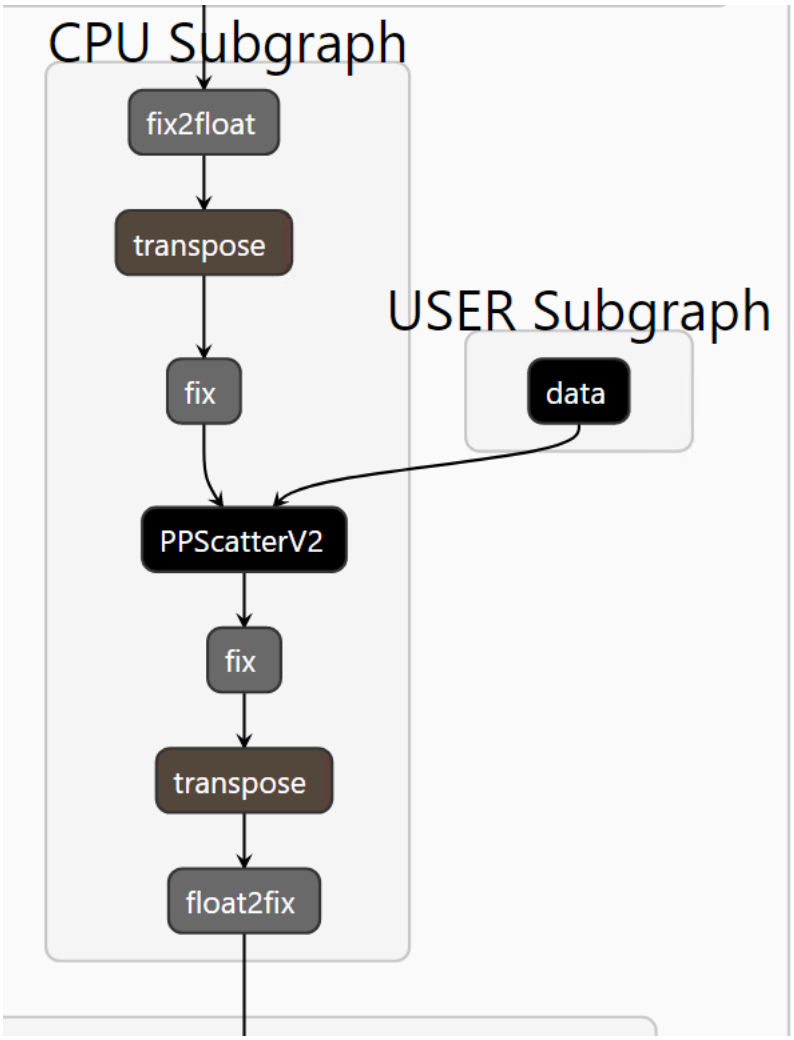
The following commands apply to PyTorch:

```
conda activate vitis-ai-pytorch
cd <path of Vitis-AI>/examples/custom_operator/pytorch_example/model/
quantized
vai_c_xir -x VoxelNet_int.xmodel -a /opt/vitis_ai/compiler/arch/DPUCZDX8G/
ZCU102/arch.json -o ./ -n pointpillars_custom_op
```

Custom OP Registration

Before custom op registration, you can use the latest Netron program to check the compiled model. From the following graph, PPScatter is assigned to the CPU. You have to implement and register `PPScatter` op.

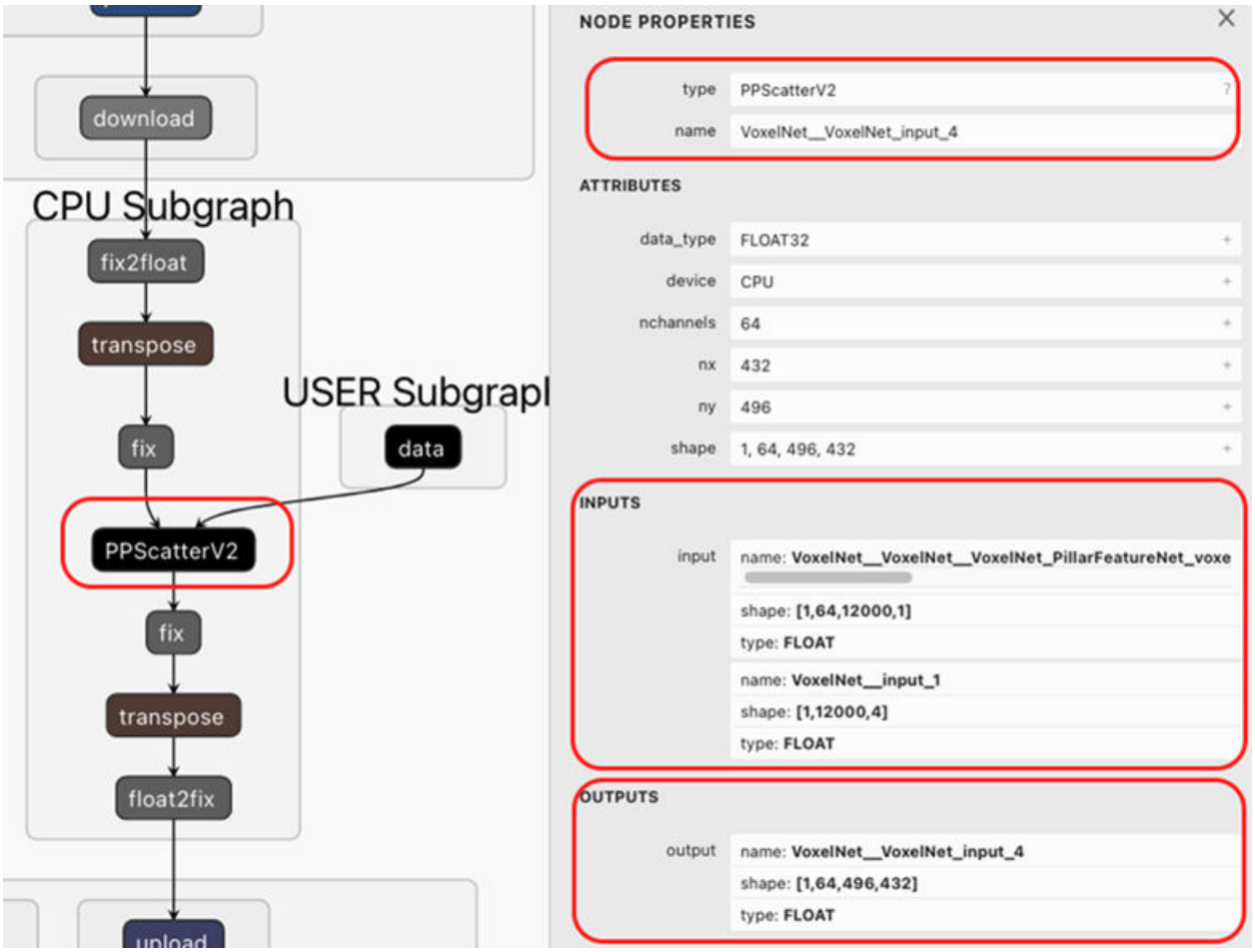
Figure 28: **PPScatter Op in CPU Subgraph**



Steps

1. Use `Netron` to open the compiled model and find the custom op in the CPU subgraph with op information.

Figure 29: **The Inputs and Outputs of PPScatter Op**



You can observe from the previous model structure image that the operation type (op type) is PPScatterV2. PPScatterV2 represents the name of the custom operation (op) to create.

To obtain detailed information about the custom op, you can use the xdputil tool. Run the following command to check the custom_layer op:

```
xdputil xmodel pointpillars_custom_op.xmodel --op
VoxelNet__VoxelNet_input_4
```

2. Write your own implementation of this op.

   Custom op registration supports both C++ and Python. The following steps show how to implement the op in C++. For the op Python implementation, refer to `Vitis-AI/examples/custom_operator/pytorch_example/op_registration/python/`

   Note: In `Vitis-AI/examples/custom_operator/op_add` you can find a 'README.md' file that comprehensively describes the procedure to implement the custom op. For guidance and instructions on implementing and registering the custom op, refer to this 'README.md' file.

   a. Create the `my_PPScatter_op.cpp` source file and place it in the new folder, `op_PPScatter`.

You can also copy one existing op and rename to your op, and then rename `my_tanh_op.cpp` to `my_PPScatter_op.cpp`.

```
cp - r Vitis-AI/src/vai_library/cpu_task/examples/op_tanh/
op_PPScatter
```

b. Create the Makefile.

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_PPScatter_op.o: my_PPScatter_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so: $(OUTPUT_DIR)/
my_PPScatter_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -
lglog -lvitis_ai_library-runner_helper -lvart-runner -lxir
```

c. Write the implementation of the op.

In the `my_PPScatter_op.cpp` file, use the constructor function to initialize any required variables. In this example, no variables need to be initialized.

Implement your custom logic in the 'calculate()' function. The primary objective of this logic is to retrieve input data from the `inputs` variable, perform the necessary calculations, and write the output data to the `output` variable.

In the calculate() function, implementation your own logic. The logic is mainly getting input data from “inputs” variable, calculating the logic, writing output data to the “output” variable.

The following is the code for `my_PPScatter_op.cpp`

```
#include <vart/op_imp.h>

class MyPPScatterOp {
  public:
  MyPPScatterOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {
  // op and attrs is not in use.
}

int calculate(vart::simple_tensor_buffer_t output,
              std::vector<vart::simple_tensor_buffer_t<float>>
inputs) {
  CHECK_EQ(inputs.size(), 2);
  auto input_data_shape = inputs[0].tensor->get_shape();
  auto input_coord_shape = inputs[1].tensor->get_shape();
  auto output_shape = output.tensor->get_shape();
  CHECK_EQ(input_data_shape.size(), 4); // 1 12000 1 64 --> 1 64
12000 1
  CHECK_EQ(input_coord_shape.size(), 3); // 1 12000 4
```

```
  CHECK_EQ(output_shape.size(), 4); // 1 496 432 64 ---> 1 64 496 432

  auto coord_numbers = input_coord_shape[1];
  auto coord_channel = input_coord_shape[2];
  CHECK_EQ(coord_numbers, input_data_shape[2]);

  auto batch = output_shape[0];
  auto height = output_shape[2];
  auto width = output_shape[3];
  auto channel = output_shape[1];
  CHECK_EQ(input_data_shape[0], batch);
  CHECK_EQ(channel, input_data_shape[1]);

  auto output_idx = 0;
  auto input_idx = 0;
  auto x_idx = 0;

  memset(output.data, 0,
output_shape[0]*output_shape[1]*output_shape[2]*output_shape[3]*sizeo
f(float));

  for (auto n = 0; n < coord_numbers; n++) {
    auto x = (int)inputs[1].data[x_idx + 3];
    auto y = (int)inputs[1].data[x_idx + 2];
    if (x < 0) break; // stop copy data when coord x == -1 .
    for(int i=0; i < channel; i++) {
    output_idx =i*height*width + y*width+x;
    input_idx = n+i*coord_numbers;
    output.data[output_idx] = inputs[0].data[ input_idx ];
    }
    x_idx += coord_channel;
  }
  return 0;
}


public:
  const xir::Op* const op;
};

DEF_XIR_op_IMP(MyPPScatterOp)
```

d. Build the library. The target directory is `$(HOME)/build/custom_op/` . You can modify the path in Makefile.

   When executing the make command using your provided Makefile, the custom-defined op library is generated in the following directory: `$(HOME)/build/custom_op/`.

   The name of the file will be 'libvart_op_imp_PPScatterV2.so'.

e. Copy the `libvart_op_imp_PPScatterV2.so` to `/usr/lib` on the target.

3. Verify the Op on the target.

   a. Use `run_op` command in `xdputil` to test the op:

```
xdputil run_op pointpillars_op.xmodel VoxelNet__VoxelNet_input_4 -r
ref -d dump
```

   Before running the command, prepare the reference inputs of the op. After you run the command, `VoxelNet__VoxelNet_input_4.bin` file is generated.

b. Compare the output with the golden file:

```
 xdputil comp_float ref/VoxelNet__VoxelNet_input_4.bin dump/
VoxelNet__VoxelNet_input_4.bin
```

If the op implementation is successful, you will see the following result:

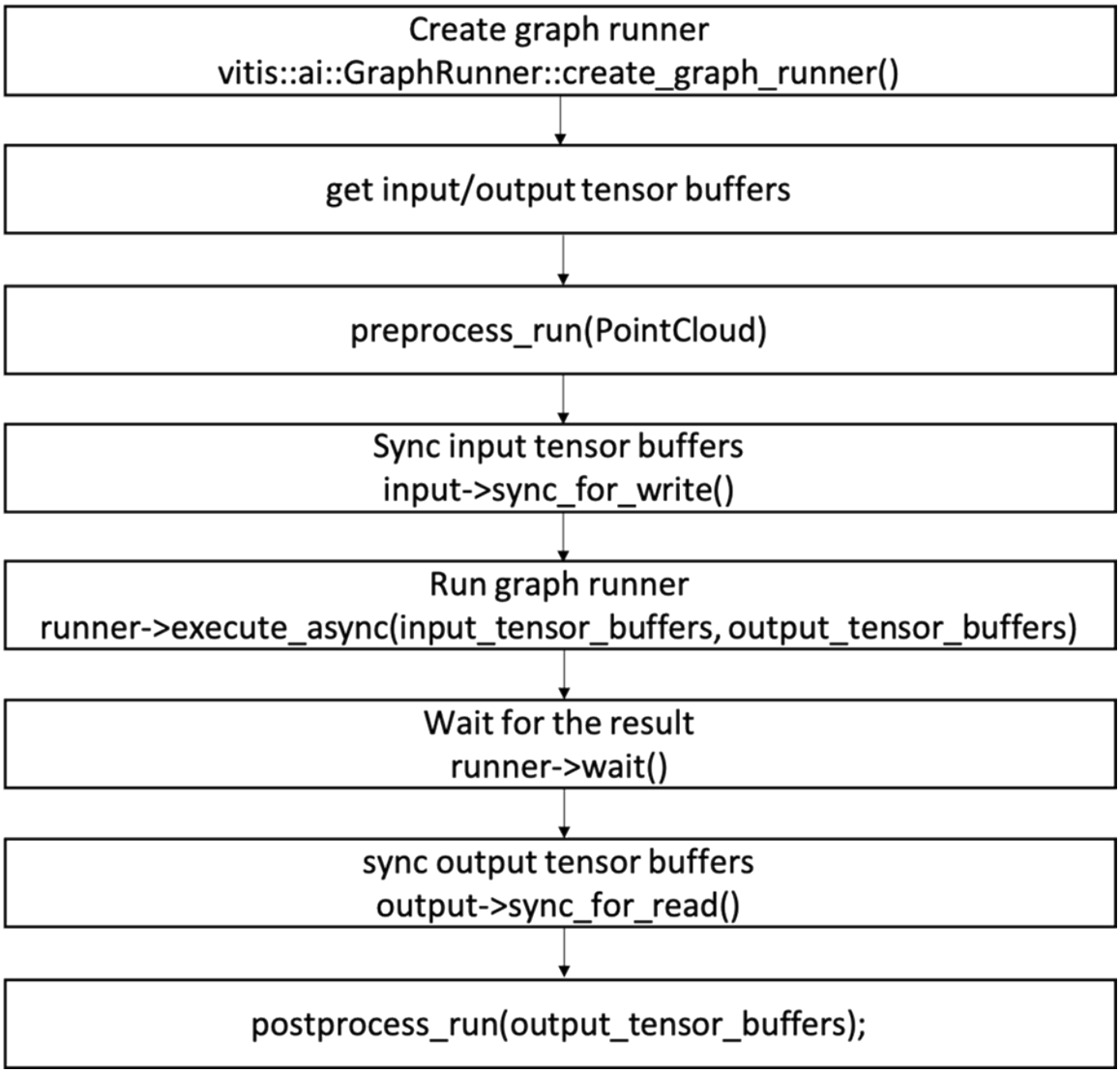
Deployment

This section describes deploying the PyTorch model with custom op in `graph_runner` APIs. `graph_runner` APIs support both C++ and Python. You can refer to the `graph_runner` samples in `Vitis-AI/examples/vai_library/samples/graph_runner`:

1. Create a new directory for your test code.
2. Create a source file and implement the following functions for this sample:
   a. Parameter parse and initialize
   b. Preprocess
   c. Model_run
   d. Postprocess

   The basic flow of this sample is shown in the following image.

Figure 30: **The basic flow of this sample**



3. Build the program.

   a. If your project is relatively simpler, consisting of a single or few .cpp source files, you can easily set up the build process by duplicating any existing build.sh script from `Vitis-AI/examples/vai_library/samples/graph_runner` and adapting it to your specific needs. After you have customized the build.sh script, execute the following command to build your program:

   ```
   cd <your sample folder>
   bash build.sh
   ```

   The following figure shows the build.sh of the resnet50_graph_runner sample.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
        OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
        OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
        -o resnet_v1_50_tf_graph_runner \
        resnet_v1_50_tf_graph_runner.cpp \
        -lglog \
        -lxir \
        -lvart-runner \
        -lvitis_ai_library-graph_runner \
        ${OPENCV_FLAGS} \
        -lopencv_core \
        -lopencv_imgcodecs \
        -lopencv_imgproc
```

b. If your project is more complex, such as this sample, using CMakeLists.txt for easy compiling is better. For more details about CMakeLists.txt, refer to `Vitis-AI/examples/custom_operator/pytorch_example/deployment/cpp/pointpillars_graph_runner/CMakeLists.txt`

   Then, run the following command to build the program:

```
cd <your sample folder>
mkdir build
cd build
cmake ..
make
```

After a successful compilation, the executable program `sample_pointpillars_graph_runner` will be generated under `<your sample folder>`

4. Test the program.

   Before you test the program, copy the XMODEL, test image, custom op library `libvart_op_imp_PPScatterV2.so` and the `sample_pointpillars_graph_runner` executable program to the board. Place the custom op library in `/usr/lib`. Then, run the following command to do the test:

```
./sample_pointpillars_graph_runner ./
pointpillars_full_customer_op.xmodel sample_pointpillars.bin
```

The following is a running result snapshot of this sample:

```
root@xilinx-zcu102-2021_2:~/pointpillars_graph_runner# ./
sample_pointpillars_graph_runner pointpillars_op.xmodel
sample_pointpillars.bin
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 05:59:20.517452 1307 tool_function.cpp:177] [UNILOG][WARNING] The
operator named VoxelNet__VoxelNet_input_4, type: PPScatterV2, is not
defined in XIR. XIR creates the definition of this operator
automatically. You should specify the shape and the data_type of the
output tensor of this operation by set_attr("shape", std::vector) and
set_attr("data_type", std::string)
result: 0
0 18.541065 3.999999 -1.732742 1.703191 4.419279 1.465484 1.679375
0.880797
0 34.522400 1.505865 -1.515198 1.503061 3.550991 1.420396 1.710625
0.851953
0 10.917599 4.705865 -1.622433 1.650789 4.350764 1.634866 1.632500
0.851953
1 21.338514 -2.400001 -1.681677 0.600000 1.963422 1.784916 4.742843
0.777300
0 57.891731 -4.188268 -1.536627 1.575194 3.780010 1.512004 2.007500
0.679179
```

