

---

# TPU-MLIR Technical Reference Manual

Release 1.20-1-gd3964d047

SOPHGO

Jun 30, 2025

# Table of Contents

<b>1 TPU-MLIR Introduction</b>	<b>4</b>
<b>2 Environment Setup</b>	<b>6</b>
2.1 Code Download . . . . .	6
2.2 Docker Configuration . . . . .	6
2.3 ModelZoo (Optional) . . . . .	7
2.4 Compilation . . . . .	7
2.5 Code Development . . . . .	8
<b>3 User Interface</b>	<b>10</b>
3.1 Model Conversion Process . . . . .	10
3.1.1 Support for Image Input . . . . .	11
3.1.2 Support for Multiple Inputs . . . . .	11
3.1.3 Support for INT8 Symmetric and Asymmetric . . . . .	11
3.1.4 Support for Mixed Precision . . . . .	12
3.1.5 Support for Quantized TFLite Models . . . . .	12
3.1.6 Support for Caffe Models . . . . .	13
3.1.7 Support LLM models . . . . .	13
3.2 Introduction to Tool Parameters . . . . .	13
3.2.1 model_transform.py . . . . .	13
3.2.2 run_calibration.py . . . . .	15
3.2.3 model_deploy.py . . . . .	18
3.2.4 llm_convert.py . . . . .	21
3.2.5 model_runner.py . . . . .	21
3.2.6 npz_tool.py . . . . .	22
3.2.7 visual.py . . . . .	22
3.2.8 mlir2graph.py . . . . .	23
3.2.9 gen_rand_input.py . . . . .	23
3.2.10 model_tool . . . . .	25
<b>4 Overall Design</b>	<b>29</b>
4.1 Layered . . . . .	29
4.2 Top Pass . . . . .	29
4.3 Tpu Pass . . . . .	31
4.4 Other Passes . . . . .	32
<b>5 Front-end Conversion</b>	<b>33</b>
5.1 Main Work . . . . .	33

5.2	Workflow . . . . .	33
5.3	Example . . . . .	35
<b>6</b>	<b>Quantization</b>	<b>39</b>
6.1	Basic Concepts . . . . .	39
6.1.1	Asymmetric Quantization . . . . .	39
6.1.2	Symmetric Quantization . . . . .	40
6.2	Scale Conversion . . . . .	41
6.3	Quantization derivation . . . . .	41
6.3.1	Convolution . . . . .	41
6.3.2	InnerProduct . . . . .	42
6.3.3	Add . . . . .	42
6.3.4	AvgPool . . . . .	43
6.3.5	LeakyReLU . . . . .	43
6.3.6	Pad . . . . .	44
6.3.7	PReLU . . . . .	44
<b>7</b>	<b>Calibration</b>	<b>45</b>
7.1	General introduction . . . . .	45
7.2	Introduction to the Default Process . . . . .	47
7.3	Calibration data screening and preprocessing . . . . .	49
7.3.1	Screening Principles . . . . .	49
7.3.2	Input format and preprocessing . . . . .	49
7.4	Quantization Threshold Algorithm Implementation . . . . .	50
7.4.1	KLD Algorithm . . . . .	50
7.4.2	Auto-tune Algorithm . . . . .	51
7.4.3	Octav Algorithm . . . . .	51
7.4.4	Aciq Algorithm . . . . .	53
7.5	optimization algorithms Implementation . . . . .	53
7.5.1	sq Algorithm . . . . .	54
7.5.2	we Algorithm . . . . .	54
7.5.3	bc Algorithm . . . . .	56
7.5.4	search_threshold Algorithm . . . . .	56
7.5.5	search_qtable Algorithm . . . . .	57
7.6	Example: yolov5s calibration . . . . .	57
7.7	visual tool introduction . . . . .	62
<b>8</b>	<b>Lowering</b>	<b>66</b>
8.1	Basic Process . . . . .	66
8.2	Mixed precision . . . . .	67
<b>9</b>	<b>SubNet</b>	<b>68</b>
<b>10</b>	<b>LayerGroup</b>	<b>69</b>
10.1	Basic Concepts . . . . .	69
10.2	BackwardH . . . . .	70
10.3	Dividing the Mem Cycle . . . . .	70
10.4	LMEM Allocation . . . . .	72

10.5	Divide the optimal Group . . . . .	74
<b>11</b>	<b>GMEM Allocation</b>	<b>75</b>
11.1	1. Purpose . . . . .	75
11.2	1. Principle . . . . .	75
	11.2.1 2.1. GMEM allocation in weight tensor . . . . .	75
	11.2.2 2.2. GMEM allocation in global neuron tensors . . . . .	76
<b>12</b>	<b>CodeGen</b>	<b>78</b>
12.1	Main Work . . . . .	78
12.2	Workflow . . . . .	78
12.3	BM168X and Related classes in TPU-MLIR . . . . .	79
12.4	Backend Function Loading . . . . .	80
12.5	Backend store_cmd . . . . .	81
<b>13</b>	<b>MLIR Definition</b>	<b>83</b>
13.1	Top Dialect . . . . .	83
	13.1.1 Operations . . . . .	83
<b>14</b>	<b>Accuracy Validation</b>	<b>102</b>
14.1	Introduction . . . . .	102
	14.1.1 Objects . . . . .	102
	14.1.2 Metrics . . . . .	102
	14.1.3 Datasets . . . . .	103
14.2	Validation Interface . . . . .	104
14.3	Validation Example . . . . .	104
	14.3.1 mobilenet_v2 . . . . .	104
	14.3.2 yolov5s . . . . .	106
<b>15</b>	<b>Quantization aware training</b>	<b>108</b>
15.1	Basic Principles . . . . .	108
15.2	tpu-mlir QAT implementation scheme and characteristics . . . . .	108
	15.2.1 Main body flow . . . . .	108
	15.2.2 Features of the Scheme . . . . .	109
15.3	Installation Method . . . . .	109
	15.3.1 Install with setup package . . . . .	109
	15.3.2 Install from source . . . . .	110
15.4	Basic Steps . . . . .	110
	15.4.1 Step 1: Interface import and model prepare . . . . .	110
	15.4.2 Step 2: Calibration and quantization training . . . . .	111
	15.4.3 Step 3: Export tuned fp32 model . . . . .	112
	15.4.4 Step 4: Initiate the training . . . . .	112
15.5	Use Examples-resnet18 . . . . .	113
15.6	Tpu-mlir QAT test environment . . . . .	115
	15.6.1 Adding a cfg File . . . . .	115
	15.6.2 Modify and execute run_eval.py . . . . .	115
15.7	Use Examples-yolov5s . . . . .	116
15.8	Use Examples-bert . . . . .	116

<b>16 TpuLang Interface</b>	<b>117</b>
16.1 Main Work . . . . .	117
16.2 Work Process . . . . .	117
16.3 Operator Conversion Example . . . . .	119
16.4 Tpulang API usage method . . . . .	123
16.4.1 Initialization . . . . .	123
16.4.2 Prepare the tensors . . . . .	123
16.4.3 Build the graph . . . . .	123
16.4.4 compile . . . . .	124
16.4.5 deinit . . . . .	124
16.4.6 deploy . . . . .	125
<b>17 Custom Operators</b>	<b>126</b>
17.1 Overview . . . . .	126
17.2 Custom Operator Addition Process . . . . .	127
17.2.1 Add TpuLang Custom Operator . . . . .	127
17.2.2 Add Caffe Custom Operator . . . . .	134
17.3 Custom Operator Example . . . . .	135
17.3.1 Example of TpuLang . . . . .	135
17.3.2 Example of Caffe . . . . .	139
17.4 Custom AP(Application Processor) Operator Adding Process . . . . .	141
17.4.1 TpuLang Custom AP Operator Adding . . . . .	141
17.5 Custom AP(Application Processor) Operator Example . . . . .	144
17.5.1 TpuLang Example . . . . .	144
<b>18 Implementing Backend Operators with PPL</b>	<b>147</b>
18.1 How to Write and Call Backend Operators . . . . .	147
18.2 PPL Workflow in TPU-MLIR . . . . .	150
<b>19 final.mlir Truncation Method</b>	<b>151</b>
19.1 final.mlir structure introduction . . . . .	151
19.2 final.mlir Truncation Process . . . . .	153
<b>20 Superior MaskRCNN Interface Guidance</b>	<b>156</b>
20.1 MaskRCNN Basic . . . . .	156
20.1.1 Fast Block Segmentation Methods . . . . .	156
20.2 Superior MaskRCNN . . . . .	157
20.3 Quick Start . . . . .	157
20.3.1 Prepare Your Yaml . . . . .	157
20.3.2 Block Unit Test . . . . .	158
20.4 New Frontend Interface API . . . . .	158
20.4.1 [Step 1] Run model_transform . . . . .	158
20.4.2 [Step 2] Generate Input Data . . . . .	159
20.4.3 [Step 3] Run model_deploy . . . . .	160
20.5 IO_MAP Guidance . . . . .	160
20.5.1 [Step-1] Describe Block Interface . . . . .	160
20.5.2 [Step-2] Describe IO_MAP . . . . .	164
20.6 Generate IO_MAP . . . . .	167

20.7 mAP Inference . . . . .	169
<b>21 LLMC Guidance</b>	<b>170</b>
21.1 TPU-MLIR weight-only quantization . . . . .	170
21.2 llmc_tpu . . . . .	171
21.2.1 Environment Setup . . . . .	171
21.2.2 tpu Directory . . . . .	172
21.2.3 Operating Steps . . . . .	172
<b>22 Analyse TPU Performance with TPU Profile</b>	<b>177</b>
22.1 1. Software and Hardware Framework of TPU . . . . .	177
22.2 2. Compile Bmodel . . . . .	178
22.3 3. Generate Profile Binary Data . . . . .	179
22.4 4. Visualize Profile Data . . . . .	180
22.5 5. Analyse the Result . . . . .	180
22.5.1 5.1 Overall Introduction . . . . .	180
22.5.2 5.2 Global Layer . . . . .	183
22.5.3 5.3 Local Layer Group . . . . .	184
22.6 6. Summary . . . . .	185
<b>23 Appendix.01: Migrating from NNTC to tpu-mlir</b>	<b>186</b>
23.1 ONNX to MLIR . . . . .	186
23.2 Make a quantization calibration table . . . . .	187
23.3 Generating int8 models . . . . .	188
<b>24 Appendix 02: Basic Elements of TpuLang</b>	<b>189</b>
24.1 Tensor . . . . .	189
24.2 Tensor Preprocessing (Tensor.preprocess) . . . . .	190
24.3 Scalar . . . . .	192
24.4 Control Functions . . . . .	192
24.4.1 Initialization Function . . . . .	192
24.4.2 compile . . . . .	193
24.4.3 Deinitialization . . . . .	194
24.4.4 Reset Default Graph . . . . .	194
24.4.5 Get Current Default Graph . . . . .	195
24.4.6 Reset Graph . . . . .	195
24.4.7 Rounding Mode . . . . .	195
24.5 Operator . . . . .	196
24.5.1 NN/Matrix Operator . . . . .	197
24.5.2 Base Element-wise Operator . . . . .	215
24.5.3 Element-wise Compare Operator . . . . .	228
24.5.4 Activation Operator . . . . .	241
24.5.5 Data Arrange Operator . . . . .	273
24.5.6 Sort Operator . . . . .	282
24.5.7 Shape About Operator . . . . .	287
24.5.8 Quant Operator . . . . .	291
24.5.9 Up/Down Scaling Operator . . . . .	299
24.5.10 Normalization Operator . . . . .	308

24.5.11 Vision Operator . . . . .	315
24.5.12 Select Operator . . . . .	320
24.5.13 Preprocess Operator . . . . .	328
24.5.14 Transform Operator . . . . .	330
24.5.15 Transform Operator . . . . .	333
24.5.16 Transform Operator . . . . .	335
24.5.17 Transform Operator . . . . .	338

---

## TABLE OF CONTENTS

---



# SOPHON

### **Legal Notices**

Copyright © SOPHGO 2025. All rights reserved.

No part or all of the contents of this document may be copied, reproduced or transmitted in any form by any organization or individual without the written permission of the Company.

### **Attention**

All products, services or features, etc. you purchased is subject to SOPHGO's business contracts and terms. All or part of the products, services or features described in this document may not be covered by your purchase or use. Unless otherwise agreed in the contract, SOPHGO makes no representations or warranties (including express and implied) regarding the contents of this document. The contents of this document may be updated from time to time due to product version upgrades or other reasons. Unless otherwise agreed, this document is intended as a guide only. All statements, information and recommendations in this document do not constitute any warranty, express or implied.

### **Technical Support**

#### **Address**

Building 1, Zhongguancun Integrated Circuit Design Park (ICPARK), No. 9  
Fenghao East Road, Haidian District, Beijing

#### **Postcode**

100094

#### **URL**

<https://www.sophgo.com/>

#### **Email**

[sales@sophgo.com](mailto:sales@sophgo.com)

#### **Tel**

010-57590723

---

## TABLE OF CONTENTS

---

### **Release Record**

## TABLE OF CONTENTS

---

Version	Release date	Explanation
v1.20.0	2025.06.30	Support IO_RELOC; Deconv3D INT8 bugfix; BatchNorm and Conv backward operators support 128 batch training
v1.19.0	2025.05.30	Support AWQ and GPTQ models; Deconv3D F16, F32 bugfix
v1.18.0	2025.05.01	YOLO series adds automatic mixed precision setting; Added SmoothQuant option for run_calibration; New one-click compilation script for LLM
v1.17.0	2025.04.03	Significant improvement in LLM model compilation speed; TPULang supports PPL operator integration; Fixed random error issue with Trilu bf16 on Mars3
v1.16.0	2025.03.03	TPULang ROI_Extractor support; Einsum supports abcde,abfge->abcdg pattern; LLMC adds Vila model support
v1.15.0	2025.02.05	Added LLMC quantization support; Address boundary check in codegen; Fixed several comparison issues
v1.14.0	2025.01.02	Added post-processing fusion for yolov8/v11; Support for Conv3D stride > 15; Improved FAttention accuracy
v1.13.0	2024.12.02	Streamlined Release package; Performance optimization for MaxPoolWithMask training operator; Added support for large RoPE operators
v1.12.0	2024.11.06	tpuv7-runtime cmodel integration; BM1690 multi-core LayerGroup optimization; Support for PPL backend operator development
v1.11.0	2024.09.27	Added soc mode for BM1688 tdb; bmodel supports fine-grained merging; Fixed several performance degradation issues
v1.10.0	2024.08.15	Added yolov10 support; New quantization tuning section; Optimized tpu-perf log output
v1.9.0	2024.07.16	BM1690 added 40 model regression tests; New quantization algorithms: octav, aciq_guas and aciq_laplace
v1.8.0	2024.05.30	BM1690 supports multi-core MatMul operator; TPULang supports input/output order specification; tpuperf removes patchelf dependency
v1.7.0	2024.05.15	CV186X dual-core changed to single-core; BM1690 testing process aligned with BM1684X; Support for gemma/llama/qwen models
v1.6.0	2024.02.23	Added Pypi release format; Support for user-defined Global operators; Added CV186X processor platform support
v1.5.0	2023.11.03	More Global Layer support for multi-core parallel
Copyright © SOHIGO 3		
v1.4.0	2023.09.27	System dependencies upgraded to Ubuntu22.04; Added BM1684 Winograd support
v1.3.0	2023.07.27	Added manual floating-point operation region specification; Added supported frontend framework operator list; Added NNTC vs TPU-MLIR

# CHAPTER 1

## TPU-MLIR Introduction

TPU-MLIR is a compiler project for Deep Learning processors. This project provides a complete toolchain, which can convert pre-trained neural networks under different frameworks into binary files bmodel that can be efficiently run on the processors. The code has been open-sourced to github: <https://github.com/sophgo/tpu-mlir>.

The overall architecture of TPU-MLIR is as follows:

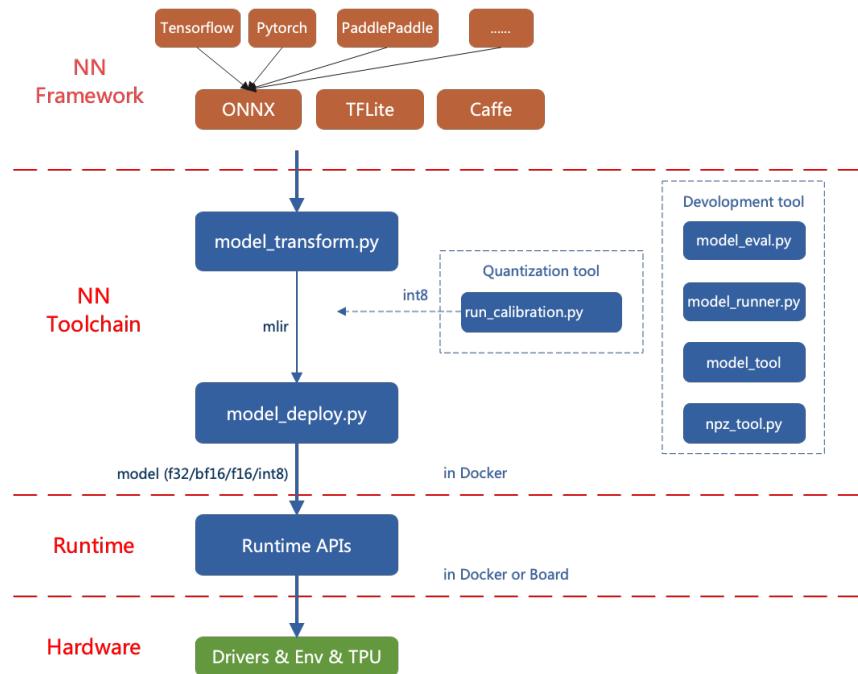


Fig. 1.1: TPU-MLIR overall architecture

The current directly supported frameworks are onnx, caffe and tflite. Models from other frameworks need to be converted to onnx models. The method of converting models from other frameworks to onnx can be found on the onnx official website: <https://github.com/onnx/tutorials>.

To convert a model, firstly you need to execute it in the specified docker. With the required environment, conversion work can be done in two steps, converting the original model to mlir file by `model_transform.py` and converting the mlir file to bmodel by `model_deploy.py`. To obtain an INT8 model, you need to call `run_calibration.py` to generate a quantization table and pass it to `model_deploy.py`.

This article presents the implementation details to guide future development.

# CHAPTER 2

---

## Environment Setup

---

This chapter describes the development environment configuration. The code is compiled and run in docker.

### 2.1 Code Download

GitHub link: <https://github.com/sophgo/tpu-mlir>

After cloning this code, it needs to be compiled in docker. For specific steps, please refer to the following.

### 2.2 Docker Configuration

TPU-MLIR is developed in the Docker environment, and it can be compiled and run after Docker is configured.

Download the required image from DockerHub [https://hub.docker.com/r/sophgo/tpuc\\_dev](https://hub.docker.com/r/sophgo/tpuc_dev) :

```
$ docker pull sophgo/tpuc_dev:v3.4
```

If the pulling fails, you can download the required image file from the official website development materials <https://developer.sophgo.com/site/index/material/86/all.html>, or use the following command to download and load the image:

```
1 $ wget https://sophon-assets.sophon.cn/sophon-prod-s3/drive/25/04/15/16/tpuc_dev_v3.4.tar.gz  
2 $ docker load -i tpuc_dev_v3.4.tar.gz
```

If you are using docker for the first time, you can execute the following commands to install and configure it (only for the first time):

```
1 $ sudo apt install docker.io
2 $ sudo systemctl start docker
3 $ sudo systemctl enable docker
4 $ sudo groupadd docker
5 $ sudo usermod -aG docker $USER
6 $ newgrp docker
```

Make sure the installation package is in the current directory, and then create a container in the current directory as follows:

```
$ docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.4
# "myname" is just an example, you can use any name you want
# use --privileged to get root permission, if you don't need root permission, please remove this ↴ parameter
```

Note that the path of the TPU-MLIR project in docker should be /workspace/tpu-mlir

## 2.3 ModelZoo (Optional)

TPU-MLIR comes with the yolov5s model. If you want to run other models, you need to download them from ModelZoo. The path is as follows:

<https://github.com/sophgo/model-zoo>

After downloading, put it in the same directory as tpu-mlir. The path in docker should be /workspace/model-zoo

## 2.4 Compilation

In the docker container, the code is compiled as follows:

```
$ cd tpu-mlir
$ source ./envsetup.sh
$ ./build.sh
```

Regression validation:

```
# This project contains the yolov5s.onnx model, which can be used directly for validation
$ pushd regression
$ python run_model.py yolov5s
$ popd
```

You can validate more networks with model-zoo, but the whole regression takes a long time:

```
# The running time is very long, so it is not necessary
$ pushd regression
$ ./run_all.sh
$ popd
```

## 2.5 Code Development

To facilitate code readability and development, it is recommended to use VSCode as the editor. In VSCode, install the following extensions:

- C/C++ Intellisense : Provides intelligent suggestions, code navigation, and formatting for C++ code.
- GitLens : Assists with Git version control and code review.
- Python : Provides intelligent suggestions and code navigation for Python.
- yapf : Formats Python code.
- shell-format : Formats shell scripts.
- Remote-SSH : Enables remote connections to code on a server (essential when code is not local).

After writing your code, right-click and select “Format Document” to ensure a consistent code style.

Since TPU-MLIR uses llvm-project and relies heavily on its headers and libraries, it is recommended to install llvm-project for improved code navigation. Follow these steps:

1. At the same level as the TPU-MLIR repository, create a third-party directory and clone llvm-project into it:

```
$ mkdir third-party
$ cd third-party
$ git clone git@github.com:llvm/llvm-project.git
```

2. Inside the TPU-MLIR Docker environment, build llvm-project (you may be prompted to install missing components during the build—follow the prompts to install them):

```
$ cd llvm-project
$ mkdir build && cd build
# If prompted for missing components (e.g., nanobind), install them:
# pip3 install nanobind
$ cmake -G Ninja ..\llvm \
    -DLLVM_ENABLE_PROJECTS="mlir" \
    -DLLVM_INSTALL_UTILS=ON \
    -DLLVM_TARGETS_TO_BUILD="" \
    -DLLVM_ENABLE_ASSERTIONS=ON \
    -DMLIR_INCLUDE_TESTS=OFF \
    -DLLVM_INSTALL_GTEST=ON \
```

(continues on next page)

(continued from previous page)

```
-DMLIR_ENABLE_BINDINGS_PYTHON=ON \
-DCMAKE_BUILD_TYPE=DEBUG \
-DCMAKE_INSTALL_PREFIX=../install \
-DCMAKE_C_COMPILER=clang \
-DCMAKE_CXX_COMPILER=clang++ \
-DLLVM_ENABLE_LLD=ON
$ cmake --build . --target install
```

After installation, you can link code navigation to the llvm-project sources.

# CHAPTER 3

---

## User Interface

---

This chapter introduces the user interface, including the basic process of converting models and the usage methods of various tools.

