Model Compiler Suite for Aries

Developers Guide

version 0.8.1

December 8, 2023

Important Notice

MOBILINT, Inc. reserves the right to make changes to the information in this publication at any time without prior notice. All information provided is for reference purpose only. MOBILINT assumes no responsibility for possible errors or omissions, or for any consequences resulting from the use of the information contained herein.

This publication on its own does not convey any license, either express or implied, relating to any MOBILINT and/or third-party products, under the intellectual property rights of MOBILINT and/or any third parties.

MOBILINT makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does MOBILINT assume any liability arising out of the application or use of any product or circuit and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Customers are responsible for their own products and applications. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by the customer's technical experts.

MOBILINT products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the MOBILINT product could reasonably be expected to create a situation where personal injury or death may occur.

Customers acknowledge and agree that they are solely responsible to meet all other legal and regulatory requirements regarding their applications using MOBILINT products notwithstanding any information provided in this publication. Customer shall indemnify and hold MOBILINT and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim (including but not limited to personal injury or death) that may be associated with such unintended, unauthorized and/or illegal use.

WARNING No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electric or mechanical, by photocopying, recording, or otherwise, without the prior written consent of MOBILINT. This publication is intended for use by designated recipients only. This publication contains confidential information (including trade secrets) of MOBILINT protected by Competition Law, Trade Secrets Protection Act and other related laws, and therefore may not be, in part or in whole, directly or indirectly publicized, distributed, photocopied or used (including in a posting on the Internet where unspecified access is possible) by any unauthorized third party. MOBILINT reserves its right to take any and all measures both in equity and law available to it and claim full damages against any party that misappropriates MOBILINT's trade secrets and/or confidential information.

Document Revision History

Doc Revision Number	Date	Description
0.8.1	December 8, 2023	Revised for v0.8.1
0.8.0	November 2, 2023	Revised for v0.8.0
0.7.12	September 12, 2023	Revised for v0.7.12
0.7.11	August 31, 2023	Revised for v0.7.11
0.7.10	August 11, 2023	Revised for v0.7.10
0.7.9	August 11, 2023	Revised for v0.7.9
0.7.8	August 8, 2023	Revised for v0.7.8
0.7.7	June 30, 2023	Revised for v0.7.7
0.7	March 23, 2023	Revised for v0.7
0.6	August 10, 2022	Revised for v0.6
0.5	July 1, 2022	Revised for v0.5
0.4	February 23, 2022	Revised for v0.4
0.3	February 5, 2022	Revised for v0.3
0.2	December 1, 2021	Revised for v0.2

Table of Contents

List of Tables

List of Figures

1. Introduction

Mobilint® Model Compiler (i.e., Compiler) is a tool that converts models from deep learning frameworks (ONNX, PyTorch, Keras, TensorFlow, etc...) into Mobilint® Model eXeCUtable (i.e., MXQ), a format executable by Mobilint® Neural Processing Unit (NPU). This is the manual for the **qubee**, Mobilint's SDK. In this manual, you can learn how to use it, what frameworks it supports, etc. A set of functions you can use to interact with the SDK will be given below.

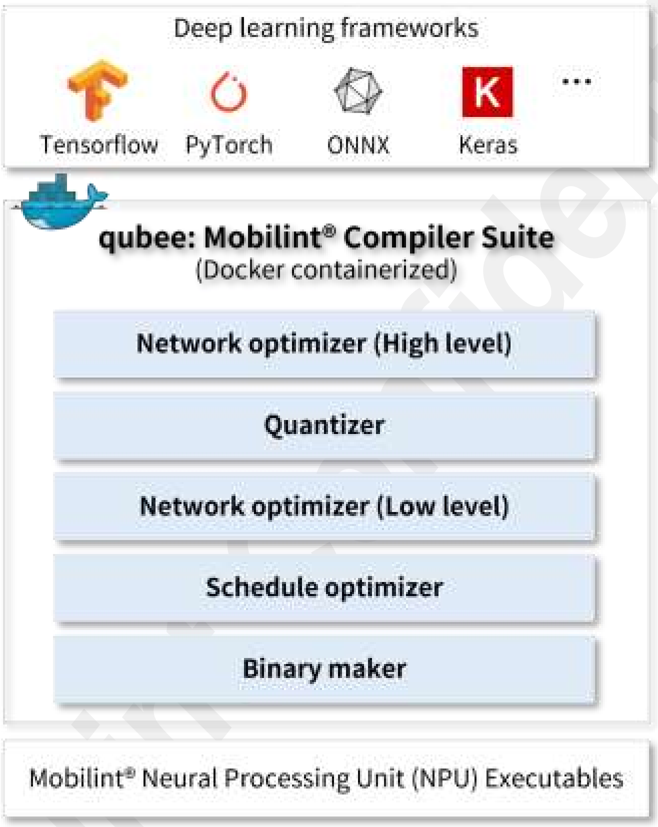

Figure 1-1. SDK Components

Inputs to qubee are a trained deep learning model, its input shape, and calibration data. It will return MXQ (compiled model) as an output.

Figure 1-2. Input and output of qubee

2. Changelog

2.1 qubee v0.8.1 (December 2023)

2.2 qubee v0.8.0 (November 2023)

2.3 qubee v0.7.12 (September 2023)

2.4 qubee v0.7.11 (August 2023)

- API
  - Torchscript Backend

2.5 qubee v0.7.10 (August 2023)

2.6 qubee v0.7.9 (August 2023)

2.7 qubee v0.7.8 (August 2023)

2.8 qubee v0.7.7 (June 2023)

- API
  - CPU offloading (beta version)
- Improve CPU efficiency
- Support more operations
- Docker
  - torch: 1.10.1 -> 1.13.0
  - tensorflow: 2.3.0 -> 2.9.0
  - onnx:1.11.0 -> 1.12.0

2.9 qubee v0.7 (March 2023)

- Multi-channel quantization
- Support more operations
- API
  - Calibration dataset
- CPU Offloading (Beta Version)

2.10 qubee v0.6 (August 2022)

- Minor updates

2.11 qubee v0.5 (July 2022)

- Docker

Conda -> Virtualenv

Python: 3.7.7 -> 3.8.10

torch: 1.8.1 -> 1.10.1

tensorflow: 1.15.0 -> 2.3.0

onnx:1.6.0 -> 1.11.0

Parser

Code refactoring

API

Enable saving sample inference results (inputs and outputs)

2.12 qubee v0.4 (February 2022)

Optimizer

Minor updates in fusing reshape

2.13 qubee v0.3 (February 2022)

Parser

Identify preprocess and postprocess of the model

Exclude preprocess and postprocess if they are unsupported by the NPU

API

Simulate integer inference in Python API

2.14 qubee v0.2 (December 2021)

First release

3. Installation

3.1 System requirements

We recommend to use NVIDIA GPU for faster compile wtih qubee, but it is not necessary. Currently, CPU version qubee is also supported.