If you want to profile the sample of custom op, use the `DEEPHI_PROFILING=1` environment variable:

```
env DEEPHI_PROFILING=1 ./sample_pointpillars_graph_runner ./
pointpillars_full_customer_op.xmodel sample_pointpillars.bin
```

The profiling result is as follows:

```
I1130 01:29:53.038476 15571 cpu_task.cpp:163] CPU_UPDATE_INPUT : 5us
I1130 01:29:53.038684 15571 cpu_task.cpp:166] CPU_UPDATE_OUTPUT : 55us
I1130 01:29:53.038872 15571 cpu_task.cpp:169] CPU_SYNC_FOR_READ : 46us
I1130 01:29:53.039050 15571 cpu_task.cpp:181] CPU_OP_EXEC : 32us
I1130 01:29:53.039232 15571 cpu_task.cpp:181] CPU_OP_EXEC : 36us
I1130 01:29:53.039597 15571 cpu_task.cpp:181] CPU_OP_EXEC : 232us
I1130 01:29:53.066352 15571 cpu_task.cpp:181] CPU_OP_EXEC : 26575us
I1130 01:29:53.066745 15571 cpu_task.cpp:195] CPU_SYNC_FOR_WRITE : 1us
```

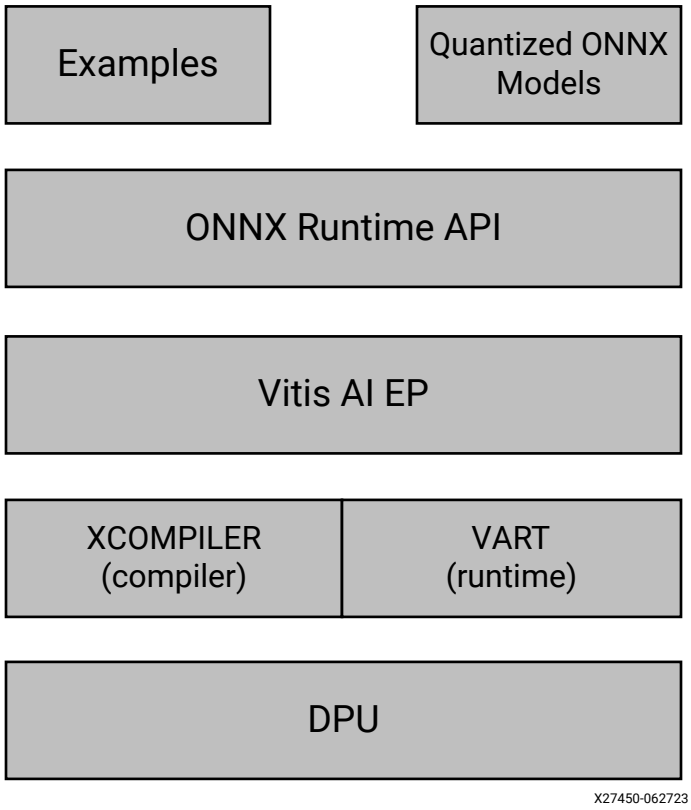
Programming with VOE

ONNX Runtime

Vitis AI Execution Provider (Vitis AI EP) offers hardware-accelerated AI inference with AMD's DPU. It enables users to run the quantized ONNX model on the target board directly. The current Vitis AI EP inside ONNX Runtime enables Neural Network model inference acceleration on embedded devices, including Zynq UltraScale+ MPSoC, Versal, Versal AI Edge and Kria cards.

Vitis AI ONNXRuntime Engine (VOE) serves as the implementation library of Vitis AI EP.

Figure 31: **VOE Overview**



Features

- Supports ONNX Opset version 18, ONNX Runtime 1.16.0 and ONNX version 1.13
- C++ and Python API (supported Python version 3)
- Supports incorporation of other execution providers, such as ACL EP, to accelerate the inference with AMD DPU in addition to the Vitis AI EP
- Supports computation on the ARM64 Cortex®-A72 cores and the supported target is VEK280 in VAI3.5

Benefits

- Versatility: You can deploy subgraphs on the AMD DPU while using other execution providers like Arm® NN and Arm® ACL for additional operators. This flexibility enables the deployment of models that might not be directly supported on target boards.
- Improved performance: By leveraging specialized execution providers such as the AMD DPU for specific operations and using other providers for the remaining operators, you can achieve optimized performance for their models.
- Expanded model support: Enhancing ONNX Runtime enables the deployment of models with operators that the DPU does not natively support. By incorporating additional execution providers, you can execute many models, including those from the ONNX model zoo.

Runtime Options

Vitis AI ONNX Runtime integrates a compiler responsible for compiling the model graph and weights into a micro-coded executable. This executable is deployed on the target accelerator.

During the initiation of the ONNX Runtime session, the model is compiled, and this compilation process must be completed before the first inference pass. The compilation duration might vary, but it could take a few minutes. After the model is compiled, the model executable is cached. For subsequent inference runs, you can use the cached executable model.

Several runtime variables can be set to configure the inference session, as listed in the following table. The `config_file` variable is not optional and must be set to point to the configuration file's location. The `cacheDir` and `cacheKey` variables are optional.

Table 33: **Runtime Variables**

Runtime Variable	Default Value	Details
config_file	""	required, the configuration file path, the configuration file vaip_config.json is contained in Vitis_ai_2023.1-r3.5.0.tar.gz
cacheDir	/tmp/{user}/vaip/.cache/	optional, cache directory
cacheKey	{onnx_model_md5}	optional, cache key, used to distinguish between different models.

The final cache directory is {cacheDir}/{cacheKey}. In addition, environment variables can be set to customize the Vitis AI EP.

Table 34: **Environment Variables**

Environment Variable	Default Value	Details
XLNX_ENABLE_CACHE	1	Whether to use the cache, if it is 0, it will ignore the cached executable, and the model will be recompiled.
XLNX_CACHE_DIR	/tmp/$USER/vaip/.cache/{onnx_model_md5}	optional, configure cache path

Install and Deploy

Vitis AI 3.5 offers over ten deployment examples based on ONNX Runtime. You can find the examples at https://github.com/Xilinx/Vitis-AI/tree/v3.5/examples/vai_library/samples_onnx. The following steps describe how to use VOE to deploy the ONNX model:

1. Prepare the quantized model in ONNX format. Use Vitis AI Quantizer to quantize the model and output the quantized model in ONNX format.

2. Download the ONNX runtime package vitis_ai_2023.1-r3.5.0.tar.gz and install it on the target board.

```
tar -xzvf vitis_ai_2023.1-r3.5.0.tar.gz -C /
```

   Then, download the voe-0.1.0-py3-none-any.whl and onnxruntime_vitisai-1.16.0-py3-none-any.whl. Ensure the device is online and install them online.

```
pip3 install voe*.whl
pip3 install onnxruntime_vitisai*.whl
```

3. Vitis AI 3.5 supports ONNX Runtime `C++` API and `Python` API. For details on ONNX Runtime API, refer to https://onnxruntime.ai/docs/api/. The following is an ONNX model deployment code snippet based on the C++ API:

   C++ example

```
// ...
#include <experimental_onnxruntime_cxx_api.h>
// include user header files
// ...

auto onnx_model_path = "resnet50_pt.onnx"
Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "resnet50_pt");
auto session_options = Ort::SessionOptions();

auto options = std::unorderd_map<std::string,std::string>({});
options["config_file"] = "/etc/vaip_config.json";
// optional, eg: cache path : /tmp/my_cache/abcdefg // Replace abcdefg
with your model name, eg. onnx_model_md5
options["cacheDir"] = "/tmp/my_cache";
options["cacheKey"] = "abcdefg"; // Replace abcdefg with your model
name, eg. onnx_model_md5

// Create an inference session using the Vitis AI execution provider
session_options.AppendExecutionProvider("VitisAI", options);

auto session = Ort::Experimental::Session(env, model_name,
session_options);

auto input_shapes = session.GetInputShapes();
// preprocess input data
// ...

// Create input tensors and populate input data
std::vector<Ort::Value> input_tensors;
input_tensors.push_back(Ort::Experimental::Value::CreateTensor<float>(
input_data.data(), input_data.size(), input_shapes[0]));

auto output_tensors = session.Run(session.GetInputNames(), input_tensors,
session.GetOutputNames());
// postprocess output data
// ...
```

To leverage the Python APIs, use the following example for reference:

```
import onnxruntime

# Add other imports
# ...

# Load inputs and do preprocessing
# ...

# Create an inference session using the Vitis-AI execution provider

session = onnxruntime.InferenceSession(
'[model_file].onnx',
providers=["VitisAIExecutionProvider"],
provider_options=[{"config_file":"/etc/vaip_config.json"}])

input_shape = session.get_inputs()[0].shape
input_name = session.get_inputs()[0].name

# Load inputs and do preprocessing by input_shape
input_data = [...]
result = session.run([], {input_name: input_data})
```

4. Create a `build.sh` file or copy one from the Vitis AI Library ONNX examples and modify it. Then, build the program:

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

lib_x=" -lglog -lunilog -lvitis_ai_library-xnnpp -lvitis_ai_library-
model_config -lprotobuf -lxrt_core -lvart-xrt-device-handle -lvaip-core -
lxcompiler-core -labsl_city -labsl_low_level_hash -lvart-dpu-controller -
lxir -lvart-util -ltarget-factory -ljson-c"
lib_onnx=" -lonnxruntime"
lib_opencv=" -lopencv_videoio -lopencv_imgcodecs -lopencv_highgui -
lopencv_imgproc -lopencv_core "

if [[ "$CXX"  == *"sysroot"* ]];then
 inc_x="-I=/usr/include/onnxruntime -I=/install/Release/include/
onnxruntime -I=/install/Release/include -I=/usr/include/xrt"
 link_x="  -L=/install/Release/lib"
else
 inc_x=" -I/usr/include/onnxruntime  -I/usr/include/xrt"
 link_x="  "
fi

name=$(basename $PWD)

CXX=${CXX:-g++}
$CXX -O2 -fno-inline -I. \
 ${inc_x} \
 ${link_x} \
 -o ${name}_onnx -std=c++17 \
```

```
$PWD/${name}_onnx.cpp \
${OPENCV_FLAGS} \
${lib_opencv} \
${lib_x} \
${lib_onnx}
```

5. Copy the executable program and the quantized ONNX model to the target. Then, run the program.

   ***Note*:** For the ONNX model deployment, the input model is the quantized ONNX model. If the environmental variable `WITH_XCOMPILER` is on, it first performs the model compiling online when you run the program. It might take some time to compile the model.

Multi-FPGA Programming

Most modern servers have multiple AMD Alveo™ cards, and you would want to take advantage of scaling up and scaling out deep-learning inference. Vitis AI provides support for multi-FPGA servers using the following building blocks.

XRM

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

- Enables multi-FPGA heterogeneous support
- C++ API and CLI for the clients to allocate, use, and release resources
- Enables resource allocation at FPGA, compute unit (CU), and service granularity
- Auto-release resource
- Multi-client support: Enables multi-client/users/processes request
- XCLBIN-to-DSA auto-association
- Resource sharing amongst clients/users
- Containerized support
- User-defined function
- Logging support

https://github.com/Xilinx/XRM

AI Kernel Scheduler

Real world deep learning applications involve multi-stage data processing pipelines, including many compute-intensive preprocessing operations like data loading from disk, decoding, resizing, color space conversion, scaling, and cropping multiple ML networks like CNN and various post-processing operations like NMS.

The AI kernel scheduler (AKS) application automatically and efficiently pipelines such graphs without much user effort. It provides different kinds of kernels for every stage of the complex graphs, which are highly configurable and plug-and-play. Some examples include preprocessing kernels like image decode and resize, CNN kernels like the Vitis AI DPU kernel and post-processing kernels like SoftMax and NMS. You can create their graphs using kernels and execute their jobs seamlessly for maximum performance.

For more details and examples, see the Vitis AI GitHub (AI Kernel Scheduler).

Using WeGO

WeGO (Whole Graph Optimizer) is a key feature of Vitis AI, providing a seamless solution for deploying TensorFlow and PyTorch models on the Versal Data Center DPU. By integrating the Vitis AI development kit with TensorFlow and PyTorch frameworks, WeGO enables efficient deployment of models on the DPU. WeGO supports TensorFlow 2.x and PyTorch 1.x, starting from Vitis AI 2.5. It is considered an in-framework inference solution within Vitis AI, setting it apart from non-framework approaches that use VART APIs or the AI Library.

WeGO automatically performs subgraph partitioning and applies optimizations and acceleration for the data center DPU-compatible subgraphs. The parts of the graph that are not supported by the DPU, known as CPU subgraphs, are dispatched to the TensorFlow or PyTorch framework for CPU native execution. WeGO handles the entire process, including whole graph optimization, compilation, and runtime subgraphs' dispatch and execution. This process remains entirely transparent to end-users, ensuring ease of use.

Using WeGO offers model designers a seamless transition from training to inference. With WeGO, deploying quantized models over a TensorFlow or PyTorch framework becomes straightforward through its Python programming interface. This interface enables maximum reuse of Python code, including pre-processing and post-processing, developed during the model training phase with TensorFlow or PyTorch. As a result, WeGO significantly enhances productivity when deploying and evaluating models over data center DPUs.

***Note*:** Currently, WeGO only supports data center DPU target DPUCV2DX8G on the V70 platform.

For WeGO examples and more information on applying TensorFlow and PyTorch to deploy models, see Vitis AI GitHub repo.

Visualization With OnBoard

OnBoard is an experimental tool introduced in the Vitis AI 3.5. release. Its main objective is to enhance the analysis and debugging process for WeGO users by visualizing the models' inference flow. Built as an extension of the TensorBoard web server, OnBoard writes events with simple function calls. It can visualize the images of inference results, models' graph structure before and after WeGO transformation, and DPU platform-related information. As of Vitis AI 3.5, OnBoard is available for PyTorch and TensorFlow 2.

For OnBoard examples and more information, see Vitis AI GitHub .

WeGO Programming Interface

PyTorch

WeGO-Torch, a sub-project of WeGO, aims to enhance Vitis AI EoU by integrating the Vitis AI toolchain into the PyTorch framework. Adhering to the standard WeGO workflow, WeGO-Torch facilitates automatic model partitioning, compilation, and inference, eliminating manual intervention.

The WeGO-Torch Python/C++ API is designed to receive a quantized TorchScript module, generated through Vitis AI PyTorch quantizer, as input. It then produces an optimized TorchScript module that can be readily used for immediate inference. Leverage WeGO-Torch for accelerating Vitis AI within PyTorch involves these general steps:

1. Import WeGO-Torch Python module into your application.
2. Load the quantized TorchScript module using standard PyTorch Python API.
3. Compile the quantized TorchScript module using the WeGO-Torch module API by providing the TorchScript module and input shape as inputs. It returns an optimized TorchScript module as the compiled result.
4. Run inference using the optimized TorchScript module.

The following code snippets show the basic usage of Python APIs of WeGO-Torch:

```
import torch
# Step 1: Import WeGO-Torch python module
import wego_torch

# Step 2: Load the quantized torchscript module generated by vitis-ai
PyTorch quantizer.
model_path = <quantized_torchscript_model_path>
mod = torch.jit.load(model_path)

# Step 3: Create an optimized TorchScript module through WeGO-Torch's API.
wego_mod = wego_torch.compile(mod,
  wego_torch.CompileOptions(
  inputs_meta = [ wego_torch.InputMeta(torch.float, [1, 3, 224, 224]) ]
```

```
    )
)

# Step 4: Run inference using the optmized TorchScript module.
result = wego_mod(input)
```

Using the C++ APIs in WeGO-Torch to compile a model is a similar process. Refer to Core C++ Classes and Core C++ APIs for detailed information on using C++ APIs in WeGO-Torch to compile and run a model on DPU.

WeGO-Torch C++ Classes and APIs

Core C++ Classes

wego_torch::core::CompileOptions

A C++ class object that specifies the compilation options for WeGO-Torch. You need to create an instance of this class and pass it as an argument to the `wego_torch::core::Compile` function.

Table 35: **Constructor Parameters**

Type	Name	Description
wego_torch::AccuracyMode	accuracy_mode	Determines the accuracy mode. It has two different types: • *wego_torch::AccuracyMode::kDefaultRemoveFixNeuron*: In this mode, WeGO-Torch optimizes performance by eliminating redundant fixneurons during the compilation process. These fixneurons can exist due to quantized operators, but they are not supported by the DPU target onboard, causing them to be dispatched to the CPU for inference. By removing these fixneurons from the model, the end-to-end performance can be enhanced, provided that the accuracy meets the requirements • *wego_torch::AccuracyMode::kReserveFixNeuron*: If this value is provided, WeGO-Torch retains all redundant fixneurons in the model instead of removing them. While removing these fixneurons enhances performance, there is a chance of accuracy issues in certain cases. It is recommended to experiment with this value if the end-to-end accuracy falls short of the requirement after compiling the model with WeGO-Torch.

Table 35: **Constructor Parameters** *(cont'd)*

Type	Name	Description
wego_torch::core::PartitionOptions	partition_options	Sets partition options. See *wego_torch::core::PartitionOptions* class for more detail.
std::vector<InputMeta>	inputs_meta	A vector of *wego_torch::InputMeta* for each input of the model.
uint32_t	thread_parallel	Parameter to optimize performance.
uint32_t	core_parallel	Parameter to optimize performance.
wego_torch::core::DebugOptions	debug_options	Sets debug options. See *wego_torch::core::DebugOptions* class for more detail.

wego_torch::core::PartitionOptions

Options for WeGO partiton configuration.

Table 36: **Constructor Parameters**

Type	Name	Description
uint32_t	wego_subgraph_min_ops_number	Currently, WeGO uses a greedy method to allocate operators to the DPU as long as they are compatible. However, this approach can give rise to the following issues: • If the DPU does not support many operators, the model might end up being partitioned into numerous DPU subgraphs and CPU subgraphs. When each DPU subgraph contains only a small number of operators, executing these subgraphs on the DPU might lead to performance problems due to frequent memory transfers between the host and the device. • WeGO assigns a device buffer for each DPU subgraph. In cases where the model is large and there are multiple DPU subgraphs after partitioning, there is a possibility of buffer overflow issues. To address this, the following option is added: *wego_subgraph_min_ops_number*. It sets a minimum number of operators for a DPU subgraph to be executed on DPU. Otherwise, it is executed on the CPU even if DPU supports all operators. ***Note***: *wego_subgraph_min_ops_number = 0* means no limit.

Table 36: **Constructor Parameters** *(cont'd)*

Type	Name	Description
std::vector<std::string>	extra_accel_op_list	DPU can run various DL operators with some constraints (For example, DPUCVDX8H_ISA1_F2W4_4PE only supports convolution with kernel 1-16 and stride 1-4). WeGO uses a DPU limitation check engine to partition operators based on their DPU compatibility. However, some operators have complex compatibility rules that might cause too much overhead. WeGO does not dispatch them to DPU by default, but lets you specify the operators you want to accelerate in the *extra_accel_op_list*. The following operators can be added to the *extra_accel_op_list* for DPU execution: • aten::mul • aten::mean • aten::linear • aten::unsqueeze • aten::slice ***Note*:** When WeGO encounters errors while compiling the operator(s) listed in the extra_accel_op_list, it indicates that DPU cannot accelerate this specific operator (s). However, if no errors are reported, these operator(s) can indeed be accelerated successfully.

wego_torch::core::DebugOptions

Option for WeGO debugging.

Table 37: **Constructor Parameters**

Type	Name	Description
bool	accuracy_debug	To enable the dumping of input and output values for subgraphs with accuracy issues, set the value to *true*. By doing so, the inputs and outputs of these subgraphs are logged. The default value is false.

wego_torch::InputMeta

Meta Information for describing inputs of the quantized model. Due to the limitations of Vitis AI toolchain, WeGO-Torch only supports compilation with static type and shape. You must explicitly pass each input's date type and shape information to enable WeGO-Torch for type and shape inference.

Table 38: **Constructor Parameters**

Type	Name	Description
wego_torch::DataType	type_	Data type of the current input tensor. It can be:*wego_torch::DataType::kBool*, *wego_torch::DataType::kInt32* or *wego_torch::DataType::kFloat32*.
wego_torch::ShapeType	input_shape_	Input shape of the current input tensor.

wego_torch::TargetInfo

A C++ class object that serves as a wrapper for DPU target information, providing access to the batch, name, fingerprint, and fingerprint-driven information of the DPU target on-board.

Table 39: **Constructor Parameters**

Type	Name	Description
std::string	name	Name of the DPU target
uint64_t	fingerprint	Fingerprint of the DPU target on-board
bool	is_fingerprint_driven	Indicates whether DPU subgraphs are compiled based on fingerprint or target name.
uint32_t	batch	Batch size supported by the DPU target on-board

Core C++ APIs

wego_torch::core::Compile

Prototype