### 3.1 Model Conversion Process

The basic procedure is transforming the model into a mlir file with `model_transform.py`, and then transforming the mlir into the corresponding model with `model_deploy.py`. Take the `somenet.onnx` model as an example, the steps are as follows:

```
# To MLIR
$ model_transform.py \
  --model_name somenet \
  --model_def somenet.onnx \
  --test_input somenet_in.npz \
  --test_result somenet_top_outputs.npz \
  --mlir somenet.mlir

# To Float Model
$ model_deploy.py \
  --mlir somenet.mlir \
  --quantize F32 \ # F16/BF16
  --processor BM1684X \
  --test_input somenet_in_f32.npz \
  --test_reference somenet_top_outputs.npz \
  --model somenet_f32.bmodel
```

### 3.1.1 Support for Image Input

When using images as input, preprocessing information needs to be specified, as follows:

```
$ model_transform.py \
--model_name img_input_net \
--model_def img_input_net.onnx \
--input_shapes [[1,3,224,224]] \
--mean 103.939,116.779,123.68 \
--scale 1.0,1.0,1.0 \
--pixel_format bgr \
--test_input cat.jpg \
--test_result img_input_net_top_outputs.npz \
--mlir img_input_net.mlir
```

### 3.1.2 Support for Multiple Inputs

When the model has multiple inputs, you can pass in a single npz file or sequentially pass in multiple npy files separated by commas, as follows:

```
$ model_transform.py \
--model_name multi_input_net \
--model_def multi_input_net.onnx \
--test_input multi_input_net_in.npz \ # a.npy,b.npy,c.npy
--test_result multi_input_net_top_outputs.npz \
--mlir multi_input_net.mlir
```

### 3.1.3 Support for INT8 Symmetric and Asymmetric

Calibration is required if you need to get the INT8 model.

```
$ run_calibration.py somenet.mlir \
--dataset dataset \
--input_num 100 \
-o somenet_cali_table
```

Generating Model with Calibration Table Input.

```
$ model_deploy.py \
--mlir somenet.mlir \
--quantize INT8 \
--calibration_table somenet_cali_table \
--processor BM1684X \
--test_input somenet_in_f32.npz \
--test_reference somenet_top_outputs.npz \
--tolerance 0.9,0.7 \
--model somenet_int8.bmodel
```

### 3.1.4 Support for Mixed Precision

When the precision of the INT8 model does not meet business requirements, you can try using mixed precision. First, generate the quantization table, as follows:

```
$ run_calibration.py somenet.mlir \
--dataset dataset \
--input_num 100 \
--inference_num 30 \
--expected_cos 0.99 \
--calibration_table somenet_cali_table \
--processor BM1684X \
--search search_qtable \
--quantize_method_list KL,MSE \
--quantize_table somenet_qtable
```

Then pass the quantization table to generate the model

```
$ model_deploy.py \
--mlir somenet.mlir \
--quantize INT8 \
--calibration_table somenet_cali_table \
--quantize_table somenet_qtable \
--processor BM1684X \
--model somenet_mix.bmodel
```

### 3.1.5 Support for Quantized TFLite Models

TFLite model conversion is also supported, with the following command:

```
# TFLite conversion example
$ model_transform.py \
--model_name resnet50_tf \
--model_def ../resnet50_int8.tflite \
--input_shapes [[1,3,224,224]] \
--mean 103.939,116.779,123.68 \
--scale 1.0,1.0,1.0 \
--pixel_format bgr \
--test_input ..image/dog.jpg \
--test_result resnet50_tf_top_outputs.npz \
--mlir resnet50_tf.mlir

$ model_deploy.py \
--mlir resnet50_tf.mlir \
--quantize INT8 \
--processor BM1684X \
--test_input resnet50_tf_in_f32.npz \
--test_reference resnet50_tf_top_outputs.npz \
--tolerance 0.95,0.85 \
--model resnet50_tf_1684x.bmodel
```

### 3.1.6 Support for Caffe Models

```
# Caffe conversion example
$ model_transform.py \
--model_name resnet18_cf \
--model_def ./resnet18.prototxt \
--model_data ./resnet18.caffemodel \
--input_shapes [[1,3,224,224]] \
--mean 104,117,123 \
--scale 1.0,1.0,1.0 \
--pixel_format bgr \
--test_input ./image/dog.jpg \
--test_result resnet50_cf_top_outputs.npz \
--mlir resnet50_cf.mlir
```

### 3.1.7 Support LLM models

```
$ llm_convert.py \
-m /workspace/Qwen2.5-VL-3B-Instruct-AWQ \
-s 2048 \
-q w4bf16 \
-c bm1684x \
--max_pixels 672,896 \
-o qwen2.5vl_3b
```

## 3.2 Introduction to Tool Parameters

### 3.2.1 model\_transform.py

Used to convert various neural network models into MLIR files (with .mlir suffix) and corresponding weight files ( `${model_name}_top_${quantize}_all_weight.npz`). The supported parameters are as follows:

Table 3.1: Function of model\_transform parameters

Name	Required?	Explanation
model_name	Y	Model name
model_def	Y	Model definition file (e.g., ‘.onnx’ , ‘.tflite’ or ‘.prototxt’ files)
mlir	Y	Specify the output mlir file name and path, with the suffix .mlir
input_shapes	N	The shape of the input, such as [[1,3,640,640]] (a two-dimensional array), which can support multiple inputs

continues on next page

Table 3.1 – continued from previous page

Name	Required?	Explanation
model_extern	N	Extra multi model definition files (currently mainly used for MaskRCNN). None by default. separate by ‘,’
model_data	N	Specify the model weight file, required when it is caffe model (corresponding to the ‘.caffemodel’ file)
input_types	N	When the model is a .pt file, specify the input type, such as int32; separate multiple inputs with ,. If not specified, it will be treated as float32 by default.
keep_aspect_ratio	N	When the size of test_input is different from input_shapes, whether to keep the aspect ratio when resizing, the default is false; when set, the insufficient part will be padded with 0
mean	N	The mean of each channel of the image. The default is 0.0,0.0,0.0
scale	N	The scale of each channel of the image. The default is 1.0,1.0,1.0
pixel_format	N	Image type, can be rgb, bgr, gray or rgbd. The default is bgr
channel_format	N	Channel type, can be nhwc or nchw for image input, otherwise it is none. The default is nchw
output_names	N	The names of the output. Use the output of the model if not specified, otherwise output in the order of the specified names
add_postprocess	N	add postprocess op into bmodel, set the type of post handle op such as yolov3/yolov3_tiny/yolov5/yolov8/yolov11/ssd/yolov8_seg
test_input	N	The input file for verification, which can be an jpg, npy or npz file. No verification will be carried out if it is not specified
test_result	N	Output file to save verification result with suffix .npz
excepts	N	Names of network layers that need to be excluded from verification. Separated by comma
onnx_sim	N	option for onnx-sim, currently only support ‘skip_fuse_bn’ args
debug	N	If open debug, immediate model file will keep; or will remove after conversion done
tolerance	N	Minimum Cosine and Euclidean similarity tolerance to model transform. 0.99,0.99 by default.
cache_skip	N	skip checking the correctness when generate same mlir and bmodel
dynamic_shape_input_	N	Name list of inputs with dynamic shape, like:input1,input2. If set, ‘–dynamic’ is required during model_deploy.

continues on next page

Table 3.1 – continued from previous page

Name	Required?	Explanation
shape_influencing_ir	N	Name list of inputs which influencing other tensors' shape during inference, like:input1,input2. If set, test_input is required and ‘–dynamic’ is required during model_deploy.
dynamic	N	Only valid for onnx model. If set, will automatically set inputs with dyanmic axis as dynamic_shape_input_names and set 1-d inputs as shape_influencing_input_names and ‘–dynamic’ is required during model_deploy.
resize_dims	N	The original image size ‘h,w’ , default is same as net input dims
pad_value	N	pad value when resize
pad_type	N	type of pad when resize, such as normal/center
preprocess_list	N	choose which input need preprocess, like:’1,3’ means input 1&3 need preprocess, default all inputs
path_yaml	N	the path for one single yaml file (currently mainly used for MaskRCNN)
enable_maskrcnn	N	if enable MaskRCNN transformation
yuv_type	N	Specify its type when using the ‘.yuv’ file as input

After converting to an mlir file, a \${model\_name}\_in\_f32.npz file will be generated, which is the input file for the subsequent models.

### 3.2.2 run\_calibration.py

Use a small number of samples for calibration to get the quantization table of the network (i.e., the threshold/min/max of each layer of op).

Supported parameters:

Table 3.2: Function of run\_calibration parameters

Name	Required?	Explanation
(None)	Y	Mlir file
sq	N	Open SmoothQuant
we	N	Open weight_equalization
bc	N	Open bias_correction
dataset	N	Directory of input samples. Images, npz or npy files are placed in this directory
data_list	N	The sample list (cannot be used together with “dataset” )
input_num	N	The number of input for calibration. Use all samples if it is 0

continues on next page

Table 3.2 – continued from previous page

Name	Required?	Explanation
inference_num	N	The number of images required for the inference process of search_qtable and search_threshold
bc_inference_num	N	The number of images required for the inference process of bias_correction
tune_num	N	The number of fine-tuning samples. 10 by default
tune_list	N	Tune list file contain all input for tune
histogram_bin_num	N	The number of histogram bins. 2048 by default
expected_cos	N	The expected similarity between the mixed-precision model output and the floating-point model output in search_qtable, with a value range of [0,1]
min_layer_cos	N	The minimum similarity between the quantized output and the floating-point output of a layer in bias_correction. Compensation is required for the layer when the similarity is below this threshold, with a value range of [0,1]
max_float_layers	N	The number of floating-point layers in search_qtable
processor	N	Processor type
cali_method	N	Choose quantization threshold calculation method
fp_type	N	The data type of floating-point layers in search_qtable
post_process	N	The path for post-processing
global_compare_layers	N	Specifies the global comparison layers, for example, layer1,layer2 or layer1:0.3,layer2:0.7
search	N	Specifies the type of search, including search_qtable, search_threshold, false. The default is false, which means search is not enabled
transformer	N	Whether it is a transformer model, if it is, search_qtable can allocate specific acceleration strategies
quantize_method_list	N	The threshold methods used for searching in search_qtable
benchmark_method	N	Specifies the method for calculating similarity in search_threshold
kurtosis_analysis	N	Specify the generation of the kurtosis of the activation values for each layer
part_quantize	N	Specify partial quantization of the model. The calibration table (cali_table) will be automatically generated alongside the quantization table (qtable). Available modes include N_mode, H_mode, or custom_mode, with H_mode generally delivering higher accuracy

continues on next page

Table 3.2 – continued from previous page

Name	Required?	Explanation
custom_operator	N	Specify the operators to be quantized, which should be used in conjunction with the aforementioned custom_mode
part_asymmetric	N	When symmetric quantization is enabled, if specific subnets in the model match a defined pattern, the corresponding operators will automatically switch to asymmetric quantization
mix_mode	N	Specify the mixed-precision types for the search_qtable. Currently supported options are 8_16 and 4_8
cluster	N	pecify that a clustering algorithm is used to detect sensitive layers during the search_qtable process
quantize_table	N	The mixed-precision quantization table output by search_qtable
o	Y	Name of output calibration table file
debug_cmd	N	debug cmd
debug_log	N	Log output level

A sample calibration table is as follows:

```
# generated time: 2022-08-11 10:00:59.743675
# histogram number: 2048
# sample number: 100
# tune number: 5
#####
# op_name threshold min max
images 1.0000080 0.0000000 1.0000080
122_Conv 56.4281803 -102.5830231 97.6811752
124_Mul 38.1586478 -0.2784646 97.6811752
125_Conv 56.1447888 -143.7053833 122.0844193
127_Mul 116.7435987 -0.2784646 122.0844193
128_Conv 16.4931355 -87.9204330 7.2770605
130_Mul 7.2720342 -0.2784646 7.2720342
.....
```

It is divided into 4 columns: the first column is the name of the Tensor; the second column is the threshold (for symmetric quantization); The third and fourth columns are min/max, used for asymmetric quantization.

### 3.2.3 model\_deploy.py

Convert the mlir file into the corresponding model, the parameters are as follows:

Table 3.3: Function of model\_deploy parameters

Name	Required?	Explanation
mlir	Y	Mlir file
processor	Y	The platform that the model will use. Support BM1684, BM1684X, BM1688, BM1690, CV186X, CV183X, CV182X, CV181X, CV180X
quantize	Y	Quantization type (e.g., F32/F16/BF16/INT8), the quantization types supported by different processors are shown in the table below.
quant_input	N	Strip input type cast in bmodel, need outside type conversion
quant_output	N	Strip output type cast in bmodel, need outside type conversion
quant_input_list	N	choose index to strip cast, such as 1,3 means first & third input's cast
quant_output_list	N	Choose index to strip cast, such as 1,3 means first & third output's cast
quantize_table	N	Specify the path to the mixed precision quantization table. If not specified, quantization is performed according to the quantize type; otherwise, quantization is prioritized according to the quantization table
fuse_preprocess	N	Specify whether to fuse preprocessing into the model. If this parameter is specified, the model input will be of type uint8, and the resized original image can be directly input
calibration_table	N	The quantization table path. Required when it is INT8/F8E4M3 quantization
high_precision	N	Some ops will force to be float32
tolerance	N	Tolerance for the minimum Cosine and Euclidean similarity between MLIR quantized and MLIR fp32 inference results. 0.8,0.5 by default.
test_input	N	The input file for verification, which can be an jpg, npy or npz. No verification will be carried out if it is not specified
test_reference	N	Reference data for verifying mlir tolerance (in npz format). It is the result of each operator
excepts	N	Names of network layers that need to be excluded from verification. Separated by comma

continues on next page

Table 3.3 – continued from previous page

Name	Required?	Explanation
op_divide	N	CV183x/CV182x/CV181x/CV180x only, Try to split the larger op into multiple smaller op to achieve the purpose of ion memory saving, suitable for a few specific models
model	Y	Name of output model file (including path)
debug	N	to keep all intermediate files for debug
asymmetric	N	Do INT8 asymmetric quantization
dynamic	N	Do compile dynamic
includeWeight	N	Include weight in tosa.mlir
customization_format	N	Pixel format of input frame to the model
compare_all	N	Decide if compare all tensors when lowering
num_device	N	The number of devices to run for distributed computation
num_core	N	The number of Tensor Computing Processor cores used for parallel computation
skip_verification	N	Skip checking the correctness of bmodel
merge_weight	N	Merge weights into one weight binary with previous generated cvimodel
model_version	N	If need old version cvimodel, set the verion, such as 1.2
q_group_size	N	Group size for per-group quant, only used in W4A16/W8A16 quant mode
q_symmetric	N	Do symmetric W4A16/W8A16 quant
compress_mode	N	Specify the compression mode of the model: “none” , “weight” , “activation” , “all” . Supported on BM1688. Default is “none” , no compression
opt_post_processor	N	Specify whether to further optimize the results of LayerGroup. Supported on MARS3. Default is “none” , no opt
lgcache	N	Specifies whether to cache the partitioning results of LayerGroup: “true” , “false” . The default is “true” , which saves the partitioning results of each subnet to the working directory as “cut_result_{subnet_name}.mlircache” .
cache_skip	N	skip checking the correctness when generate same mlir and bmodel
aligned_input	N	if the input frame is width/channel aligned. VPSS input alignment for CV series processors only
group_by_cores	N	whether layer groups force group by cores, auto/true/false, default is auto

continues on next page

Table 3.3 – continued from previous page

Name	Required?	Explanation
opt	N	Optimization type of LayerGroup, 1/2/3, default is 2. 1: Simple LayerGroup mode, all operators will be grouped as much as possible, and the compilation speed is faster; 2: Dynamic compilation calculates the global cycle optimal Group grouping, suitable for inference graphs; 3: Linear programming LayerGroup mode, suitable for training graphs.
addr_mode	N	set address assign mode [ ‘auto’ , ‘basic’ , ‘io_alone’ , ‘io_tag’ , ‘io_tag_fuse’ , ‘io_reloc’ ], if not set, auto as default
dis- able_layer_group	N	Whether to disable LayerGroup pass
dis- able_gdma_check	N	Whether to disable gdma address check
do_winograd	N	if do WinoGrad convolution, only for BM1684
mat- mul_perchannel	N	if matmul is quantized in per-channel mode, for BM1684X and BM1688, the performance may be decreased if enable
enable_maskrcnn	N	if enable comparison for MaskRCNN.

The following table shows the correspondence between different processors and the supported quantize types:

Table 3.4: Quantization types supported by different processors

Processor	Supported quantize
BM1684	F32, INT8
BM1684X	F32, F16, BF16, INT8, W4F16, W8F16, W4BF16, W8BF16
BM1688	F32, F16, BF16, INT8, INT4, W4F16, W8F16, W4BF16, W8BF16
BM1690	F32, F16, BF16, INT8, F8E4M3, F8E5M2, W4F16, W8F16, W4BF16, W8BF16
CV186X	F32, F16, BF16, INT8, INT4
CV183X, CV182X, CV181X, BF16, INT8	
CV180X	

The Weight-only quantization mode of W4A16 and W8A16 only applies to the MatMul operation, and other operators will still perform F16 or BF16 quantization.

### 3.2.4 llm\_convert.py

Convert the LLM model into bmodel, the parameters are as follows:

Table 3.5: llm\_convert Parameter Functions

Parameter	Re-required?	Description
model_path	Yes	Specifies the path to the model
seq_length	Yes	Specifies the maximum sequence length
quantize	Yes	Specifies the quantization type, e.g., w4bf16/w4f16/bf16/f16
q_group_size	No	Specifies the group size for quantization
chip	Yes	Specifies the processor type; supports bm1684x/bm1688/cv186ah
max_pixels	No	Multimodal parameter; specifies the maximum dimensions, either “672,896” or “602112”
num_device	No	Specifies the number of devices for bmodel deployment
num_core	No	Specifies the number of cores for bmodel deployment; 0 means use the maximum available cores
embedding_disk	No	If set, exports the word embeddings to a binary file and runs inference on the CPU
out_dir	Yes	Specifies the output directory for the bmodel file

### 3.2.5 model\_runner.py

Model inference. mlir/pytorch/onnx/tflite/bmodel/prototxt supported.

Example:

```
$ model_runner.py \
--input sample_in_f32.npz \
--model sample.bmodel \
--output sample_output.npz \
--out_fixed
```

Supported parameters:

Table 3.6: Function of model\_runner parameters

Name	Required?	Explanation
input	Y	Input npz file
model	Y	Model file (mlir/pytorch/onnx/tflite/bmodel/prototxt)
dump_all_tensors	N	Export all the results, including intermediate ones, when specified
out_fixed	N	Remain integer output when the dtype of output is int8, instead of transforming to float32 automaticall

### 3.2.6 npz\_tool.py

npz will be widely used in TPU-MLIR project for saving input and output results, etc.  
npz\_tool.py is used to process npz files.

Example:

```
# Check the output data in sample_out.npz
$ npz_tool.py dump sample_out.npz output
```

Supported functions:

Table 3.7: npz\_tool functions

Function	Description
dump	Get all tensor information of npz
compare	Compare difference of two npz files
to_dat	Export npz as dat file, contiguous binary storage

### 3.2.7 visual.py

visual.py is an visualized network/tensor compare application with interface in web browser, if accuracy of quantized network is not as good as expected, this tool can be used to investigate the accuracy in every layer.

Example:

```
# use TCP port 9999 in this example
$ visual.py \
--f32_mlir netname.mlir \
--quant_mlir netname_int8_sym_tpu.mlir \
--input top_input_f32.npz --port 9999
```

Supported functions:

Table 3.8: visual functions

Function	Description
f32_mlir	fp32 mlir file
quant_mlir	quantized mlir file
input	test input data for networks, can be in jpeg or npz format.
port	TCP port used for UI, default port is 10000, the port should be mapped when starting docker
host	Host ip, default:0.0.0.0
manual_run	if net will be automaticall inferenced when UI is opened, default is false for auto inference

Notice: --debug flag should be opened during model\_deploy.py to save intermediate files for visual.py. More details refer to ([visual tool introduction](#))

### 3.2.8 mlir2graph.py

Visualizes MLIR files based on dot, supporting MLIR files from all stages. After execution, corresponding .dot and .svg files will be generated in the MLIR directory. The .dot file can be rendered into other formats using the dot command. .svg is the default output rendering format and can be directly opened in a browser.

Execution command example:

```
$ mlir2graph.py \
--mlir netname.mlir
```

For large MLIR files, the original rendering algorithm for dot files may take a long time. You can add the --is\_big parameter to reduce the iteration time of the algorithm and generate the graph faster:

```
$ mlir2graph.py \
--mlir netname.mlir --is_big
```

Supported functions:

Table 3.9: mlir2graph functions

Function	Description
mlir	Any MLIR file
is_big	Indicates whether the MLIR file is relatively large; there is no specific criterion, usually judged based on rendering time
failed_keys	List of failed node names for comparison, separated by “,” , nodes corresponding to these keys will be rendered in red after rendering
bmodel_checker_d@	Path to the failed.npz file generated by bmodel_checker.py; when this path is specified, it will automatically parse the error nodes and render them in red
output	Path to the output file, default is the path of --mlir with the corresponding format suffix, such as netname.mlir.dot/netname.mlir.svg

### 3.2.9 gen\_rand\_input.py

During model transform, if you do not want to prepare additional test data (test\_input), you can use this tool to generate random input data to facilitate model verification.

The basic procedure is transforming the model into a mlir file with `model_transform.py`. This step does not perform model verification. And then use `gen_rand_input.py` to read

the mlir file generated in the previous step and generate random test data for model verification. Finally, use `model_transform.py` again to do the complete model transformation and verification.

Example:

```
# To MLIR
$ model_transform.py \
--model_name yolov5s \
--model_def ./regression/model/yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio --pixel_format rgb \
--output_names 350,498,646 \
--mlir yolov5s.mlir

# Generate dummy input. Here is a pseudo test picture.
$ gen_rand_input.py \
--mlir yolov5s.mlir \
--img --output yolov5s_fake_img.png

# Verification
$ model_transform.py \
--model_name yolov5s \
--model_def ./regression/model/yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--test_input yolov5s_fake_img.png \
--test_result yolov5s_top_outputs.npz \
--keep_aspect_ratio --pixel_format rgb \
--output_names 350,498,646 \
--mlir yolov5s.mlir
```

For more detailed usage, please refer to the following:

```
# Value ranges can be specified for multiple inputs.
$ gen_rand_input.py \
--mlir ernie.mlir \
--ranges [[0,300],[0,0]] \
--output ern.npz

# Type can be specified for the input.
$ gen_rand_input.py \
--mlir resnet.mlir \
--ranges [[0,300]] \
--input_types si32 \
--output resnet.npz

# Generate random image
$ gen_rand_input.py \
```

(continues on next page)

(continued from previous page)

```
--mlir yolov5s.mlir \
--img --output yolov5s_fake_img.png
```

Supported functions:

Table 3.10: gen\_rand\_input functions

Name	Re- quired?	Explanation
mlir	Y	Specify the output mlir file name and path, with the suffix .mlir
img	N	Used for CV tasks to generate random images, otherwise generate npz files. The default image value range is [0,255], the data type is ‘uint8’ , and cannot be changed.
ranges	N	Set the value ranges of the model inputs, expressed in list form, such as [[0,300],[0,0]]. If you want to generate a picture, you do not need to specify the value range, the default is [0,255]. In other cases, value ranges need to be specified.
input_types	N	Set the model input types, such as ‘si32,f32’ . ‘si32’ and ‘f32’ types are supported. False by default, and it will be read from mlir. If you generate an image, you do not need to specify the data type, the default is ‘uint8’ .
output	Y	The names of the output.

Notice: CV-related models usually perform a series of preprocessing on the input image. To ensure that the model is verified correctly, you need to use ‘-img’ to generate a random image as input. Random npz files cannot be used as input. It is worth noting that random input may cause model correctness verification to fail, especially NLP-related models, so it is recommended to give priority to using real test data.

### 3.2.10 model\_tool

This tool is used to process the final model file “bmodel” or “cvimodel” . All arguments and corresponding function descriptions can be viewed by executing the following command:

```
$ model_tool
```

The following uses “xxx.bmodel” as an example to introduce the main functions of this tool.

- 1) show basic info of bmodel

Example:

```
$ model_tool --info xxx.bmodel
```

Displays the basic information of the model, including the compiled version of the model, the compilation date, the name of the network in the model, input and output parameters, etc. The display effect is as follows:

```
bmodel version: B.2.2+v1.7.beta.134-ge26380a85-20240430
processor: BM1684X
create time: Tue Apr 30 18:04:06 2024

kernel_module name: libbm1684x_kernel_module.so
kernel_module size: 3136888
=====
net 0: [block_0] static
-----
stage 0:
input: input_states, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
input: position_ids, [1, 512], int32, scale: 1, zero_point: 0
input: attention_mask, [1, 1, 512, 512], bfloat16, scale: 1, zero_point: 0
output: /layer/Add_1_output_0_Add, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Add_1_output_0_Add, [1, 1, 512, 256], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Transpose_2_output_0_Transpose, [1, 1, 512, 256], bfloat16, scale: 1,[F]
→zero_point: 0
=====
net 1: [block_1] static
-----
stage 0:
input: input_states, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
input: position_ids, [1, 512], int32, scale: 1, zero_point: 0
input: attention_mask, [1, 1, 512, 512], bfloat16, scale: 1, zero_point: 0
output: /layer/Add_1_output_0_Add, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Add_1_output_0_Add, [1, 1, 512, 256], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Transpose_2_output_0_Transpose, [1, 1, 512, 256], bfloat16, scale: 1,[F]
→zero_point: 0

device mem size: 181645312 (weight: 121487360, instruct: 385024, runtime: 59772928)
host mem size: 0 (weight: 0, runtime: 0)
```

## 2) combine multi bmodels

Example:

```
$ model_tool --combine a.bmodel b.bmodel c.bmodel -o abc.bmodel
```

Merge multiple bmodels into one bmodel. If there is a network with the same name in the bmodel, it will be divided into different stages.

## 3) extract model to multi bmodels

Example:

```
$ model_tool --extract abc.bmodel
```

Decomposing a bmodel into multiple bmodels is the opposite operation to the combine command. It will be divided into different stages.