3.1.1 Reference System

```
Ubuntu 20.04.4 LTS
NVIDIA Graphics Driver 525.147.05
```

3.1.2 Recommended Packages

```
NVIDIA Graphics Driver 450.80.02 or Above
Docker
nvidia-docker
```

3.2 SDK Installation

We recommend installing qubee on the mobilint docker container.

(Docker image: mobilint/qbcompiler:v0.8.1, https://hub.docker.com/r/mobilint/qbcompiler)

3.2.1 Building Docker Image

Run the following commands to build the docker image.

```
$ # Docker image download
$ docker pull mobilint/qbcompiler:v0.8.1
$ # Make a docker container (if needed)
$ # mkdir {WORKING DIRCTORY}
$ cd {WORKING DIRCTORY}
$ docker run -it --gpus all --ipc=host --name mxq_compiler -v $(pwd):/workspace
mobilint/qbcompiler:v0.8.1
```

(Recommended) If the trained models and calibration dataset are stored in different directories, you can mount them to the docker container as follows:

```
$ docker run -it --gpus all --ipc=host --name mxq_compiler -v $(pwd):/workspace -v {PATH TO MODEL
DIR}:/models -v {PATH TO CALIBRATION DATASET DIR}:/datasets mobilint/qbcompiler:v0.8.1
```

(Option, not available yet) Build the docker image for WSL2

```
$ # Docker image download
$ docker pull mobilint/qbcompiler:v0.8.1-wsl
$ # Make a docker container
$ cd {WORKING DIRCTORY}
$ docker run -it --gpus all --ipc=host --name mxq_compiler -v $(pwd):/data
mobilint/qbcompiler:v0.8.1-wsl
```

3.2.2 Installation of qubee

qubee compiler packages are available in [Mobilint® Software Development Kit (SDK)]

Run the following commands to install qubee on the docker container.

```
$ # Download qubee-0.8.1-py3-none-any.whl file
$ # Copy qubee whl file to Docker
$ docker cp {Path to qubee-0.8.1-py3-none-any.whl} mxq_compiler:/
$ # Start docker
$ docker start mxq_compiler
$ # Attach docker
$ docker exec -it mxq_compiler /bin/bash
$ # Install qubee
$ cd /
$ python -m pip install qubee-0.8.1-py3-none-any.whl
```

(Option, for WSL2) Run the following commands to install qubee on the docker container.

```
$ # Download qubee-0.8.1_wsl-py3-none-any.whl file
$ # Copy qubee whl file to Docker
$ docker cp {Path to qubee-0.8.1_wsl-py3-none-any.whl} mxq_compiler:/
$ # Start docker
$ docker start mxq_compiler
$ # Attach docker
$ docker exec -it mxq_compiler /bin/bash
$ # Install qubee
$ cd /
$ python -m pip install qubee-0.8.1_wsl-py3-none-any.whl
```

4. Tutorials

The tutorials below go through the steps for preparing the calibration dataset, model compiling, and inference.

4.1 Preparing Calibration Data

To compile the model, you should prepare the calibration dataset (the pre-processed inputs for the model) for quantization. There are three ways to make the calibration dataset as follows:

(i) Pre-process the raw calibration dataset and save it as numpy tensors.

(ii) Utilize a pre-processing configuration YAML file (only for images with **uniform format**).

(iii) Use a manually defined pre-processing function (only for images with **uniform format**).

(iv) Use Mobilint® Processor (will be available soon)

Important The process of making a calibration dataset may vary depending on whether you compile the model for CPU offloading or not. Currently, qubee compiles the model without CPU offloading by default. In this scenario, the pre-processed input shape should be in the format (H, W, C). On the other hand, when CPU offloading is employed, the pre-processed input shape should match the input shape that the original model takes.

4.1.1 Pre-process raw calibration dataset and save it as numpy tensors

You can save the pre-processed calibration dataset as numpy tensors with your custom pre-processing function and use them to compile the model.

An example code is shown below. The following code assumes that we hold an image folder consisting of 1000 randomly selected images from the Imagenet dataset for calibration prepared in directory `/datasets/imagenet/cali_1000`.

```
import os
import numpy as np
import cv2

def get_img_paths_from_dir(dir_path: str, img_ext = ["jpg", "jpeg", "png"]):
    assert os.path.exists(dir_path)
    candidates = os.listdir(dir_path)
    return [os.path.join(dir_path, y) for y in candidates if any([y.lower().endswith(e) for e in
    img_ext])]

def pre_process(img_path: str, target_h: int, target_w: int):
    img = cv2.imread(img_path, cv2.IMREAD_COLOR)
    resized_img = cv2.resize(img, dsize=(target_w, target_h)).astype(np.float32)
    return resized_img

if __name__ == "__main__":
    img_dir = "/datasets/imagenet/cali_1000"
    save_dir = "/workspace/calibration/custom_single_input"
    target_h, target_w = 224, 224
    os.makedirs(save_dir, exist_ok=True)
    img_paths = get_img_paths_from_dir(img_dir)
    for i, img_path in enumerate(img_paths):
```

```
        fname = f"{i}".zfill(3) + ".npy"
        fpath = os.path.join(save_dir, fname)
        x = pre_process(img_path, target_h, target_w)
        np.save(fpath, x)
```

The above results in a directory containing the pre-processed calibration dataset (numpy tensors of shape (224,224, 3)), located at `/workspace/calibration/custom_single_input`.