```
torch::jit::Module Compile(const torch::jit::Module &module,
                           CompileOptions options);
```

Table 40: **Parameters**

Type	Name	Description
const torch::jit::Module &	module	A quantized PyTorch module to be compiled, represented as a *torch::jit::Module* object.
CompileOptions	options	Compiler options used for WeGO-Torch compilation, specified as the *wego_torch::core::CompileOptions* data type. For further details on this object class, refer to the core classes section.

Return

An optimized *torch::jit::Module* object.

wego_torch::core::GetTargetInfo

Prototype

```
TargetInfo GetTargetInfo();
```

Parameters

None

Return

A *wego_torch::TargetInfo* object. See the classes section for more details about this object class type.

wego_torch::getVersionInfo

Prototype

```
std::string getVersionInfo();
```

Parameters

None

Return

A raw string representing wego_torch version information.

WeGO-Torch Python Classes and APIs

Core Python Classes

wego_torch.CompileOptions(accuracy_mode : wego_torch.AccuracyMode = wego_torch.AccuracyMode.Default, inputs_meta = [], partition_options : wego_torch.PartitionOptions = None)

A python class object representing WeGO-Torch compilation options. It is created and passed into wego_torch.compile interface by the users.

Table 41: **Constructor Parameters**

Parameter	Description	Values
accuracy_mode	Decides the accuracy mode.	• *wego_torch.AccuracyMode.Default*: This is the default value for accuracy mode. In this mode, WeGO-Torch removes all redundant fixneurons to enhance performance after compilation. The redundant fixneurons exist because even some operators are quantized. Because the DPU target on board does not support them, they are dispatched to the CPU for inference. These fixneurons can be removed from the model to improve the end-to-end performance if the accuracy can meet the required criteria. • *wego_torch.AccuracyMode.ReserveReduantFixNeurons*: If this value is provided, WeGO-Torch keeps all the redundant fixneurons in the model rather than removing them. Although removing redundant fixneurons improves performance, accuracy issues are possible in some cases. You are encouraged to try this value if the end-to-end accuracy cannot meet the requirement after WeGO-Torch compiles your modelWeGO-Torch.
inputs_meta	A list of `wego_torch.InputMeta` for each input of the model.	See the following section for more details about wego_torch.InputMeta type.
partition_options	Partition options with type `PartitionOptions`.	See the following section for more details about it.

wego_torch.InputMeta (dtype = None, input_shape = [])

Meta Information for describing inputs of the quantized model. Due to the limitations of the Vitis AI toolchain, WeGO-Torch only supports compilation with static type and shape. You must explicitly pass each input's date type and shape information to enable WeGO-Torch for type and shape inference.

Table 42: **Constructor Parameters**

Parameter	Description
dtype	Data type of the current input tensor. It can be torch.int32, torch.float, or torch.bool.
input_shape	Input shape of the current input tensor.

wego_torch.PartitionOptions (wego_subgraph_min_ops_number = 0, extra_accel_op_list = [])

Options for WeGO partiton configuration.

Table 43: **Constructor Parameters**

Parameter	Descriptions
wego_subgraph_min_ops_number	Currently, WeGO uses greedy method to dispatch operators into DPU as long as they are supported by DPU. It might cause the following issues: • If there are a lot of operators not supported by DPU, the whole model might be partitioned into many DPU subgraphs and CPU subgraphs. Suppose each DPU subgraph only contains a minor number of operators. In that case, the subgraph only contains a minor number of operators. Dispatching these subgraphs into DPU for execution might lead to performance issues due to frequent memory transfer between the host and device. • WeGO allocates a device buffer for each DPU subgraph. There might be a buffer overflow issue when the model is large and there are many DPU subgraphs after partition. Added the option `wego_subgraph_min_ops_number` in WeGO to control dispatching a DPU subgraph into the DPU for execution. Suppose the number of operators in a DPU subgraph is below or equal to the wego_subgraph_min_ops_number threshold. In that case, the subgraph is below or equal to the `wego_subgraph_min_ops_number` threshold. WeGO will dispatch the subgraph to the CPU side for execution even if DPU can support all operators in the subgraph. ***Note***: If `wego_subgraph_min_ops_number` is 0, then there are no limitations.
extra_accel_op_list	DPU can support diverse DL operators but with some limitations (For example. DPUCVDX8H_ISA1_F2W4_4PE can only support convolution with kernel 1-16 and stride 1-4). WeGO leverages a DPU limitation check engine to decide an operator can be supported by DPU or not when performing a partition. But for some operators, the rules to decide whether they are supported are complicated. To avoid introducing too much overhead, WeGO does not dispatch them into DPU by default but relies on users explicitly specifying the operator type desiring for acceleration in the `extra_accel_op_list`. Currently, the following operators can be specified through `extra_accel_op_list` for DPU execution: • aten::mul • aten::mean • aten::linear ***Note***: If errors occur during the compilation in WeGO after specifying the operator(s) in the `extra_accel_op_list` list, it indicates that the DPU cannot accelerate the supplied operator(s). Otherwise, they can be.

wego_torch.TargetInfo()

A Python class object designed to wrap DPU target information, providing access to details such as batch size, name, and fingerprint of the DPU target on-board.

***Note*:** You are recommended not to create this object manually. Instead, you should rely on the wego_torch.get_target_info() API, which returns the pre-configured wego_torch.TargetInfo object containing various property fields:

1. Batch: batch size supported by the DPU target on-board.
2. Name: name of the DPU target.
3. Fingerprint: fingerprint of the DPU target on-board.

The following is a general approach to use wego_torch.TargetInfo:

```
import wego_torch
...
# Detect the DPU target on-board and return an object with type
wego_torch.TargetInfo.
target_info = wego_torch.get_target_info()
# The target_info object can be printed directly.
print(target_info)
# Retrieve diverse propery fields of the DPU target.
batch, name, fingerprint = target_info.batch, target_info.name,
target_info.fingerprint
...
```

Core Python APIs

Table 44: **wego_torch.compile(module: Any, options: wego_torch.CompileOptions)**

Description	Parameters	Return
Compiles a pytorch torchscript module for Vitis AI acceleration.	• module: A quantized PyTorch module with type torch.jit.ScriptModule to be compiled. • options: Compiler options for WeGO-Torch compilation purpose, with type wego_torch.CompileOptions. See the core classes section for more details.	An optimized TorchScript module.

Table 45: **wego_torch.get_target_info()**

Description	Parameters	Return
Detects DPU target on-board and return the target information.	None	A wego_torch.TargetInfo object. See the core classes section for more details.

Table 46: **wego_torch.version()**

Description	Parameters	Return
Get WeGO-Torch version.	None	A raw string representing wego_torch version information.

WeGO-PyTorch Limitations

The WeGO-Torch project is currently in an early access state and might encounter a few known usage issues. Here are the details of these issues and the necessary steps to address them:

1. WeGO-Torch cannot support RCNN models (with control-flow) because:
    a. RCNN Model Support: WeGO-Torch does not currently support RCNN models with control-flow due to dynamic shape problems. In RCNN models, the shape of tensors can change during runtime when different images are provided as inputs. This poses challenges for deployment in WeGO. To make RCNN models compatible with WeGO, manual modifications are required to eliminate this constraint.
    b. Input Type Compatibility: RCNN models often accept `Tensor[]` as an input type, which WeGO's compile API does not support. Additionally, using `Tensor []` as the input type implies that the float model itself is batch-sensitive, and the quantized models obtained through tracing differ based on the batch size used during the TorchScript tracing phase. To deploy these models in WeGO, the following steps are recommended:
        i. Replace `Tensor []` with `Tensor` or `Tensor, Tensor, ...` (when the number of inputs is known) as the input type in the original float model.
        ii. The batch size used for inference in WeGO must be the same as the one used in the export phase during quantization.
2. WeGO-Torch currently covers only a subset of operators that data center DPUs can support. Consequently, certain operators might be dispatched to the CPU for execution, even if data center DPUs can support them.

Examples

For WeGO-Torch examples, see the Vitis AI GitHub page.

TensorFlow 2.x

WeGO-TensorFlow2.x, a sub-project of WeGO, is designed to enhance Vitis AI EoU by seamlessly integrating the Vitis AI toolchain into the TensorFlow 2.x framework. For VAI 2.5, TensorFlow v2.8.0 is the supported version. To use WeGO-TensorFlow2.x, you should provide a quantized model in HDF5 format, usually named quantized.h5, which can be generated using the vai_q_tensorflow2 quantizer.

The core API of WeGO, create_wego_model(), automatically converts the quantized Keras model into a new concrete function. This function transforms data center DPU-compatible subgraphs into TensorFlow operators with the VaiWeGOOp kind.

The entire process of WeGO-TensorFlow2.x inference can be abstracted into the following four steps:

1. Import the WeGO TensorFlow2.x Python module into the application.
2. Obtain the batch information of the DPU target using vitis_vai.get_target_info() for the inputs batching process.
3. Create the WeGO model using vitis_vai.create_wego_model() to obtain the concrete function.
4. Execute the concrete function.

WeGO-TensorFlow2.x Python APIs

Core Python Classes

DeviceInfo()

An object that wraps DPU target information.

***Note*:** You should not create this object on your own but should rely on the API Vitis_vai.get_target_info() to return this object, which will contain diverse property fields:

- batch: batch size supported by the DPU target on-board.
- target: name of the DPU target.
- fingerprint: the fingerprint of the DPU target on-board.

The general way to use DeviceInfo is as follows:

```
from tensorflow.compiler import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

Core Python APIs

Table 47: **get_target_info()**

Description	Parameters	Return
Gets target information including batch, fingerprint, and target name. You can use info for batching or get target name information.	None	A DeviceInfo object. For more details about this object type, see the core classes section.

Table 48: **create_wego_model(input_h5, feed_dict={}, accuracy_mode= vitis_vai.enums.AccuracyMode.Default)**

Description	Parameters	Return
Creates WeGO model, convert Keras h5 file into concrete function	1. input_h5: Path to the h5 file. 2. feed_dict: Infer shape configuration when input model without fixed input shape. 3. accuracy_mode: • vitis_vai.enums.AccuracyMode.Default: Inference without CPU FixNeuron. • vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons: Inference with CPU FixNeruon	New concrete function with VaiWeGOOps. ***Note*:** WeGO eliminates CPU FixNeurons operators within a quantized model to achieve optimal performance by default. However, for those models containing many CPU FixNeurons operators, their accuracy may decrease by deploying them with default values (vitis_vai.enums.AccuracyMode.Default) .In such cases, you can switch to vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons to achieve better accuracy.

Environment Variable

WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE

This environment variable can be enabled when some operators need to rely on batchsize to infer static shapes. Export WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1 will set the batch size to 1. For example, the static shape of the reshape operator cannot be obtained for some models (For example, ssd_resnet_50_fpn_coco_tf model with input shape [-1,640,640,3]), resulting in an error when some WeGO subgraph is compiled by Vitis AI toolchain. Here is the error message:

```
AssertionError: [ERROR] Invalid shape of input layer: shape: [1, -1, -1,
256] (N,H,W,C), name: input1
[INFO] parse raw model     :  0%|          | 0/52 [00:00<?, ?it/
s]
*** Check failure stack trace: ***
```

To solve this problem, you must set the environment variables as follows before running the sample to enable WeGO aggressive shape inference:

```
export WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1
```

Otherwise, you do not need this environment variable. Or, cancel the environment variable that the following command has set:

```
unset WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE
```

Examples

For WeGO-TensorFlow 2.x samples, see Vitis AI GitHub page.

TensorFlow 1.x

WeGO-TensorFlow1.x is a sub-project of WeGO, which is designed to improveVitis AI EoU by integrating the Vitis AI toolchain into TensorFlow 1.x framework. Vitis AI 2.5 supports TensorFlow v1.15. The input for WeGO-TensorFlow1.x is the quantized model usually named as quantize_eval_model.pb, which is generated by vai_q_tensorflow. The core WeGO API create_wego_graph() automatically converts the quantized graph into a new TensorFlow graph called as WeGO graph, where the data center DPU compatible subgraphs are transformed into TensorFlow operator with the kind of VaiWeGOOp.

The whole WeGO-TensorFlow1.x inference can be abstracted into the following for steps

1. Execute some graph-level optimizations on the original graph to meet DPU-specific requirements.
2. Traverse the whole graph of the input quantized model and detect nodes which are supported by data center DPU.
3. Perform graph auto-partitioning over the quantized graph over the node list detected in step 2.
4. Transform all data center DPU compatible subgraphs into new TensorFlow nodes with kind of VaiWeGOOp within the input quantized model.
5. Return the optimized new WeGO graph and then invoke TensorFlow sess.run() to execute the whole graph.

WeGO-TensorFlow 1.x Python APIs

Core Python Classes

DeviceInfo()

An object that wraps DPU target information.

***Note*:** You should not create this object by yourself but should rely on the API vitis_vai.get_target_info() to return this object with diverse property fields:

- batch: batch size supported by the DPU target on-board.
- target: name of the DPU target.
- fingerprint: the fingerprint of the DPU target on-board.

The general way to use DeviceInfo is as follows:

```
from tensorflow.contrib import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

Core Python APIs

Table 49: **get_target_info()**

Description	Parameters	Return
Gets target information including batch, fingerprint, and target name. You can use info for batching or get target name information.	None	A DeviceInfo object. For more details about this object type, see the core classes section.

Table 50: **create_wego_graph(input_graph_def, feed_dict={}, accuracy_mode= vitis_vai.enums.AccuracyMode.Default)**

Description	Parameters	Return
Python wrapper for the VAI transformation.	1. input_graph_def: GraphDef object containing a model to be transformed. 2. feed_dict: Infer shape configuration when input model without fixed input shape. 3. accuracy_mode: • vitis_vai.enums.AccuracyMode.Default: Running without CPU FixNeuron. • vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons: Running with CPU FixNeruon	New GraphDef with VaiWeGOOps placed in graph replacing subgraphs. ***Note***: WeGO eliminates CPU FixNeurons operators within a quantized model to achieve optimal performance by default. However, for those models containing many CPU FixNeurons operators, their accuracy might decrease by deploying them with the default value(Vitis_vai.enums.AccuracyMode.Default).In such cases, you can switch to Vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons to achieve better accuracy.

Environment Variable

WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE

This environment variable is used by WeGO TensorFlow 1.x and WeGO TensorFlow 2.x. Refer to WeGO TensorFlow 2.x section for its usage.

Examples

For WeGO-TensorFlow 1.x samples, see Vitis AI GitHub page.

On-the-fly Quantization in WeGO

In the original WeGO workflow, because WeGO only accepts a quantized INT8 model as input, it is necessary to perform a separate quantization process initially. It can be achieved by explicitly using the Vitis AI quantizer, which converts the float32 model into an INT8 model. It creates the need to perform extra tasks for the users, such as performing conda environment switch operations between quantizer and WeGO and figuring out the relationship between Vitis AI quantizer and WeGO. To improve the ease of use and make the entire process from quantization to deployment smoother, WeGO integrates the Vitis AI quantizer into its flow, enabling on-the-fly quantization when a float32 model is offered as WeGO's input. Besides the original WeGO API for compilation, a new API is introduced in WeGO for quantization purposes, and the quantizer details are transparent to the end users. The quantization integration in WeGO is in the early stage with the following limitations:

1. The integration flow currently supports Only PTQ (Post-Training Quantization). If the model's accuracy is significantly lower than expected, fine-tuning or QAT (Quantization Aware Training) must be used to improve the accuracy by following the native Vitis AI quantization flow.
2. Only CPUs are adopted for quantization in WeGO; currently, GPUs are not supported. This might introduce some issues when quantizing large models, as the process may consume much time.

Quantization APIs

This section introduces WeGO's API for PTQ quantization targeting different frameworks.

PyTorch

Quantization API