#### 4) show weight info

Example:

```
$ model_tool --weight xxx.bmodel
```

Display the weight range information of each operator in different networks. The display effect is as follows:

```
net 0 : "block_0", stage:0
=====
tpu.Gather : [0x0, 0x40000)
tpu.Gather : [0x40000, 0x80000)
tpu.RMSNorm : [0x80000, 0x81000)
tpu.A16MatMul : [0x81000, 0x2b1000)
tpu.A16MatMul : [0x2b1000, 0x2f7000)
tpu.A16MatMul : [0x2f7000, 0x33d000)
tpu.A16MatMul : [0x33d000, 0x56d000)
tpu.RMSNorm : [0x56d000, 0x56e000)
tpu.A16MatMul : [0x56e000, 0x16ee000)
tpu.A16MatMul : [0x16ee000, 0x286e000)
tpu.A16MatMul : [0x286e000, 0x39ee000)
=====
net 1 : "block_1", stage:0
=====
tpu.Gather : [0x0, 0x40000)
tpu.Gather : [0x40000, 0x80000)
tpu.RMSNorm : [0x80000, 0x81000)
tpu.A16MatMul : [0x81000, 0x2b1000)
tpu.A16MatMul : [0x2b1000, 0x2f7000)
tpu.A16MatMul : [0x2f7000, 0x33d000)
tpu.A16MatMul : [0x33d000, 0x56d000)
tpu.RMSNorm : [0x56d000, 0x56e000)
tpu.A16MatMul : [0x56e000, 0x16ee000)
tpu.A16MatMul : [0x16ee000, 0x286e000)
tpu.A16MatMul : [0x286e000, 0x39ee000)
=====
```

#### 5) update weight from one bmodel to dst bmodel

Example:

```
# Update the weight of the network named src_net in src.bmodel at the 0x2000 position to the F
→ 0x1000 position of dst_net in dst.bmodel
$ model_tool --update_weight dst.bmodel dst_net 0x1000 src.bmodel src_net 0x2000
```

The model weights can be updated. For example, if the weight of an operator of a certain

model needs to be updated, compile the operator separately into bmodel, and then update its weight to the original model.

#### 6) model encryption and decryption

Example:

```
# -model specifies the combined or regular bmodel, -net specifies the network to be encrypted, -  
→lib specifies the library implementing the encryption algorithm, -o specifies the name of the F  
→encrypted model output  
$ model_tool --encrypt -model combine.bmodel -net block_0 -lib libcipher.so -o encrypted.bmodel  
$ model_tool --decrypt -model encrypted.bmodel -lib libcipher.so -o decrypted.bmodel
```

This can achieve the encryption of model weights, flatbuffer structured data, and headers. The encryption and decryption interfaces must be implemented in C style, not using C++. The interface specifications are as follows:

```
extern "C" uint8_t* encrypt(const uint8_t* input, uint64_t input_bytes, uint64_t* output _  
→bytes);  
extern "C" uint8_t* decrypt(const uint8_t* input, uint64_t input_bytes, uint64_t* output _  
→bytes);
```

# CHAPTER 4

---

## Overall Design

---

### 4.1 Layered

TPU-MLIR treats the compilation process of the network model in two layers.

#### Top Dialect

Hardware-independent layer, including graph optimization, quantization and inference, etc.

#### Tpu Dialect

Hardware-related layer, including weight reordering, operator slicing, address assignment, inference, etc.

The overall flow is shown in the ([TPU-MLIR overall process](#)) diagram, where the model is gradually converted into final instructions by Passes. Here is a detailed description of what functions each Pass does in the Top layer and the Tpu layer. The following chapters will explain the key points of each Pass in detail.

### 4.2 Top Pass

#### shape-infer

Do shape inference, and constant folder

#### canonicalize

Graph optimization related to specific OP, such as merging relu into conv, shape merge, etc.

#### extra-optimize

Do extra patterns, such as get FLOPs, remove unuse output, etc.

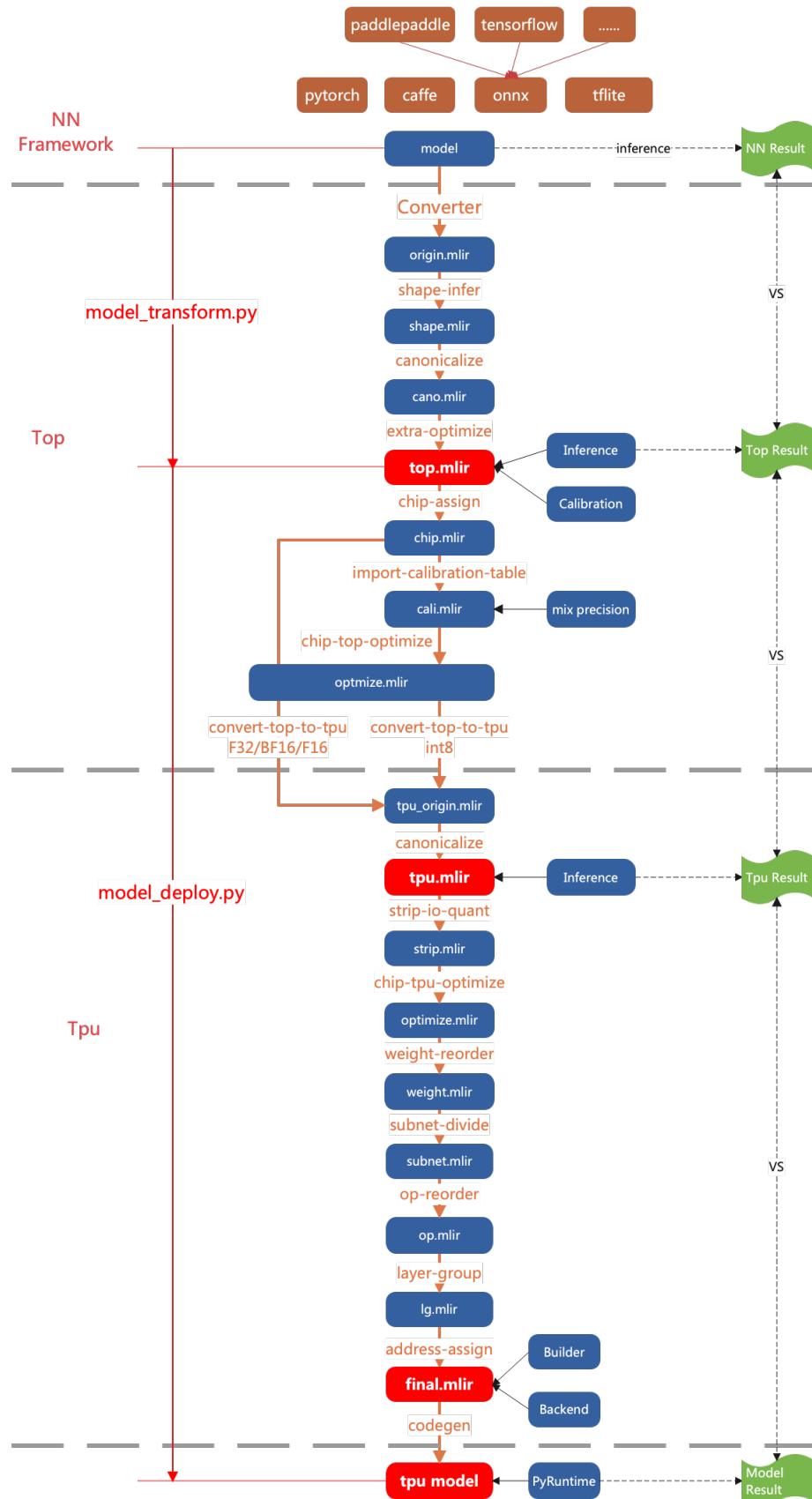


Fig. 4.1: TPU-MLIR overall process  
Copyright © SOPHGO

**processor-assign**

Assign processor, such as BM1684X, CV183X, etc; and adjust top mlir by processor, for example, make all CV18XX input types as F32.

**import-calibration-table**

Import calibration table, assign min and max for all ops, for quantization later.

**processor-top-optimize**

Do top ops optimization by processor.

**convert-top-to-tpu**

Lower top ops to tpu ops; if for mode F32/F16/BF16, top op normally convert to tpu op directly; if INT8, quantization is needed.

## 4.3 Tpu Pass

**canonicalize**

Graph optimization related to specific OP, such as merging of consecutive Requestants, etc.

**strip-io-quant**

Input and output types will be quantized if true; or be F32

**processor-tpu-optimize**

Do tpu ops optimization by processor.

**weight-reorder**

Reorder the weights of individual OP based on processor characteristics, such as filter and bias for convolution.

**subnet-divide**

Divide the network into various subnets based on the processor type. If the Tensor Competing Processor can compute all operators, then it forms a single subnet.

**op-reorder**

Reorder op to make sure ops are close to their users.

**layer-group**

Slice the network so that as many OPs as possible are computed consecutively in the local mem.

**address-assign**

Assign addresses to the OPs that need global mem.

**codegen**

Use Builder module to generate the final model in flatbuffers format.

## 4.4 Other Passes

There are some optional passes, not in the diagram, used for special functions.

### **fuse-preprocess**

Fuse image preprocess to model.

### **add-postprocess**

add postprocess to model, only support ssd/yolov3/yolov5.

# CHAPTER 5

---

## Front-end Conversion

---

This chapter takes the onnx model as an example to introduce the front-end conversion process of models/operators in this project.

### 5.1 Main Work

The front-end is mainly responsible for transforming the original model into a Top (hardware-independent) mlir model (without the Canonicalize part, so the generated file is named “\*\_origin.mlir”). This process creates and adds the corresponding operators (Op) based on the original model and the input arguments when running `model_transform.py`. The transformed model and weights will be saved in mlir and npz file respectively.

### 5.2 Workflow

1. Prereq: definition of the Top layer operator in `TopOps.td`.
  2. Input: input the original onnx model and arguments (preprocessing arguments mainly).
  3. Initialize OnnxConverter (`load_onnx_model + initMLIRImporter`).
    - `load_onnx_model` part is mainly to refine the model, intercept the model according to the `output_names` in arguments, and extract the relevant information from the refined model.
    - The `init_MLIRImporter` part generates the initial mlir text.
  4. generate\_mlir
-

- Create the input op, the model intermediate nodes op and the return op in turn and add them to the mlir text (if the op has tensors, additional weight op will be created).

## 5. Output

- Save the simplified model as a “\*\_opt.onnx” file
- Generate a “.prototxt” file to save the model information except the weights
- Convert the generated text to str and save it as a “.mlir” file
- Save model weights (tensors) in “.npz” file

The workflow of the front-end conversion is shown in the figure (Front-end conversion workflow).

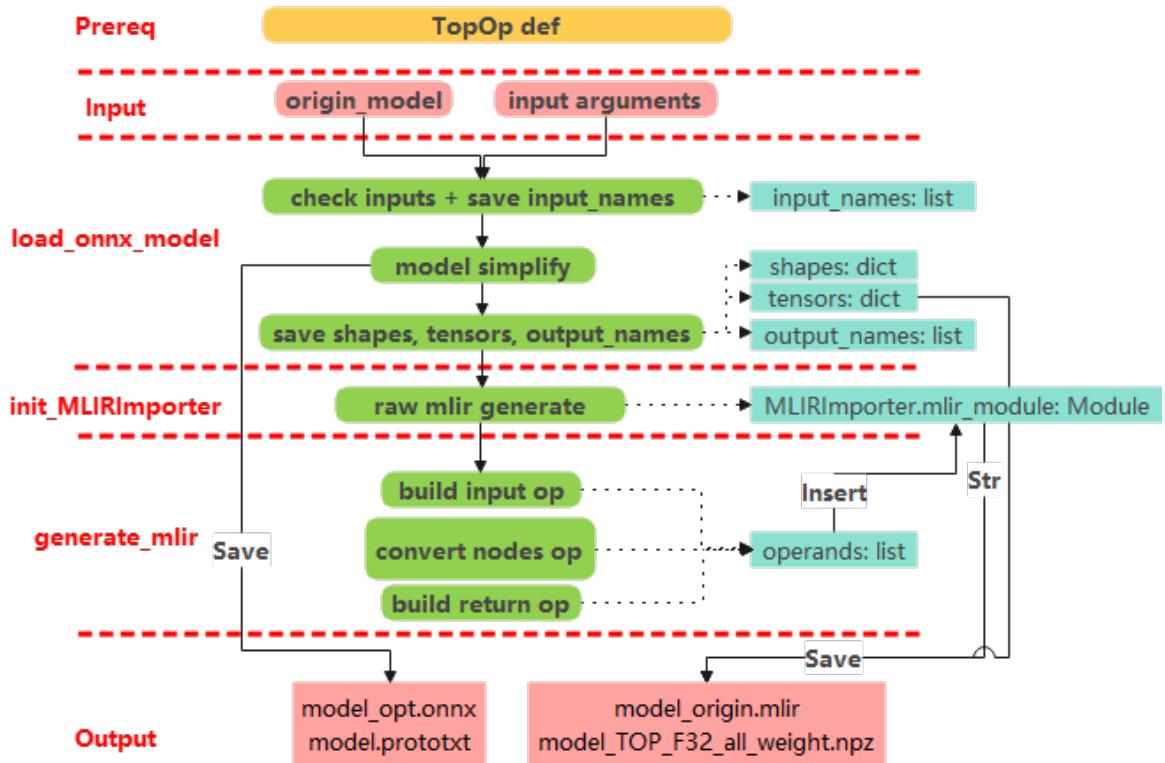


Fig. 5.1: Front-end conversion workflow

### Additional Notes:

- Build input op requires:
  1. input\_names.
  2. index for each input.
  3. preprocessing arguments (if the input is an image).
- Convert nodes op requires:

1. former ops.
  2. the output\_shape from `shapes`.
  3. attrs extracted from the onnx node. Attrs are set by MLIRImporter according to definition in TopOps.td.
- Build return op requires:
    - output ops according to `output_names`.
  - Insertion operation is performed for each op conversion or creation. The operator is inserted into the mlir text so that the final generated text can one-to-one correspond with the original onnx model.

### 5.3 Example

This section takes the Conv onnx operator as an example for Top mlir conversion. The original model is shown in the figure ([Conv onnx model](#)).

The conversion process:

#### 1. Conv op definition

Define the Top.Conv operator in TopOps.td. The definition is shown in the figure ([Top.Conv definition](#)).

#### 2. Initialize OnnxConverter

`load_onnx_model`:

- Since this example uses the simplest model, the resulting Conv\_opt.onnx model is the same as the original one.
- `input_names` for saving input name “input” of Conv op.
- The weight and bias of the Conv op are stored in `tensors`.
- `shapes` saves `input_shape` and `output_shape` of conv op.
- `output_names` holds the output name of the Conv op “output” .

`init_MLIRImporter`:

The initial mlir text `MLIRImporter.mlir_module` is generated based on model name, input shape and output shape from `shapes`, as shown in the figure ([Initial mlir text](#)).

#### 3. generate\_mlir

- build `input_op`, the generated `Top.inputOp` will be inserted into `MLIRImporter.mlir_module`.
- call `convert_conv_op()`, which calls `MLIRImporter.create_conv_op` to create a `ConvOp`, and the `create` function takes the following arguments.

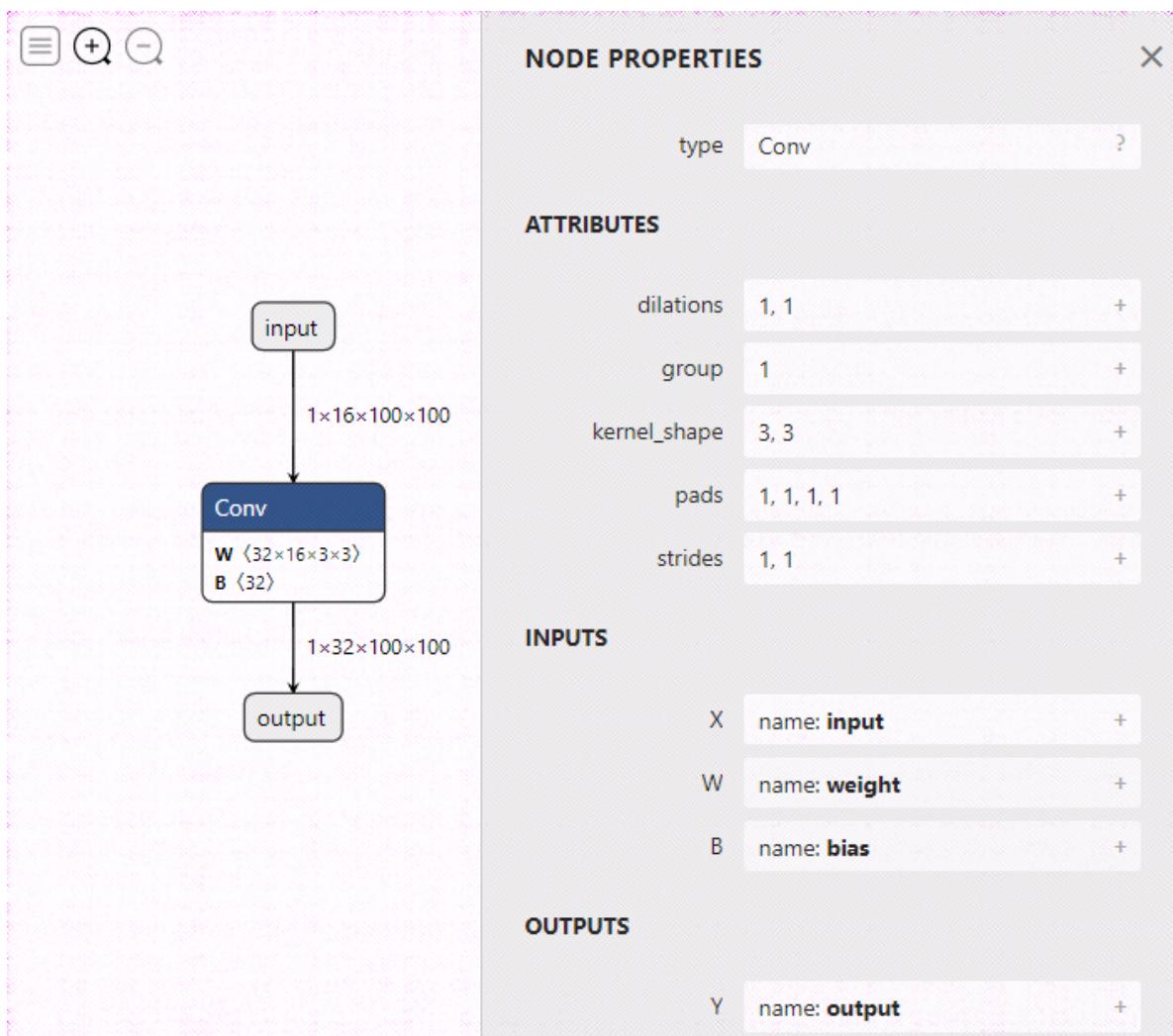


Fig. 5.2: Conv onnx model

```
include > tpu_mlir > Dialect > Top > IR > ≡ TopOps.td
157 def Top_ConvOp: Top_Op<"Conv", [SupportFuseRelu]> {
158     let summary = "Convolution operator";
159
160     let description = [
161         In the simplest case, the output value of the layer with input size
162         .....
163     ];
164
165     let arguments = (ins
166         AnyTensor:$input,
167         AnyTensor:$filter,
168         AnyTensorOrNone:$bias,
169         I64ArrayAttr:$kernel_shape,
170         I64ArrayAttr:$strides,
171         I64ArrayAttr:$pads, // top,left,bottom,right
172         DefaultValuedAttr<I64Attr, "1">:$group,
173         OptionalAttr<I64ArrayAttr>:$dilations,
174         OptionalAttr<I64ArrayAttr>:$inserts,
175         DefaultValuedAttr<BoolAttr, "false">:$do_relu,
176         OptionalAttr<F64Attr>:$upper_limit,
177         StrAttr:$name
178     );
179
180     let results = (outs AnyTensor:$output);
181     let extraClassDeclaration = [
182         void parseParam(int64_t &n, int64_t &ic, int64_t &ih, int64_t &iw, int64_t &oc,
183                     int64_t &oh, int64_t &ow, int64_t &g, int64_t &kh, int64_t &kw, int64_t &
184                     ins_h,
185                     int64_t &ins_w, int64_t &sh, int64_t &sw, int64_t &pt, int64_t &pb,
186                     int64_t &pl,
187                     int64_t &pr, int64_t &dh, int64_t &dw, bool &is_dw, bool &with_bias, bool &
188                     do_relu,
189                     float &relu_upper_limit);
190     ];
191 }
192 }
```

Fig. 5.3: Top.Conv definition

```
module attributes {module.chip = "ALL", module.name = "Conv2d", module.state = "TOP_F
32", module.weight_file = "conv2d_top_f32_all_weight.npz"} {
  func.func @main(%arg0: tensor<1x16x100x100xf32>) -> tensor<1x32x100x100xf32> {
    %0 = "top.None"() : () -> none
  }
}
```

Fig. 5.4: Initial mlir text

- 1) inputOp: from ([Conv onnx model](#)), we can see that inputs of the Conv operator contain input, weight and bias. inputOp has been created, and the op of weight and bias will be created by getWeightOp().
- 2) output\_shape: use onnx\_node.name to get the output shape of the Conv operator from shapes.
- 3) Attributes: get attributes such as ([Conv onnx model](#)) from the onnx Conv operator.

The attributes of the Top.Conv operator are set according to the definition in ([Top.Conv definition](#)). Top.ConvOp will be inserted into the MLIR text after it is created.

- Get the output op from operands based on output\_names to create return\_op and insert it into the mlir text. Up to this point, the generated mlir text is shown ([Complete mlir text](#)).

```
onnx_test > ≡ Conv2d_origin.mlir
1 module attributes {module.chip = "ALL", module.name = "Conv2d", module.state = "TOP_F32",
2   module.weight_file = "conv2d_top_f32_all_weight.npz"} {
3     func.func @main(%arg0: tensor<1x16x100x100xf32>) -> tensor<1x32x100x100xf32> [
4       %0 = "top.None"() : () -> none
5       inputOp %1 = "top.Input"(%arg0) {name = "input"} : (tensor<1x16x100x100xf32>) -> tensor<1x16x100x100xf32>
6       weightOp %2 = "top.Weight"() {name = "weight"} : () -> tensor<32x16x3x3xf32>
7       %3 = "top.Weight"() {name = "bias"} : () -> tensor<32xf32>
8       ConvOp %4 = "top.Conv"(%1, %2, %3) {dilations = [1, 1], do_relu = false, group = 1 : i64, kernel_shape = [3, 3],
9         name = "output_Conv", pads = [1, 1, 1, 1], strides = [1, 1]} : (tensor<1x16x100x100xf32>,
10           tensor<32x16x3x3xf32>, tensor<32xf32>) -> tensor<1x32x100x100xf32>
11       returnOp return %4 : tensor<1x32x100x100xf32>
12     ]
13 }
```

Fig. 5.5: Complete mlir text

#### 4. Output

Save the mlir text as Conv\_origin.mlir and the weights in the tensors as Conv\_TOP\_F32\_all\_weight.npz.

# CHAPTER 6

---

## Quantization

---

The theory of quantization is based on: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

Paper link: <https://arxiv.org/abs/1712.05877>

This chapter introduces the quantization design of TPU-MLIR, focusing on the application of the paper in practical quantization.

### 6.1 Basic Concepts

INT8 quantization is divided into symmetric and asymmetric quantization. Symmetric quantization is a special case of asymmetric quantization, and usually, the performance of the former will be better than the latter, while the accuracy is in contrast.

#### 6.1.1 Asymmetric Quantization

As shown in the figure ([Asymmetric quantization](#)), asymmetric quantization is actually the fixed-pointing of values in the range [min,max] to the interval [-128, 127] or [0, 255].

The quantization formula from int8 to float is:

$$\begin{aligned} r &= S(q - Z) \\ S &= \frac{\max - \min}{q_{\max} - q_{\min}} \\ Z &= \text{Round}\left(-\frac{\min}{S} + q_{\min}\right) \end{aligned}$$

where  $r$  is the real value of type float and  $q$  is the quantized value of type INT8 or UINT8.

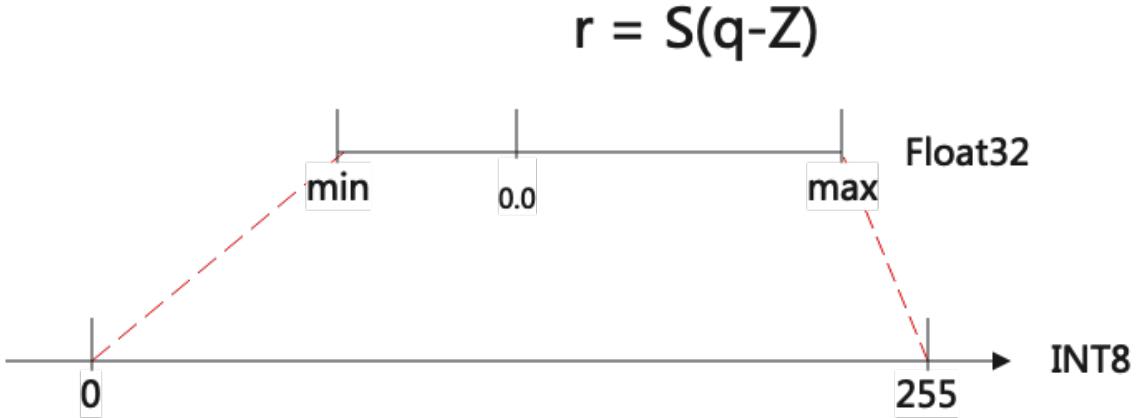


Fig. 6.1: Asymmetric quantization

$S$  denotes scale, which is float;  $Z$  is zeropoint, which is of type INT8.