4.1.2 Use a pre-processing configuration YAML file

Image pre-processing techniques such as resizing, cropping, and normalization are often applied in machine vision tasks. Users can construct a pre-processing configuration using a YAML file and prepare the calibration dataset via the API provided by qubee, *make_calib*. Please be aware that this method can only be employed when the raw data is an image. An example code is shown below. The following code assumes that images for calibration are prepared in the directory `/workspace/cali_1000`.

```
from qubee import make_calib
make_calib(
    args_pre="/workspace/mobilenet_v2.yaml", # path to pre-processing configuration yaml file
    data_dir="/datasets/imagenet/cali_1000", # path to folder of original calibration data files such as images
    save_dir="/workspace/calibration/", # path to folder to save pre-proceessed calibration data files
    save_name="mobilenet_v2", # tag for the generated calibration dataset
    max_size=50 # Maximum number of data to use for calibration
)
```

```
# mobilenet_v2.yaml
Datatype: Image
GetImage:
    to_float32: false
    channel_order: RGB

Pre-Order: [ResizeTorch, CenterCrop, Normalize, SetOrder]
Pre-processing:
    ResizeTorch:
        size: [256, 256]
        interpolation: bilinear
    CenterCrop:
        size: [224, 224]
    Normalize:
        mean: [0.485, 0.456, 0.406]
        std: [0.229, 0.224, 0.225]
        to_float_div255: true
    SetOrder:
        shape: HWC
```

The above results are in a directory containing the pre-processed calibration dataset (numpy tensors), located at `/workspace/calibration/mobilenet_v2`. In addition, a calibration meta txt file containing the paths to the pre-processed numpy files is created, named `/workspace/calibration/mobilenet_v2.txt`.

Remark The sample dataset for calibration should be composed of images with the same format. If some are in color images and others are in grayscale images, the calibration dataset will not be created properly.

4.1.3 Use a manually defined pre-processing function

You can use your pre-processing function to make the calibration dataset via the API provided by qubee, *make_calib_man*. In this case, the pre-processing function should take the image path as input and return a numpy tensor. An example of the code is shown below. The following code assumes that images for calibration are prepared in the directory `/datasets/imagenet/cali_1000`.

```
import torch
import numpy as np
from PIL import Image
import torchvision.transforms.functional as F
from qubee import make_calib_man

def preprocess_resnet50(img_path: str):
    img = Image.open(img_path)
    resize_size=(232, 232)
    crop_size=(224, 224)
    mean=[0.485, 0.456, 0.406]
    std=[0.229, 0.224, 0.225]
    out = F.pil_to_tensor(img)
    out = F.resize(out, size=resize_size)
    out = F.center_crop(out, output_size=crop_size)
    out = out.to(torch.float, copy=False) / 255.
    out = F.normalize(out, mean, std)
    out = np.transpose(out.numpy(), axes=[1, 2, 0])
    return out

make_calib_man(
    pre_ftn=preprocess_resnet50, # callable function to pre-process the calibration data
    data_dir="/datasets/imagenet/cali_1000", # path to folder of original calibration data files
such as images
    save_dir="/workspace/calibration/", # path to folder to save pre-proceessed calibration data
files
    save_name="resnet50", # tag for the generated calibration dataset
    max_size=50 # Maximum number of data to use for calibration
)
```

The above results are in a directory containing the pre-processed calibration dataset (numpy tensors), located at `/workspace/sample/calibration/resnet50`. In addition, a calibration meta txt file containing the paths to the pre-processed numpy files is created, named `/workspace/sample/calibration/resnet50.txt`.

Remark Unless the custom pre-processing function contains proper exception handling, the sample dataset for calibration should be composed of images with the same format. Like the previous method, the calibration dataset will not be created properly if some are in color images and others are in grayscale images.

4.2 Compiling ONNX Models

ONNX model is recommended to use for compiling the trained model. With simple code, the ONNX model can be directly parsed to obtain Mobilint IR. example code is shown below. The following code assumes that the calibration dataset and the model are prepared in the directory `/workspace/calibration/resnet50` and `/workspace/resnet50.onnx`, respectively.

```
""" Compile ONNX model"""
from qubee import mxq_compile
onnx_model_path = "/workspace/resnet50.onnx"
```

```
calib_data_path = "/workspace/calibration/resnet50"
# calib_data_path can be replaced with the path to the calibration meta file such as
"/workspace/calibration/resnet50.txt"

mxq_compile(
    model=onnx_model_path,
    calib_data_path=calib_data_path,
    save_path="resnet50.mxq",
    backend="onnx"
)
```

4.3 Compiling PyTorch Models

PyTorch models can be compiled in two different ways. The first approach involves converting the PyTorch model to ONNX, which is then further converted into Mobilint IR. The second approach involves converting the PyTorch model to Torchscript, which is then further converted into Mobilint IR. Once the model is converted to Mobilint IR, then it is be compiled into MXQ. Examples of the code are shown below. The following codes assume that the calibration dataset is prepared in directory `/workspace/calibration/resnet50`.

```
""" Compile PyTorch model, first way """
from qubee import mxq_compile
from qubee.utils import convert_pytorch_to_onnx
import torchvision

input_shape = (224, 224, 3)
calib_data_path = "/workspace/calibration/resnet50"
# A calibration meta file such as "/workspace/calibration/resnet50.txt" can be used instead.

### get resnet50 from torchvision and convert it to ONNX
torch_model = torchvision.models.resnet50(pretrained=True)
onnx_model_path = "/workspace/resnet50.onnx"
convert_pytorch_to_onnx(torch_model, input_shape, onnx_model_path)

mxq_compile(
    model=onnx_model_path,
    calib_data_path=calib_data_path,
    save_path="resnet50.mxq",
    backend="onnx"
)
```

```
""" Compile PyTorch model, second way """
from qubee import mxq_compile
### get resnet50 from torchvision
import torchvision
import torch
calib_data_path = "/workspace/calibration/resnet50"
# A calibration meta file such as "/workspace/calibration/resnet50.txt" can be used instead.

### get resnet50 from torchvision and convert it to torchscript
torch_model = torchvision.models.resnet50(pretrained=True)
torchscript_model_path = "/workspace/resnet50.pt"

example_input = torch.rand(1, 3, 224, 224)
```

```
scripted_model = torch.jit.script(torch_model, example_input)
torch.jit.save(scripted_model, torchscript_model_path)

mxq_compile(
    model=torchscript_model_path,
    calib_data_path=calib_data_path,
    backend="torchscript",
    save_path="resnet50.mxq",
    example_input=example_input
)
```

4.4 Compiling Keras/TensorFlow Models

Since Keras works as an interface for TensorFlow 2, models on the Keras framework can be converted to Mobilint IR via TensorFlow. First, we load and save the Keras/TensorFlow model into the format of the frozen graph, which ends with `.pb`. Then, with the directory containing the frozen graph, qubee will compile the model. The following code assumes the calibration dataset is prepared in the directory `/workspace/calibration/resnet50`.