```
wego_torch.quantize(
module: torch.nn.Module,
input_shapes: Sequence[Sequence],
dataloader: Iterable,
calibrator: Callable[[torch.nn.Module, Any, int, torch.device], None],
export_dataloader: Iterable = None,
device: torch.device = torch.device("cpu"),
output_dir: str = "quantize_result",
bitwidth: int = None,
quant_config_file: Optional[str] = None,
*args, **kwargs) -> torch.jit.ScriptModule
```

This function quantizes a torch float model with Post-Training Quantization (PTQ) method, and a quantized TorchScript Module is returned for WeGO compilation usage.

If PTQ cannot achieve the required accuracy, you might need to consider using Quantization Aware Training (QAT) with Vitis AI Quantizer API. For an in-depth understanding of the quantization process, see Quantizing the Model in the Vitis AI User Guide.

Parameters

- **module:** `(torch.nn.Module)` The input PyTorch float model.
- **input_shapes:** `(Sequence[Sequence]` Input shapes for the model- a sequence of lists or tuples.
- **dataloader:** `(Iterable)` Dataloader for calibration dataset. It must be iterable. API iterates through it and passes the returned values to the calibrator.
- **calibrator:** `(Callable)` Callable object to perform batch data pre-processing and forwarding. Get batch data from the dataloader, pre-process it if necessary, and use the module to forward it. This calibrator is called N + 1 times in the calibration and export stages.
  - Stage 1 is for calibration. In this stage, your dataloader is iterated, and the data is passed through the module to collect quantization statistics. The calibrator is called N times(N = len(dataloader)). At stage 1, if you did not pass the optional export_dataloader, the first batch returned by dataloader is saved and later used in stage 2. In this case, ensure the first batch is unchanged by calibrator or iteration side effects.
  - Stage 2 is for exporting the quantized TorchScript module. In this stage, the calibrator is only called once with one batch of data. If you pass in an export_dataloader, this export_dataloader is iterated, and only the first batch is used. The program breaks out of iteration after processing the first batch. If you did not pass in an export_dataloader, the saved first batch from stage 1 is used.

    Calibrator arguments:

    - **module:** `(torch.nn.Module)` Module for quantization. This is a modified version of the module you passed in, with the necessary mechanisms to collect data statistics. You should use this module instead of the original float model to forward your data.
    - **batch_data:** `(Any)` Batch data returned from dataloader.
    - **batch_index:** `(int)` Index of the batch. Use the device if necessary (torch.device) for forwarding. Currently only supports CPU.

    ***Note*:** Extra positional and keyword arguments to quantize API will be forwarded to the calibrator. For more information, see Quantizing the Model.
- **export_dataloader:** `(Iterable)` An optional dataloader for the export stage. The default value is None. If None, it uses the first batch saved from stage 1.
- **device:** `(torch.device)` Device to use for calibration. Currently only supports CPU.
- **output_dir:** `(str)` A temporary working directory. The default value is quantize_result. Some intermediary files are saved here.
- **bitwidth:** `(int)` Global quantization bit width. The default value is 8.
- **quant_config_file:** `(str)` Path to the quantizer configuration file. The default value is None.

- **args:** Extra positional arguments to pass to the calibrator.
- **kwargs:** Extra keyword arguments to pass to the calibrator.

For more information on how to use on-the-fly quantization in WeGO, see WeGO examples .

TensorFlow 2.x

Quantization API

```
vitis_vai.quantize(
input_float,
quantize_strategy = 'pof2s',
custom_quantize_strategy = None,
calib_dataset = None,
calib_steps = None,
calib_batch_size = None,
save_path = './vai_wego/quantized.h5',
verbose = 0,
add_shape_info = False,
dump = False,
dump_output_dir = './vai_dump/')
```

This function performs the float model's post-training quantization (PTQ), including model optimization, weight quantization, and activation post-training quantization.

Parameters

- **input_float:** A `tf.keras.Model` float object to be quantized.
- **quantize_strategy:** A string object of the quantize strategy type. Available values are *pof2s*, *pof2s_tqt*, *fs* , and *fsx*. *pof2s* is the default strategy that uses a power-of-2 scale quantizer and the Straight-Through-Estimator. *pof2s_tqt* is a strategy introduced in Vitis AI 1.4 that uses Trained-Threshold in power-of-2 scale quantizers and can yield better QAT results. *fs* is a quantize strategy introduced in Vitis AI 2.5 that does float scale quantization for inputs and weights of Conv2D, DepthwiseConv2D, Conv2DTranspose, and Dense layers. On the other hand, *fsx* extends the quantize strategy for more layer types than *the fs* quantize strategy, including Add, MaxPooling2D, and AveragePooling2D, and also includes biases and activations in quantization.

  ***Note*:**

  - *pof2s_tqt* strategy should only be used in QAT with init_quant=True for the best performance.
  - *fs* and *fsx* strategies are designed for target devices with floating-point supports. DPU does not have floating-point support now, so models quantized with these quantize strategies cannot be deployed to them.
- **custom_quantize_strategy:** A *string* object. The file path of the custom quantize strategy JSON file.

- **calib_dataset:** A *tf.data.Dataset*, *keras.utils.Sequence*, or *np.numpy* object. The representative dataset for calibration. You can use the whole or part of eval_dataset, train_dataset, or other datasets as calib_dataset.
- **calib_steps:** An *int* object. The total number of steps for calibration. Ignored with the default value of None. If *calib_dataset* is a *tf.data* dataset, generator, or *keras.utils.Sequence* instance and steps are None, calibration runs until the dataset is exhausted. Array inputs do not support this argument.
- **calib_batch_size:** An *int* object. The number of samples per batch for calibration. If the *calib_dataset* is in the form of a dataset, generator, or *keras.utils.Sequence* instances, the batch size is controlled by the dataset itself. If the *calib_dataset* is in the form of a numpy.array object, the default batch size is set to 32.
- **save_path:** A *string* object. The directory to save the quantized model.
- **verbose:** An *int* object. The verbosity of the logging. Greater verbose value generates more detailed logging. The default value is 0.
- **add_shape_info:** A *bool* object. Determines whether to add shape inference information for custom layers. It must be set to True for models with custom layers.
- **dump:** A *flag* to enable/disable dump. If *dump=False*, dump is disabled, and if *dump=True*, dump is enabled.
- **dump_output_dir:** A *string* object. The directory to save the dump results.

For more information on how to use on-the-fly quantization in WeGO TensorFlow 2.x, see WeGO examples.

TensorFlow 1.x

Quantization API

```
def quantize(
input_frozen_graph = "",
input_nodes = "",
input_shapes = "",
output_nodes = "",
input_fn = "",
method = 1,
calib_iter = 100,
output_dir = "./quantize_results",
**kargs)
```

This function invokes the `vai_q_tensorflow` command tool in WeGO TensorFlow r1.15 and converts the input floating-point model to a fixed-point model for DPU deployment acceleration. To be fully compatible with the native `vai_q_tensorflow` quantizer, all parameters received from this API are forwarded to `vai_q_tensorflow` command tool directly. This function returns a quantized `GraphDef` object or `None` on failure.

Note: Only PTQ is supported now for on-the-fly quantization in WeGO. For more information on fast fine-tuning and QAT quantization, see vai_q_tensorflow Quantization Aware Training.

Parameters

- **input_frozen_graph:** string. Path to input frozen graph(.pb) (default:)
- **input_nodes:** string: The comma-separated name list of input nodes of the subgraph to be quantized and used together with output_nodes. Only the subgraph between input_nodes and output_nodes is included when generating the deployment model. Set it to the beginning of the main body of the model to quantize, such as the nodes after data pre-processing and augmentation. (default:)
- **input_shapes:** string. The comma-separated shape list of input_nodes. The shape must be a 4-dimension shape for each node, separated by commas, for example, `1,224,224,3`; Unknown size for batch size is supported, 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` (default:)
- **output_nodes:** string: The comma-separated name list of output nodes of the subgraph to be quantized that is used together with input_nodes. Only the subgraph between input_nodes and output_nodes is included when generating the deployment model. Set it to the end of the main body of the model to quantize, such as the nodes, before post-processing. (default:)
- **input_fn:** string: The Python importable function that provides the input data. The format is `module_name.input_fn_name`, for example, `my_input_fn.input_fn`. The `input_fn` should take an `int` object as input indicating the calibration step and return a dict (`placeholder_node_name : numpy.Array`) object for each call, which will be fed into the model's placeholder nodes. (default:)
- **method:** int32: {0,1,2}, default: 1. The quantization method, options are:
  - 0: non-overflow method. Ensures no values are saturated during quantization. It might cause inaccurate results
  - 1: min-diffs method. Enables saturation for large values during quantization to get smaller quantization errors. This method is slower than method 0 but has higher endurance to outliers.
  - 2: min-diffs method with a strategy for depthwise. Enables saturation for large values during quantization to get smaller quantization errors. Apply a special strategy for depthwise weights, but implement method 1 to standard weights and activation. This method is slower than method 0 but has higher endurance to outliers.
- **calib_iter:** int32. The iterations of calibration. The total number of images for calibration = calib_iter * batch_size (default: 100)
- **output_dir:** string. The directory to save the quantization results (default: ./quantize_results).

Note: For more information on other parameters for `**kargs`, see vai_q_tensorflow Usage.

***Note*:** For more information on the on-the-fly quantization examples for WeGO TensorFlow 1.x, see examples.

Optimize Performance with AMD ZenDNN

ZenDNN library, which includes APIs for basic neural network building blocks optimized for AMD CPU architecture, targets deep learning application and framework developers to improve deep learning inference performance on AMD CPUs. To improve the performance of DPU-unsupported operators, CPUs, especially AMD CPUs, ZenDNN is integrated into WeGO flow.

***Note*:** This is an experimental feature. The performance gain using ZenDNN in WeGO is not guaranteed currently. Enable/disable ZenDNN per your requirement.

ZenDNN in WeGO PyTorch

Enable ZenDNN in WeGO PyTorch

The ZenDNN is initially disabled, but you have the option to enable it through the WeGO-Torch's compile API:

```
wego_mod = wego_torch.compile(mod, wego_torch.CompileOptions(
...
optimize_options = wego_torch.OptimizeOptions(zendnn_enable = True))
)
```

After ZenDNN is enabled, the CPU operators (the operators not supported by DPU) in the compiled WeGO graph are replaced with the ZenDNN operators, and they will be executed using ZenDNN kernels for acceleration.

Environment Variables

ZenDNN provides some environment variables for performance tuning.

Table 51: **Environment Variables**

Name	Description
OMP_DYNAMIC	Set it explicitly with FALSE when you want to enable ZenDNN.
ZENDNN_GEMM_ALGO	The default value is 3. You can set [0, 1, 2, 3, 4] to tune different GEMM ALGO paths.
OMP_NUM_THREADS	The default value is the number of physical cores of the user system. You need to tune per the inference thread number to achieve better performance. For more details, see tuning guidelines.

Tunning Guidelines

ZenDNN uses OpenMP as the underlying library. The `OMP_NUM_THREADS` environment variable controls intra-op parallelism, which is multi-core parallelism in ZenDNN kernels. For OpenMP, different application threads or inter-op threads can use different OpenMP thread pools for intra-op tasks. Thus, many OpenMP threads might be used in a multi-thread application, which will consume lots of CPU core resources and reduce the overall performance. So, the recommended tuning `OMP_NUM_THREADS` value is set per the number of cores in the target CPU platform and the thread number used in your application to avoid over-subscription. For example, if you launch 16 threads in an application and have 64 CPU cores on your platform, you can set `OMP_NUM_THREADS <= 4` to avoid CPU cores contention.

ZenDNN in WeGO TensorFlow 2

Enable ZenDNN in WeGO TensorFlow 2

ZenDNN is disabled by default. Set `export TF_ENABLE_ZENDNN_OPTS=1` to enable it.

Environment Variables

You must export the following environment variables explicitly to enable ZenDNN to work correctly in WeGO TensorFlow 2.

Table 52: **Environment Variables**

Name	Description
OMP_DYNAMIC	Set it to FALSE explicitly when ZenDNN is enabled.
OMP_NUM_THREADS	Set it explicitly to achieve better performance. See tuning guidelines for more details.
ZENDNN_GEMM_ALGO	The default value is 3. You can set [0, 1, 2, 3, 4] to tune different GEMM ALGO paths.
ZENDNN_TENSOR_POOL_LIMIT	The default value is 32. See tuning guidelines for more details.
ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE	Default is 0. • 0: Enable reduced memory pool tensor. • 1: Enable increased memory pool tensor.
TF_ENABLE_ZENDNN_OPTS	The default value is 0. Set it to 1 to enable ZenDNN.

Tuning Guidelines

Set `OMP_NUM_THREADS` per the core number of the user system. AMD recommends setting a small number like 1 or 2.

In some cases, set `ZENDNN_TENSOR_POOL_LIMIT` to a small number like 1, so some layers use default memory allocation instead of the tensor pool once it hits the pool limit with `ZENDNN_TENSOR_POOL_LIMIT`.

Profiling the Model

Vitis AI Profiler

The AMD Vitis™ AI profiler is a set of tools that helps profile and visualize AI applications based on VART:

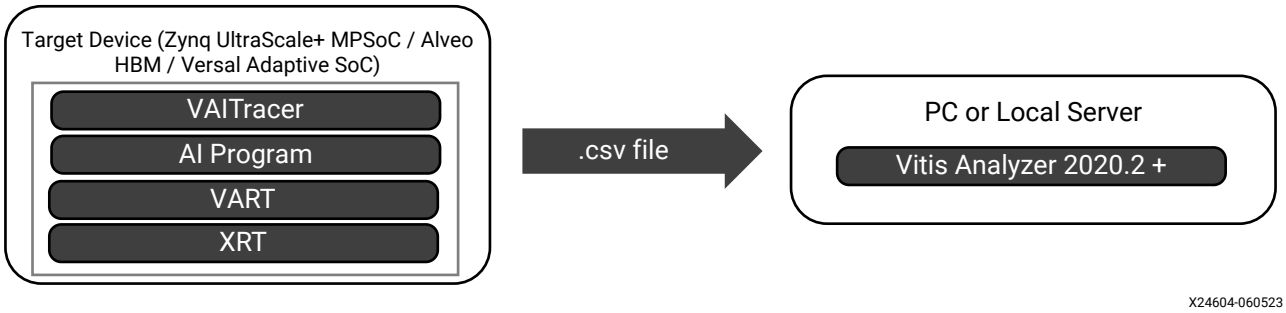
- Easy to use as it neither requires any change in the user code nor any re-compilation of the program.
- Visualize system performance bottlenecks.
- Illustrate the execution state of different compute units (CPU/DPU).

The Vitis AI Profiler is an all-in-one profiling solution for Vitis AI. It is an application-level tool to profile and visualize AI applications based on VART. For an AI application, some components run on the hardware. For example, neural network computation usually runs on the DPU, and some components run on a CPU as a function implemented by C/C++ code-like image pre-processing. This tool helps you to put the running status of all these different components together.

Vitis AI Profiler Architecture

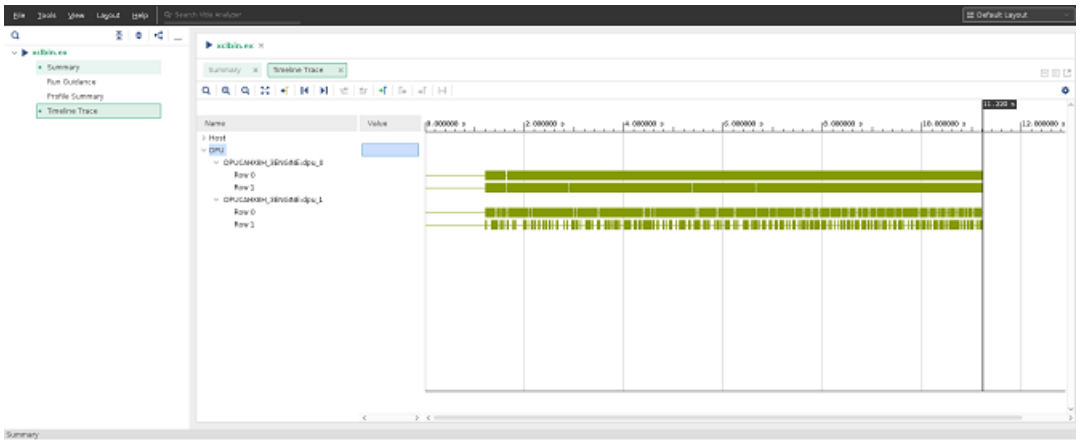
The Vitis AI Profiler architecture is shown in the following figure:

Figure 32: **Vitis AI Profiler Architecture**



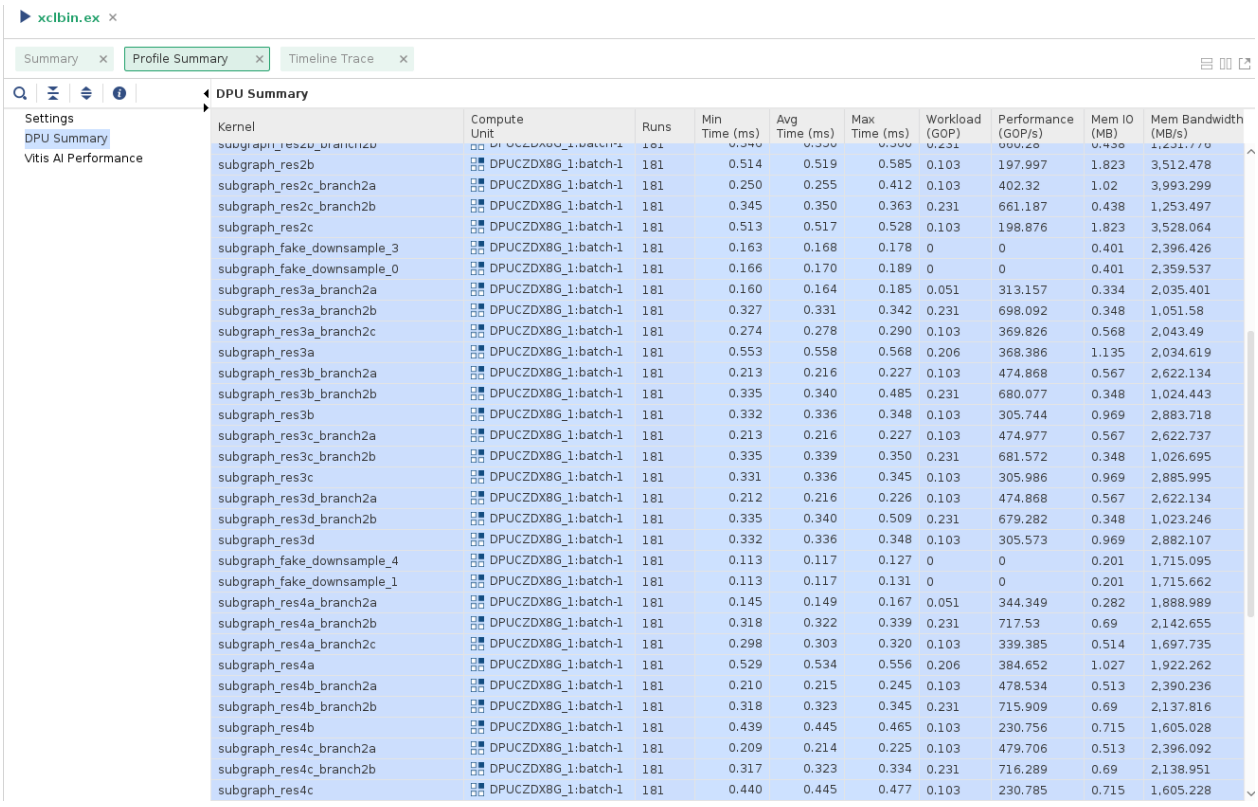
Vitis AI Profiler GUI Overview

Figure 33: **Vitis AI Profiler GUI Overview**



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

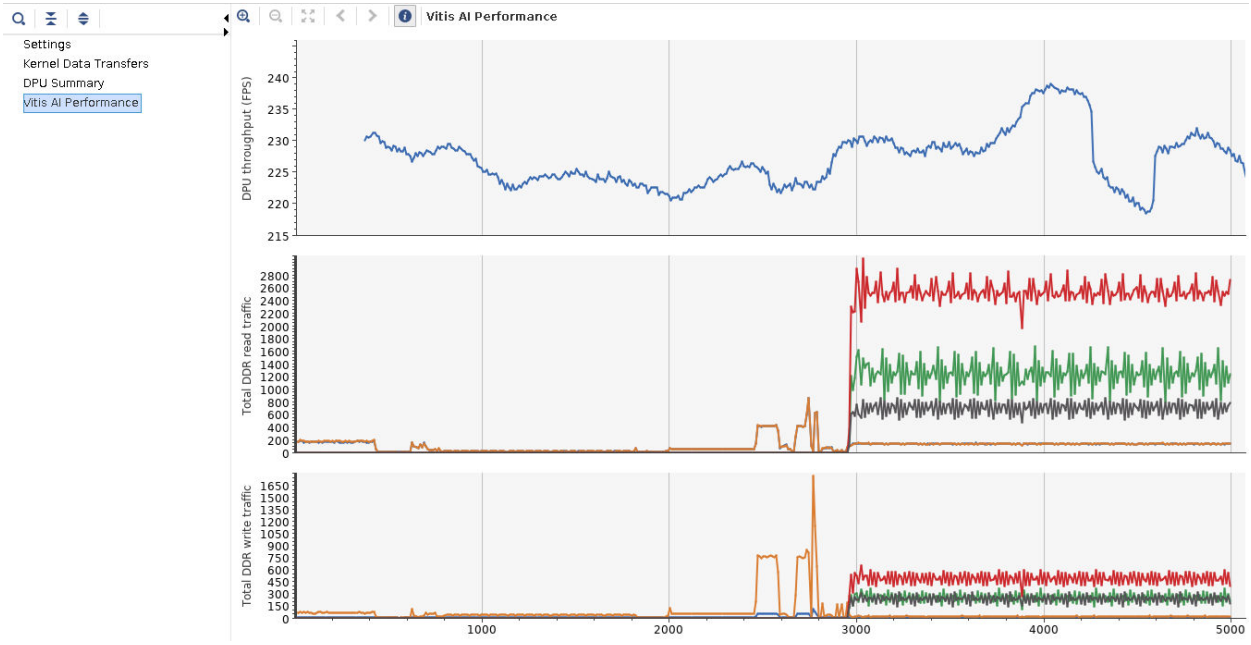
Figure 34: **DPU Summary**



Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Workload (GOP)	Performance (GOP/s)	Mem IO (MB)	Mem Bandwidth (MB/s)
subgraph_res2b	DPUCZDX8G_1:batch-1	181	0.514	0.519	0.585	0.103	197.997	1.823	3,512.478
subgraph_res2c_branch2a	DPUCZDX8G_1:batch-1	181	0.250	0.255	0.412	0.103	402.32	1.02	3,993.299
subgraph_res2c_branch2b	DPUCZDX8G_1:batch-1	181	0.345	0.350	0.363	0.231	661.187	0.438	1,253.497
subgraph_res2c	DPUCZDX8G_1:batch-1	181	0.513	0.517	0.528	0.103	198.876	1.823	3,528.064
subgraph_fake_downsample_3	DPUCZDX8G_1:batch-1	181	0.163	0.168	0.178	0	0	0.401	2,396.426
subgraph_fake_downsample_0	DPUCZDX8G_1:batch-1	181	0.166	0.170	0.189	0	0	0.401	2,359.537
subgraph_res3a_branch2a	DPUCZDX8G_1:batch-1	181	0.160	0.164	0.185	0.051	313.157	0.334	2,035.401
subgraph_res3a_branch2b	DPUCZDX8G_1:batch-1	181	0.327	0.331	0.342	0.231	698.092	0.348	1,051.58
subgraph_res3a_branch2c	DPUCZDX8G_1:batch-1	181	0.274	0.278	0.290	0.103	369.826	0.568	2,043.49
subgraph_res3a	DPUCZDX8G_1:batch-1	181	0.553	0.558	0.568	0.206	368.386	1.135	2,034.619
subgraph_res3b_branch2a	DPUCZDX8G_1:batch-1	181	0.213	0.216	0.227	0.103	474.868	0.567	2,622.134
subgraph_res3b_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.340	0.485	0.231	680.077	0.348	1,024.443
subgraph_res3b	DPUCZDX8G_1:batch-1	181	0.332	0.336	0.348	0.103	305.744	0.969	2,883.718
subgraph_res3c_branch2a	DPUCZDX8G_1:batch-1	181	0.213	0.216	0.227	0.103	474.977	0.567	2,622.737
subgraph_res3c_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.339	0.350	0.231	681.572	0.348	1,026.695
subgraph_res3c	DPUCZDX8G_1:batch-1	181	0.331	0.336	0.345	0.103	305.986	0.969	2,885.995
subgraph_res3d_branch2a	DPUCZDX8G_1:batch-1	181	0.212	0.216	0.226	0.103	474.868	0.567	2,622.134
subgraph_res3d_branch2b	DPUCZDX8G_1:batch-1	181	0.335	0.340	0.509	0.231	679.282	0.348	1,023.246
subgraph_res3d	DPUCZDX8G_1:batch-1	181	0.332	0.336	0.348	0.103	305.573	0.969	2,882.107
subgraph_fake_downsample_4	DPUCZDX8G_1:batch-1	181	0.113	0.117	0.127	0	0	0.201	1,715.095
subgraph_fake_downsample_1	DPUCZDX8G_1:batch-1	181	0.113	0.117	0.131	0	0	0.201	1,715.662
subgraph_res4a_branch2a	DPUCZDX8G_1:batch-1	181	0.145	0.149	0.167	0.051	344.349	0.282	1,888.989
subgraph_res4a_branch2b	DPUCZDX8G_1:batch-1	181	0.318	0.322	0.339	0.231	717.53	0.69	2,142.655
subgraph_res4a_branch2c	DPUCZDX8G_1:batch-1	181	0.298	0.303	0.320	0.103	339.385	0.514	1,697.735
subgraph_res4a	DPUCZDX8G_1:batch-1	181	0.529	0.534	0.556	0.206	384.652	1.027	1,922.262
subgraph_res4b_branch2a	DPUCZDX8G_1:batch-1	181	0.210	0.215	0.245	0.103	478.534	0.513	2,390.236
subgraph_res4b_branch2b	DPUCZDX8G_1:batch-1	181	0.318	0.323	0.345	0.231	715.909	0.69	2,137.816
subgraph_res4b	DPUCZDX8G_1:batch-1	181	0.439	0.445	0.465	0.103	230.756	0.715	1,605.028
subgraph_res4c_branch2a	DPUCZDX8G_1:batch-1	181	0.209	0.214	0.225	0.103	479.706	0.513	2,396.092
subgraph_res4c_branch2b	DPUCZDX8G_1:batch-1	181	0.317	0.323	0.334	0.231	716.289	0.69	2,138.951
subgraph_res4c	DPUCZDX8G_1:batch-1	181	0.440	0.445	0.477	0.103	230.785	0.715	1,605.228

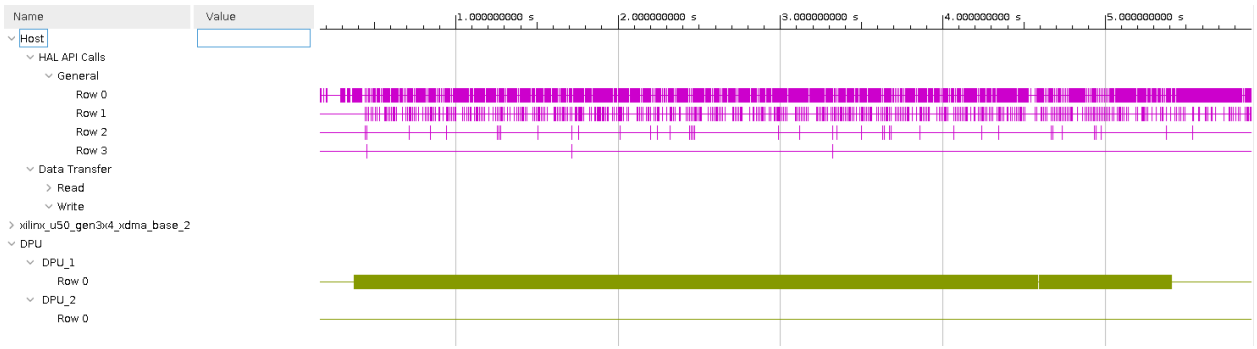
- **DPU Throughput and DDR Transfer Rates:** Line graphs of achieved FPS and read/write transfer rates (in Mbps) as sampled during the application.

Figure 35: **DPU Throughput and DDR Transfer Rates**



- **Timeline Trace:** This includes timed events from VART, HAL APIs, and the DPUs.

Figure 36: **Timeline Trace**



Note: The Vitis Analyzer is the default GUI for vaitrace in Vitis AI 1.3 and later releases.

Getting Started with the Vitis AI Profiler

System Requirements

- **Hardware:**
  - Supports AMD Zynq™ UltraScale+™ MPSoC (DPUCZDX8G)

    Supports AMD Versal™ adaptive SoC (DPUCVDX8G/ DPUCVDX8H)
- **Software:**

- Supports VART v1.2+

Installing the Vitis AI Profiler

1. Prepare the debug environment for vaitrace in the Zynq UltraScale+ MPSoC PetaLinux platform.
   a. Configure and build PetaLinux by running `petalinux-config -c kernel`.
   b. Enable the following settings for the Linux kernel.
      - General architecture-dependent options ---> [*] Kprobes
      - Kernel hacking ---> [*] Tracers
      - Kernel hacking ---> [*] Tracers --->

        [*] Kernel Function Tracer

        [*] Enable kprobes-based dynamic events

        [*] Enable uprobes-based dynamic events
   c. Run `petalinux-config -c rootfs` and enable the following setting for root-fs.

      Petalinux package Groups ---> packaggroup-petalinux-self-hosted ---> [*] packagegroup-petalinux-self-hosted
   d. Run `petalinux-build`.
2. Install vaitrace. vaitrace is integrated into the VART runtime. If VART runtime is installed, vaitrace is installed into `/usr/bin/vaitrace`.

Starting a Simple Trace with vaitrace

The following example uses the VART ResNet50 sample:

1. Download and set up Vitis AI.
2. Start testing and tracing.
   - For C++ programs, add `vaitrace` in front of the test command as follows:

     ```
     # cd ~/Vitis_AI/examples/vai_runtime/resnet50
     # vaitrace ./resnet50 /usr/share/vitis_ai_library/models/resnet50/
     resnet50.xmodel
     ```
   - For Python programs, add `-m vaitrace_py` to the Python interpreter command as follows:

     ```
     # cd ~/Vitis_AI/examples/vai_runtime/resnet50_mt_py
     # python3 -m vaitrace_py ./resnet50.py 2 /usr/share/vitis_ai_library/
     models/resnet50/resnet50.xmodel
     ```

   vaitrace and XRT generate some files in the working directory.

3. Copy all .csv files and `xclbin.ex.run_summary` to your system. You can open the `xclbin.ex.run_summary` using vitis_analyzer 2020.2 and above:
   - If using the command line, run `# vitis_analyzer xclbin.ex.run_summary`.
   - If using the GUI, select **File → Open Summary → xclbin.ex.run_summary**.

To know more about the Vitis Analyzer, see Using the Vitis Analyzer in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

vaitrace Usage

Command Line Usage

```
# vaitrace --help
usage: vaitrace [-h] [-c [CONFIG]] [-d] [-o [TRACESAVETO]] [-t [TIMEOUT]] [-
v]
                [-b] [-p] [--va] [--xat] [--txt_summary] [--fine_grained]
                ...

positional arguments:
  cmd

optional arguments:
  -h, --help         show this help message and exit
  -c [CONFIG]        Specify the config file
  -d                 Enable debug
  -o [TRACESAVETO]   Save report to, only available for txt summary mode
  -t [TIMEOUT]       Tracing time limitation
  -v                 Show version
  -b                 Bypass vaitrace, just run command
  -p                 Trace python application
  --va               Generate trace data for Vitis Analyzer
  --xat              Save raw data for debug usage
  --txt_summary      Display txt summary
  --fine_grained     Fine-grained mode
```

Following are some important and frequently used arguments:

- **cmd:** cmd is an executable Vitis AI program that traces, including the program name and arguments.
- **-t:** Controlling the tracing time (in seconds) starting from the [cmd] being launched, the default value is 30. In other words, if no -t is specified for vaitrace, the tracing stops after [cmd] runs for 30 seconds. The [cmd] continues running normally but stops collecting tracing data.
- **-c:** You can start a tracing with more custom options by writing these options on a JSON configuration file and specifying the configuration by -c. Details of the configuration file are explained in the next section.
- **-o:** Location of the report. This is only available for the text summary mode. By default, the test summary outputs to STDOUT.

- **--va:** Generate trace data for Vitis Analyzer, enabled by default, cannot work together with --txt_summary.
- **--txt_summary or --txt:** Output text summary. vaitrace does not generate a report for the Vitis Analyzer in this mode and cannot work together with --va.
- **--fine_grained:** Start trace in the fine-grained mode. This mode generates a mass of trace data, limiting the trace time to 10 seconds.

Other arguments are used for debugging.

Configuration

Using a configuration file to record trace options for vaitrace is recommended. You can start a trace with configuration by using `vaitrace -c trace_cfg.json`.

Configuration priority: **Configuration File → Command Line → Default**.

Here is an example of a vaitrace configuration file.