When quantized to INT8,  $q_{\max}=127, q_{\min}=-128$ , and for UINT8,  $q_{\max}=255, q_{\min}=0$ .

The quantization formula from float to INT8 is:

$$q = \frac{r}{S} + Z$$

### 6.1.2 Symmetric Quantization

Symmetric quantization is a special case of asymmetric quantization when  $Z=0$ . The formula is:

$$\begin{aligned} i8\_value &= f32\_value \times \frac{128}{threshold} \\ f32\_value &= i8\_value \times \frac{threshold}{128} \end{aligned}$$

The range of Tensor is  $[-\text{threshold}, \text{threshold}]$ .

For activation, usually  $S = \text{threshold}/128$ .

For weight, usually  $S = \text{threshold}/127$ .

In the case of UINT8, the Tensor range is  $[0, \text{threshold}]$ , at this time  $S = \text{threshold}/255.0$ .

## 6.2 Scale Conversion

The formula in the paper:

$$M = 2^{-n} M_0, \text{ where the range of } M_0 \text{ is } [0.5, 1], \text{ and } n \text{ is a non-negative number}$$

In other words, it is the floating point Scale, which can be converted to Multiplier and rshift:

$$\text{Scale} = \frac{\text{Multiplier}}{2^{\text{rshift}}}$$

For example:

$$\begin{aligned} y &= x \times 0.1234 \\ &\Rightarrow y = x \times 0.9872 \times 2^{-3} \\ &\Rightarrow y = x \times (0.9872 \times 2^{31}) \times 2^{-34} \\ &\Rightarrow y = x \times \frac{2119995857}{1 \ll 34} \\ &\Rightarrow y = (x \times 2119995857) \gg 34 \end{aligned}$$

The higher the number of bits supported by Multiplier, the closer to Scale it will be, but that leads to worse performance. Therefore, generally, the hardware will use a 32-bit or 8-bit Multiplier.

## 6.3 Quantization derivation

We can use quantization formulas and derive quantization for different OPs to get their corresponding INT8 calculations.

Both symmetric and asymmetric are used for Activation, and for weights generally only symmetric quantization is used.

### 6.3.1 Convolution

The abbreviation of Convolution:  $Y = X_{(n,ic,ih,iw)} \times W_{(oc,ic,kh,kw)} + B_{(1,oc,1,1)}$ .

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned} \text{float : } Y &= X \times W + B \\ \text{step0} &\Rightarrow S_y(q_y - Z_y) = S_x(q_x - Z_x) \times S_w q_w + B \\ \text{step1} &\Rightarrow q_y - Z_y = S_1(q_x - Z_x) \times q_w + B_1 \\ \text{step2} &\Rightarrow q_y - Z_y = S_1 q_x \times q_w + B_2 \\ \text{step3} &\Rightarrow q_y = S_3(q_x \times q_w + B_3) + Z_y \\ \text{step4} &\Rightarrow q_y = (q_x \times q_w + b_{i32}) * M_{i32} \gg rshift_{i8} + Z_y \end{aligned}$$

In particular, for asymmetric quantization, Pad is filled with Zx.

In the symmetric case, Pad is filled with 0 (both Zx and Zy are 0).

In PerAxis (or PerChannal) quantization, each OC of Filter will be quantized, and the derivation formula will remain unchanged, but there will be OC Multiplier and rshift.

### 6.3.2 InnerProduct

Expression and derivation are the same as ([Convolution](#)).

### 6.3.3 Add

The expression for addition is:  $Y = A + B$

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned}
& \text{float : } Y = A + B \\
& \text{step0} \Rightarrow S_y(q_y - Z_y) = S_a(q_a - Z_a) + S_b(q_b - Z_b) \\
& \text{step1(Symmetric)} \Rightarrow q_y = (q_a * M_a + q_b * M_b)_{i16} >> rshift_{i8} \\
& \text{step1(Asymmetric)} \Rightarrow q_y = requant(dequant(q_a) + dequant(q_b))
\end{aligned}$$

The way to implement Add with Tensor Computing Processor is related to specific processor instructions.

The symmetric method here is to use INT16 as the intermediate buffer.

The asymmetric method is to first de-quantize into the float, do the addition and then re-quantize into INT8.

### 6.3.4 AvgPool

The expression of average pooling can be abbreviated as:  $Y_i = \frac{\sum_{j=0}^k (X_j)}{k}$ ,  $k = kh \times kw$ .

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned}
 & \text{float : } Y_i = \frac{\sum_{j=0}^k (X_j)}{k} \\
 & \text{step0 : } \Rightarrow S_y(y_i - Z_y) = \frac{S_x \sum_{j=0}^k (x_j - Z_x)}{k} \\
 & \text{step1 : } \Rightarrow y_i = \frac{S_x}{S_y k} \sum_{j=0}^k (x_j - Z_x) + Z_y \\
 & \text{step2 : } \Rightarrow y_i = \frac{S_x}{S_y k} \sum_{j=0}^k (x_j) - (Z_y - \frac{S_x}{S_y} Z_x) \\
 & \text{step3 : } \Rightarrow y_i = (Scale_{f32} \sum_{j=0}^k (x_j) - Offset_{f32})_{i8} \\
 & Scale_{f32} = \frac{S_x}{S_y k}, Offset_{f32} = Z_y - \frac{S_x}{S_y} Z_x
 \end{aligned}$$

### 6.3.5 LeakyReLU

The expression of LeakyReLU can be abbreviated as:  $Y = \begin{cases} X, & \text{if } X \geq 0 \\ \alpha X, & \text{if } X < 0 \end{cases}$

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned}
 & \text{float : } Y = \begin{cases} X, & \text{if } X \geq 0 \\ \alpha X, & \text{if } X < 0 \end{cases} \\
 & \text{step0 : } \Rightarrow S_y(q_y - Z_y) = \begin{cases} S_x(q_x - Z_x), & \text{if } q_x \geq 0 \\ \alpha S_x(q_x - Z_x), & \text{if } q_x < 0 \end{cases} \\
 & \text{step1 : } \Rightarrow q_y = \begin{cases} \frac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{if } q_x \geq 0 \\ \alpha \frac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{if } q_x < 0 \end{cases}
 \end{aligned}$$

In INT8 symmetric quantization:  $S_y = \frac{\text{threshold}_y}{128}$ ,  $S_x = \frac{\text{threshold}_x}{128}$ . In INT8 asymmetric quantization:  $S_y = \frac{\max_y - \min_y}{255}$ ,  $S_x = \frac{\max_x - \min_x}{255}$ . After BackwardCalibration,  $\max_y = \max_x$ ,  $\min_y = \min_x$ ,  $\text{threshold}_y = \text{threshold}_x$ , so  $S_x/S_y = 1$ .

$$\begin{aligned}
 & \text{step2 : } \Rightarrow q_y = \begin{cases} (q_x - Z_x) + Z_y, & \text{if } q_x \geq 0 \\ \alpha(q_x - Z_x) + Z_y, & \text{if } q_x < 0 \end{cases} \\
 & \text{step3 : } \Rightarrow q_y = \begin{cases} q_x - Z_x + Z_y, & \text{if } q_x \geq 0 \\ M_{i8} >> rshift_{i8}(q_x - Z_x) + Z_y, & \text{if } q_x < 0 \end{cases}
 \end{aligned}$$

In the symmetric case, both  $Z_x$  and  $Z_y$  are 0.

### 6.3.6 Pad

The expression of Pad can be abbreviated as:  $Y = \begin{cases} X, & \text{origin location} \\ \text{value}, & \text{padded location} \end{cases}$

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned} \text{float : } Y &= \begin{cases} X, & \text{origin location} \\ \text{value}, & \text{padded location} \end{cases} \\ \text{step0 : } &\Rightarrow S_y(q_y - Z_y) = \begin{cases} S_x(q_x - Z_x), & \text{origin location} \\ \text{value}, & \text{padded location} \end{cases} \\ \text{step1 : } &\Rightarrow q_y = \begin{cases} \frac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{origin location} \\ \frac{\text{value}}{S_y} + Z_y, & \text{padded location} \end{cases} \end{aligned}$$

After BackwardCalibration,  $\max_y = \max_x$ ,  $\min_y = \min_x$ ,  $\text{threshold}_y = \text{threshold}_x$ , so  $Sx/Sy = 1$ .

$$\text{step2 : } \Rightarrow q_y = \begin{cases} (q_x - Z_x) + Z_y, & \text{origin location} \\ \frac{\text{value}}{S_y} + Z_y, & \text{padded location} \end{cases}$$

In the symmetric case, both  $Zx$  and  $Zy$  are 0, so the padded value is  $\text{round}(\text{value}/Sy)$ . When asymmetric quantization, the padded value is  $\text{round}(\text{value}/Sy + Zy)$ .

### 6.3.7 PReLU

The expression of PReLU can be abbreviated as:  $Y_i = \begin{cases} X_i, & \text{if } X_i \geq 0 \\ \alpha_i X_i, & \text{if } X_i < 0 \end{cases}$

Substitute it into the int8 quantization formula, the derivation is as follows:

$$\begin{aligned} \text{float : } Y_i &= \begin{cases} X_i, & \text{if } X_i \geq 0 \\ \alpha_i X_i, & \text{if } X_i < 0 \end{cases} \\ \text{step0 : } &\Rightarrow S_y(y_i - Z_y) = \begin{cases} S_x(x_i - Z_x), & \text{if } x_i \geq 0 \\ S_\alpha q_{\alpha_i} S_x(x_i - Z_x), & \text{if } x_i < 0 \end{cases} \\ \text{step1 : } &\Rightarrow y_i = \begin{cases} \frac{S_x}{S_y}(x_i - Z_x) + Z_y, & \text{if } x_i \geq 0 \\ S_\alpha q_{\alpha_i} \frac{S_x}{S_y}(x_i - Z_x) + Z_y, & \text{if } x_i < 0 \end{cases} \end{aligned}$$

After BackwardCalibration,  $\max_y = \max_x$ ,  $\min_y = \min_x$ ,  $\text{threshold}_y = \text{threshold}_x$ , so  $Sx/Sy = 1$ .

$$\begin{aligned} \text{step2 : } &\Rightarrow y_i = \begin{cases} (x_i - Z_x) + Z_y, & \text{if } x_i \geq 0 \\ S_\alpha q_{\alpha_i} (x_i - Z_x) + Z_y, & \text{if } x_i < 0 \end{cases} \\ \text{step3 : } &\Rightarrow y_i = \begin{cases} (x_i - Z_x) + Z_y, & \text{if } x_i \geq 0 \\ q_{\alpha_i} * M_{i8}(x_i - Z_x) \gg rshift_{i8} + Z_y, & \text{if } x_i < 0 \end{cases} \end{aligned}$$

There are oc Multipliers and 1 rshift. When symmetric quantization,  $Zx$  and  $Zy$  are both 0.

# CHAPTER 7

---

## Calibration

---

### 7.1 General introduction

Calibration is the use of real scene data to tune the proper quantization parameters. Why do we need calibration? When we perform asymmetric quantization of the activation, we need to know the overall dynamic range, i.e., the minmax value, in advance. When applying symmetric quantization to activations, we need to use a suitable quantization threshold algorithm to calculate the quantization threshold based on the overall data distribution of the activation. However, the general trained model does not have the activation statistics. Therefore, both of them need to inference on a miniature sub-training set to collect the output activation of each layer.

The calibration process in tpu-mlir includes automatic threshold search method (`search_threshold`), SmoothQuant(`sq`), cross-layer weight equalization (`we`), bias correction (`bc`), and an automatic mixed precision feature (`search_qtable`), among other methods. The overall process is shown in([Overall process of quantization](#)). Among these, `sq`, `we`, `bc`, `search_qtable`, and `search_threshold` are optional and can be combined according to the actual situation of the model to be quantized. Subsequent sections will also provide specific instructions for the use of each method. The above processes are integrated and executed collectively, and the optimized thresholds and min/max values of each operation are output to a quantization calibration parameter file called “`cali_table`.” Subsequently, in the “`model_deploy.py`” script, these parameters can be used for further int8 quantization. If you have utilized the automatic mixed-precision feature, along with generating the “`cali_table`,” a mixed-precision table “`qtable`” will also be produced. In the following “`model_deploy.py`” script, both of these files are required for subsequent int8 mixed-precision quantization.

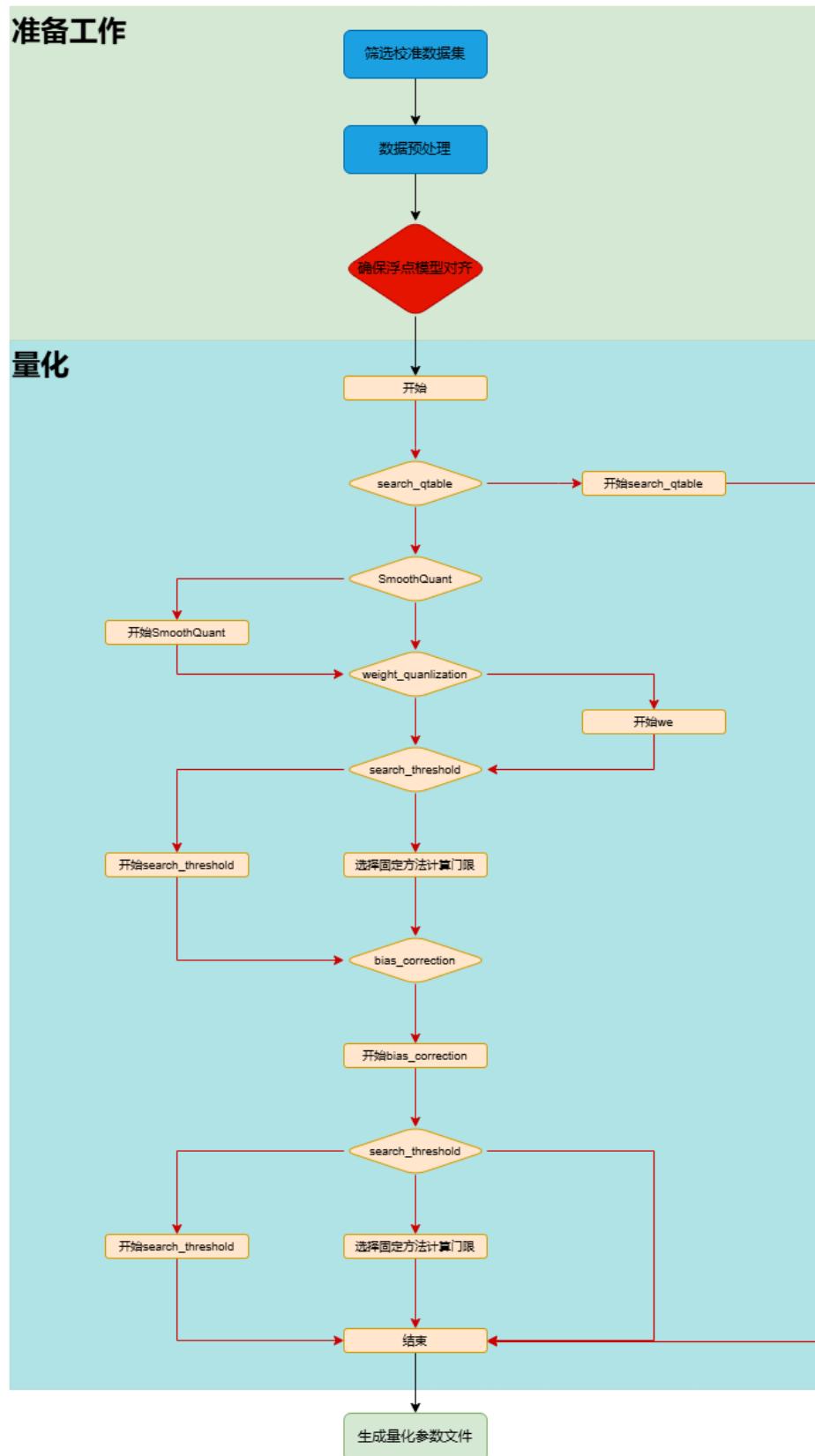


Fig. 7.1: Overall process of quantization

## 7.2 Introduction to the Default Process

The current calibration process encompasses multiple methods and also provides a default calibration workflow, as illustrated in the figure referred to as (Default Process).



Fig. 7.2: Default Process

As shown in the following figure (cali\_table), the final output of the calibration is the cali\_table.

If you use the search\_qtable feature, it will also generate the mixed precision table qtable as shown in the following figure.

```
# generated time: 2022-08-11 10:00:59.743675
# histogram number: 2048
# sample number: 100
# tune number: 5
###
# op_name    threshold      min      max
images 1.0000080 0.0000000 1.0000080
122_Conv 56.4281803 -102.5830231 97.6811752
124_Mul 38.1586478 -0.2784646 97.6811752
125_Conv 56.1447888 -143.7053833 122.0844193
127_Mul 116.7435987 -0.2784646 122.0844193
128_Conv 16.4931355 -87.9204330 7.2770605
130_Mul 7.2720342 -0.2784646 7.2720342
131_Conv 51.5455152 -56.4878578 26.2175255
133_Mul 22.2855371 -0.2784646 26.2175255
134_Conv 19.6111164 -28.0139256 21.4674854
136_Mul 20.8639418 -0.2784646 21.4674854
137_Add 20.5015809 -0.5569289 21.8679256
138_Conv 14.7106976 -87.1445465 32.7312393
```

Fig. 7.3: cali\_table

```
622_Conv F32
646_Transpose F32
474_Conv F32
498_Transpose F32
326_Conv F32
350_Transpose F32
```

## 7.3 Calibration data screening and preprocessing

### 7.3.1 Screening Principles

Selecting about 100 to 200 images covering each typical scene style in the training set for calibration. Using a approach similar to training data cleaning to exclude some anomalous samples.

### 7.3.2 Input format and preprocessing

Table 7.1: Input format

Format	Description
Original Image	For CNN-like vision networks, image input is supported. Image preprocessing arguments must be the same as in training step when generating the mlir file by <code>model_transform.py</code> .
npz or npy file	For cases where non-image inputs or image preprocessing types are not supported at the moment, it is recommended to write an additional script to save the preprocessed input data into npz/npy files (npz file saves multiple tensors in the dictionary, and npy file only contains one tensor). <code>run_calibration.py</code> supports direct input of npz/npy files.

There is no need to specify the preprocessing parameters for the above two formats when calling `run_calibration.py` to call the mlir file for inference.

Table 7.2: Methods of specifying parameters

Method	Description
<code>-dataset</code>	For single-input networks, place images or preprocessed input npy/npz files (no order required). For multi-input networks, place the pre-processed npz files of each sample.
<code>-data_list</code>	Place the path of the image, npz or npy file of each sample (one sample per line) in a text file. If the network has more than one input file, separate them by commas (note that the npz file should have only 1 input path).

```

1 /data/cali_100pics/n01440764_9572.JPG
2 /data/cali_100pics/n01531178_12753.JPG
3 /data/cali_100pics/n01537544_17475.JPG
4 /data/cali_100pics/n01608432_4202.JPG
5 /data/cali_100pics/n01608432_4203.JPG

```

Fig. 7.4: Example of data\_list required format

## 7.4 Quantization Threshold Algorithm Implementation

tpu-mlir currently implements seven quantization threshold calculation methods: Kullback-Leibler divergence with auto-tuning (kld+auto-tune), Octav (octav), MinMax, Percentile (percentile9999), ACIQ with Gaussian assumption and auto-tuning (aciq\_gauss+auto-tune), ACIQ with Laplace assumption and auto-tuning (aciq\_laplace+auto-tune), and a histogram-based algorithm derived from Torch. Below, we will introduce the KLD, Octav, ACIQ, and auto-tune algorithms.

### 7.4.1 KLD Algorithm

The KLD algorithm implemented by tpu-mlir refers to the implementation of tensorRT. In essence, it cuts off some high-order outliers (the intercepted position is fixed at 128 bin, 256bin ... until 2048 bin) from the distribution of abs(fp32\_tensor) (represented by the histogram of 2048 fp32 bins) to get the fp32 reference probability distribution P. This fp32 distribution is expressed in terms of 128 ranks of int8 type. By merging multiple adjacent bins (e.g., 256 bins are 2 adjacent fp32 bins) into 1 rank of int8 values, calculating the distribution probability, and then expanding bins to ensure the same length as P, the probability distribution Q of the quantized int8 values can be got. The KL divergences of P and Q are calculated for the interception positions of 128bin, 256bin, ..., and 2048 bin, respectively in each loop until the interception with the smallest divergence is found. Interception here means the probability distribution of fp32 can be best simulated with the 128 quantization levels of int8. Therefore, it is most appropriate to set the quantization threshold here. The pseudo-code for the implementation of the KLD algorithm is shown below:

```

1 the pseudocode of computing int8 quantize threshold by kld:
2     Prepare fp32 histogram H with 2048 bins
3     compute the absmax of fp32 value
4
5     for i in range(128,2048,128):
6         Outliers_num = sum(bin[i], bin[i+1], ..., bin[2047])
7         Fp32_distribution = [bin[0], bin[1], ..., bin[i-1] + Outliers_num]
8         Fp32_distribution /= sum(Fp32_distribution)
9
10    int8_distribution = quantize [bin[0], bin[1], ..., bin[i]] into 128 quant level
11    expand int8_distribution to i bins
12    int8_distribution /= sum(int8_distribution)
13    kld[i] = KLD(Fp32_distribution, int8_distribution)

```

(continues on next page)

(continued from previous page)

```
14 end for  
15  
16 find i which kld[i] is minimal  
17 int8 quantize threshold = (i + 0.5)*fp32 absmax/2048
```

### 7.4.2 Auto-tune Algorithm

From the actual performance of the KLD algorithm, its candidate threshold is relatively coarse and does not take into account the characteristics of different scenarios, such as object detection and key point detection, in which tensor outliers may be more important to the performance. In these cases, a larger quantization threshold is required to avoid saturation which will affect the expression of distribution features. In addition, the KLD algorithm calculates the quantization threshold based on the similarity between the quantized int8 and the fp32 probability distribution, while there are other methods to evaluate the distribution similarity such as Euclidean distance, cos similarity, etc. These metrics evaluate the tensor numerical distribution similarity directly without the need for a coarse interception threshold, which most of the time has better performance. Therefore, with the basis of efficient KLD quantization threshold, tpu-mlir proposes the auto-tune algorithm to fine-tune these activations quantization thresholds based on Euclidean distance metric, which ensures a better accuracy performance of its int8 quantization.

Implementation: firstly, uniformly pseudo-quantize layers with weights in the network, i.e., quantize their weights from fp32 to int8, and then de-quantize to fp32 for introducing quantization error. After that, tune the input activation quantization threshold of op one by one (i.e., uniformly select 20 candidates among the initial KLD quantization threshold and maximum absolute values of activations. Use these candidates to quantize fp32 reference activation values for introducing quantization error. Input op for fp32 calculation, calculating the Euclidean distance between the output and the fp32 reference activations. The candidate with a minimum Euclidean distance will be selected as the tuning threshold). For the case where the output of one op is connected to multiple subsequent ones, the quantization thresholds are calculated for the multiple branches according to the above method, and then the larger one is taken. For instance, the output of layer1 will be adjusted for layer2 and layer3 respectively as shown in the figure ([Implementation of auto-tune](#)).

### 7.4.3 Octav Algorithm

The OCTAV algorithm implemented by tpu-mlir is based on the paper “Optimal Clipping and Magnitude-aware Differentiation for Improved Quantization-aware Training.” It is commonly believed that quantization error stems from rounding error and truncation error. Computing the optimal truncation (threshold) for each tensor can minimize the quantization error. OCTAV uses mean squared error (MSE) to measure quantization error and employs a recursive approach based on the fast Newton-Raphson method to dynamically determine the optimal threshold that minimizes MSE. Below is the iterative formula for computing the optimal threshold using this method, as illustrated in figure ([octav迭代公式](#)).