Remark According to the annotations and old version instructions, the TensorFlow compilation should work by providing the directory containing the frozen graph or just the frozen graph file. However, the current version makes various errors, such as kernel parsing errors, incompatible tag errors, etc. We are currently working on this issue.

```
""" Compile Keras/TensorFlow model """
from qubee import mxq_compile
import tensorflow as tf

keras_model = tf.keras.applications.resnet50.ResNet50() # Load a Keras model
input_shape = (224, 224, 3)
calib_data_path = "/workspace/calibration/resnet50"
# A calibration meta file such as "/workspace/calibration/resnet50.txt" can be used instead.

keras_model_save_path = "/workspace/tf_models/resnet50" # directory to save the model
keras_model.save(keras_model_save_path) # Save the model in the format of frozen graph.
saved_model.pb file will be created in the directory.
keras_model.summary() # if you are not aware of the input name, you can check it by this command.

mxq_compile(
    model=keras_model_save_path,
    calib_data_path=calib_data_path,
    backend="tf1", # or "tf2". It will be unified to "tf" in the future.
    save_path="resnet50.mxq",
    input_shape={'input_1':(224, 224, 3)} # dictionary of input shape
)
```

5. CPU Offloading

From qubee v0.7, we provide a Beta version of CPU offloading for mxq compile. CPU offloading makes it easier for users to compile their models by automatically offloading the computation that Mobilint NPU does not support to the CPU. For example, if a pre-processing or post-processing included in the model involves operations that the NPU does not support, the user would have to implement them manually after compiling, but CPU offloading covers most of these operations and eliminates the need for additional work.

When CPU offloading is employed, the procedures for preparing the calibration dataset and compiling the model vary slightly as follows:

(i) The pre-processed input shape should match the original model's input shape, whereas the pre-processed input shape should be in the format (H, W, C) to compile the model without CPU offloading.

(ii) Set the argument *cpu_offload* of function *mxq_compile* True to enable CPU offloading.

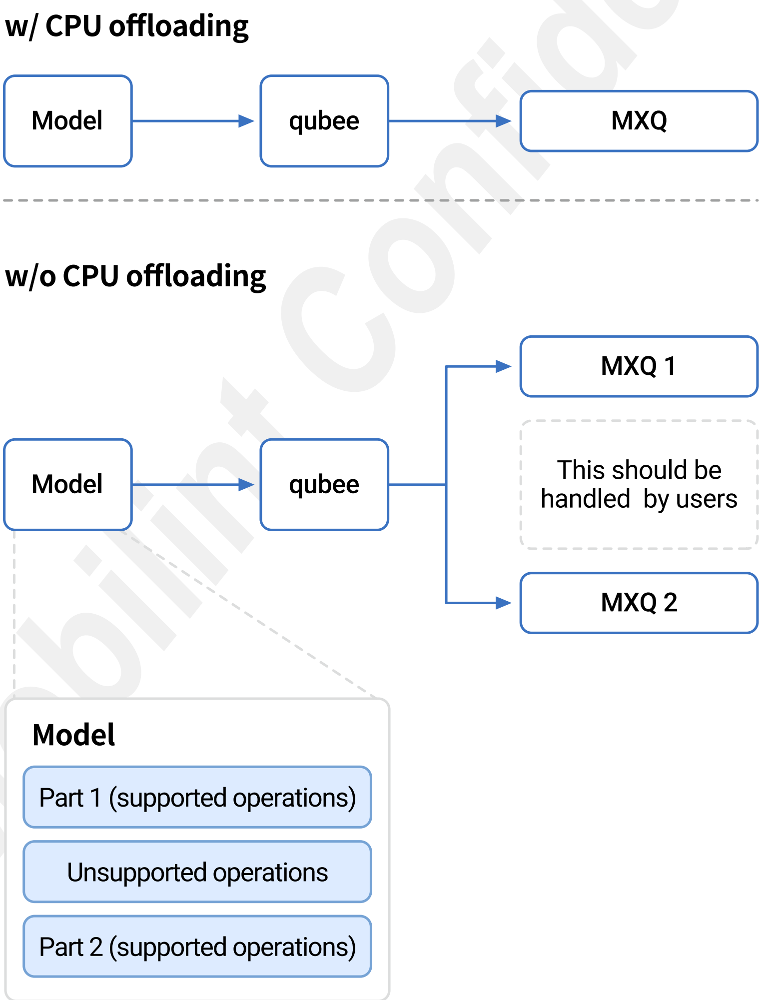

Figure 5-1. SDK CPU Offloading

6. Supported Frameworks

We support almost all the commonly used Machine Learning frameworks & libraries such as ONNX, PyTorch, Keras, and TensorFlow.

Figure 6-1. Supported deep-learning frameworks

6.1 Supported Operations (ONNX)

Table 6-1. ONNX Supported Operations

API Name	Comments
Add	Broadcast only for specific cases of constant addition: Adding scalar, Adding channel-size vector.
And	
ArgMax	
AveragePool	Only dilation=1, count_include_pad=1.
BatchNormalization	Only training_mode=0.
Cast	
Ceil	
Clip	
Concat	Only along channel axis.
Constant	
ConstantOfShape	
Conv	
ConvTranspose	
DepthToSpace	
Div	Only constant division. Support broadcast same as Add.
Elu	
Equal	
Erf	
Exp	
Expand	
Flatten	Only axis=1 and before fully connected layer or Conv w/ 1x1 kernel.
Floor	
Gather	
GatherND	
Gemm	Only transA=0.