```
{
  "trace": {
      "enable_trace_list": ["vitis-ai-library", "vart", "custom"]
  }
  "trace_custom": []
}
```

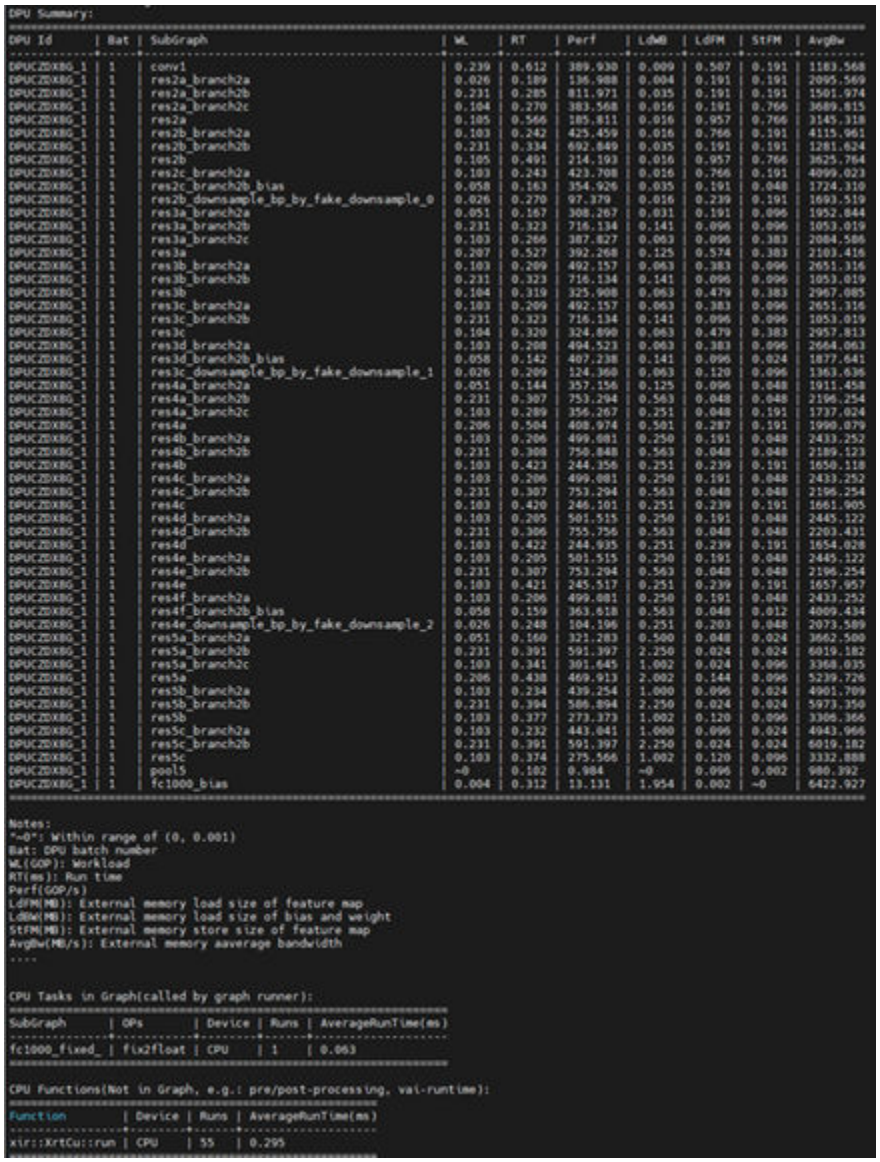
Table 53: **Contents of the Configuration File**

Key Name		Value Type	Description
trace		object	
	enable_trace_list	list	Built-in trace function list to be enabled, available value "vitis-ai-library", "vart", “opencv”, "custom." Custom for function in trace_custom list.
trace_custom		list	The list of functions to be traced that the user implements. For the name of the function, naming space is supported. You can see an example of using a custom trace function later in this document.

Text Summary

When the `--txt` or `--txt_summary` option is used, vaitrace prints an ASCII table, as shown in the following figure:

Figure 37: **ASCII Table**



The fields are defined in the following list:

- **DPU Id:** Name of the DPU instance.
- **Bat:** Batch size of the DPU instance.
- **SubGraph:** Name of subgraph in the xmodel.
- **WL (Workload):** Computation workload (MAC indicates two operations). The unit is GOP.
- **RT (Runtime):** The execution time in milliseconds, unit is ms.
- **Perf:** The DPU performance in unit of GOP per second. The unit is GOP/s.

- **LdFM (Load Size of Feature Map):** The external memory load size of the feature map. The unit is MB.
- **LdWB (Load Size of Weight and Bias):** The external memory load size of bias and weight. The unit is MB.
- **StFM (Store Size of Feature Map):** The external memory store size of the feature map. The unit is MB.
- **AvgBw (Average bandwidth):** Average DDR memory access bandwidth.

  AvgBw = (total load size of the subgraph (including feature map and weight/bias, from DDR/HBM to DPU bank mem) + total store size of the subgraph (from DPU bank mem to DDR/HBM)) / subgraph runtime

DPU Profiling Examples

You can find advanced DPU profiling examples with the Vitis AI Profiler on the Vitis AI Profiler GitHub page.

Appendix A

Error Codes

Table 54: **Error Codes**

Error Code ID	Error Message
OPTIMIZER_DATA_PARALLEL_NOT_ALLOWED_ERROR	torch.nn.DataParallel module is not allowed.
OPTIMIZER_INVALID_ANA_RESULT_ERROR	Model analysis result is not valid. This is usually caused by PyTorch or Python version changes.
OPTIMIZER_INVALID_ARGUMENT_ERROR	Invalid argument.
OPTIMIZER_TORCH_MODULE_ERROR	The operation is not an instance of torch.nn.Module.
OPTIMIZER_NOT_EXCLUDE_NODE_ERROR	Some nodes must be excluded from pruning.
OPTIMIZER_NO_ANA_RESULT_ERROR	Model analysis results not found.
OPTIMIZER_SUBNET_ERROR	Subnet candidates not found. Must do subnet searching first.
OPTIMIZER_UNSUPPORTED_OP_ERROR	The operation is not supported yet.
OPTIMIZER_KERAS_MODEL_ERROR	The given object is not an instance of keras.Model.
OPTIMIZER_KERAS_LAYER_ERROR	The operation is not an instance of keras.Layer.
OPTIMIZER_DATA_FORMAT_ERROR	The data format for saving weights is not allowed in pruning.
OPTIMIZER_INVALID_GRAPH	The parsed graph is invalid.
OPTIMIZER_IO_ERROR	IO error. Usually occurs during disk read/write.
OPTIMIZER_MODEL_ANALYSIS_ERROR	An error occurred while performing model analysis.
OPTIMIZER_PARSE_GRAPH_FAILED	Unable to parse the model to a computation graph.
OPTIMIZER_WEIGHTS_NOT_FOUND	The weights for the operation can not be found.
QUANTIZER_TF1_INVALID_BITWIDTH	invalid parameter
QUANTIZER_TF1_INVALID_METHOD	invalid parameter
QUANTIZER_TF1_INVALID_TARGET_DTYPE	invalid parameter
QUANTIZER_TF1_MISSING_QUANTIZE_INFO	not found
QUANTIZER_TF1_INVALID_INPUT	not found
QUANTIZER_TF1_UNSUPPORTED_OP	Unsupported Op type
QUANTIZER_TF1_LENGTH_MISMATCH	invalid parameter
QUANTIZER_TF1_INVALID_INPUT_FN	fail to import
QUANTIZER_TF2_UNSUPPORTED_MODEL	Unsupported model type
QUANTIZER_TF2_UNSUPPORTED_LAYER	Unsupported layer type
QUANTIZER_TF2_INVALID_CALIB_DATASET	Invalid calibration dataset
QUANTIZER_TF2_INVALID_INPUT_SHAPE	Invalid input shape
QUANTIZER_TF2_INVALID_TARGET	Invalid Target

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
QUANTIZER_TORCH_BIAS_CORRECTION	Bias correction file in quantization result directory does not match current model.
QUANTIZER_TORCH_CALIB_RESULT_MISMATCH	Node name mismatch is found when loading quantization steps of tensors. Please ensure the vai_q_pytorch version and PyTorch version for test mode are the same as those in calibration (or QAT training) mode.
QUANTIZER_TORCH_EXPORT_ONNX	The quantized module, which is based PyTorch traced model, can not be exported to ONNX due to PyTorch internal failure. The PyTorch internal failure reason is listed in message text. May needs adjust the float model code.
QUANTIZER_TORCH_EXPORT_XMODEL	Fail to convert the graph to XMODEL. Needs to check the reasons in the message text.
QUANTIZER_TORCH_FAST_FINETINE	Fast fine-tuned parameter file does not exist. Call load_ft_paramthe in the model code to load them.
QUANTIZER_TORCH_FIX_INPUT_TYPE	Data type or value is illegal in arguments of quantization OP when exporting the ONNX model.
QUANTIZER_TORCH_ILLEGAL_BITWIDTH	The configuration of tensor quantization is illegal. It should be an integer and in the range given in message text.
QUANTIZER_TORCH_IMPORT_KERNEL	Importing vai_q_pytorch library file error. Check PyTorch version matching vai_q_pytorch version (pytorch_nndct.__version__) or not.
QUANTIZER_TORCH_NO_CALIB_RESULT	Quantization result file does not exist. Check whether the calibration is done or not.
QUANTIZER_TORCH_NO_CALIBRATION	Quantization calibration is not performed completely, check if module FORWARD function is called! FORWARD function of torch_quantizer.quant_model needs to be called in the user code explicitly. Please refer to the example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_NO_FORWARD	torch_quantizer.quant_model FORWARD function must be called before exporting quantization result. Please refer to example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_OP_REGIST	The type of OP can't be registered multiple times.
QUANTIZER_TORCH_PYTORCH_TRACE	Failed to get PyTorch traced graph from model and input arguments. The PyTorch internal failure reason is reported in message text. May needs adjust float model code.
QUANTIZER_TORCH_QUANT_CONFIG	Quantization configuration items are illegal. Refer to the message text.
QUANTIZER_TORCH_SHAPE_MISMATCH	Tensors shape are mismatched. Refer to the message text.
QUANTIZER_TORCH_VERSION	PyTorch version is not supported for the function or does not match vai_q_pytorch version (pytorch_nndct.__version__). Refer to the message text.
QUANTIZER_TORCH_XMODEL_BATCHSIZE	Batch size must be 1 when exporting XMODEL.
QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT	Inspector only supports dump SVG or PNG format.
QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT	Inspector no longer supports the fingerprint. Please provide architecture name instead.
QUANTIZER_TORCH_UNSUPPORTED_OPS	The quantization of the op is not supported.
QUANTIZER_TORCH_TRACED_NOT_SUPPORT	The model produced by 'torch.jit.script' is not supported in vai_q_pytorch.
QUANTIZER_TORCH_NO_SCRIPT_MODEL	vai_q_pytorch does not find any script model.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
QUANTIZER_TORCH_REUSED_MODULE	The quantized module has been called multiple times in forward pass. If you want to share quantized parameters in multiple calls, call trainable_model with "allow_reused_module=True"
QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED	torch.nn.DataParallel object is not allowed.
QUANTIZER_TORCH_INPUT_NOT_QUANTIZED	Input is not quantized. Please use QuantStub/DeQuantStub to define quantization scope.
QUANTIZER_TORCH_NOT_A_MODULE	Quantized operation must be instance of "torch.nn.Module". Replace the function by a "torch.nn.Module" object. Original source range is indicated in the message text.
QUANTIZER_TORCH_QAT_PROCESS_ERROR	Must call "trainable_model" first before getting deployable model.
QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR	The given trained model has BN fused to CONV and cannot be converted to a deployable model. Make sure model.fuse_conv_bn() is not called.
QUANTIZER_TORCH_XMODEL_DEVICE	XMODEL can only be exported in CPU mode, use deployable_model(src_dir, used_for_xmodel=True) to get a CPU model.
WEGO_TORCH_UNKNOWN_ERROR	Unknown error
WEGO_TORCH_INTERNAL_ERROR	Internal error
WEGO_TORCH_INVALID_ARGUMENT	Invalid argument error
WEGO_TORCH_INVALID_MODEL	Invalid model error
WEGO_TORCH_OUT_OF_RANGE	Out of range error
WEGO_TORCH_UNIMPLEMENTED	Unimplemented error
WEGO_TORCH_RUNTIME_ERROR	Runtime error
XCOM_OP_CONV_PARAM_ERROR	Convolution parameter out of range or error, including feature map height, width, depth, channel, dilation size, transposition size, kernel height, kernel width, stride height, stride width or padding left, right, top, bottom, depth or fixed-point shift range or other network designed parameters.
XCOM_OP_IO_TENSOR_TYPE_ERROR	Error tensor type for IO operator such as load and save.
XCOM_OP_MEM_TYPE_ERROR	The op's output tensor's memory type is error.
XCOM_OP_PAD_SMF_MISSING	Failed to generate padding in pool since smf data missing.
XCOM_OP_POOL_SIZE_ERROR	Failed to calculate pooling size with formula.
XCOM_OP_SIGMOID_HEIGHT_NE	sigmoid operator need input and output have same height.
XCOM_OP_SIGMOID_WIDTH_NE	sigmoid operator need input and output have same width.
XCOM_OP_SIGMOID_CHANNEL_NE	sigmoid operator need input and output have same channel.
XCOM_OP_REORG_HEIGHT_NE	reorg operator need input height and output height have scale multiple.
XCOM_OP_REORG_WIDTH_NE	reorg operator need input width and output width have scale multiple.
XCOM_OP_REORG_CHANNEL_NE	reorg operator need input channel and output channel have scale multiple.
XCOM_OP_REORG_CHANNEL_OVERFLOW	feature channel size overflows regorg channel input.
XCOM_OP_TYPE_UNMATCH	unmatch operator type, it could be unknown operator or inappropriate suffix operator type.
XCOM_OP_TYPE_ERROR	op involved type error, unrecognized op involved type here.
XCOM_TILING_SIZE_ERROR	Tiling bank group size not enough and tiling failed. Perhaps input tensor or kernel size is too large or tiling bank aligned has calculation fault.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_DPUOP_DATA_SIZE_ERROR	Size not enough or unaligned while mapping smf onto banks with channel, width, depth, height, stride_h and other dimension incompatible. Please check bank info
XCOM_ACGEN_POOL_KERNEL_OUTRANGE	Pooling layer kernel size out of range. check network design or input data size.
XCOM_OP_NONLINEAR_TYPE_ERROR	Error of operator non-linear type.
XCOM_ACGEN_UNSUPPORT_QUANTIZATION	Unsupport quantization bit shift while assembly code generation.
XCOM_ACGEN_NONLINEAR_TYPE_ERROR	Unrecognized or unmatched type of non-linear type while assembly code generation
XCOM_ACGEN_BANK_IO_ERROR	Bank input or output addr count error while assembly code generation, it may exceed hardware capapcity.
XCOM_ACGEN_PRELU_ERROR	parameter-relu data info error while convolution assembly code generating, might be error with parameter data input width or number error.
XCOM_ACGEN_CONV_WEIGHTS_OC_NE	Convolution output channel number should be equal to convolution weights input, if not, check data life-circle.
XCOM_ACGEN_BANK_OC_WEIGHTS_UNALIGNED	Weights bank address need be aligned to convolution output channel.
XCOM_BANK_UNALIGNED_ADDRESSING	Trying to address in the middle of bank addr which is illegal, bank addressing only support aligned operation, for example, stride or address mod 16 == 0.
XCOM_ACGEN_CONV_FAKE_WEIGHTS_BANKID	Convolution weights input bank id need be equal to base bank id, check bank assignation of weights.
XCOM_ACGEN_CONV_FAKE_BIAS_BANKID	Convolution bias input bank id need be equal to base bank id, check bank assignation of bias.
XCOM_ACGEN_CONV_OCG_WEIGHTS_CNT_NE	Convolution weights input number need be equal to output channel group size, check weights input number.
XCOM_ACGEN_KERNEL_ALL_PAD	kernel are fulfilled with pad value, this is an unexpected situation, check kernel size and dilation value.
XCOM_ACGEN_BANK_ADDR_IN_OUTRANGE	Bank address input number need be ordered in hardware limitation.
XCOM_ACGEN_BANK_ADDR_OUT_OUTRANGE	Bank address output number need be ordered in hardware limitation.
XCOM_ACGEN_ELEW_IO_ERROR	Element wise operator have more than hardware capability input number, or, input and output number is not equal. check bank assignation.
XCOM_ACGEN_ELEW_IO_CHANNEL_NE	Element wise operator need input and output have same channel group size.
XCOM_ACGEN_ELEW_IO_LENGTH_NE	Element wise operator need input and output have same length.
XCOM_ACGEN_MUL_IO_ERROR	mul operator have more than hardware capability input number, or, input and output number is not equal. check bank assignation.
XCOM_ACGEN_INPUT_MISSING	Operator assembly code generation input bank address missing, check bank assignation.
XCOM_ACGEN_BLOB_MISSING	Blob shifting failed because cannot find specific blob id in blob area. check blob assignation.
XCOM_ACGEN_OUTPUT_MISSING	Operator assembly code generation ouptut bank address missing, check bank assignation.
XCOM_ACGEN_IO_TUPLE_NE	Some operator need input and output number be equal but here is not. check bank assignation.
XCOM_ACGEN_WEIGHTS_NOT_UNIQ	Some operator need uniq weights input bank, check bank assignation.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_ACGEN_PRELU_NOT_UNIQ	Some operator need uniq prelu input bank, check bank assignation.
XCOM_ACGEN_PARAM_NOT_UNIQ	Some operator need uniq param input bank like sigmoid, check bank aggregation.
XCOM_ACGEN_BIAS_NOT_UNIQ	Some operator need uniq bias input bank, check bank assignation.
XCOM_ACGEN_V3ME_BANK_MISSING	Operator's dest bank have no virtual bank or constant bank, on v3me conv operator at least need 1 type of bank.
XCOM_ACGEN_IC_WEIGHTS_CHANNEL_NE	Depth operator input channel and weights channel are not equal, check bank assignation or network structure.
XCOM_ACGEN_BIAS_WEIGHTS_CHANNEL_NE	Depth operator weighs channel and bias channel are not equal, check bank assignation.
XCOM_ACGEN_INVALIDE_STATUS	Compiler internal error, some status is invalid for assembly code generation like object was not inited, missing input or output data on dpu bank. check data flow and code logic.
XCOM_ACGEN_BANK_JUMP_READ_ERROR	The bank cannot jump read.
XCOM_ACGEN_BANK_JUMP_WRITE_ERROR	The bank cannot jump write.
XCOM_OP_ARGMAX_IO_HEIGHT_NE	argmax need input and output have equal height.
XCOM_OP_ARGMAX_IO_WIDTH_NE	argmax need input and output have equal width.
XCOM_OP_ARGMAX_IO_DEPTH_NE	argmax need input and output have equal depth.
XCOM_OP_ARGMAX_OC_NOT_UNIQ	argmax need output channel is 1.
XCOM_OP_CONCAT_IO_CHANNEL_NE	concat operator need input and output channel equal.
XCOM_OP_CORR_ELT_MUL_OUTPUT_ERROR	correlation eltwise multiply output depth calculate error.
XCOM_OP_CORR_ELT_MUL_IO_HEIGHT_NE	Correlation eltwise multiply need input height is equal to output height
XCOM_OP_CORR_ELT_MUL_IO_WIDTH_NE	Correlation eltwise multiply need input width is equal to output width
XCOM_OP_CORR_ELT_MUL_IO_CHANNEL_NE	Correlation eltwise multiply need input channel is equal to output channel
XCOM_OP_CORR_ELT_MUL_INPUT_CHANNEL_NE	Correlation eltwise multiply need all input channel are equal
XCOM_OP_COST_STRIDE_OUTPUT_DEPTH_NE	Cost operator need stride is equal to output depth
XCOM_OP_COST_IO_HEIGHT_NE	Cost operator need input and output have same height
XCOM_OP_COST_IO_WIDTH_NE	Cost operator need input and output have same width
XCOM_OP_COST_INPUT_DEPTH_NOT_UNIQ	Cost operator need input depth = 1
XCOM_OP_COST_IO_CHANNEL_NE	Cost operator need input channel = 1/2 output channel
XCOM_OP_DOWNSAMPLE_IO_HEIGHT_NE	downsample need input height ceiling divide scale height equal to output height
XCOM_OP_DOWNSAMPLE_IO_WIDTH_NE	downsample need input width ceiling divide scale width equal to output width
XCOM_OP_DOWNSAMPLE_IO_CHANNEL_NE	downsample need input and output channel equal.
XCOM_OP_TDPTCONV3D_ICG_NOT_ENOUGH	Transposed depth conv 3d input channel group mult channel parallel is less than feature channel size.
XCOM_OP_TDPTCONV3D_KERNEL_HEIGHT_OVERFLOW	Transposed depth conv 3d kernel height overflow.
XCOM_OP_TDPTCONV3D_KERNEL_WIDTH_OVERFLOW	Transposed depthconv 3d kernel width overflow.
XCOM_OP_TDPTCONV3D_KERNEL_DEPTH_OVERFLOW	Transposed depth conv 3d kernel depth overflow.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_OP_TDPTCONV3D_STRIDE_HEIGHT_OVERFLOW	Transposed depth conv 3d stride height overflow.
XCOM_OP_TDPTCONV3D_STRIDE_WIDTH_OVERFLOW	Transposed depth conv 3d stride width overflow.
XCOM_OP_TDPTCONV3D_STRIDE_DEPTH_OVERFLOW	Transposed depth conv 3d stride depth overflow.
XCOM_OP_TDPTCONV3D_OCG_NOT_ENOUGH	Transposed depth conv 3d output channel group mult channel parallel is less than feature channel size.
XCOM_OP_DPTCONV_IC_WEIGHT_DEPTH_NE	depth conv need kernel mult input channel group equal to weight bank depth
XCOM_OP_DPTCONV3D_KERNEL_HEIGHT_OVERFLOW	depth conv 3d kernel height overflow.
XCOM_OP_DPTCONV3D_KERNEL_WIDTH_OVERFLOW	depth conv 3d kernel width overflow.
XCOM_OP_DPTCONV3D_KERNEL_DEPTH_OVERFLOW	depth conv 3d kernel depth overflow.