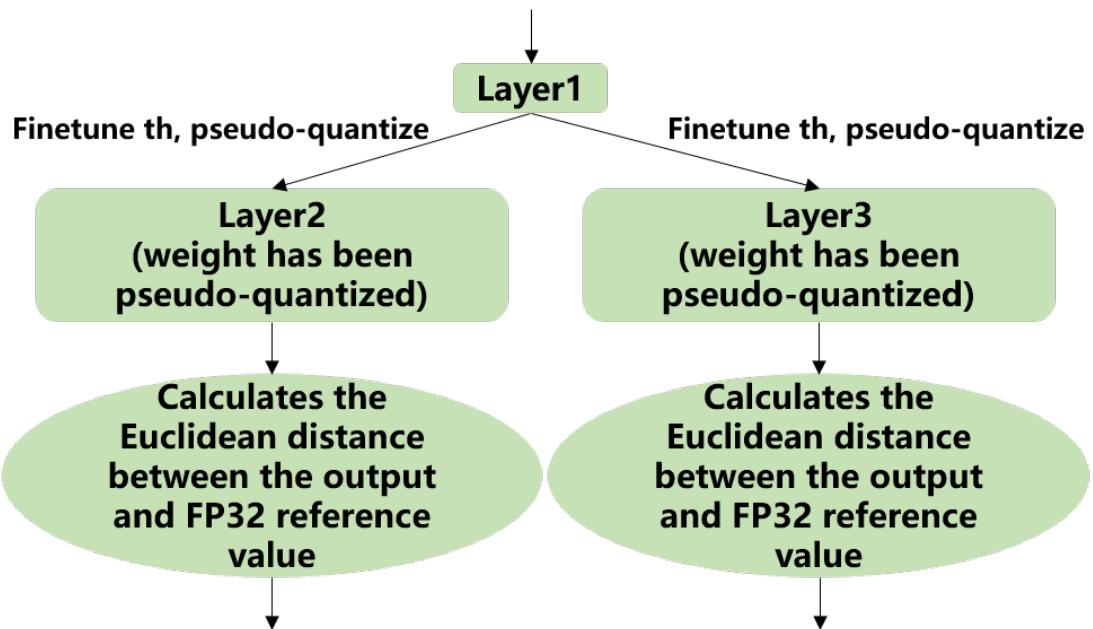


Fig. 7.5: Implementation of auto-tune

$$s_{n+1} = \frac{\mathbb{E} [|X| \cdot \mathbb{1}_{\{|X| > s_n\}}]}{\frac{4-B}{3} \mathbb{E} [\mathbb{1}_{\{|X| \leq s_n\}}] + \mathbb{E} [\mathbb{1}_{\{|X| > s_n\}}]}$$

Fig. 7.6: octav迭代公式

It was initially designed for use in Quantization-Aware Training (QAT), but it is also effective in Post-Training Quantization (PTQ). Below is the pseudocode for its implementation:

```

1 the pseudocode of computing int8 quantize threshold by octav:
2     Prepare T: Tensor to be quantized,
3         B: Number of quantization bits,
4         epsilon: Convergence threshold (e.g., 1e-5),
5         s_0: Initial guess for the clipping scalar (e.g., max absolute value in tensor T)
6     compute s_star: Optimal clipping scalar
7
8     for n in range(20):
9         Compute the indicator functions for the current clipping scalar:
10            I_clip = 1{|T| > s_n} (applied element-wise to tensor T)
11            I_disc = 1{0 < |T| <= s_n}
12
13         Update the clipping scalar s_n to the next one s_(n+1) using:
14             s_(n+1) = ( $\Sigma|x| * I_{clip}$ ) / (( $4^{-B}$  / 3) *  $\Sigma I_{disc}$  +  $\Sigma I_{clip}$ )
15             where  $\Sigma$  denotes the summation over the corresponding elements
16
17         If  $|s_{(n+1)} - s_n| < \text{epsilon}$ , the algorithm is considered to have converged
18     end for
19     s_star = s_n

```

#### 7.4.4 Aciq Algorithm

The ACIQ algorithm implemented in TPU-MLIR is based on the paper “ACIQ: Analytical Clipping for Integer Quantization of Neural Networks.” This method assumes that the activation values follow a fixed distribution, then calculates the statistical measures of the corresponding distribution of the activation values, and derives the optimal threshold based on the theoretically calculated optimal clipping quantile.

Implementation approach: TPU-MLIR provides two variants of the algorithm, aciq\_gauss and aciq\_laplace, which assume Gaussian and Laplace distributions for the activation values. Then, based on the optimal clipping quantile corresponding to 8-bit theoretically, the optimal threshold is calculated.

### 7.5 optimization algorithms Implementation

During the calibration process, to further enhance the precision of the quantized model, TPU-MLIR offers a variety of optimization algorithms, including SmoothQuant (SQ), Cross-Layer Weight Equalization (WE), Bias Correction (BC), search\_qtable, and search\_threshold. Below is an introduction to the aforementioned optimization algorithms.

### 7.5.1 sq Algorithm

The SmoothQuant algorithm implemented in TPU-MLIR is based on the paper “SmoothQuant: Accurate and Efficient Post-Training Quantisation for Large Language Models”. This method improves the accuracy of the quantised model by smoothly assigning the tensor scales of the model and adjusting the range of inputs and weights of the model to a more suitable range for quantization, thus improving the accuracy of the quantised model. It solves the problem of accuracy degradation of large-scale pre-trained models (e.g., language models and visual models) during the quantisation process.

SmoothQuant redistributes the range of activations and weights by adjusting the tensor ratio of the model, which makes the quantisation process more stable. Specifically, SmoothQuant introduces a smoothing factor before quantisation, which partially transfers the range of the activation values to the weights, adjusts the model weights with a mathematically equivalent transformation, thus reducing the quantisation error of the activation values. The technical principle is illustrated in the figure ([SmoothQuant](#)).

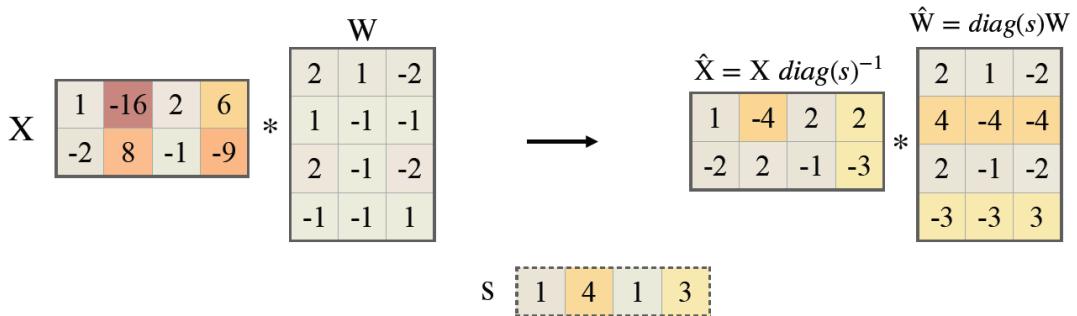


Fig. 7.7: SmoothQuant

### 7.5.2 we Algorithm

The cross-layer weight equalization algorithm implemented in TPU-MLIR is based on the paper “Data-Free Quantization Through Weight Equalization and Bias Correction.” This method primarily targets model weights and equalizes weights that fit the patterns of conv-conv and conv-relu-conv, aiming to make the distribution of two adjacent weights as uniform as possible.

Previous studies have found that in networks with a high number of depthwise separable convolutions, such as MobileNet, there is a significant variation in the data distribution across channels. This variation can lead to substantial quantization errors when per-layer quantization is used. The WE algorithm effectively addresses this issue by leveraging the linear characteristics of the ReLU function, allowing for the equalization of adjacent convolutional weights. This equalization reduces the distribution disparity between convolutional channels, enabling per-layer quantization to achieve results comparable to per-channel quantization. The technical principle is illustrated in the figure ([weight\\_equalization](#)).

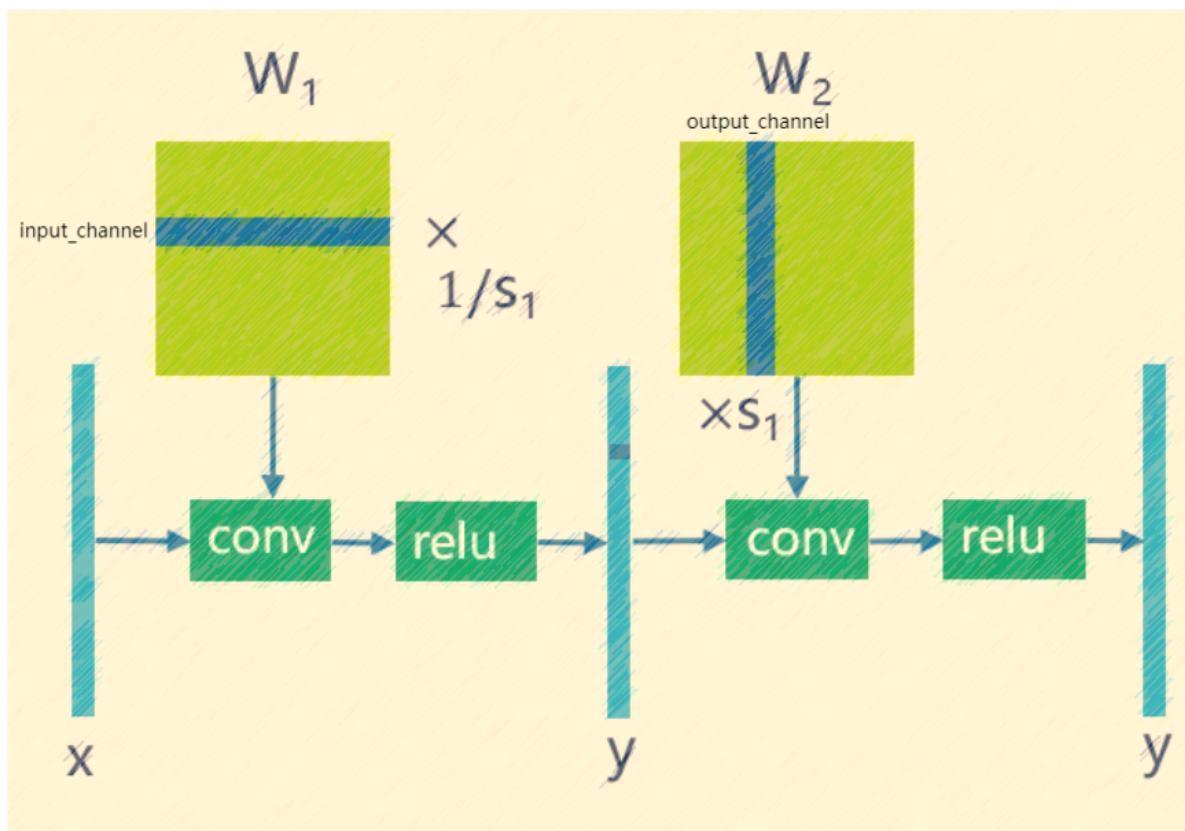


Fig. 7.8: weight\_equalization

### 7.5.3 bc Algorithm

The bias correction algorithm implemented in TPU-MLIR is referenced from the paper “Data-Free Quantization Through Weight Equalization and Bias Correction.” It’s commonly assumed that the output error of a quantized model is unbiased, meaning its expected value is zero. However, in many practical scenarios, the output error of a quantized model is biased, indicating a deviation in the expected value between the outputs of the quantized model and the floating-point model. This deviation can impact the accuracy of the quantized model.

The bias correction algorithm calculates the statistical deviation between the quantized model and the floating-point model on calibration data. It then compensates for this deviation by adjusting the bias term of Conv/Gemm operators in the model, aiming to minimize the expected value deviation between their outputs as much as possible. The effect is illustrated in the figure([bias\\_correction](#)).

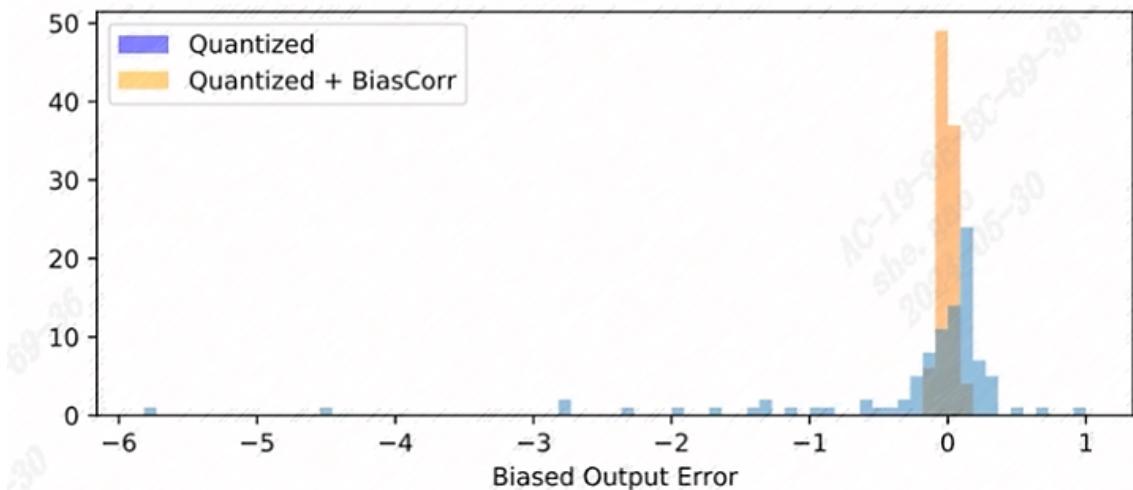


Fig. 7.9: bias\_correction

### 7.5.4 search\_threshold Algorithm

TPU-MLIR offers seven independent threshold calculation methods, and when we have a model that needs to be quantized, choosing the best threshold calculation method becomes an issue. `search_threshold` provides a solution for this problem.

Implementation: `search_threshold` initially computes the threshold values using four methods: `kld+tune`, `octav`, `max`, and `percentile9999`. It then calculates the similarity between the outputs of the quantized model generated by different threshold values and the floating-point model. By comparing the similarity of the four threshold methods, the threshold value corresponding to the highest similarity is selected as the quantization parameter for the current model. During the use of `search_threshold`, the following points need to be noted: 1.

search\_threshold currently offers two similarity calculation methods, cos and snr, with cos being the default method. 2. If the cos similarity between the quantized model and the floating-point model is below 0.9, the accuracy of the quantized model may be significantly reduced. Search\_threshold results may not be accurate. After performing actual accuracy validation, it is recommended to try mixed precision with search\_qtable.

### 7.5.5 search\_qtable Algorithm

search\_qtable is an automatic mixed-precision functionality integrated into the calibration process. When the accuracy of a fully int8 quantized model does not meet the requirements, you can try enabling the search\_qtable algorithm. This algorithm is faster compared to run\_sensitive\_lyer. It also offers the ability to mix custom threshold algorithms and automatically generate a qtable.

Implementation: The output of search\_qtable will generate mixed thresholds, meaning it performs optimal selection for the threshold of each layer of the model. That is, it chooses the best result from multiple threshold calculation methods specified by the user for each layer. This choice is based on the comparison of the similarity between the quantized model's current layer output and the original model's current layer output. In addition to generating mixed thresholds, search\_qtable will also output the layers of the model that are mixed precision. When the user specifies the desired similarity between the mixed precision model and the original model's output, search\_qtable will automatically output the minimum number of mixed precision layers required to achieve that similarity level.

## 7.6 Example: yolov5s calibration

In the docker environment of tpu-mlir, execute `source envsetup.sh` in the tpu-mlir directory to initialize the environment, then enter any new directory and execute the following command to complete the calibration process for yolov5s.

```
1 $ model_transform.py \
2   --model_name yolov5s \
3   --model_def ${REGRESSION_PATH}/model/yolov5s.onnx \
4   --input_shapes [[1,3,640,640]] \
5   --keep_aspect_ratio \ #keep_aspect_ratio、mean、scale、pixel_format are preprocessing[E]
6   ↪arguments
7   --mean 0.0,0.0,0.0 \
8   --scale 0.0039216,0.0039216,0.0039216 \
9   --pixel_format rgb \
10  --output_names 350,498,646 \
11  --test_input ${REGRESSION_PATH}/image/dog.jpg \
12  --test_result yolov5s_top_outputs.npz \
13  --mlir yolov5s.mlir
```

Table 7.3: The arguments of model\_transform.py

Argument	Description
model_name	Model name
-model_def	Model definition file (.onnx,.pt.,tflite or .prototxt)
-model_data	Specify the model weight file, required when it is caffe model (corresponding to the ‘.caffemodel’ file)
-input_shapes	The shape of the input, such as [[1,3,640,640]] (a two-dimensional array), which can support multiple inputs
-resize_dims	The size of the original image to be adjusted to. If not specified, it will be resized to the input size of the model
-	Whether to maintain the aspect ratio when resize. False by default.
keep_aspect_ratio	It will pad 0 to the insufficient part when setting
-mean	The mean of each channel of the image. The default is 0.0,0.0,0.0
-scale	The scale of each channel of the image. The default is 1.0,1.0,1.0
-pixel_format	Image type, can be rgb, bgr, gray or rgbd
-output_names	The names of the output. Use the output of the model if not specified, otherwise use the specified names as the output
-test_input	The input file for validation, which can be an image, npy or npz. No validation will be carried out if it is not specified
-test_result	Output file to save validation result
-excepts	Names of network layers that need to be excluded from validation. Separated by comma
-debug	if open debug, immediate model file will keep; or will remove after conversion done
-mlir	The output mlir file name (including path)

Default process

```

1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --tune_num 10 \
5   -o yolov5s_cali_table

```

Using different quantization threshold calculation methods.

octav:

```

1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --cali_method use_mse \
5   -o yolov5s_cali_table

```

minmax:

```
1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --cali_method use_max \
5   -o yolov5s_cali_table
```

percentile9999:

```
1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --cali_method use_percentile9999 \
5   -o yolov5s_cali_table
```

aciq\_gauss:

```
1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --cali_method use_aciq_gauss \
5   -o yolov5s_cali_table
```

aciq\_laplace:

```
1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --cali_method use_aciq_laplace \
5   -o yolov5s_cali_table
```

Using optimization methods:

sq:

```
1 $ run_calibration.py yolov5s.mlir \
2   --sq \
3   --dataset $REGRESSION_PATH/dataset/COCO2017 \
4   --input_num 100 \
5   --cali_method use_mse \
6   -o yolov5s_cali_table
```

we:

```
1 $ run_calibration.py yolov5s.mlir \
2   --we \
3   --dataset $REGRESSION_PATH/dataset/COCO2017 \
4   --input_num 100 \
5   --cali_method use_mse \
6   -o yolov5s_cali_table
```

we+bc:

```

1 $ run_calibration.py yolov5s.mlir \
2   --we \
3   --bc \
4   --dataset $REGRESSION_PATH/dataset/COCO2017 \
5   --input_num 100 \
6   --processor bm1684x \
7   --bc_inference_num 200 \
8   --cali_method use_mse \
9   -o yolov5s_cali_table

```

we+bc+search\_threshold:

```

1 $ run_calibration.py yolov5s.mlir \
2   --we \
3   --bc \
4   --dataset $REGRESSION_PATH/dataset/COCO2017 \
5   --input_num 100 \
6   --processor bm1684x \
7   --bc_inference_num 200 \
8   --search search_threshold \
9   -o yolov5s_cali_table

```

search\_qtable:

```

1 $ run_calibration.py yolov5s.mlir \
2   --dataset $REGRESSION_PATH/dataset/COCO2017 \
3   --input_num 100 \
4   --processor bm1684x \
5   --max_float_layers 5 \
6   --expected_cos 0.99 \
7   --transformer False \
8   --quantize_method_list KL,MSE \
9   --search search_qtable \
10  --quantize_table yolov5s_qtable \
11  -o yolov5s_cali_table

```

Table 7.4: The arguments of run\_calibration.py

Argument	Description
mlir_file	mlir file
-sq	open SmoothQuant
-we	open weight_equalization
-bc	open bias_correction
-dataset	dataset for calibration
-data_list	Input list file contain all input
-input_num	num of images for calibration
-inference_num	The number of images required for the inference process of search_qtable and search_threshold

continues on next page

Table 7.4 – continued from previous page

Argument	Description
<code>-bc_inference_num</code>	The number of images required for the inference process of bias_correction
<code>-tune_list</code>	Tune list file contain all input for tune
<code>-tune_num</code>	num of images for tune
<code>-histogram_bin_num</code>	Specify histogram bin numer for kld calculate
<code>-expected_cos</code>	The expected similarity between the mixed-precision model output and the floating-point model output in search_qtable, with a value range of [0,1]
<code>-min_layer_cos</code>	The minimum similarity between the quantized output and the floating-point output of a layer in bias_correction. Compensation is required for the layer when the similarity is below this threshold, with a value range of [0,1]
<code>-max_float_layers</code>	The number of floating-point layers in search_qtable
<code>-processor</code>	processor type
<code>-cali_method</code>	Choose quantization threshold calculation method, Options include use_kl, use_mse, use_percentile9999, use_max, with the default being use_kl
<code>-fp_type</code>	The data type of floating-point layers in search_qtable
<code>-post_process</code>	The path for post-processing
<code>-global_compare_layers</code>	Specifies the global comparison layers, for example, layer1,layer2 or layer1:0.3,layer2:0.7
<code>-search</code>	Specifies the type of search, including search_qtable, search_threshold, false. The default is false, which means search is not enabled
<code>-transformer</code>	Whether it is a transformer model, if it is, search_qtable can allocate specific acceleration strategies
<code>-quantize_method_list</code>	The threshold methods used for searching in search_qtable, The default is only MSE, but it supports free selection among KL, MSE, MAX, and Percentile9999
<code>-benchmark_method</code>	Specifies the method for calculating similarity in search_threshold
<code>-kurtosis_analysis</code>	Specify the generation of the kurtosis of the activation values for each layer
<code>-part_quantize</code>	Specify partial quantization of the model. The calibration table (cali_table) will be automatically generated alongside the quantization table (qtable). Available modes include N_mode, H_mode, or custom_mode, with H_mode generally delivering higher accuracy
<code>-custom_operator</code>	Specify the operators to be quantized, which should be used in conjunction with the aforementioned custom_mode
<code>-part_asymmetric</code>	When symmetric quantization is enabled, if specific subnets in the model match a defined pattern, the corresponding operators will automatically switch to asymmetric quantization

continues on next page

Table 7.4 – continued from previous page

Argument	Description
-mix_mode	Specify the mixed-precision types for the search_qtable. Currently supported options are 8_16 and 4_8
-cluster	Specify that a clustering algorithm is used to detect sensitive layers during the search_qtable process
-quantize_table	The mixed-precision quantization table output by search_qtable
-o	output threshold table
-debug_cmd	debug command to specify calibration mode; “percentile9999” initialize the threshold via percentile function, “use_max” specifies the maximum of absolute value to be the threshold, “use_torch_observer_for_cali” adopts Torch observer for calibration. “use_mse” adopts Octav for calibration.
-debug_log	Log output level

The result is shown in the following figure ([yolov5s\\_cali calibration result](#)).