API Name	Comments
	Only for the following specific case: Input A is a flatten activation and input B is 2D tensor.
GlobalAveragePool	
Greater	
HardSigmoid	
HardSwish	
Identity	
InstanceNormalization	
LayerNormalization	
LeakyRelu	
Less	
Loop	
MatMul	Only for the following specific case: Input A is a flatten activation and input B is 2D tensor or vice-versa.
Max	
MaxPool	Only dilation=1.
Min	
Mod	
Mul	Only constant multiplication. Support broadcast same as Add.
NonMaxSuppression	
NonZero	
Not	
Or	
Pad	Constant, reflect, edge modes are supported
Pow	
PRelu	
Range	
Reciprocal	
ReduceMax	Only along height and width.
ReduceMean	Only along height and width.
ReduceMin	Only along height and width.
ReduceProd	Only along height and width.
ReduceSum	Only along height and width.
Relu	Only scalar slope.
Reshape	Only channel-wise flatten and before fully connected layer or Conv w/ 1x1 kernel. Only allowzero=0.
Resize	Only for the following specific case: Only mode = "nearest" and coordinate_transformation_mode = "half_pixel" or "pytorch_half_pixel", Only mode = "linear" and coordinate_transformation_mode = "half_pixel"

API Name	Comments
	or "pytorch_half_pixel", Attributes axes, antialias, keep_aspect_ratio_policy nearest_mode are not supported.
ScatterND	
Shape	
Sigmoid	
Slice	Only channel-wise slice.
Softmax	
Softplus	
Split	
Sqrt	
Squeeze	Only when resulting tensor has 2D shape. Squeeze along batch axis is unsupported.
Sub	Support broadcast same as Add.
Tanh	
Tile	Batch-wise tile is unsupported.
TopK	
Transpose	Only for the following specific case: Transpose-Flatten-Linear.
Unsqueeze	
Upsample	Only mode "nearest" and "linear".
Xor	

6.2 Supported operations (PyTorch)

Remark Since the Torchscript backend framework is based on Torchscript-Based-ONNX-Exporter, even if the operation is not listed below, it can be supported if it has corresponding ONNX operation.

Table 6-2. PyTorch Supported Operations

API Name	Comments
nn.Conv2d	
nn.ConvTranspose2d	
nn.Linear	
nn.BatchNorm2d	
nn.MaxPool2d	Only dilation=1.
nn.AvgPool2d	Only dilation=1, count_include_pad=1.
nn.AdaptiveAvgPool2d	
nn.functional.interpolate	See supported operations (ONNX): resize.
nn.Upsample	Only mode "nearest" and "linear". Only scales=[2,2].
Add	Only alpha=1.
Sub	Only alpha=1.

API Name	Comments
Mul	Only constant multiplication.
Div	Only constant division.
Cat	Only along channel axis.
Relu	
PRelu	
LeakyRelu	
Sigmoid	
Tanh	
Softplus	Only beta=1.
Hardswish	
Clip	
Exp	
Reshape	Only channel-wise flatten and before fully connected layer or Conv w/ 1x1 kernel.
Transpose	Only before fully connected layer.
Squeeze	Only when resulting tensor has 2D shape. Squeeze along batch axis is unsupported.
nn.Flatten	Only channel-wise flatten and before fully connected layer or Conv w/ 1x1 kernel.
nn.Identity	
Pad	
Ceil	
Clamp	See supported operations (ONNX): clip
Erf	
Exp	
Floor	
Pow	
Reciprocal	

6.3 Supported operations (TensorFlow)

Table 6-3. TensorFlow Supported Operations

API Name	Comments
tf.placeholder	
tf.constant	
tf.identity	
tf.identity_n	
tf.pad	
tf.nn.conv2d	
tf.nn.depthwise_conv2d	
tf.nn.conv2d_backprop_input	

API Name	Comments
tf.linalg.matmul	
tf.nn.fused_batch_norm	
tf.nn.max_pool2d	
tf.nn.avg_pool	
tf.math.reduce_mean	Only along height, width, and channel.
tf.image.resize_nearest_neighbor	
tf.raw_ops.ConcatV2	Only along channel axis.
tf.math.add_n	
tf.nn.bias_add	
tf.math.multiply	Only constant multiplication.
tf.realdiv	Only constant division.
tf.math.subtract	
tf.nn.relu	
tf.nn.leaky_relu	
tf.nn.relu6v	
tf.math.sigmoid	
tf.math.softplus	
tf.math.tanh	Only beta=1.
tf.math.exp	
tf.switch_case	
tf.shape	
tf.reshape	Only channel-wise flatten and before fully connected layer or Conv w/ 1x1 kernel.
tf.transpose	Only before fully connected layer.
tf.expand_dims	
tf.squeeze	Only when resulting tensor has 2D shape. Squeeze along batch axis is unsupported.
tf.strided_slice	ellipsis_mask, new_axis_mask, shrink_axis_mask are unsupported.
tf.tile	Batch-wise tile is unsupported.

6.4 Supported operations (Keras)

Table 6-4. Keras Supported Operations

API Name	Comments
layers.Conv2D	
layers.Conv2DTranspose	
layers.Dense	
layers.BatchNormalization	Only training=False.
layers.ZeroPadding2D	
layers.Concatenate	Only along channel axis.
layers.Add	

API Name	Comments
layers.Subtract	
layers.Multiply	Only constant multiplication.
layers.MaxPooling2D	
layers.AveragePooling2D	
layers.GlobalAveragePooling2D	
layers.Dropout	Only inference mode.
layers.Flatten	Only channel-wise flatten and before fully connected layer or Conv w/ 1x1 kernel.
activations.relu	Only alpha=0, max_value=None, threshold=0.
activations.sigmoid	
activations.softplus	
activations.tanh	
activations.exponential	

7. API Reference

7.1 Class: Model_Dict

This class serves two main functions:

1. Compile

2. Inference (Note that this inference is only for testing and done by CPU or GPU.)

Table 7-1. Model_Dict Class

Attributes	Type	Description
model_dict	ONNX_Model_Dict, TF_Model_Dict	Mobilint IR, which holds information of layers in the model.
model_from	string	Backend for holding information of the model.
output_name _list	List[string]	List of the keys (in model_dict) corresponding to the output layer of the model. (It could be different from the original model, because qubee parses deep learning related operations only.)
model_from	string	Deep learning framework where the input model comes from.
c_model	qubee.mmc.Compiler	Low-level compiler. (defined in C++ code). It compiles Mobilint IR into MXQ format.
p_model	qubee.model_dict.Model	Model restored from Mobilint IR. This enables full-precision inference for testing.
is_compiled	bool	This indicates whether the model is compiled.
device	string	Device to be used for compile and inerence. Either cpu or gpu.
has_c_model	bool	This indicates whether the c_model is prepared.
has_p_model	bool	This indicates whether the p_model is prepared.