XCOM_OP_DPTCONV3D_STRIDE_HEIGHT_OVERFLOW	depth conv 3d stride height overflow.
XCOM_OP_DPTCONV3D_STRIDE_WIDTH_OVERFLOW	depth conv 3d stride width overflow.
XCOM_OP_DPTCONV3D_STRIDE_DEPTH_OVERFLOW	depth conv 3d stride depth overflow.
XCOM_OP_DPTCONV3D_OCG_NOT_ENOUGHT	depth conv 3d output channel group mult channel parallel is less than feature channel size.
XCOM_OP_THRESHOLD_HEIGHT_NE	Threshold operator needs input and output have same height.
XCOM_OP_THRESHOLD_WIDTH_NE	Threshold operator needs input and output have same width.
XCOM_OP_THRESHOLD_CHANNEL_NE	Threshold operator needs input and output have same channel.
XCOM_OP_TILE_HEIGHT_NE	Tile operator needs input and output height have scale multiple relationship.
XCOM_OP_TILE_WIDTH_NE	Tile operator needs input and output width have scale multiple relationship.
XCOM_OP_TILE_CHANNEL_NE	Tile operator needs input and output channel have scale multiple relationship.
XCOM_OP_UPSAMPLE_HEIGHT_NE	upsample operator needs input and output height have scale multiple relationship.
XCOM_OP_UPSAMPLE_WIDTH_NE	upsample operator needs input and output width have scale multiple relationship.
XCOM_OP_UPSAMPLE_CHANNEL_NE	upsample operator needs input and output channel have scale multiple relationship.
XCOM_OP_ELEW_IO_CHANNEL_NE	element wise operator needs input and output channel equal
XCOM_OP_ELEW3D_IO_CHANNEL_NE	element wise 3d operator needs input and output channel equal
XCOM_OP_MUL_IO_HEIGHT_NE	mul operator needs input and output have same height.
XCOM_OP_MUL_IO_WIDTH_NE	mul operator needs input and output have same width.
XCOM_OP_MUL_IO_DEPTH_NE	mul operator needs input and output have same depth.
XCOM_OP_MUL_IO_CHANNEL_NE	mul operator needs input and output have same height.
XCOM_OP_MVR_IO_HEIGHT_NE	mvr operator needs input and output have same height.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_OP_MVR_IO_WIDTH_NE	mvr operator needs input and output have same width.
XCOM_OP_MVR_IO_DEPTH_NE	mvr operator needs input and output have same depth.
XCOM_OP_MVR_IO_CHANNEL_NE	mvr operator needs input and output have same height.
XCOM_OP_CONV_KERNEL_WIDTH_OVERFLOW	Kernel width is larger than input width plus padding. that makes window cannot slide
XCOM_OP_CONV_KERNEL_HEIGHT_OVERFLOW	Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_CONV_STRIDE_WIDTH_OVERFLOW	Kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_CONV_STRIDE_HEIGHT_OVERFLOW	Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_CONV_KERNEL_DEPTH_OVERFLOW	Kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_KERNEL_HEIGHT_OVERFLOW	Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_STRIDE_WIDTH_OVERFLOW	Kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_STRIDE_HEIGHT_OVERFLOW	Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_KERNEL_DEPTH_OVERFLOW	Kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation
XCOM_OP_TCONV3D_DILATION_HEIGHT_OVERFLOW	Dilation height is too large for input height
XCOM_OP_TCONV3D_DILATION_WIDTH_OVERFLOW	Dilation height is too large for input height
XCOM_OP_TCONV3D_DILATION_DEPTH_OVERFLOW	Dilation height is too large for input height
XCOM_OP_TCONV3D_ICG_WEIGHT_DEPTH_OVERFLOW	Input channel group stride overflows the weight bank depth.
XCOM_OP_CONV_DILATION_WIDTH_ALL_PAD	Padding width too large for dilation. Make all value in slide window are padding without input.
XCOM_OP_CONV_DILATION_HEIGHT_ALL_PAD	padding height too large for dilation, make all value in slide window are padding without input.
XCOM_OP_CONV_DILATION_DEPTH_ALL_PAD	padding depth too large for dilation, make all value in slide window are padding without input.
XCOM_OP_CONV_STRIDE_OVERFLOW	input channel stride overflow the weight bank depth.
XCOM_OP_TCONV3D_KENREL_DEPTH_OVERLARGE	kernel depth too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_KENREL_WIDTH_OVERLARGE	kernel width too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_KENREL_HEIGHT_OVERLARGE	kernel height too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation.
XCOM_OP_TCONV3D_DILATION_WIDTH_ALL_PAD	padding width too large for dilation, makes all value in slide window are padding without input feature.
XCOM_OP_TCONV3D_DILATION_HEIGHT_ALL_PAD	padding height too large for dilation, makes all value in slide window are padding without input feature.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_OP_TCONV3D_DILATION_DEPTH_ALL_PAD	padding depth too large for dilation, makes all value in slide window are padding without input feature.
XCOM_ACGEN_CONV_ERROR	error parameters while generating convolution, like some convolution parameter exceed hardware limitation or unexpected middle result generated.
XCOM_BANK_CONV_ERROR	error banking status or behavior for convolution operation wile generating assembly code. check conv op data flow and tensor aggregation.
XCOM_BANK_INVALID_ID	invalid id while parsing for finding bank id. check bank name in target_factory.
XCOM_STR_PARSE_FAILED	Failed to parse specific string, perhaps it's an illegal string or empty string.
XCOM_DATA_SEGMENT_FAULT	data tensor or const tensor index exceed max tensor size, check index value.
XCOM_OP_CONFIG_MISSING	Failed to get specific op config. check op type config file.
XCOM_AIE_TARGET_INIT_FAILED	Failed to init aie target.
XCOM_AIE_SHIMTILE_OVERFLOW	aie tiling shim index or shim size out of range.
XCOM_AIE_MEMTILE_OVERFLOW	aie tiling memory index or memory size out of range.
XCOM_AIE_AIETILE_OVERFLOW	aie tiling index or bd index out of range.
XCOM_AIE_OUT_OF_BD	cannot find free bd for mem tiling.
XCOM_SLNODE_UNREGISTED	slnode target, type, name, or any info unregistered.
XCOM_INVOKE_BASE	An improper function invoking occurred!
XCOM_VALUE_UNMATCH	The value is not supposed!
XCOM_MEANINGLESS_VALUE	The value is meaningless.
XCOM_SIZE_UNMATCH	The object's size is not matching the requirement.
XCOM_OPERATOR_UNSUPPORT	This operator is not supposed!
XCOM_LEAF_SUBGRAPH_REQUIRED	Here requires a leaf subgraph.
XCOM_UNACCEPTABLE_SUBGRAPH	The subgraph is not allowed or meeting the requirements.
XCOM_PASS_MISS	Some compiler pass is missed.
XCOM_PASS_DEPENDENCY	Something wrong about pass dependency.
XCOM_DEBUGMANAGER_NOT_RECORDING	Invalid status for DebugManager recording.
XCOM_NO_PASS_RECORDED	No Pass has been recorded in DebugManager.
XCOM_DEBUGMANAGER_UNRECORDED_OP	Unrecorded op found.
XCOM_DDR_ADDR_ASSIGNMENT_FAILED	DDR address assignment is failed.
XCOM_DDR_PARAM_SPACE_INITIALIZTION_SIZE_ERROR	DDR parameter space initialization size error. Please contact us.
XCOM_UNIMPLEMENT	This part of function is unimplemented.
XCOM_UNDEFINED_STATE	This behavior is undefined.
XCOM_EXECUTE_SYSTEM_CMD_FAILED	Error occurred when execute the system command.
XCOM_TENSOR_DIMENSION_UNMATCH	The tensor dimension is unexpected.
XCOM_DATA_OUTRANGE	Data value is out of range!
XCOM_TYPE_UNMATCH	Unmatched type!

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_ITEM_UNDEFINED	The requested item was not found or defined!
XCOM_OPERATION_FAILED	The supposed operation is failed!
XCOM_DIR_OPEN_FILE_FAILED	The file can't be read or can't access it.
XCOM_INVALID_GRAPH	Subgraph is null or error subgraph type like cpu subgraph using for dpu
XCOM_UNREGISTED_STRATEGY	This error code only used in dead code
XCOM_INVALID_ARCH_PARAM	This error code only used in dead code
XCOM_ACGEN_ERROR	Instruction generating fail, contact us.
XCOM_UNEXPECTED_VALUE	Inappropriate value at this place like nullptr or non-one value
XCOM_UNEXPECTED_ARCH	Unknown architecture name, check target_factory
XCOM_ARCH_UNREGISTED	Unknown architecture name, check target_factory
XCOM_ARCH_INVALID_NAME	Failed to file arch name in config dict, target factory or arch name serialization, check arch name.
XCOM_FILE_NOT_EXISTS	The file is not exists
XCOM_TARGET_REQUIRED	The compiler requires the target.
XCOM_GRAPH_REQUIRED	The compiler requires an input graph.
XCOM_LOGICAL_CONDITION_ERROR	The logical condition is wrong.
XCOM_INVALID_SUPERLAYER	The superlayer subgraph is invalid.
XCOM_UNSUPPORT_QUANTIZATION	The fix info is error or unsupported.
XCOM_SWIM_NODE_TYPE_ERROR	mismatch slnode type
XCOM_SWIM_OUTPUT_MISSING	swim lane output bank smf missing.
XCOM_SWIM_UNDEFINED_TYPE	undefined swim program or lane type.
XCOM_ALLOCATE_BANK_FAIL	XCompiler occurs error when allocating bank. Please contact us.
XCOM_TILING_FAIL	XCompiler occurs error when tiling. Please contact us.
XCOM_PM_FAIL	The compiler occurs an error when generating instructions, contact us.
XCOM_SUBGRAPH_ATTR_MISSING	The subgraph attribute is missing.
XCOM_ASSIGN_OUTPUT_OPS_FAILED	Assign output ops failed.
XCOM_DDR_REG_ID_SIZE_UNMATCH	The DDR reg id size unmatch. Please contact us.
XCOM_DDR_OPTIMIZATION_0_FAILED	DDR assignment optimization failed (code 0). Please contact us.
XCOM_DDR_OPTIMIZATION_1_FAILED	DDR assignment optimization failed (code 1). Please contact us.
XCOM_TRANSPOSED_CONV_WEIGHTS_DDR_OPTIMIZATION_ERROR	DDR assignment optimization failed during optimizing the transposed convolution's weights. Please contact us.
XCOM_DIRECTORY_EXIST	The directory is existing, can't be created multiple times.
XCOM_DIRECTORY_NOT_EXIST	The directory is not existing.
XCOM_INVALID_COMPILE_MODE	The compile mode is not supported now.
XCOM_INVALID_TARGET	Invalid DPU target
XCOM_DPU_MEMORY_ALLOCATION_FAILED	error mapping smfs onto dpu banks since error input group of smfs like no specific type (data, const data (weights, bias ...)) found in given Smf group. Or unaligned smf info on data width, height or their stride, channel and channel group stride. Unaligned smfs cannot be aggregated on aggregation dimension.
XCOM_UNSUPPORT_NONLIEAR_TYPE	The nonlinear type is unsupported by DPU.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_PAD_KERNEL_SIZE_UNMATCH	The pad size is not correct comparing with the kernel size.
XCOM_DATA_DEPENDCY_MISSING	Generate code failed.
XCOM_MULTIPLE_WEIGHTS	There are more than one weights for some ops.
XCOM_MULTIPLE_BIAS	There are more than one bias for some ops.
XCOM_MULTIPLE_PRELU	There are more than oen prelu for some ops.
XCOM_TOO_MANY_INPUTS	There are too many inputs for the op.
XCOM_TENSOR_SHAPE_UNMATCH	Tensor shapes for some ops are not matching.
XCOM_UNSUPPORT_KERNEL_SIZE	The op's kernel size is not supported.
XCOM_CODE_GEN_ERROR	Code generation fail.
XCOM_UNSUPPORT_ROUND_MODE	The round mode is not supported.
XCOM_ADDITION_OVERFLOW	The addition is overflow.
XCOM_INT_COMPOSITION_INVALID_RANGE	The integer composition's output range constraint is invalid. Please contact us.
XCOM_TO_XINT_DATA_SIZE_UNMATCH	The input data vector's size is unmatching with the xint's bit width
XCOM_REVERT_XINT_DATA_SIZE_UNMATCH	The input data vector's size is unmatching with the xint's original bit width.
XCOM_DWCONV_PARAM_REORDER_SIZE_UNMATCH	size unmatching occurred during reordering the DepthwiseConv2d's parameter.
XCOM_CONV_PARAM_REORDER_SIZE_UNMATCH	size unmatching occured during reordering the Conv2d's parameter.
XCOM_AIE_CORE_NUM_MISMATCH	The config of aie core number mismatchs.
XCOM_AIE_TIMING_CONFIG_MISMATCH	The config inside aie timing calculating mismatchs.
XCOM_AIE_TILING_FAIL	There is not enough bank space inside AIE local memory for this tensor
XCOM_AIE_UNSUPPORTED_OP	Unsupported op for aie tiling
XCOM_PARTITIONENGINE_HINTS_ERROR	The partition engine's hints are invalid. Please contact us.
XCOM_PARSE_FAIL	Failed to parse structured data!
XCOM_PARTITION_REPEATED_REGISTRATION	Repeated checker registration for the op_type and arch_type in partition.
XCOM_UNREGISTED_SLNODE	Unregisted Slnode
XCOM_GET_CHANNEL_FAILED	xGet channel failed.
XCOM_GET_PACKET_ID_FAILED	xGet packet id failed.
XCOM_GET_PACKET_TYPE_FAILED	xGet packet type failed.
XCOM_GET_LOCK_ID_FAILED	xGet lock id failed.
XCOM_GET_BD_FAILED	xGet bd failed.
XCOM_SMF_SPEC_MISSING	no found such spec for the smf
XCOM_SMF_MISSING	no found such smf
XCOM_SMF_Y_SIZE_ERROR	smf y direction size overflow bank height with padding.
XCOM_SMF_C_SIZE_ERROR	channel direction smf need height = 1, width = 1, pad top and bottom = 0.
XCOM_SMF_COORDINATE_ERROR	smf on coordinate have error size or missing.
XCOM_SMF_CONCAT_ERROR	smf on concatenate have error size or missing.
XCOM_SMF_BIAS_ERROR	bias smf size error or missing.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_SMF_PARAM_ERROR	param smf size error or missing.
XCOM_SMF_TYPE_ERROR	error smf type.
XCOM_SMF_TENSOR_TYPE_ERROR	tensor type error while smf arrangment
XCOM_SMF_RESERVED_ERROR	smf dynamic size error or missing or addressing failed.
XCOM_SMF_BANK_MANAGEMENT_ERROR	error happens in bank management, error name addressing or missing.
XCOM_BANK_SMF_EXIST	smf name already exists in bank.
XCOM_BANK_ADDR_MISSING	bank addressing missing.
XCOM_BANK_ADDR_OVERFLOW	bank addressing overflow.
XCOM_BANK_ADDR_ERROR	bank block addr have error sequence or non-exist addressing.
XCOM_USER_FILE_NOT_EXISTS	The file does not exist
XCOM_USER_FILE_OPERATION_ERR	The file operation failed
XCOM_USER_INVALID_COMPILE_MODE	The compile mode is not supported now.
XCOM_USER_DIRECTORY_NOT_EXIST	The directory does not exist, create it first.
XCOM_USER_DIRECTORY_ALREADY_EXIST	The directory already exist, remove it first.
XCOM_USER_INVALID_CMD_PARAM	Invalid cmdline parameter
XCOM_USER_INVALID_TARGET	Invalid DPU target is given by cmdline.
XCOM_USER_INVALID_OUTPUT_OPS	The output ops user specified can't be found in the network. Please check the op names.
XCOM_USER_INVALID_OUTPUT_TENSORS	The output tensors user specified can't be found in the network. Please check the tensor names.
XCOM_USER_UNSUPPORTED_SYSTEM	The function is not supported by current operating system.
XCOM_PASS_DEPENDENCY_ERROR	Something wrong about pass dependency.
XCOM_PASS_UNREGISTERED	Pass has not been registered.
XCOM_PASS_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_PASS_TARGET_UNIMPLEMENTED	The target has not been implemented yet.
XCOM_PASS_GRAPH_ATTR_MISSING	Necessary attribution has not been set for the graph.
XCOM_PASS_OP_INVALID_ATTR	The parameter of the operator is invalid. Please check the input network.
XCOM_PASS_OP_ATTR_MISMATCH	The parameter of the operator is unexpected.
XCOM_PASS_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_PASS_OP_ATTR_MISSING	The requisite parameter is missing. Please check input network.
XCOM_PASS_TENSOR_SIZE_ERROR	Size of the tensor is invalid. Please check input network.
XCOM_PASS_TENSOR_SIZE_MISMATCH	Size mismatch between correlative tensors.
XCOM_PASS_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_COMGRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_COMGRAPH_OP_CONNECTION_MISSING	The connection is missing in the graph.
XCOM_COMGRAPH_OP_CONNECTION_INVALID	The connection between two specific op is unexpected. Check input network.
XCOM_COMGRAPH_ATTR_NOT_ASSIGNED	The requisite attribution of xcomgraph has not be assigned.
XCOM_COMGRAPH_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_COMGRAPH_SUBGRAPH_MISSING	The subgraph is missing.
XCOM_COMGRAPH_BANK_UNEXPECTED_STATE	Bank assignment is rejected by a particular subgraph.
XCOM_COMGRAPH_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_FRONTEND_SUBGRAPH_MISSING	The subgraph is missing.
XCOM_FRONTEND_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_FRONTEND_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_FRONTEND_GRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_FRONTEND_GRAPH_TEMPLATE_MISMATCH	Pattern mismatch between template and replacing process.
XCOM_FRONTEND_GRAPH_LEVEL_MISMATCH	A unsuitable level of graph is given for current function.
XCOM_FRONTEND_GRAPH_ATTR_MISSING	Necessary attribution has not been set for the graph.
XCOM_FRONTEND_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_FRONTEND_OP_INVALID_ATTR	The parameter of the operator is invalid. Please check the input network.
XCOM_FRONTEND_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_FRONTEND_OP_UNSUPPORTED_BLOB_NUMBER	Blob number of the operator is unsupported by target DPU.
XCOM_FRONTEND_OP_UNSUPPORTED_ATTR	An attribution of the operator is unsupported by target DPU.
XCOM_FRONTEND_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_FRONTEND_OP_UNSUPPORTED_NONLIEAR	The nonlinear(or activation) is unsupported by target DPU
XCOM_FRONTEND_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_FRONTEND_OP_ATTR_MISMATCH	The parameter of the operator is unexpected.
XCOM_FRONTEND_QUANT_ATTR_MISSING	Quantizing information for the op is mission
XCOM_FRONTEND_QUANT_ATTR_OUT_OF_RANGE	The shiftbit for quantizing is out of range, that the range is restricted by target DPU.
XCOM_FRONTEND_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_FRONTEND_TENSOR_SHAPE_MISMATCH	Shape mismatch between 2 correlative tensors.
XCOM_FRONTEND_TENSOR_SIZE_MISMATCH	Size mismatch between 2 correlative tensors.
XCOM_FRONTEND_DATA_TYPE_MISMATCH	Data type mismatch between 2 correlative tensors.
XCOM_FRONTEND_BITWIDTH_MISMATCH	The bit width of the tensor is unexpected.
XCOM_GENINST_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_GENINST_INVALID_VALUE	A invalid value is given for specific function.
XCOM_GENINST_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_GENINST_OP_UNSUPPORTED	The operator has not been implemented by target dpu.