## 7.7 visual tool introduction

visual.py is an visualized net/tensor compare tool with UI in web browser. When quantized net encounters great accuracy decrease, this tool can be used to investigate the accuracy loss layer by layer. This tool is started in docker as an server listening to TCP port 10000 (default), and by input localhost:10000 in url of browser in host computer, the tool UI will be displayed in it, the port must be mapped to host in advance when starting the docker, and the tool must be start in the same directory where the mlir files located, start command is as following:

```
root@80ab6476536b:/workspace/code/tpu-mlir/doc/developer_manual/tmp1# run_calibration.py yolov5s.mlir \
> --dataset $REGRESSION_PATH/dataset/COCO2017 \
> --input_num 10 \
> --tune_num 2 \
> -o yolov5s_cali_table
SOPHGO Toolchain v0.3.10-g3630539-20220816
2022/08/17 17:18:16 - INFO :
load_config Preprocess args :
    resize_dims          : [640, 640]
    keep_aspect_ratio    : True
    pad_value            : 0
    pad_type             : center
    input_dims           : [640, 640]
    -----
    mean                 : [0.0, 0.0, 0.0]
    scale                : [0.0039216, 0.0039216, 0.0039216]
    -----
    pixel_format         : rgb
    channel_format       : nchw

mem info before _activations_generator_and_find_minmax:total mem is 32802952, used mem is 7537492
inference and find Min Max *000000281447.jpg: 100%|██████████| 100/100 [00:00<00:00, 100.00it/s]
mem info after _activations_generator_and_find_minmax:total mem is 32802952, used mem is 7794272
calculate histogram..
mem info before calc_thresholds:total mem is 32802952, used mem is 7793756
calc_thresholds: 000000281447.jpg: 100%|██████████| 100/100 [00:00<00:00, 100.00it/s]
mem info after calc_thresholds:total mem is 32802952, used mem is 7785728
[2048] threshold: images: 100%|██████████| 100/100 [00:00<00:00, 100.00it/s]
mem info after find_threshold:total mem is 32802952, used mem is 7786352
start fake_quant_weight
tune op: 646_Transpose: 100%|██████████| 100/100 [00:00<00:00, 100.00it/s]
root@80ab6476536b:/workspace/code/tpu-mlir/doc/developer_manual/tmp1# ll
total 573036
drwxr-xr-x 3 root root      4096 Aug 17 17:23 ../
drwxrwxr-x 7 1003 1003      4096 Aug 17 17:15 ../
drwxr-xr-x 2 root root      4096 Aug 17 17:19 tmpdata/
-rw-r--r-- 1 root root     38065 Aug 17 17:17 yolov5s.mlir
-rw-r--r-- 1 root root      6233 Aug 17 17:23 yolov5s_cali_table
-rw-r--r-- 1 root root    4915466 Aug 17 17:17 yolov5s_in_f32.npz
-rw-r--r-- 1 root root   28931068 Aug 17 17:17 yolov5s_opt.onnx
-rw-r--r-- 1 root root    126202 Aug 17 17:17 yolov5s_opt.onnx.prototxt
-rw-r--r-- 1 root root     50069 Aug 17 17:17 yolov5s_origin.mlir
```

Fig. 7.10: yolov5s\_cali calibration result

```
sophgo@3cc464b1891d:/workspace/regression/mlir_deploy.1$ visual.py --f32_mlir transformed.mlir --quant_mlir openpose_bm1684x_int8_asym_tpu.mlir --input
openpose_in_f32.npz --port 9999
Dash is running on http://0.0.0.0:9999/
* Serving Flask app 'visual'
* Debug mode: off
-----
f32 mlir is :transformed.mlir
quant mlir is:openpose_bm1684x_int8_asym_tpu.mlir
input is   :openpose_in_f32.npz
-----
```

Table 7.5: visual tool parameters

Param	Description
-port	the TCP port used to listen to browser as server, default value is 10000
-f32_mlir	the float mlir net to compare to, this file is produced by model_transform, and usually with the name of netname.mlir, it is the base float32 mlir net.
-quant_mlir	the quantized mlir net to compare with float net, this file is generated in model_deploy, usually with netname_int8_sym_tpu.mlir, _final.mlir to generate bmodel can't be used here.
-input	input data to run the float net and quantized net for data compare, can be image or npy/npz file, can be the test_input when graph_transform
-manual_run	if run the nets when browser connected to server, default is true, if set false, only the net structure will be displayed

Open browser in host computer and input localhost:9999, the tool UI will be displayed. The float and quantized net will automatically inference to get output of every layer, if the nets are huge, it would took a long time to wait! UI is as following:



**Areas of the UI is marked with light blue rectangle for reference, dark green comments on the areas, includeing:**

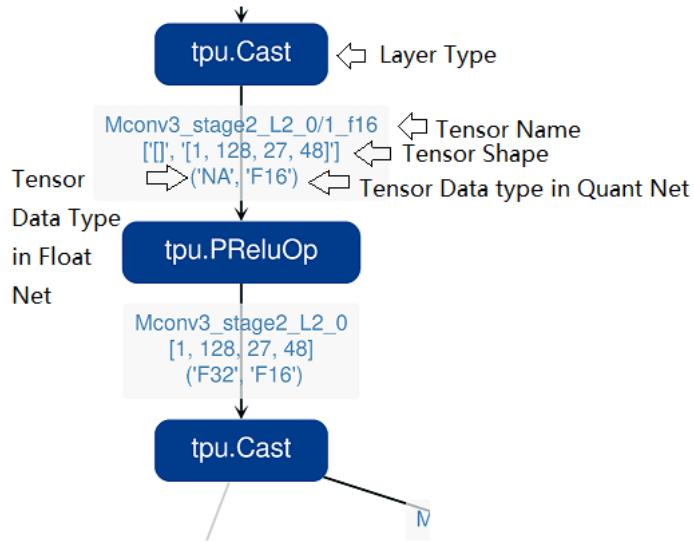
1. working directory and net file indication
2. accuracy summary area
3. layer information area

4. graph display area
5. tensor data compare figure area
6. infomation summary and tensor distribution area (by switching tabs)

With scroll wheel over graph display area, the displayed net graph can be zoomed in and out, and hover or click on the nodes (layer), the attributes of it will be displayed in the layer information card, by clicking on the edges (tensor), the compare of tensor data in float and quantized net is displayed in tensor data compare figure, and by clicking on the dot in accuracy summary or information list cells, the layer/tensor will be located in graph display area.

**Notice:** the net graph is displayed according to quantized net, and there may be difference in it comparing to float net, some layer/tensor may not exist in float net, but the data is copied from quantized net for compare, so the accuracy may seem perfect, but in fact, it should be ignored. Typical layer is Cast layer in quantized net, in following picture, the non-exist tensor data type will be NA. Notice: without `-debug` parameter in deployment of the net, some essential intermediate files needed by visual tool would have been deleted by default, please re-deploy with `-debug` parameter.

information displayed on edge (tensor) is illustrated as following:



# CHAPTER 8

## Lowering

Lowering lowers the Top layer OP to the Tpu layer OP, it supports types of F32/F16/BF16/INT8 symmetric/INT8 asymmetric.

When converting to INT8, it involves the quantization algorithm. For different processors, the quantization algorithm is different. For example, some support per-channel and some do not. Some support 32-bit Multiplier and some only support 8-bit, etc.

Therefore, lowering converts op from the hardware-independent layer (TOP), to the hardware-related layer (TPU).

### 8.1 Basic Process

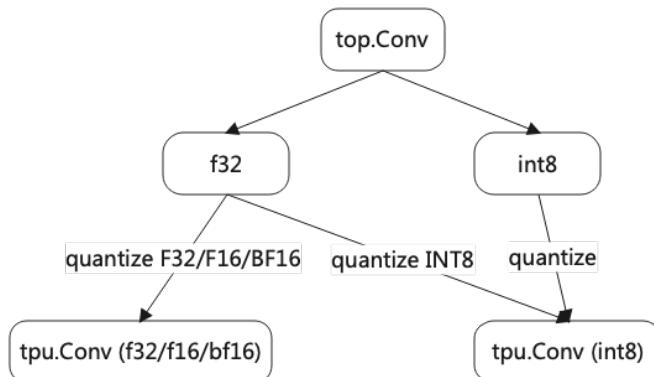


Fig. 8.1: Lowering process

The process of lowering is shown in the figure (Lowering process).

- Top op can be divided into f32 and int8. The former is the case of most networks and the latter is the case of quantized networks such as tflite.
- f32 op can be directly converted to f32/f16/bf16 tpu layer operator. If it is to be converted to int8, the type should be calibrated\_type.
- int8 op can only be directly converted to tpu layer int8 op.

## 8.2 Mixed precision

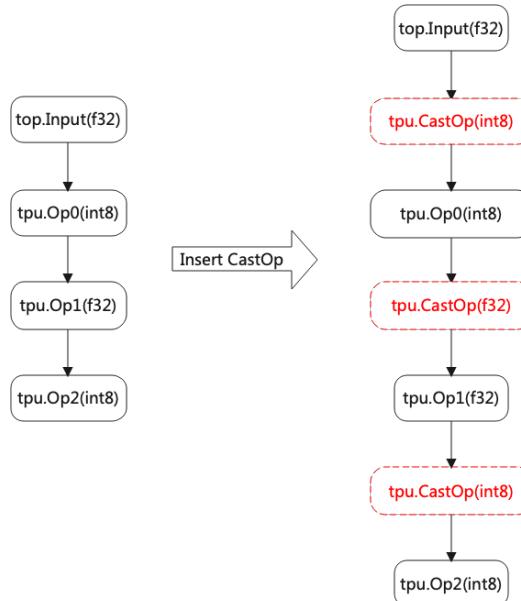


Fig. 8.2: Mixed precision

When the type is not the same between OPs, CastOp is inserted as shown in the figure ([Mixed precision](#)).

It is assumed that the type of output is the same as the input. Otherwise, special treatment is needed. For example, no matter what the type of embedding output is, the input is of type uint.

# CHAPTER 9

---

SubNet

---

# CHAPTER 10

---

## LayerGroup

---

### 10.1 Basic Concepts

The memory in a Tensor Computing Processor can be categorized into global memory (GMEM) and local memory (LMEM).

Usually the global memory is very large (e.g., 4GB) while the local memory is quite limited (e.g., 16MB).

In general, the amount of data and computation of neural network model is very large, so the OP of each layer usually needs to be sliced and put into local memory for operation, and then the result is saved to global memory.

LayerGroup enables as many OPs as possible to be executed in local memory after being sliced, so that it can avoid too many copy operations between local and global memory.

#### **Problem to be solved:**

How to keep Layer data in the limited local memory for computing, instead of repeatedly making copies between local and global memory.

#### **Basic idea:**

Slicing the N and H of activation, make the operation of each layer always in local memory, as shown in the figure ([Network slicing example](#)).

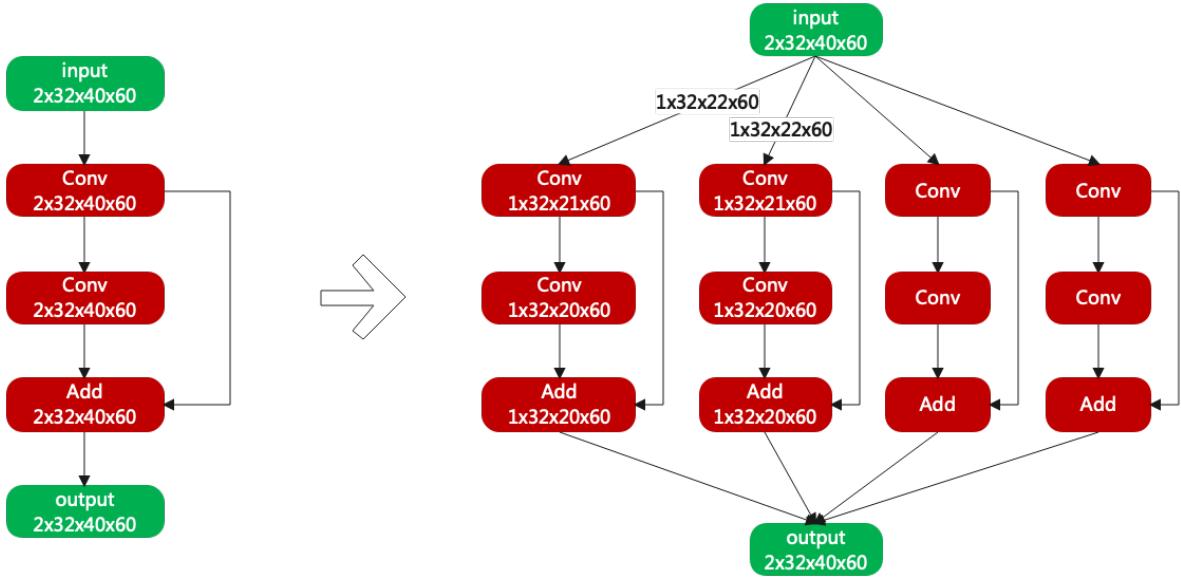


Fig. 10.1: Network slicing example

## 10.2 BackwardH

When slicing along the axis of H, the input and output H of most layers are consistent. But for Conv, Pool, etc., additional calculations are needed.

Take Conv for example, as shown in the figure ([Convolutional BackwardH example](#)).

## 10.3 Dividing the Mem Cycle

How to divide the group? First of all, list the lmem needed for each layer, which can be broadly classified into three categories:

1. Activation Tensor, which is used to save the input and output results, and is released directly after there is no user.
2. Weight, used to save the weights, released when there is no slice. Otherwise, always resides in the lmem.
3. Buffer, used for Layer operation to save intermediate results, released after use.

Then configure the ids in a breadth-first manner, for example, as shown in the figure ([LMEM's ID assignment](#)).

Then configure the period as shown in ([TimeStep assignment](#)).

Details of configuring period are as follows:

- [T2,T7], which means that lmem should be requested at the beginning of T2 and released at the end of T7.

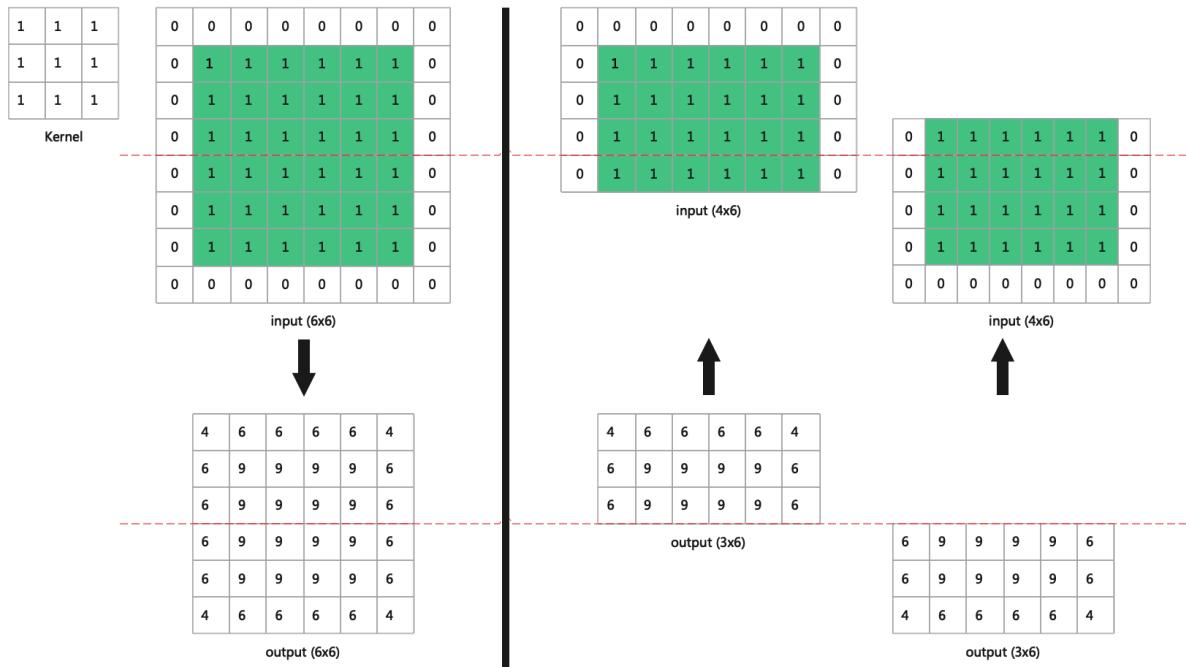


Fig. 10.2: Convolutional BackwardH example

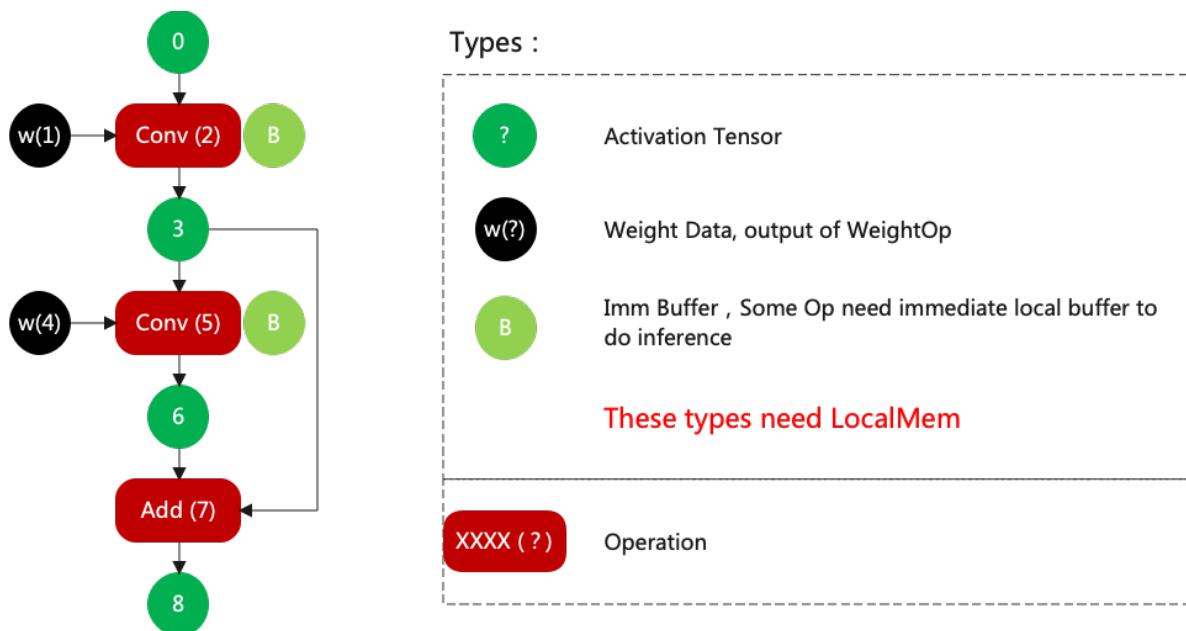


Fig. 10.3: LMEM's ID assignment

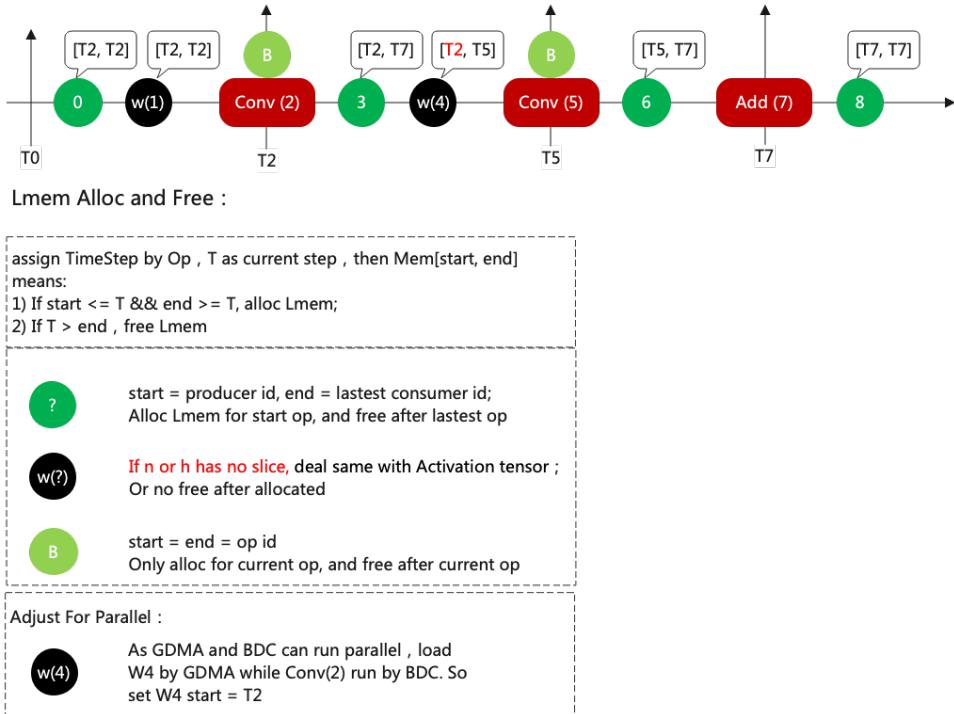


Fig. 10.4: TimeStep assignment

- The original period of w4 should be [T5, T5], but it is corrected to [T2, T5], because w4 can be loaded at the same time when T2 does the convolution operation.
- When N or H is sliced, weight does not need to be reloaded and its end point will be corrected to positive infinity.

## 10.4 LMEM Allocation

When the slice exists in N or H, weight is resident in LMEM so that each slice can use it.

At this point weight will be allocated first, as shown in the figure ([Allocation in the case of slice](#))

When there is no slice, weight and activation are handled the same way, and released when not in use.

The allocation process at this point is shown in the figure ([Allocation in the case of no slice](#)).

Then the LMEM allocation problem can be converted into a problem of how to place these squares (note that these squares can only be moved left and right, not up and down).

In addition, LMEM allocation is better not to cross the bank.

The current strategy is to allocate them in order of op, giving priority to those with long timestep, followed by those with large LMEM.

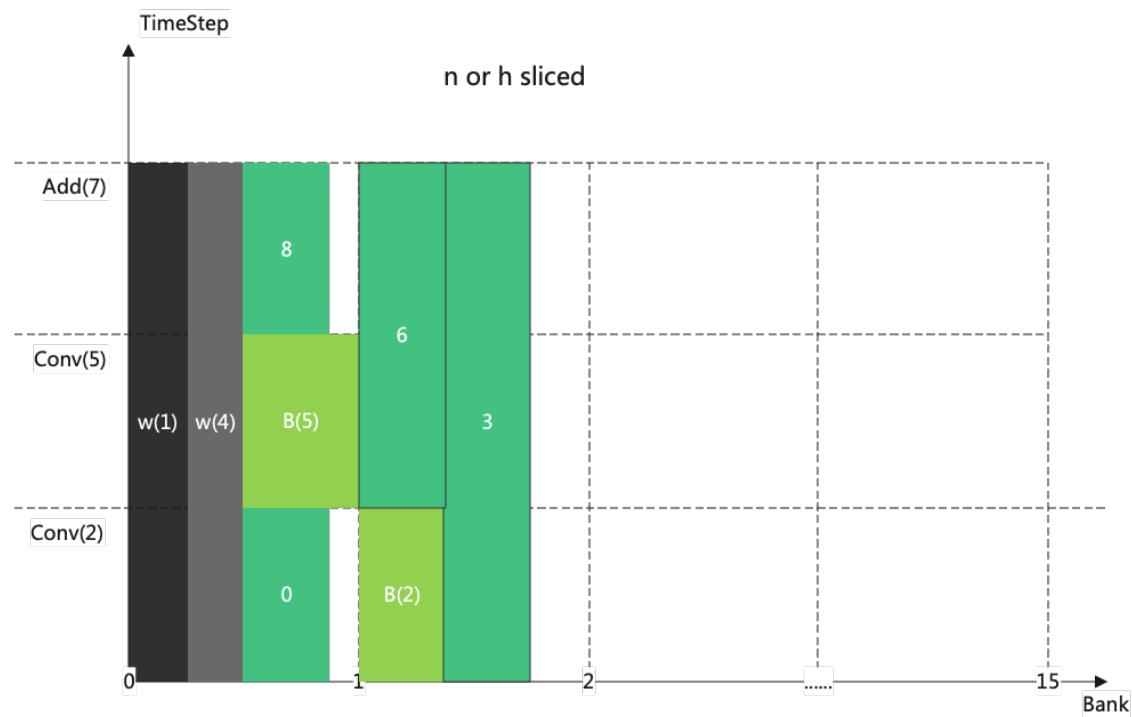


Fig. 10.5: Allocation in the case of slice

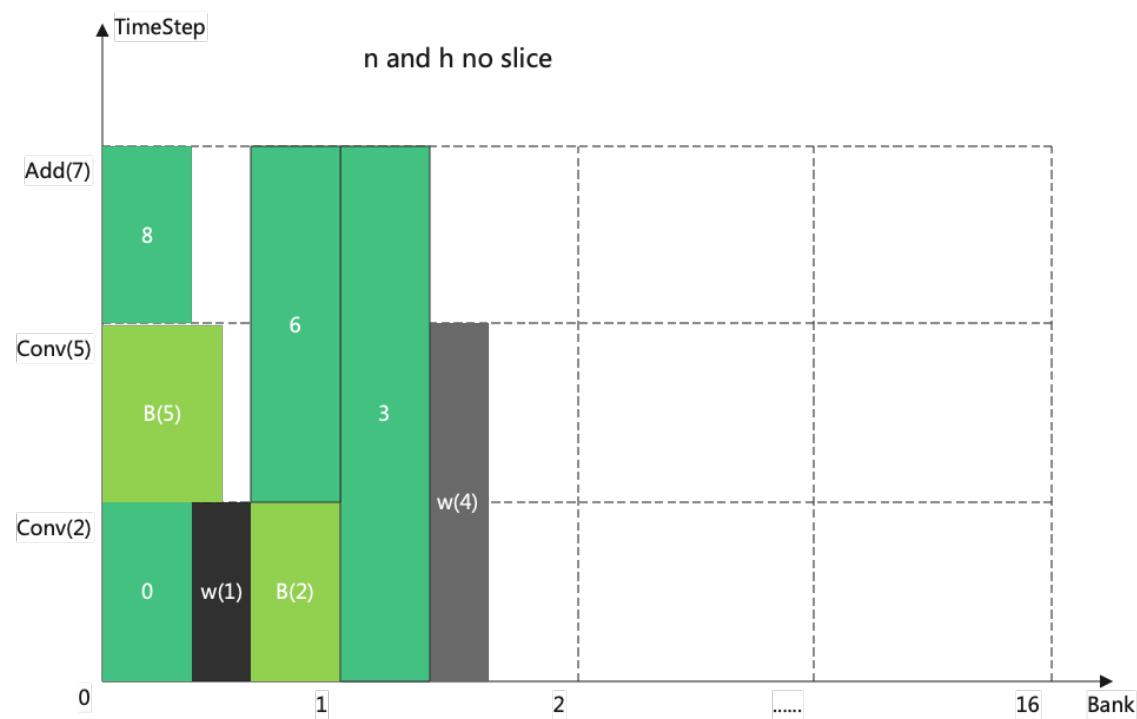


Fig. 10.6: Allocation in the case of no slice

## 10.5 Divide the optimal Group

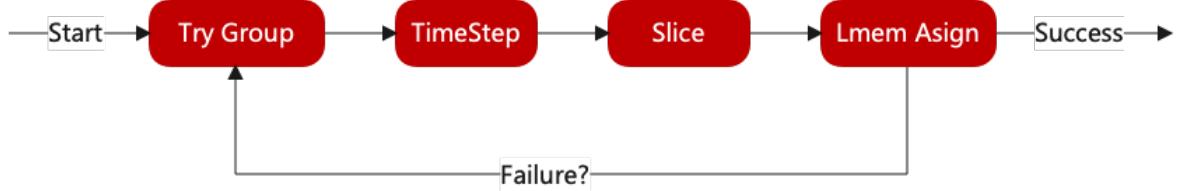


Fig. 10.7: Group process

At present, the group is divided from the tail to the head. N will be sliced first till the smallest unit, then H when it is needed.