7.1.1 Methods

Table 7-2. Model_Dict Methods

Methods	Description
__init__	Constructor of Mobilint IR model.
compile	Compile the given model into MXQ format.
inference	Floating inference with the Mobilint IR. This can be used to check the built IR returns the same output as the model.
inference_int8	Integer inference with the compiled and quantized model. The model must be compiled before executing this function.
inference_int8_input_dict	Same as "inference_int8", but get a dictionary input which has a form of {node name: node input} instead. This can be used for models with multiple inputs.
cal_ops	Return the number of add/multiplication operations in the build Mobilint IR. This can be reduced in later optimization steps.

Methods	Description
to	Set the operating device (CPU or GPU).

7.1.2 Method Details

Table 7-3. Model_Dict.__init__

Parameter	Type	Description
model	string or model class of the corresponding framework	Model path or model instance. The following cases are supported: Using backend="onnx" and a ONNX model path, Using backend="torchscript" and a PyTorch model path, Using backend="tf1" or "tf2" and a fronzen TensorFlow PB graph.
backend	string (optional)	Which framework to use to get the Mobilint IR. It must be one of "onnx", "tf1", "tf2", and "torchscript". They correspond to deep learning frameworks as follows: "onnx": ONNX, "tf1" and "tf2": TensorFlow, Defaults to "onnx".
input_shape	tuple or list (optional)	Input shape in HWC. Required only for using PyTorch model.
device	string (optional)	Device to be used for compile and inerence. Either "cpu" or "gpu". Defaults to "cpu".

Table 7-4. Model_Dict.compile

Parameter	Type	Description
calib_data_path	string	A path to the calibration dataset. It can be either of a path to the text file (or json) containing the paths to the pre-processed numpy files or a directory containing the pre-processed numpy files.
save_path	string	Filename of the resulting .mxq.
model_nickname	string (optional)	Model nickname used in qubee. It is used in qubee to facilitate quicker recompilation of the same models. Qubee stores prior optimization information under this nickname, enabling it to locate and utilize the previously compiled results for faster processing. It is auto-generated from the model's base name, if not provided. For instance, a model "/workspace/onnx/resnet50.onnx" results in "resnet50". If not derivable, "temporary" is the default nickname.
quantize_method	string (optional)	Quantization method to determine the scale parameter in the quantization. Currently, "Max", "Percentile", "MaxPercentile" and "KL" are supported. Defaults to "Percentile".
quantize_per	float (optional)	Percentile used for the quantization method "Percentile" and

Parameter	Type	Description
centile		"MaxPercentile". This should be between 0 and 1. (Ex. 0.999, 0.9999). Defaults to 0.9999.
topk_ratio	float (optional)	It is used for quantization method "maxpercentile". Defaults to 0. The larger this value is, the more data is used for calibration. This should be between 0 and 1, but using a value of 0.01 or less is recommended.
smooth_factor	float (optional)	Smoothing factor that is required for Gaussian kernel construction on KL divergence estimation. Defaults to 1.6.
is_quant_ch	bool (optional)	Use multi-channel quantization if True. Defaults to False.
optimization	bool (optional)	If True, it compiles the model with optimization process. If False, qubee uses previous optimization information when stored in previous compiling. (Nickname should be the same.) It must be set to True on the first compile. Defaults to True.
optimization_level	int (optional)	Optimization level in the compiler. If optimization level is high, NPU inference could be faster, but it takes more time for compiling. (Recommend: 3~6.) Defaults to 5.
save_sample	bool (optional)	If True, create the "sampleInOut" folder in the current directory and store the input and output binary files in it. Defaults to False.
use_random_calib	bool (optional)	If True, it compiles the given model with random calibration data. This is just used to check if the model is compilable without making a calibration data. Defaults to False.
cpu_offload	bool (optional)	Use CPU offloading for NPU inference if True. Defaults to False.
quant_output	string (optional)	Quantization method that applied to the output layer. "layer", "ch" and "sigmoid" options are available. If "layer", per-layer quantization is applied to the output layer. If is_quant_ch is true, then the computed quantization scale for each channel of the output layer will be merged into single value. If "ch", per-channel quantization is applied to the output layer. This option is valid only when is_quant_ch is true. If "sigmoid", assign quantization scale that computed with sigmoid function. Defaults to "layer".
adaq_useadaquant	bool (optional)	If True, enable the finetuning with AdaQuant after quantization. Defaults to False.

Parameter	Type	Description
adaq_weightDeltaLR	float (optional)	Learning rate for finetuning weight delta(weight update) of AdaQuant. (Recommend: 1e-6 ~ 5e-5) Defaults to 0.
adaq_biasDeltaLR	float (optional)	Learning rate for finetuning bias delta(bias update) of AdaQuant. (Recommend: weightDeltaLR/10 ~ weightDeltaLR/2) Defaults to 0.
adaq_weightScaleLR	float (optional)	Learning rate for finetuning weight quantization scale of AdaQuant. Defaults to 0.
adaq_biasScaleLR	float (optional)	Learning rate for finetuning bias quantization scale of AdaQuant. Defaults to 0.
adaq_actScaleLR	float (optional)	Learning rate for finetuning activation quantization scale of AdaQuant. Defaults to 0.
adaq_batchSize	int (optional)	Batch size for running AdaQuant. Defaults to 16.
adaq_epoch	int (optional)	Epochs for repeating AdaQuant update. Defaults to 10.

Table 7-5. Model_Dict.inference

Parameter	Type	Description
input_tensor	numpy.array torch.Tensor Dict[string, numpy.array or torch.Tensor] List[numpy.array or torch.Tensor]	Input tensor with layout BCHW.
cast_cpu	bool (optional)	If True, enable CPU casting on full precision inference. Defaults to False.