XCOM_GENINST_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_GENINST_OP_UNSUPPORTED_NONLIEAR	The nonlinear(or activation) is unsupported by target DPU
XCOM_GENINST_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_GENINST_TARGET_UNIMPLEMENTED	The target has not been implemented yet.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_GENINST_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_GENINST_GRAPH_TEMPLATE_MISMATCH	Pattern mismatch between template and replacing process.
XCOM_GENINST_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_GENINST_TILING_FAIL	Failed to get a available tiling scheme.
XCOM_GENINST_CODE_GEN_ERROR	Code generation fail.
XCOM_OPFACTORY_ILLEGAL_NAME	The object name is illegal.
XCOM_OPFACTORY_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_OPFACTORY_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_OPFACTORY_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_OPFACTORY_OP_UNSUPPORTED_BLOB_NUMBER	Blob number of the operator is unsupported by target DPU.
XCOM_OPFACTORY_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_OPFACTORY_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_OPFACTORY_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_OPTIMIZE_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_OPTIMIZE_GRAPH_OPERATION_FAILED	A graphic problem has caused by graphic operation.
XCOM_OPTIMIZE_OP_UNSUPPORTED_ATTR	An attribution of the operator is unsupported by target DPU.
XCOM_OPTIMIZE_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_OPTIMIZE_TENSOR_SHAPE_INVALID	Tensor shape is invalid. Please check the input network.
XCOM_OPTIMIZE_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_OPTIMIZE_OP_ATTR_MISMATCH	The parameter of the operator is unexpected.
XCOM_UTIL_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_UTIL_OPERATION_DENIED	The operation is not permitted.
XCOM_UTIL_PARSE_FAIL	Failed to parse structured data.
XCOM_UTIL_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_UTIL_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_UTIL_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_UTIL_TENSOR_SHAPE_INVALID	Tensor shape is invalid. Please check the input network.
XCOM_UTIL_DATA_TYPE_INVALID	Data type is invalid for current context.
XCOM_UTIL_ATTR_OUT_OF_RANGE	The value is out of range.
XCOM_UTIL_INVALID_VALUE	A invalid value is given for specific function.
XCOM_UTIL_INVALID_VALUE_RANGE	The data range is illegal.
XCOM_UTIL_ADDITION_OVERFLOW	The addition is overflow.
XCOM_UTIL_DATA_SIZE_MISMATCH	Size mismatch between 2 data chunk.
XCOM_UTIL_BANK_ALLOC_FAILED	Failed to get a available scheme of bank assignment.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XCOM_BANKASSIGN_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_BANKASSIGN_INVALID_ITEM	The item is not available yet.
XCOM_BANKASSIGN_OUT_OF_BANK_SIZE	Insufficiency of bank size, input model might be too large.
XCOM_BANKASSIGN_TARGET_ENGINE_UNREGISTERED	The engine is not registered for dpu target.
XCOM_BANKASSIGN_INVALID_VALUE	A invalid value is given for specific function.
XCOM_BANKASSIGN_TARGET_BANK_ERR	No output bank associated with current op within target DPU.
XCOM_BANKASSIGN_BANK_UNEXPECTED_STATE	Bank space manager might be in disorder.
XCOM_BANKASSIGN_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_BANKASSIGN_BANK_INFO_MISMATCH	Bank requirement mismatch between 2 ops.
XCOM_PARTITION_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_PARTITION_NULL_POINTER	Accessing of uninitialized object or pointer
XCOM_DDRALLOC_NULL_POINTER	Accessing of uninitialized object or pointer.
XCOM_DDRALLOC_UNEXPECTED_STATE	A impossible state occurs. There might be a logical error of programming.
XCOM_DDRALLOC_INVALID_ITEM	Failed to get a available scheme of bank assignment.
XCOM_DDRALLOC_ITEM_UNDEFINED	The item has not be defined.
XCOM_DDRALLOC_DATA_SIZE_MISMATCH	Size mismatch between 2 data chunk.
XCOM_DDRALLOC_MEM_ACCESS_OVERFLOW	Memory access overflow.
XCOM_DDRALLOC_OUT_OF_MEM	DDR space is not enough for current model.
XCOM_DDRALLOC_FEATURE_UNIMPLEMENT	The feature of DDR allocating optimization is not implemented.
XCOM_DDRALLOC_OPERATION_DENIED	The operation is not permitted.
XCOM_DDRALLOC_TARGET_UNIMPLEMENTED	The target has not been implemented yet.
XCOM_DDRALLOC_ASSIGNMENT_STATE_ERROR	The state of DDR allocation is unexpected.
XCOM_DDRALLOC_PARAM_SPACE_INITIALIZTION_SIZE_ERROR	DDR parameter space initialization size error. Please contact us.
XCOM_DDRALLOC_GRAPH_LEVEL_MISMATCH	A unsuitable level of graph is given for current function.
XCOM_DDRALLOC_GRAPH_OP_MISSING	The op is missing in specific graph.
XCOM_DDRALLOC_GRAPH_INVALID_STRUCTURE	There is a unexpected pattern in the graph. Please check the input network.
XCOM_DDRALLOC_OP_UNSUPPORTED	The operator has not been implemented by target DPU.
XCOM_DDRALLOC_OP_UNSUPPORTED_MODE	The mode has not been implemented for specific operator.
XCOM_DDRALLOC_OP_INVALID_BLOB_NUMBER	Blob number of the operator is invalid. Please check the input network.
XCOM_DDRALLOC_OP_UNSUPPORTED_NONLIEAR_TYPE	The nonlinear(or activation) is unsupported by target DPU
XCOM_DDRALLOC_TENSOR_DIMS_MISMATCH	Dimensions of the tensor is unexpected.
XCOM_DDRALLOC_TENSOR_SHAPE_MISMATCH	Shape mismatch between 2 correlative tensors.
XCOM_DDRALLOC_TENSOR_SIZE_ERROR	Size of the tensor is invalid. Please check input network.
XCOM_DDRALLOC_TENSOR_SIZE_MISMATCH	Size mismatch between 2 correlative tensors.

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
VAILIB_DPU_TASK_NOT_FIND	Model files not find
VAILIB_DPU_TASK_OPEN_ERROR	Open file failed
VAILIB_DPU_TASK_CONFIG_PARSE_ERROR	Parse model config file failed
VAILIB_DPU_TASK_TENSORS_EMPTY	Runner has no input tensors
VAILIB_DPU_TASK_SUBGRAPHS_EMPTY	Runner has no subgraphs
VAILIB_CPU_RUNNER_OPEN_LIB_ERROR	dlopen can not open lib
VAILIB_CPU_RUNNER_LOAD_LIB_SYM_ERROR	dlsym load symbol error
VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_FIND	Can not find tensor buffer with this name
VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_CONTINOUS	Tensor buffer not continuous
VAILIB_CPU_RUNNER_READ_FILE_ERROR	Fail to read file
VAILIB_CPU_RUNNER_WRITE_FILE_ERROR	Fail to write file
VAILIB_CPU_RUNNER_CPU_OP_NOT_FIND	Can not find op with this name
VAILIB_GRAPH_RUNNER_NOT_FIND	GraphTask can not find tensor or tensor buffer
VAILIB_GRAPH_RUNNER_DPU_BATCH_ERROR	GraphTask get dpu batch not equal
VAILIB_GRAPH_RUNNER_NOT_SUPPORT	The function or value are not supported in graph runner
VAILIB_GRAPH_RUNNER_NOT_OVERRIDE	The function has not been overridden
VAILIB_MATH_NOT_SUPPORT	The function or value are not supported in vai-math
VAILIB_MATH_FIX_POS_ERROR	Softmax table not support the fix position value
VAILIB_MODEL_CONFIG_NOT_FIND	Model config info not find
VAILIB_MODEL_CONFIG_OPEN_ERROR	Model config file or directory open error
VAILIB_MODEL_CONFIG_CONFIG_PARSE_ERROR	Model config file parse error
VAILIB_BENCHMARK_LIST_EMPTY	Can not found images. List of images are empty
VAILIB_DEMO_CANVAS_ERROR	Canvas image size is too small
VAILIB_DEMO_GST_ERROR	Failed to open gstreamer
VAILIB_DEMO_IMAGE_LOAD_ERROR	Failed to load image
VAILIB_DEMO_OUT_OF_BOUNDARY	Gui rects out of boundary
VAILIB_DEMO_VIDEO_OPEN_ERROR	Cannot open video stream
VART_OPEN_DEVICE_FAIL	Cannot open device
VART_LOAD_XCLBIN_FAIL	Bitstream download failed
VART_LOCK_DEVICE_FAIL	Cannot lock device
VART_FUNC_NOT_SUPPORT	Function not support
VART_XMODEL_ERROR	XMODEL error
VART_GRAPH_ERROR	Graph error
VART_TENSOR_INFO_ERROR	Tensor info error
VART_DPU_INFO_ERROR	DPU info error
VART_SYSTEM_ERROR	File system error
VART_DEVICE_BUSY	Device busy
VART_DEVICE_MISMATCH	Device mismatch

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
VART_DPU_ALLOC_ERROR	DPU allocate error
VART_VERSION_MISMATCH	Version mismatch
VART_OUT_OF_RANGE	Array index out of range
VART_SIZE_MISMATCH	Array size not match
VART_NULL_PTR	Nullptr
VART_XRT_NULL_PTR	Nullptr
VART_XRT_DEVICE_BUSY	Device busy
VART_XRT_READ_ERROR	Read error
VART_XRT_READ_CU_ERROR	Read cu fatal
VART_XRT_FUNC_FAULT	XRT function fault
VART_XRT_DEVICE_AVAILABLE_ERROR	No devices available
VART_XRT_CU_AVAILABLE_ERROR	No CU available
VART_XRT_OPEN_CONTEXT_ERROR	xclOpenContext failed
VART_XRM_CREATE_CONTEXT_ERROR	failed to create XRM context
VART_XRM_CONNECT_ERROR	Failed to connect to XRM
VART_XRM_ACQUIRE_CU_ERROR	Could not acquire CU
VART_DEVICE_BUFFER_ALLOC_ERROR	Cannot alloc device buffer -- unknown datatype
VART_XCLBIN_READ_ERROR	Failed to open xclbin file for reading
VART_XCLBIN_DOWNLOAD_ERROR	Bitstream download failed !
VART_CONTROLLER_VIR_MEMORY_ALLOC_ERROR	not enough virtual space on host
VART_VERSION_MISMATCH_ERROR	subgraph's version is mismatch with xclbin
VART_CONTROLLER_DUMP_FOLDER_CREATE_ERROR	Create dump folder error
VART_CONTROLLER_DUMP_SUBFOLDER_CREATE_ERROR	Create sub dump folder error
VART_DEVICE_MEMORY_ALLOC_ERROR	Device memory not enough, alloc fail
VART_TENSOR_BUFFER_CREATE_ERROR	TensorBuffer create fail
VART_TENSOR_BUFFER_INVALID	invalid tensorbuffer input or output
VART_DPU_EXEC_ERROR	DPU run fail
VART_DPU_TIMEOUT_ERROR	DPU timeout
VART_CONTROLLER_DUMP_ERROR	dump failed
VART_XCLBIN_PATH_INVALID	xclbinPath is not set, consider setting XLNX_VART_FIRMWARE.
VART_GRAPH_FINGERPRINT_ERROR	no hardware info in subgraph
VART_TENSOR_BUFFER_CHECK_ERROR	TensorBuffer size less than offset, input shape invalid
VART_TENSOR_BUFFER_DIMS_ERROR	input dims shape is invalid
XIR_ACCESS_ADDRESS_OVERFLOW	The address you try to access does not exsit!
XIR_ADD_OP_FAIL	Failed to add an op!
XIR_FILE_NOT_EXIST	File doesn’t exist!
XIR_INTERNAL_ERROR	it is an internal bug supposed never happen

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XIR_INVALID_ARG_OCCUR	Invalid arg occurrence!
XIR_INVALID_ATTR_DEF	Invalid attribute definition!
XIR_INVALID_ATTR_OCCUR	Invalid attr occurrence!
XIR_INVALID_DATA_TYPE	The data type is invalid.
XIR_MEANINGLESS_VALUE	The value you set for this parameter makes no sense.
XIR_MULTI_DEFINED_OP	Multiple definition of OP!
XIR_MULTI_DEFINED_TENSOR	Multiple definition of Tensor!
XIR_MULTI_REGISTERED_ARG	Multiple registration of argument!
XIR_MULTI_REGISTERED_ATTR	Multiple registration of attribute!
XIR_MULTI_REGISTERED_EXPANDED_ATTR	Multiple registration of static attr!
XIR_MULTI_REGISTERED_OP	Multiple registration of operator!
XIR_OPERATION_FAILED	Fail to execute command!
XIR_OP_DEF_SHAPE_HINT_MISSING	A shape hint is required by the op definition
XIR_OP_NAME_CONFLICT	There are at least two ops assigned the same name, check the ops' name.
XIR_OUT_OF_RANGE	idx out of range!
XIR_PROTECTED_MEMORY	The content in protected momory for tensor can not be modified!
XIR_READ_PB_FAILURE	failed to read pb file
XIR_REMOVE_OP_FAIL	Failed to remove an op!
XIR_SHAPE_UNMATCH	The shape is unmatching.
XIR_SUBGRAPH_ALREADY_CREATED_ROOT	Already created root subgraph for the graph!
XIR_SUBGRAPH_CREATE_CHILDREN_FOR_NONLEAF	
XIR_SUBGRAPH_HAS_CYCLE	Children from a same subgraph depend each other!
XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONCHILD	Cannot merge subgraphs which are not children!
XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONLEAF	Cannot merge subgraphs which are not leaves!
XIR_UNDEFINED_ATTR	Undefined attribute!
XIR_UNDEFINED_INPUT_ARG	Undefined input arg!
XIR_UNDEFINED_OP	Access undefined OP!
XIR_UNEQUIVALENT_ATTRIBUTE	These two attributes/parameters are not equivalent.
XIR_UNEXPECTED_VALUE	Unexpected value!
XIR_UNKNOWNTYPE_TENSOR	The datatype of this tensor is not specified.
XIR_UNREGISTERED_ARG	Unregistered argument!
XIR_UNREGISTERED_ATTR	Unregistered attribute!
XIR_UNREGISTERED_OP	Unregistered operator!
XIR_UNSUPPORTED_ROUND_MODE	unsupported round mode.
XIR_UNSUPPORTED_TYPE	unsupported data type for attr value
XIR_VALUE_UNMATCH	Value unmatch!

Table 54: **Error Codes** *(cont'd)*

Error Code ID	Error Message
XIR_WRITE_PB_FAILURE	failed to write pb file
XIR_XIR_UNDEFINED_OPERATION	Undefined operation!
TARGET_EXPLORER_XCLBIN_ERROR	No xclbin specified
TARGET_EXPLORER_XCLBIN_ENV_ERROR	DPU xclbin path specified by 'XLNX_VART_FIRMWARE' not exist, check!
TARGET_EXPLORER_XCLBIN_ENV_VAL_ERROR	'XLNX_VART_FIRMWARE' need to be specified like /path/to/xxx.xclbin, check!
TARGET_EXPLORER_SYS_DEVICE_CHECK_ERROR	The system has no device
TARGET_EXPLORER_XCLBIN_SET_ERROR	xclbinPath is not set, consider setting XLNX_VART_FIRMWARE.
TARGET_EXPLORER_NO_DPU_ERROR	xclbin is not for DPU, can't find DPU kernel in xclbin
TARGET_EXPLORER_BATCH_ERROR	Only support multiple DPU cores with same batch and fingerprint
TARGET_EXPLORER_DEVICE_CHECK_ERROR	No device available for current xclbin
TARGET_FACTORY_INVALID_ARCH	Invalid target arch!
TARGET_FACTORY_INVALID_FEATURE_CODE	Invalid target feature code!
TARGET_FACTORY_INVALID_ISA_VERSION	Invalid target ISA version!
TARGET_FACTORY_INVALID_TYPE	Invalid target type!
TARGET_FACTORY_MULTI_REGISTERED_TARGET	Multiple registration of target!
TARGET_FACTORY_PARSE_TARGET_FAIL	Fail to parse target from prototxt!
TARGET_FACTORY_UNREGISTERED_TARGET	Unregistered target!

Appendix B

Additional Resources and Legal Notices

Finding Additional Documentation

Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to https://docs.xilinx.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD 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.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

1. Release Notes and Known Issues - https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/release_notes.html
2. *Vitis AI Library User Guide* (UG1354)
3. *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide* (PG338)
4. *DPUCAHX8H for Convolutional Neural Networks Product Guide* (PG367)
5. *DPUCVDX8G for Versal Adaptive SoCs Product Guide* (PG389)
6. *DPUCV2DX8G for Versal Adaptive SoCs Product Guide*(PG425)
7. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
8. *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393)
9. *PetaLinux Tools Documentation: Reference Guide* (UG1144)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
09/28/2023 Version 3.5	
N/A	Editorial updates only.
06/29/2023 Version 3.5	
Chapter 2: Optimizing the Model	Added the chapter.
Chapter 3: Quantizing the Model	Added: • Quantization Strategy Configuration. • Using the New Data Format. • Inference of Quantized Model. • def export_onnx_model(self, output_dir, verbose)-New. • ONNX Runtime Version (vai_q_onnx).

Section	Revision Summary
Chapter 5: Deploying and Running the Model	• Added Visualization With OnBoard. • Added WeGO-Torch C++ Classes and APIs. • Updated Programming with VOE.
Entire document	Made editorial and technical updates.
02/24/2023 Version 3.0	
General Updates	Updated the links.
01/12/2023 Version 3.0	
General Updates	• Made technical updates for the new release • Editorial updates
06/15/2022 Version 2.5	
General Updates	• Made technical updates for the new release • Editorial updates
01/20/2022 Version 2.0	
General Updates	• Updated minor technical details • Updated the supported card versions • Editorial updates
Deep-Learning Processor Unit	Updated to include the Versal DPUs
vitis_quantize.VitisQuantizer.get_qat_model	Updated the argument descriptions
Supported Operators and DPU Limitations	Updated the supported operators table
vaitrace Usage	Updated command line usage text
12/13/2021 Version 1.4.1	
Chapter 3: Quantizing the Model	Updated vai_q_tensorflow2 Supported Operations and APIs section
07/22/2021 Version 1.4	
Chapter 1: Vitis AI Overview	Added Versal AI Core Series: DPUCVDX8G section
TensorFlow 2.x Version (vai_q_tensorflow2)	Added vai_q_tensorflow2 Quantization Aware Training, Quantizing with Custom Layers, and vai_q_tensorflow2 Usage sections
PyTorch Version (vai_q_pytorch)	Updated vai_q_pytorch QAT
Chapter 5: Deploying and Running the Model	Updated Apache TVM, Microsoft ONNX Runtime, and TensorFlow Lite
Chapter 6: Profiling the Model	Added Text Summary Updated vaitrace Usage
02/03/2021 Version 1.3	
Entire document	Updated links
12/17/2020 Version 1.3	
Entire document	Minor changes
Deep-Learning Processor Unit	Added new topics: Alveo U200/U250 Card: DPUCADF8H and Versal AI Core Series: DPUCVDX8G.
TensorFlow 2.x Version (vai_q_tensorflow2)	Added new section
PyTorch Version (vai_q_pytorch)	Added new topics: Module Partial Quantization, vai_q_pytorch Fast fine-tuning, and vai_q_pytorch QAT.
Chapter 4: Compiling the Model	Added new section: Compiling with an XIR-based Toolchain.

Section	Revision Summary
Integrating the DPU into Custom Platforms	Added new chapter.
VART Programming APIs	Added new section: VART APIs.
07/21/2020 Version 1.2	
Entire document	Minor changes
07/07/2020 Version 1.2	
Entire document	• Added Vitis AI Profiler topic. • Added Vitis AI unified API introduction.
DPU Naming	Added new topic
Getting Started	Updated the chapter
03/23/2020 Version 1.1	
DPUCAHX8H	Added new topic
Entire document	Added contents for Alveo U50 support, U50 DPUV3 enablement, including compiler usage and model deployment description.

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

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