When the network is very deep, because Conv, Pool and other operators have duplicate computation parts, too much H slice leads to too many duplicate parts.

In order to avoid too much duplication, it is considered as failed when the input of layer after backward has duplicated part of  $h\_slice > h/2$ .

Example: if the input has  $h = 100$ , and it is sliced into two inputs,  $h[0, 80]$  and  $h[20, 100]$ , then the duplicate part is 60. It is considered as failed. The repeated part is 40 when two inputs are  $h[0, 60]$  and  $h[20, 100]$ , which is considered as success.

# CHAPTER 11

---

## GMEM Allocation

---

### 11.1 1. Purpose

In order to save global memory space and reuse memory space to the greatest extent, GMEM will be allocated to weight tensor first, and then allocated to all global neuron tensors according to their life cycle. In addition, allocated GMEM will be reused during the allocation process.

---

**Note:** global neuron tensor definition: the tensor that needs to be saved in GMEM after the Op operation. If it is a LayerGroup op, only the input/output tensor is considered as global neuron tensor.

---

### 11.2 1. Principle

#### 11.2.1 2.1. GMEM allocation in weight tensor

Iterate through all WeightOp and allocate GMEM sequentially with 4K alignment. Address space will keep accumulating.

### 11.2.2 2.2. GMEM allocation in global neuron tensors

Maximize the reuse of memory space. Allocate GMEM to all global neuron tensors according to their life cycle, and reuse the allocated GMEM during the allocation process.

#### a. Introduction of data structure:

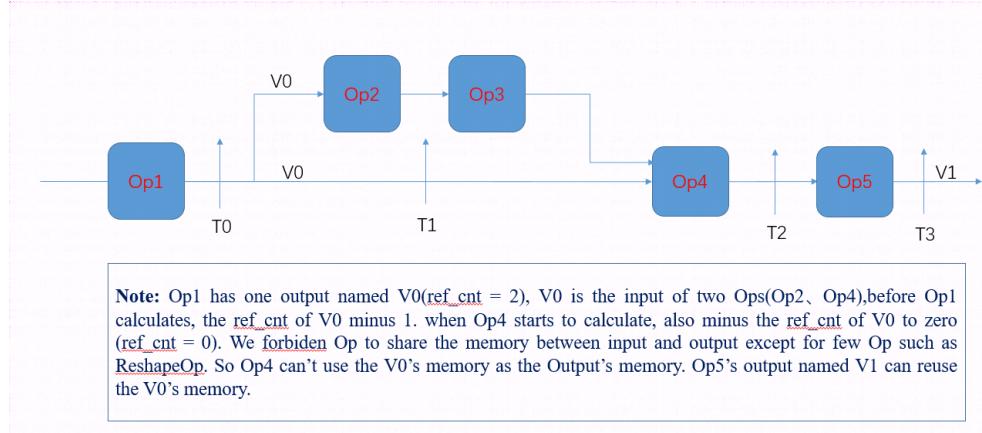
The corresponding tensor, address, size, ref\_cnt (how many OPs are using this tensor) are recorded in rec\_tbl at each allocation. The tensor and address are recorded in the auxiliary data structures hold\_edges,in\_using\_addr respectively.

```
//Value, offset, size, ref_cnt
using gmem_entry = std::tuple<mlir::Value, int64_t, int64_t, int64_t>;
std::vector<gmem_entry> rec_tbl;
std::vector<mlir::Value> hold_edges;
std::set<int64_t> in_using_addr;
```

#### b. Flow description:

- **Iterate through each Op, and determine if the input tensor of the Op is in rec\_tbl, if yes, then determine if ref\_cnt >= 1, if still yes, ref\_cnt**
  - **This operation means that the number of references to the input tensor is reduced by one.**  
If ref\_cnt is equal to 0, it means that the life cycle of the tensor is over, and later tensors can reuse its address space.
- **When allocating the output tensor to each Op, we first check whether the EOL tensor address can be reused. In other words, the rec\_tbl must meet the following 5 conditions before it can be reused:**
  - The corresponding tensor is not in the hold\_edges.
  - The address of the corresponding tensor is not in \_using\_addr
  - The corresponding tensor is already EOL.
  - The address space of the corresponding tensor >= the space required by the current tensor.
  - The address of the input tensor of the current Op is different from the address of the corresponding tensor (e.g., the final result of some Op operations is incorrect, except for reshapeOp).
- **Allocate GMEM to the output tensor of the current Op. Reuse it if step2 shows that it can be reused. Otherwise, open a new GMEM in ddr.**
- **Adjust the lifecycle of the current Op's input tensor and check if it is in hold\_edges. If yes, look in rec\_tbl and check if its ref\_cnt is 0. If yes, remove it from hold\_edges as well as its addr from in\_using\_addr. This**

operation means that the input tensor has finished its life cycle and the address space has been released.



---

**Note:** EOL definition: end-of-life.

---

# CHAPTER 12

---

## CodeGen

---

The code generation (CodeGen) in TPU-MLIR is the final step of BModel creation. Its purpose is to convert MLIR files into the final BModel. This chapter introduces the CodeGen of models/operators in this project.

### 12.1 Main Work

The purpose of CodeGen is to convert the MLIR file into the BModel file. This process will execute the CodeGen interface of each op to generate cmdbuf, and use the Builder module to generate the final BModel in flatbuffers format.

### 12.2 Workflow

The general process of CodeGen can be divided into three parts: instruction generation, instruction storage and instruction retrieval.

Instruction generation: Encapsulate the back-end functions of different processors into classes, execute the op's CodeGen interface, and generate corresponding instructions (binary code);

Instruction storage: Store the instruction (binary code) in the specified data structure through store\_cmd;

Instruction retrieval: After the binary codes of all ops are generated, the compiler will call the function encapsulated in the BM168X series class to retrieve the instructions, and finally generate the Bmodel.

The workflow is as follows:

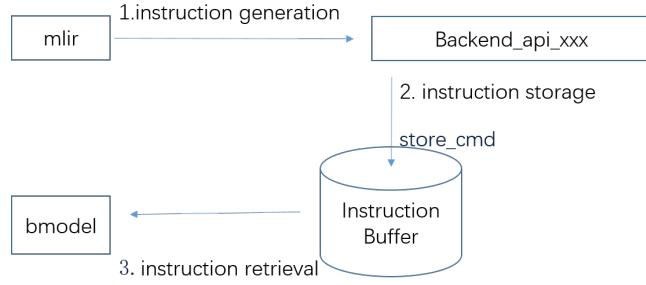


Fig. 12.1: CodeGen Workflow

The following introduces the data structures required in the CodeGen process:

The instructions differ based on the processor's engine, e.g., 1684 has GDMA and TIU, while new architecture processors like bm1690 have sdma, cdma, etc. Using the most common engines, BDC (later renamed to TIU) and GDMA, as examples:

```

std::vector<uint32_t> bdc_buffer;
std::vector<uint32_t> gdma_buffer;
uint32_t gdma_total_id = 0;
uint32_t bdc_total_id = 0;
std::vector<uint32_t> gdma_group_id;
std::vector<uint32_t> bdc_group_id;
std::vector<uint32_t> gdma_bytes;
std::vector<uint32_t> bdc_bytes;
int cmdid_groupnum = 0;
CMD_ID_NODE *cmdid_node;
CMD_ID_NODE *bdc_node;
CMD_ID_NODE *gdma_node;
    
```

bdc\_buffer: stores bdc instructions

gdma\_buffer: stores gdma instructions

gdma\_total\_id: The total number of gdma instructions stored

bdc\_total\_id: The total number of bdc instructions stored

gdma\_bytes: number of gdma instruction bytes

bdc\_bytes: bdc instruction byte number

## 12.3 BM168X and Related classes in TPU-MLIR

These related classes are defined in the folder tpu-mlir/include/tpu\_mlir/Backend. Their purpose is to encapsulate different processor backends, thereby isolating the backend from the CodeGen process.

The inheritance relationship is as follows:

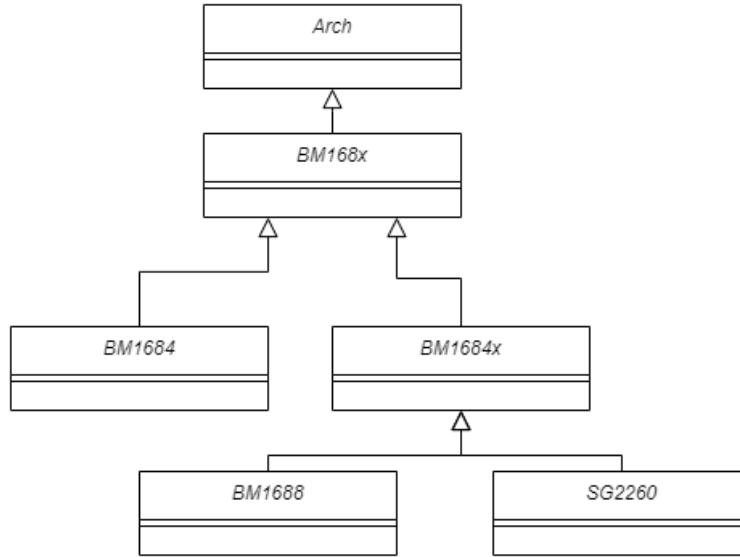


Fig. 12.2: BM168X and its related class inheritance relationships in TPU-MLIR

Only one class exists during a single run (singleton design pattern). When this class is initialized, it undergoes: reading the backend dynamic link library, loading functions (setting backend function pointers), initializing instruction data structures, and setting some hardware-related parameters like NPU\_NUM, L2\_SRAM starting address, etc.

## 12.4 Backend Function Loading

The backend is placed as a dynamic library in the TPU-MLIR project, specifically at third\_party/nntoolchain/lib/libbackend\_xxx.so. The loading method of the backend function is: first define the function pointer, and then load the dynamic library so that the function pointer points to the function in the dynamic library.

**Take the synchronization function tpu\_sync\_all as an example, as we will add multi-core support later, it needs to be well-defined in the relevant backend cmodel library.**

1. Make sure to keep the function name and parameters consistent: `typedef void (*tpu_sync_all)();`
2. Add this function member within the class: `tpu_sync_all, dl_tpu_sync_all;`
3. Add the macro, `CAST_FUNCTION(tpu_sync_all)`, to the implementation of this type of `load_functions` function; This macro can point `dl_tpu_sync_all` to the function in the dynamic library.

After obtaining an instance of this class, we can use the functions in the dynamic library.

## 12.5 Backend store\_cmd

The function store\_cmd in the backend refers to the process where the compiler calls the operators and saves the configured instructions to the designated space. The key function in the backend is in store\_cmd.cpp; for example, cmodel/src/store\_cmd.cpp; cmodel/include/store\_cmd.h. store\_cmd has a series of EngineStorer and CmdStorer classes:

1. EngineStoreInterface (interface class), GDMAEngineStorer, BDEngineStorer and other specific classes that inherit from the EngineStoreInterface interface, EngineStorerDecorator (decoration class interface), VectorDumpEngineStorerDecorator and other specific decoration classes that inherit from EngineStorerDecorator
2. CmdStorerInterface (interface), ConcretCmdStorer inherited from the interface, StorerDecorator: decoration interface, VectorDumpStorerDecorator specific decoration class.

### Relationship and Logic Among the Classes:

1. Using the singleton design pattern, there is only one ‘ConcretCmdStorer’ class in ‘store\_cmd’ , which will store all ‘EngineStorer’ classes. When different engines are called, different ‘EengineStorers’ will be called, as shown in the code below.

```
virtual void store_cmd(int engine_id, void *cmd, CMD_ID_NODE *cur_id_
→node, int port) override
{
    switch (engine_id)
    {
        case ENGINE_BD:
        case ENGINE_GDMA:
        case ENGINE_HAU:
        case ENGINE_SDMA:
            port = 0;
            break;
        case ENGINE_CDMA:
            ASSERT(port < CDMA_NUM);
            break;
        case ENGINE_VSDMA:
            engine_id = ENGINE_SDMA;
            break;
        default:
            ASSERT(0);
            break;
    }
    return this->get(engine_id, port)->store(cmd, cur_id_node);
}
```

2. The function of ‘EngineStorer’ is to parse commands. ‘VectorDumpEngineStorerDecorator’ executes the ‘store’ function and ‘take\_cmds’ function in the ‘EngineStorer’ class to store all instructions in output\_ .

```
class VectorDumpEngineStorerDecorator : public EngineStorerDecorator
{
private:
    std::vector<uint32_t> *output_;

    void take_cmds()
    {
        auto cmd = EngineStorerDecorator::get_cmds();
        (*output_).insert((*output_).end(), cmd.begin(), cmd.end());
    }

public:
    VectorDumpEngineStorerDecorator(ComponentPtr component, std::vector
→<uint32_t> **output)
        : EngineStorerDecorator(component), output_(*output) {}

    virtual void store(void *cmd, CMD_ID_NODE *cur_id_node) override
    {
        EngineStorerDecorator::store(cmd, cur_id_node);
        if (!enabled_)
            return;
        this->take_cmds();
    }

    virtual void store_cmd_end(unsigned dep) override
    {
        EngineStorerDecorator::store_cmd_end(dep);
        this->take_cmds();
    }
};
```

# CHAPTER 13

---

## MLIR Definition

---

This chapter introduces the definition of each element of MLIR, including Dialect, Interface, etc.

### 13.1 Top Dialect

#### 13.1.1 Operations

##### AddOp

###### Brief intro

Add operation,  $Y = coeff_0 * X_0 + coeff_1 * X_1$

###### Input

- inputs: tensor array, corresponding to 2 or more input tensors

###### Output

- output: tensor

###### Attributes

- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number
- coeff: the coefficient corresponding to each tensor, 1.0 by default

**Output**

- output: tensor

**Interface**

None

**Example**

```
%2 = "top.Add"(%0, %1) {do_relu = false} : (tensor<1x3x27x27xf32>, tensor
    ↪<1x3x27x27xf32>) -> tensor<1x3x27x27xf32> loc("add")
```

**AvgPoolOp****Brief intro**

Perform average pooling on the input tensor,  $S = \frac{1}{width * height} \sum_{i,j} a_{ij}$ , where *width* and *height* represent the width and height of the kernel\_shape.  $\sum_{i,j} a_{ij}$  means to sum the kernel\_shape. A sliding window of a given size will sequentially pool the input tensor

**Input**

- input: tensor

**Output**

- output: tensor

**Attributes**

- kernel\_shape: controls the size of the sliding window
- strides: step size, controlling each step of the sliding window
- pads: controls the shape of the padding
- pad\_value: padding content, constant, 0 by default
- count\_include\_pad: whether the result needs to count the pads filled
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Interface**

None

**Example**

```
%90 = "top.AvgPool"(%89) {do_relu = false, kernel_shape = [5, 5], pads = [2, 2,
    ↪ 2, 2], strides = [1, 1]} : (tensor<1x256x20x20xf32>) -> tensor
    ↪<1x256x20x20xf32> loc("resnetv22_pool1_fwd_GlobalAveragePool")
```

## Depth2SpaceOp

### Brief intro

Depth to space operation,  $Y = Depth2Space(X)$

### Input

- inputs: tensor

### Output

- output: tensor

### Attributes

- block\_h: tensor block size of h dimension, i64 type
- block\_w: tensor block size of w dimension, i64 type
- is\_CRD: column-row-depth. If true, the data is arranged in the depth direction according to the order of HWC, otherwise it is CHW, bool type
- is\_inversed: if true, the shape of the result is:  $[n, c * block_h * block_w, h/block_h, w/block_w]$ , otherwise it is:  $[n, c/(block_h * block_w), h * block_h, w * block_w]$ , bool type

### Output

- output: tensor

### Interface

None

### Example

```
%2 = "top.Depth2Space"(%0) {block_h = 2, block_w = 2, is_CRD = true, is_inversed = false} : (tensor<1x8x2x3xf32>) -> tensor<1x2x4x6xf32> loc("add")
```

## BatchNormOp

### Brief intro

Perform Batch Normalization on a 4D input tensor. More details on batch normalization can be found in the paper “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”.

The specific calculation formula is as follows:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

### Input

- input: 4D input tensor

- mean: mean of the input tensor
- variance: variance of the input tensor
- gamma:  $\gamma$  tensor in the formula, can be None
- beta:  $\beta$  tensor in the formula, can be None

### Output

- output: tensor

### Attributes

- epsilon: constant  $\epsilon$  in formula, 1e-05 by default
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

### Interface

None

### Example

```
%5 = "top.BatchNorm"(%0, %1, %2, %3, %4) {epsilon = 1e-05, do_relu = false}
  ↪ : (tensor<1x3x27x27xf32>, tensor<3xf32>, tensor<3xf32>, tensor<3xf32>, F
  ↪ tensor<3xf32>) -> tensor<1x3x27x27xf32> loc("BatchNorm")
```

## CastOp

(To be implemented)

## ClipOp

### Brief intro

Constrain the given input to a certain range

### Input

- input: tensor

### Output

- output: tensor

### Attributes

- min: the lower limit
- max: the upper limit

### Output

- output: tensor

**Interface**

None

**Example**

```
%3 = "top.Clip"(%0) {max = 1%: f64,min = 2%: f64} : (tensor<1x3x32x32xf32>
    ↪) -> tensor<1x3x32x32xf32> loc("Clip")
```

**ConcatOp****Brief intro**

Concatenates the given sequence of tensors in the given dimension. All input tensors either have the same shape (except the dimension to be concatenated) or are all empty.

**Input**

- inputs: tensor array, corresponding to 2 or more input tensors

**Output**

- output: tensor

**Attributes**

- axis: the subscript of the dimension to be concatenated
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Interface**

None

**Example**

```
%2 = "top.Concat"(%0, %1) {axis = 1, do_relu = false} : (tensor
    ↪<1x3x27x27xf32>, tensor<1x3x27x27xf32>) -> tensor<1x6x27x27xf32> loc(
    ↪"Concat")
```

## ConvOp

### Brief intro

Perform 2D convolution operation on the input tensor.

In simple terms, the size of the given input is  $(N, C_{\text{in}}, H, W)$ . The output  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  is calculated as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k),$$

where  $\star$  is a valid cross-correlation operation,  $N$  is the batch size,  $C$  is the number of channels,  $H, W$  is the input image height and width.

### Input

- input: tensor
- filter: parameter tensor. The shape is  
 $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$
- bias: learnable bias tensor with the shape of  $(\text{out\_channels})$

### Output

- output: tensor

### Attributes

- kernel\_shape: the size of the convolution kernel
- strides: strides of convolution
- pads: the number of layers to add 0 to each side of the input
- group: the number of blocked connections from the input channel to the output channel, the default is 1
- dilations: the spacing between convolution kernel elements, optional
- inserts: optional
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

### Interface

None

### Example

```
%2 = "top.Conv"(%0, %1) {kernel_shape = [3, 5], strides = [2, 1], pads = [4, 2]}\n  ↳ : (tensor<20x16x50x100xf32>, tensor<33x3x5xf32>) -> tensor\n  ↳ <20x33x28x49xf32> loc("Conv")
```

## DeconvOp

### Brief intro

Perform a deconvolution operation on the input tensor.

### Input

- input: tensor
- filter: parameter tensor. The shape is  
 $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$
- bias: learnable bias tensor with the shape of  $(\text{out\_channels})$

### Output

- output: tensor

### Attributes

- kernel\_shape: the size of the convolution kernel
- strides: strides of convolution
- pads: the number of layers to add 0 to each side of the input
- group: the number of blocked connections from the input channel to the output channel, the default is 1
- dilations: the spacing between convolution kernel elements, optional
- inserts: optional
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

### Interface

None

### Example

```
%2 = "top.Deconv"(%0, %1) {kernel_shape = (3, 5), strides = (2, 1), pads = (4,
    ↪ 2)} : (tensor<20x16x50x100xf32>, tensor<33x3x5xf32>) -> tensor
    ↪ <20x33x28x49xf32> loc("Deconv")
```

## DivOp

### Brief intro

Division operation,  $Y = X_0/X_1$

### Input

- inputs: tensor array, corresponding to 2 or more input tensors

### Output

- output: tensor

### Attributes

- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number
- multiplier: the multiplier for quantization, the default is 1
- rshift: right shift for quantization, 0 by default

### Output

- output: tensor

### Interface

None

### Example

```
%2 = "top.Div"(%0, %1) {do_relu = false, relu_limit = -1.0, multiplier = 1, F
    ↪rshift = 0} : (tensor<1x3x27x27xf32>, tensor<1x3x27x27xf32>) -> tensor
    ↪<1x3x27x27xf32> loc("div")
```

## InputOp

(To be implemented)

## LeakyReluOp

### Brief intro

Apply the LeakyRelu function on each element in the tensor. The function can be expressed as:  $f(x) = \alpha * x$  for  $x < 0$ ,  $f(x) = x$  for  $x \geq 0$

### Input

- input: tensor

### Output

- output: tensor

**Attributes**

- alpha: the coefficients corresponding to each tensor

**Output**

- output: tensor

**Interface**

None

**Example**

```
%4 = "top.LeakyRelu"(%3) {alpha = 0.67000001668930054 : f64} : (tensor
 ↳<1x32x100x100xf32>) -> tensor<1x32x100x100xf32> loc("LeakyRelu")
```

## LSTMOp

**Brief intro**

Perform the LSTM operation of the RNN

**Input**

- input: tensor

**Output**

- output: tensor

**Attributes**

- filter: convolution kernel
- recurrence: recurrence unit
- bias: parameter of LSTM
- initial\_h: Each sentence in LSTM will get a state after the current cell. The state is a tuple(c, h), where h=[batch\_size, hidden\_size]
- initial\_c: c=[batch\_size, hidden\_size]
- have\_bias: whether to set bias, the default is false
- bidirectional: set the LSTM of the bidirectional loop, the default is false
- batch\_first: whether to put the batch in the first dimension, the default is false

**Output**

- output: tensor

**Interface**

None

**Example**

```
%6 = "top.LSTM"(%0, %1, %2, %3, %4, %5) {batch_first = false, bidirectional=F
    ↪= true, have_bias = true} : (tensor<75x2x128xf32>, tensor<2x256x128xf32>,
    ↪ tensor<2x256x64xf32>, tensor<2x512xf32>, tensor<2x2x64xf32>, tensor
    ↪<2x2x64xf32>) -> tensor<75x2x2x64xf32> loc("LSTM")
```

**LogOp****Brief intro**

Perform element-wise logarithm on the given input tensor

**Input**

- input: tensor

**Output**

- output: tensor

**Attributes**

None

**Output**

- output: tensor

**Interface**

None

**Example**

```
%1 = "top.Log"(%0) : (tensor<1x3x32x32xf32>) -> tensor<1x3x32x32xf32> loc(
    ↪"Log")
```

**MaxPoolOp****Brief intro**

Perform max pool on the given input tensor

**Input**

- input: tensor

**Output**

- output: tensor

**Attributes**

- kernel\_shape: controls the size of the sliding window
- strides: step size, controlling each step of the sliding window

- pads: controls the shape of the padding
- pad\_value: padding content, constant, 0 by default
- count\_include\_pad: whether the result needs to count the pads filled
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Interface**

None

**Example**

```
%8 = "top.MaxPool"(%7) {do_relu = false, kernel_shape = [5, 5], pads = [2, 2,
 ↳ 2, 2], strides = [1, 1]} : (tensor<1x256x20x20xf32>) -> tensor
 ↳<1x256x20x20xf32> loc("resnetv22_pool0_fwd_MaxPool")
```

**MatMulOp****Brief intro**2D matrix multiplication operation,  $C = A * B$ **Input**

- input: tensor: matrix of size m\*k
- right: tensor: matrix of size k\*n

**Output**

- output: tensor: matrix of size m\*n

**Attributes**

- bias: the bias\_scale will be calculated according to the bias during quantization (can be empty)
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Output**

- output: tensor

**Interface**

None

**Example**

```
%2 = "top.MatMul"(%0, %1) {do_relu = false, relu_limit = -1.0} : (tensor
    ↳<3x4xf32>, tensor<4x5xf32>) -> tensor<3x5xf32> loc("matmul")
```

## MulOp

### Brief intro

multiplication operation,  $Y = X_0 * X_1$

### Input

- inputs: tensor array, corresponding to 2 or more input tensors

### Output

- output: tensor

### Attributes

- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number
- multiplier: the multiplier for quantization, the default is 1
- rshift: right shift for quantization, default is 0

### Output

- output: tensor

### Interface

None

### Example

```
%2 = "top.Mul"(%0, %1) {do_relu = false, relu_limit = -1.0, multiplier = 1, F
    ↳rshift = 0} : (tensor<1x3x27x27xf32>, tensor<1x3x27x27xf32>) -> tensor
    ↳<1x3x27x27xf32> loc("mul")
```

## MulConstOp

### Brief intro

Multiply with a constant,  $Y = X * ConstVal$

### Input

- inputs: tensor

### Output

- output: tensor

**Attributes**

- const\_val: constants of type f64
- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Output**

- output: tensor

**Interface**

None

**Example**

```
%1 = arith.constant 4.7 : f64
%2 = "top.MulConst"(%0) {do_relu = false, relu_limit = -1.0} : (tensor
 ↳<1x3x27x27xf64>, %1) -> tensor<1x3x27x27xf64> loc("mulconst")
```

**PermuteOp****Brief intro**

Change the tensor layout. Change the order of tensor data dimensions, and rearrange the input tensor according to the given order

**Input**

- inputs: tensor array, tensor of any types

**Attributes**

- order: the order in which tensors are rearranged

**Output**

- output: rearranged tensor

**Interface**

None

**Example**

```
%2 = "top.Permute"(%1) {order = [0, 1, 3, 4, 2]} : (tensor<4x3x85x20x20xf32>
 ↳) -> tensor<4x3x20x20x85xf32> loc("output_Transpose")
```

## ReluOp

### Brief intro

Performs the ReLU function on each element in the input tensor, if the limit is zero, the upper limit is not used

### Input

- input: tensor

### Output

- output: tensor

### Attributes

- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

### Output

- output: tensor

### Interface

None

### Example

```
%1 = "top.Relu"(%0) {relu_limit = 6.000000e+00 : f64} : (tensor
 ↳<1x3x32x32xf32>) -> tensor<1x3x32x32xf32> loc("Clip")
```

## ReshapeOp

### Brief intro

Reshape operator, which returns a tensor of the given shape with the same type and internal values as the input tensor. Reshape may operate on any row of the tensor. No data values will be modified during the reshaping process

### Input

- input: tensor

### Output

- output: tensor

### Attributes

None

### Interface

None

### Example

```
%133 = "top.Reshape"(%132) : (tensor<1x255x20x20xf32>) -> tensor  
↳ <1x3x85x20x20xf32> loc("resnetv22_flatten0_reshape0_Reshape")
```

## ScaleOp

### Brief intro

Scale operation  $Y = X * S + B$ , where the shape of X/Y is [N, C, H, W], and the shape of S/B is [1, C, 1, , 1].