Table 7-6. Model_Dict.inference_int8

Parameter	Type	Description
input_tensor	torch.Tensor or np.ndarray	Input tensor with layout BCHW.

Table 7-7. Model_Dict.inference_int8_input_dict

Parameter	Type	Description
input_dict	Dict[str, torch.Tensor or np.ndarray]	Dictionary that contains input information such as {input node name: input tensor}.

Table 7-8. Model_Dict.to

Parameter	Type	Description
device	string	Target device to use, which must be one of "cpu", "gpu", "cuda".

7.2 Function: mxq_compile

Compile a given model directly without creating an instance of "Model_Dict".

Table 7-9. mxq_compile

Parameter	Type	Description
model	string or model instance	Model path or model instance. Model should be instance for the following cases: Using backend="onnx" and a ONNX model path, Using backend="torchscript" and a PyTorch model, Using backend="tf1" or "tf2" and a fronzen TensorFlow PB graph.
calib_data_path	string	A path to the calibration dataset. It can be either of a path to the text file (or json) containing the paths to the pre-processed numpy files or a directory containing the pre-processed numpy files.
model_nickname	string (optional)	Model nickname used in qubee. It is used in qubee to facilitate quicker recompilation of the same models. Qubee stores prior optimization information under this nickname, enabling it to locate and utilize the previously compiled results for faster processing. It is auto-generated from the model's base name, if not provided. For instance, a model "/workspace/onnx/resnet50.onnx" results in "resnet50". If not derivable, "temporary" is the default nickname.
save_path	string (optional)	Filename of the resulting .mxq. If it is None, then it is set to "model_nickname".mxq Defaults to None.
input_shape	tuple or list (optional)	Input shape in HWC. Required only for using PyTorch model and backend="torchscript".
backend	string (optional)	Which framework to use to get the Mobilint IR. It must be one of "onnx", "tf1", "tf2", and "torchscript". They correspond to deep learning frameworks as follows: "onnx": ONNX, "tf1" or "tf2": TensorFlow, Keras "torchscript": PyTorch Defaults to "onnx".
device	string (optional)	Device to be used for compile and inerence. Either "cpu" or "gpu". Defaults to "cpu".
quantize_method	string (optional)	Quantization method to determine the scale parameter in the quantization. Currently, "Max", "Percentile", "MaxPercentile" and "KL" are supported.

Parameter	Type	Description
		Defaults to "Percentile".
quantize_percentile	float (optional)	Percentile used for the quantization method "Percentile" and "MaxPercentile". This should be between 0 and 1. (Ex. 0.999, 0.9999) Defaults to 0.99995.
topk_ratio	float (optional)	It is used for quantization method "maxpercentile". Defaults to 0. The larger this value is, the more data is used for calibration. This should be between 0 and 1, but using a value of 0.01 or less is recommended.
smooth_factor	float (optional)	Smooth factor for Gaussian kernel construction, which is required on KL divergence estimation. Defaults to 1.6.
is_quant_ch	bool (optional)	Use multi-channel quantization if True. Defaults to False.
optimization	bool (optional)	If True, it compiles the model with optimization process. If false, qubee uses previous optimization information when stored in previous compiling. (Nickname should be the same.) It must be set to True on the first compile. Defaults to True.
optimization_level	int (optional)	Optimization level in the compiler. If optimization level is high, NPU inference could be faster, but it takes more time for compiling. (Recommend: 3~6) Defaults to 5.
save_sample	bool (optional)	If True, create the "sampleInOut" folder in the current directory and store the input and output binary files in it. Defaults to False.
use_random_calib	bool (optional)	If True, it compiles the given model with random calibration data. This is just used to check if the model is compilable without making a calibration data. Defaults to False.
cpu_offload	bool (optional)	Use CPU offloading for NPU inference if True. Defaults to False.
quant_output	string (optional)	Quantization method that applied to the output layer. "layer", "ch" and "sigmoid" options are available. If "layer", per-layer quantization is applied to the output layer. If is_quant_ch is true, then the computed quantization scale for each channel of the output layer will be merged into single value. If "ch", per-channel quantization is applied to the output layer. This option is valid only when is_quant_ch is true. If "sigmoid", assign quantization scale that computed with sigmoid function. Defaults to "layer".
adaq_useada	bool (optional)	If True, enable the finetuning with AdaQuant after

Parameter	Type	Description
quant		quantization. Defaults to False.
adaq_weight DeltaLR	float (optional)	Learning rate for finetuning weight delta(weight update) of AdaQuant. (Recommend: 1e-6 ~ 5e-5) Defaults to 0.
adaq_biasDel taLR	float (optional)	Learning rate for finetuning bias delta(bias update) of AdaQuant. (Recommend: weightDeltaLR/10 ~ weightDeltaLR/2) Defaults to 0.
adaq_weight ScaleLR	float (optional)	Learning rate for finetuning weight quantization scale of AdaQuant. Defaults to 0.
adaq_biasSc aleLR	float (optional)	Learning rate for finetuning bias quantization scale of AdaQuant. Defaults to 0.
adaq_actScal eLR	float (optional)	Learning rate for finetuning activation quantization scale of AdaQuant. Defaults to 0.
adaq_batchSi ze	int (optional)	Batch size for running AdaQuant. Defaults to 16.
adaq_epoch	int (optional)	Epochs for repeating AdaQuant update. Defaults to 10.

7.2.1 Tips for choosing quantization methods

"Percentile" and "MaxPercentile" quantization methods each take a hyperparameter called *percentile*. An increase in this value corresponds to a broader quantization interval. To elaborate further, a higher *percentile* results in reduced overflow, albeit at the expense of accuracy.

The "MaxPercentile" method determines the percentile value from data that has been filtered once. As a result, a lower *percentile* is needed for "MaxPercentile" compared to the "Percentile" method. For instance, for the "Percentile" method, we suggest using a value of 0.9999 to 0.999999. For the "MaxPercentile" method, we recommend *percentile* between 0.9 and 0.9999.

7.3 Function: make_calib

From given images and preprocessing configuration, create the preprocessed numpy files and a txt file containing their paths.

Table 7-10. make_calib

Parameter	Type	Description
args_pre	string or Dict	Path to a Yaml file or dictionary containing preprocessing configuration information. Refer to 7.4. for details.
data_dir	string	Directory of data to be used for calibration.
save_dir	string	Directory to save the pre-processed numpy files and txt file which contains their paths.
save_name	string (optional)	Name for resulting files. Numpy files will be saved under {save_dir}/{save_name}_npy

Parameter	Type	Description
		directory. Text file will be saved in {save_dir}/{save_name}.txt. If it is not provided, it is set to the basename of data_dir.
anno_json	string (optional)	Path to an annotation json file for COCO format. When provided, make_calib function randomly selects samples considering class balance. Defaults to None.
file_format	string (optional)	Filename format using image_idx. Defaults to '%012d.jpg'.
max_size	int (optional)	Maximum size of the resulting calibration data. Defaults to -1, which means no limit on the number of the calibration data.
remove_npy	bool (optional)	If True, remove pre-existing numpy files. Defaults to False.
seed	int (optional)	Random seed. Defaults to 2023.
save_calib_msg	bool (optional)	If True, save calibration data dictionary as MSGpack file. Defaults to False.
msg_path	string (optional)	Path to save MSGpack file If not provided, it automatically generate the path with dataname and number of calibration data. Defaults to None.

7.4 Fuction: make_calib_man

From given images and manually written function that takes an image path as input, create the preprocessed numpy files and a txt file containing their paths.

Table 7-11. make_calib_man

Parameter	Type	Description
pre_ftn	Callable	Pre-processing function that takes an image path as input.
data_dir	string	Directory of data to be used for calibration.
save_dir	string	Directory to save the pre-processed numpy files and txt file which contains their paths.
save_name	string (optional)	Name for resulting files. Numpy files will be saved under {save_dir}/{save_name}_npy directory. Text file will be saved in {save_dir}/{save_name}.txt. If it is not provided, it is set to the basename of data_dir.
anno_json	string (optional)	Path to an annotation json file for COCO format. When provided, make_calib function randomly selects samples considering class balance. Defaults to None.
file_format	string (optional)	Filename format using image_idx. Defaults to '%012d.jpg'.
max_size	int (optional)	Maximum size of the resulting calibration data.

Parameter	Type	Description
		Defaults to -1, which means no limit on the number of the calibration data.
remove_npy	bool (optional)	If True, remove pre-existing numpy files. Defaults to False.
seed	int (optional)	Random seed. Defaults to 2023.
save_calib_msg	bool (optional)	If True, save calibration data dictionary as MSGpack file. Defaults to False.
msg_path	string (optional)	Path to save MSGpack file If not provided, it automatically generate the path with dataname and number of calibration data. Defaults to None.

Example codes for using these functions are provided in the ##Preparing Calibration Data section.

7.5 Pre-processing Configurations

qubee supports the following pre-processing functions to make calibration data.

Table 7-12. Pre-processing function API

Pre-processing Type	Description
GetImage	Get image tensor from image path using cv2 backend or image tensor. Note that this should be at the top of the list.
Pad	Pad image tensor.
Normalize	Normalize image tensor.
ResizeTorch	Resize the input image to the given size using torchvision.transforms.functional.resize
Resize	Resize image tensor to the given size using cv2.resize.
CenterCrop	Center crop the image tensor.
SetOrder	Set the order of axes of the given image tensor. Note that this should be at the very end.

You can write a yaml file as follows:

```
[Pre-processing Type]
    [Parameter]: [Argument]
    ...
```

```yaml
# Example
GetImage:
    to_float32: false
    channel_order: RGB
ResizeTorch:
    size: [256, 256]
    interpolation: bilinear
```

```
CenterCrop:
    size: [224, 224]
Normalize:
    mean: [0.485, 0.456, 0.406]
    std: [0.229, 0.224, 0.225]
    to_float: true
SetOrder:
    shape: HWC
```

7.5.1 Pre-processing Parameters

Table 7-13. GetImage

Parameter	Type	Description
to_float32	bool (optional)	If True, set dtype as float32. Defaults to False.
channel_order	string (optional)	Channel order to load. Upper cases will be converted into lower cases. Defaults to "bgr".

Table 7-14. Pad

Parameter	Type	Description
shape	Tuple[int] (optional)	Expected padding shape (h, w). Defaults to None.
size_divisor	int (optional)	Pad images so that the the resulting image's width and height are divisible by size_divisor. Defaults to None.
pad_val	float (optional)	Values to be filled in padding areas when padding_mode is 'constant'. Defaults to 0.
right_bottom	bool (optional)	If True, it only bads to right and bottom. Defaults to False.

Table 7-15. Normalize

Parameter	Type	Description
mean	List[float] or np.ndarray	Normalization mean.
std	List[float] or np.ndarray	Normalization standard deviation.
to_float	bool (optional)	Normalize image between [0, 255] into [0, 1] by dividing by 255 before normalizing with the mean and std. Defaults to False.

Table 7-16. ResizeTorch

Parameter	Type	Description
size	List[int]	Desired output size, i.e., height and width.
interpolation	string	Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "box", "hamming", "lanczos".

Table 7-17. Resize

Parameter	Type	Description
img_scale	float or Tuple[int, int]	The scaling factor or maximum size (h, w). If it is a float number, then the image will be rescaled by this factor, else if it is a tuple of 2 integers, then the image will be rescaled as large as possible within the scale.
keep_ratio	bool	Whether to keep the aspect ratio when resizing the image. Defaults to False.
interpolation	string	Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "area", "lanczos".

Table 7-18. CenterCrop

Parameter	Type	Description
size	List[int]	Desired output height and width.

Table 7-19. SetOrder

Parameter	Type	Description
shape	string	Desired data layout format, accepted values are "HWC", "CHW", "BHWC", "BCHW". Defaults to "HWC".

8. Open Source License Notice

Apache TVM

- https://github.com/apache/tvm
- Apache 2.0 License

PyTorch

- https://github.com/pytorch/pytorch
- BSD-like License

TensorFlow

- https://github.com/tensorflow/tensorflow
- Apache 2.0 License

ONNX

- https://github.com/onnx/onnx
- Apache 2.0 License

ONNX Runtime

- https://github.com/microsoft/onnxruntime
- MIT License

Keras

- https://github.com/keras-team/keras
- Apache 2.0 License

9. Copyright