### Input

- input: tensor
- scale: the magnification of the input
- bias: the bias added after scaling

### Output

- output: tensor

### Attributes

- do\_relu: whether to perform Relu operation on the result, False by default
- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

### Interface

None

### Example

```
%3 = "top.Scale"(%0, %1, %2) {do_relu = false} : (tensor<1x3x27x27xf32>, F  
↳ tensor<1x3x1x1xf32>, tensor<1x3x1x1xf32>) -> tensor<1x3x27x27xf32> loc(  
↳ "Scale")
```

## SigmoidOp

### Brief intro

The activation function, which maps elements in the tensor to a specific interval, [0, 1] by default. The calculation method is:

$$Y = \frac{scale}{1 + e^{-X}} + bias$$

### Input

- inputs: tensor array, tensor of any types

### Attributes

- scale: the magnification of the input, 1 by default
- bias: default is 0

**Output**

- output: tensor

**Interface**

None

**Example**

```
%2 = "top.Sigmoid"(%1) {bias = 0.000000e+00 : f64, scale = 1.000000e+00 : f64}
  ↪ : (tensor<1x16x64x64xf32>) -> tensor<1x16x64x64xf32> loc("output_
  ↪ Sigmoid")
```

**SiLUOp****Brief intro**

The activation function,  $Y = \frac{X}{1+e^{-X}}$  or  $Y = X * Sigmoid(X)$

**Input**

- input: tensor array, tensor of any types

**Attributes**

None

**Output**

- output: tensor

**Interface**

None

**Example**

```
%1 = "top.SiLU"(%0) : (tensor<1x16x64x64xf32>) -> tensor<1x16x64x64xf32>
  ↪ loc("output_Mul")
```

**SliceOp****Brief intro**

Tensor slice, slicing each dimension of the input tensor according to the offset and step size in the offset and steps arrays to generate a new tensor

**Input**

- input: tensor array, tensor of any types

**Attributes**

- offset: an array for storing slice offsets. The index of the offset array corresponds to the dimension index of the input tensor
- steps: an array that stores the step size of the slice. The index of the steps array corresponds to the index of the input tensor dimension

### Output

- output: tensor

### Interface

None

### Example

```
%1 = "top.Slice"(%0) {offset = [2, 10, 10, 12], steps = [1, 2, 2, 3]} : (tensor
 ↳<5x116x64x64xf32>) -> tensor<3x16x16x8xf32> loc("output_Slice")
```

## SoftmaxOp

### Brief intro

For the input tensor, the normalized index value is calculated on the dimension of the specified axis. The calculation method is as follows:

$$\sigma(Z)_i = \frac{e^{\beta Z_i}}{\sum_{j=0}^{K-1} e^{\beta Z_j}},$$

where  $\sum_{j=0}^{K-1} e^{\beta Z_j}$  does the exponential summation on the axis dimension. j ranges from 0 to K-1 and K is the size of the input tensor in the axis dimension.

For example, the size of the input tensor is  $(N, C, W, H)$ , and the Softmax is calculated on the channel of axis=1. The calculation method is:

$$Y_{n,i,w,h} = \frac{e^{\beta X_{n,i,w,h}}}{\sum_{j=0}^{C-1} e^{\beta X_{n,j,w,h}}}$$

### Input

- input: tensor array, tensor of any types

### Attributes

- axis: dimension index, which is used to specify the dimension to perform softmax. It can take the value from  $[-r, r-1]$ , where r is the number of dimensions of the input tensor. When axis is negative, it means the reverse order dimension
- beta: The scaling factor for the input in the tflite model, invalid for non-tflite models, 1.0 by default.

### Output

- output: the tensor on which the softmax is performed.

**Interface**

None

**Example**

```
%1 = "top.Softmax"(%0) {axis = 1 : i64} : (tensor<1x1000x1x1xf32>) -> tensor  
→<1x1000x1x1xf32> loc("output_Softmax")
```

**SqueezeOp****Brief intro**

Crop the input tensor with the specified dimension and return the cropped tensor

**Input**

- input: tensor

**Output**

- output: tensor

**Attributes**

- axes: specifies the dimension to be cropped. 0 represents the first dimension and -1 represents the last dimension

**Interface**

None

**Example**

```
%133 = "top.Squeeze"(%132) {axes = [-1]} : (tensor<1x255x20x20xf32>) -> tensor  
→<1x255x20xf32> loc(#loc278)
```

**UpsampleOp****Brief intro**

Upsampling op, upsampling the input tensor nearest and returning the tensor

**Input**

tensor

**Attributes**

- scale\_h: the ratio of the height of the target image to the original image
- scale\_w: the ratio of the width of the target image to the original image
- do\_relu: whether to perform Relu operation on the result, False by default

- relu\_limit: specify the upper limit value if doing Relu. There is no upper limit if it is a negative number

**Output**

- output: tensor

**Interface**

None

**Example**

```
%179 = "top.Upsample"(%178) {scale_h = 2 : i64, scale_w = 2 : i64} : (tensor
 ↳<1x128x40x40xf32>) -> tensor<1x128x80x80xf32> loc("268_Resize")
```

**WeightOp****Brief intro**

The weight op, including the reading and creation of weights. Weights will be stored in the npz file. The location of the weight corresponds to the tensor name in npz.

**Input**

None

**Attributes**

None

**Output**

- output: weight Tensor

**Interface**

- read: read weight data, the type is specified by the model
- read\_as\_float: convert the weight data to float type for reading
- read\_as\_byte: read the weight data in byte type
- create: create weight op
- clone\_bf16: convert the current weight to bf16 and create a weight Op
- clone\_f16: convert the current weight to f16 and create a weight Op

**Example**

```
%1 = "top.Weight"() : () -> tensor<32x16x3x3xf32> loc("filter")
```

# CHAPTER 14

---

## Accuracy Validation

---

### 14.1 Introduction

#### 14.1.1 Objects

The accuracy validation in TPU-MLIR is mainly for the mlir model, fp32 uses the mlir model of the top layer while the int8 symmetric and asymmetric quantization uses the mlir model of the tpu layer.

#### 14.1.2 Metrics

Currently, the validation is mainly used for classification and object detection networks. The metrics for classification networks are Top-1 and Top-5 accuracy, while the object detection networks use 12 metrics of COCO, as shown below. Generally, we record the Average

Precision when IoU=0.5 (i.e., PASCAL VOC metric).

**AveragePrecision(AP) :**

$$\begin{aligned} AP & \text{ \% AP at IoU=.50:.05:.95 (primary challenge metric)} \\ AP^{IoU} = .50 & \text{ \% AP at IoU=.50 (PASCAL VOC metric)} \\ AP^{IoU} = .75 & \text{ \% AP at IoU=.75 (strict metric)} \end{aligned}$$

**AP Across Scales :**

$$\begin{aligned} AP^{small} & \text{ \% AP for small objects: } area < 32^2 \\ AP^{medium} & \text{ \% AP for medium objects: } 32^2 < area < 96^2 \\ AP^{large} & \text{ \% AP for large objects: } area > 96^2 \end{aligned}$$

**AverageRecall(AR) :**

$$\begin{aligned} AR^{max=1} & \text{ \% AR given 1 detection per image} \\ AR^{max=10} & \text{ \% AR given 10 detections per image} \\ AR^{max=100} & \text{ \% AR given 100 detections per image} \end{aligned}$$

**AP Across Scales :**

$$\begin{aligned} AP^{small} & \text{ \% AP for small objects: } area < 32^2 \\ AP^{medium} & \text{ \% AP for medium objects: } 32^2 < area < 96^2 \\ AP^{large} & \text{ \% AP for large objects: } area > 96^2 \end{aligned}$$

### 14.1.3 Datasets

In addition, the dataset used for validation needs to be downloaded by yourself. Classification networks use the validation set of ILSVRC2012 (50,000 images, <https://www.image-net.org/challenges/LSVRC/2012/>). There are two ways to place the images in the dataset. One is that there are 1000 subdirectories under the dataset directory, corresponding to 1000 classes, and each class has 50 images. In this case, no additional label file is required. The other way is that all images are in the same dataset directory, and there is an additional label file. According to the sequence of images' names, each line in the txt file uses a number from 1 to 1000 to indicate the class of each image.

Object detection networks use the COCO2017 validation set (5000 images, <https://cocodataset.org/#download>). All images are under the same dataset directory. The corresponding json label file needs to be downloaded as well.

## 14.2 Validation Interface

TPU-MLIR provides the command for accuracy validation:

```
$ model_eval.py \
  --model_file mobilenet_v2.mlir \
  --count 50 \
  --dataset_type imagenet \
  --postprocess_type topx \
  --dataset datasets/ILSVRC2012_img_val_with_subdir
```

The supported parameters are shown below:

Table 14.1: Function of model\_eval.py parameters

Name	Required?	Explanation
model_file	Y	Model file
dataset	N	Directory of dataset
dataset_type	N	Dataset type. Currently mainly supports imagenet, coco. The default is imagenet
postprocess_type	Y	Metric. Currently supports topx and coco_mAP
label_file	N	txt label file, which may be needed when validating the accuracy of classification networks
coco_annotation	N	json label file, required when validating object detection networks
count	N	The number of images used for validation. The default is to use the entire dataset.

## 14.3 Validation Example

In this section, mobilenet\_v2 and yolov5s are used as the representative of the classification network and the object detection network for accuracy validation.

### 14.3.1 mobilenet\_v2

#### 1. Dataset Downloading

Download the ILSVRC2012 validation set to the datasets/ILSVRC2012\_img\_val\_with\_subdir directory. Images of the dataset are placed in subdirectories, so no additional label files are required.

#### 2. Model Conversion

Use the model\_transform.py interface to convert the original model to the mobilenet\_v2.mlir model, and obtain mobilenet\_v2\_cali\_table through the run\_calibration.py interface. Please refer to the “User Interface” chapter for

specific usage. The INT8 model of the tpu layer is obtained through the command below. After running the command, an intermediate file named mobilenet\_v2\_bm1684x\_int8\_sym\_tpu.mlir will be generated. We will use this intermediate file to validate the accuracy of the INT8 symmetric quantized model:

```
# INT8 Sym Model
$ model_deploy.py \
--mlir mobilenet_v2.mlir \
--quantize INT8 \
--calibration_table mobilenet_v2_cali_table \
--processor BM1684X \
--test_input mobilenet_v2_in_f32.npz \
--test_reference mobilenet_v2_top_outputs.npz \
--tolerance 0.95,0.69 \
--model mobilenet_v2_int8.bmodel
```

### 3. Accuracy Validation

Use the model\_eval.py interface to validate:

```
# F32 model validation
$ model_eval.py \
--model_file mobilenet_v2.mlir \
--count 50000 \
--dataset_type imagenet \
--postprocess_type topx \
--dataset datasets/ILSVRC2012_img_val_with_subdir

# INT8 sym model validation
$ model_eval.py \
--model_file mobilenet_v2_bm1684x_int8_sym_tpu.mlir \
--count 50000 \
--dataset_type imagenet \
--postprocess_type topx \
--dataset datasets/ILSVRC2012_img_val_with_subdir
```

The accuracy validation results of the F32 model and the INT8 symmetric quantization model are as follows:

```
# mobilenet_v2.mlir validation result
2022/11/08 01:30:29 - INFO : idx:50000, top1:0.710, top5:0.899
INFO:root:idx:50000, top1:0.710, top5:0.899

# mobilenet_v2_bm1684x_int8_sym_tpu.mlir validation result
2022/11/08 05:43:27 - INFO : idx:50000, top1:0.702, top5:0.895
INFO:root:idx:50000, top1:0.702, top5:0.895
```

### 14.3.2 yolov5s

#### 1. Dataset Downloading

Download the COCO2017 validation set to the datasets/val2017 directory, which contains 5,000 images for validation. The corresponding label file instances\_val2017.json is downloaded to the datasets directory.

#### 2. Model Conversion

The conversion process is similar to mobilenet\_v2.

#### 3. Accuracy Validation

Use the model\_eval.py interface to validate:

```
# F32 model validation
$ model_eval.py \
--model_file yolov5s.mlir \
--count 5000 \
--dataset_type coco \
--postprocess_type coco_mAP \
--coco_annotation datasets/instances_val2017.json \
--dataset datasets/val2017

# INT8 sym model validation
$ model_eval.py \
--model_file yolov5s_bm1684x_int8_sym_tpu.mlir \
--count 5000 \
--dataset_type coco \
--postprocess_type coco_mAP \
--coco_annotation datasets/instances_val2017.json \
--dataset datasets/val2017
```

The accuracy validation results of the F32 model and the INT8 symmetric quantization model are as follows:

```
# yolov5s.mlir validation result
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.369
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.561
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.393
Average Precision (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.217
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.422
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.470
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.300
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.502
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.542
Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.359
Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.602
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.670

# yolov5s_bm1684x_int8_sym_tpu.mlir validation result
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.337
```

(continues on next page)

(continued from previous page)

Average Precision	(AP) @[ IoU=0.50	area= all   maxDets=100 ] = 0.544
Average Precision	(AP) @[ IoU=0.75	area= all   maxDets=100 ] = 0.365
Average Precision	(AP) @[ IoU=0.50:0.95	area= small   maxDets=100 ] = 0.196
Average Precision	(AP) @[ IoU=0.50:0.95	area=medium   maxDets=100 ] = 0.382
Average Precision	(AP) @[ IoU=0.50:0.95	area= large   maxDets=100 ] = 0.432
Average Recall	(AR) @[ IoU=0.50:0.95	area= all   maxDets= 1 ] = 0.281
Average Recall	(AR) @[ IoU=0.50:0.95	area= all   maxDets= 10 ] = 0.473
Average Recall	(AR) @[ IoU=0.50:0.95	area= all   maxDets=100 ] = 0.514
Average Recall	(AR) @[ IoU=0.50:0.95	area= small   maxDets=100 ] = 0.337
Average Recall	(AR) @[ IoU=0.50:0.95	area=medium   maxDets=100 ] = 0.566
Average Recall	(AR) @[ IoU=0.50:0.95	area= large   maxDets=100 ] = 0.636

# CHAPTER 15

---

## Quantization aware training

---

### 15.1 Basic Principles

Compared with the precision loss caused by post-training quantization because it is not the global optimal, QAT quantization perception training can achieve the global optimal based on loss optimization and reduce the quantization precision loss as far as possible. The basic principle is as follows: In fp32 model training, weight and activation errors caused by inference quantization are introduced in advance, and task loss is used to optimize learnable weight and quantized scale and zp values on the training set. Even under the influence of this quantization error, task loss can reach relatively low loss value through learning. In this way, when the real inference deployment of quantization later, because the error introduced by quantization has already been well adapted in the training, as long as the inference and the calculation of training can be guaranteed to be completely aligned, theoretically, there will be no precision loss in the inference quantization.

### 15.2 tpu-mlir QAT implementation scheme and characteristics

#### 15.2.1 Main body flow

During user training, model QAT quantization API is called to modify the training model: In reasoning, after op fusion, a pseudo-quantization node is inserted before the input (including weight and bias) of the op that needs to be quantized (the quantization parameters of this node can be configured, such as per-chan/layer, symmetry or not, number of quantization bits, etc.), and then the user uses the modified model for normal training process. After completing a few rounds of training, Call the transformation deployment API interface to convert the

---

trained model into the FP32-weighted onnx model, extract the parameters from the pseudo-quantization node and export them to the quantization parameter text file. Finally, input the optimized onnx model and the quantization parameter file into the tpu-mlir tool chain, and convert and deploy according to the post-training quantization method mentioned above.

### 15.2.2 Features of the Scheme

Feature 1: Based on pytorch;QAT is an additional finetune part of the training pipeline, and only deep integration with the training environment can facilitate users to use various scenarios. Considering pytorch has the most extensive usage rate, the current scheme is based on pytorch only. If qat supports other frameworks in the future, the scheme will be very different. Its trace and module replacement mechanisms are deeply dependent on the support of the native training platform.

Feature 2: Users basically have no sense; Different from earlier schemes that require deep manual intervention in model transformation, this scheme is based on pytorch fx which helps automatically model tracing, pseudo-quantization node insertion, custom module replacement and other operations. In most cases, users can complete model transformation with minus change of the default configuration.

Feature 3: Based on the open-sourced mqbench training framework by SenseTime, SOPHGO-mq include particular quantization configuration for SOPHGO ASICs.

## 15.3 Installation Method

SOPHGO-mq is recommended to be used in docker container, the docker image can be pulled by:

```
docker pull sophgo/tpuc_dev:v3.3-cuda
```

The docker image includes pytorch 2.3.0 and cuda 12.1, and intergrated the environment for tpu-mlir.

### 15.3.1 Install with setup package

Download setup package from download area of open-source project <https://github.com/sophgo/sophgo-mq.git>, for example, sophgo\_mq-1.0.1-cp310-cp310-linux\_x86\_64.whl, install with:

```
pip3 install sophgo_mq-1.0.1-cp310-cp310-linux_x86_64.whl
```

### 15.3.2 Install from source

1、Run the command to get the latest code on github: git clone <https://github.com/sophgo/sophgo-mq.git>

2、Execute after entering the SOPHGO-mq directory:

```
pip install -r requirements.txt #Note: torch version 2.3.0 is currently required
python setup.py install
```

3、Execute python -c ‘import sophgo\_mq’ and if it does not return any errors, it indicates that the installation is correct. If there is an error with the installation, execute pip uninstall sophgo\_mq to uninstall it and then try again.

## 15.4 Basic Steps

### 15.4.1 Step 1: Interface import and model prepare

Add the following python module import interface to the training file:

```
import torch
import torchvision.models as models
from sophgo_mq.prepare_by_platform import prepare_by_platform # init module
from sophgo_mq.utils.state import enable_quantization, enable_calibration #calibration and F
→quantization switch
from sophgo_mq.convert_deploy import convert_deploy #deploy interface
    import tpu_mlir # with tpu-mlir introduced, bmodel can be generated in SOPHGO-
→mq environment
        from tools.model_runner import mlir_inference #tpu-mlir inference module, accuracy can be F
→checked by tpu-mlir inference module

#Use the pre-trained ResNet18 model from the torchvision model zoo.
model = models.__dict__['resnet18'](pretrained=True)

#1.Trace the model, using a dictionary to specify the processor type as BM1690 and the F
→quantization mode as weight_activation. In this quantization mode, both weights and F
→activations are quantized. Specify the quantization strategy for CNN type.
extra_prepare_dict = {
'quant_dict': {
    'chip': 'BM1690',
    'quantmode': 'weight_activation',
    'strategy': 'CNN',
},
}
model_quantized = prepare_by_platform(model, prepare_custom_config_dict=extra_prepare_
→dict)
```

When the above interface selects the BM1690 processor, the default quantization configuration is as shown in the following figure:

```
'BM1690': dict(qtype='affine',
    w_qscheme=QuantizeScheme(symmetry=True, per_channel=True, pot_scale=False, bit=8),
    a_qscheme=QuantizeScheme(symmetry=True, per_channel=False, pot_scale=False, bit=8),
    default_weight_quantize=LearnableFakeQuantize,
    default_act_quantize=LearnableFakeQuantize,
    default_weight_observer=MinMaxObserver,
    default_act_observer=EMAMinMaxObserver),
```

The meanings of the quantization configuration items in the above figure, from top to bottom, are as follows:

- 1、The weight quantization scheme is: per-chan symmetric 8bit quantization, the scale coefficient is not power-of-2, but arbitrary
- 2、The activation quantization scheme is per-layer symmetric 8bit quantization
- 3/4、The weights and activation pseudo-quantization schemes are: LearnableFakeQuantize, namely LSQ algorithm
- 5/6、The dynamic range statistics and scale calculation scheme of weights are as follows: MinMaxObserver, and the activation is EMAMinMaxObserver with moving average

#### 15.4.2 Step 2: Calibration and quantization training

```
#1.Turn on the calibration switch to allow the pytorch observer object to collect the activation[F]
→distribution and calculate the initial scale and zp when reasoning on the model
enable_calibration(model_quantized)
# iterations of calibration
for i, (images, _) in enumerate(cali_loader):
    model_quantized(images) #All you need is forward reasoning
#3.After the pseudo-quantization switch is turned on, the quantization error will be introduced[F]
→by invoking the QuantizeBase subobject to conduct the pseudo-quantization operation when[F]
→reasoning on the model
enable_quantization(model_quantized)
# iterations of training
for i, (images, target) in enumerate(train_loader):
    #Forward reasoning and calculation loss
    output = model_quantized(images)
    loss = criterion(output, target)
    #Back to back propagation gradient
    loss.backward()
    #Update weights and pseudo-quantization parameters
    optimizer.step()
```

### 15.4.3 Step 3: Export tuned fp32 model

**Set reasonable training hyperparameters. The suggestions are as follows:**

-epochs=1:About 1~3 can be;

-lr=1e-4:The learning rate should be the learning rate when fp32 converges, or even lower;

-optim=sgd:The default is sgd;

```
#Here the batch-size can be adjusted according to the need, do not have to be consistent with theF
→training batch-size
input_shape={'input': [4, 3, 224, 224]}
# Specify the exported model type as CNN.
net_type='CNN'
#4. Before export, the conv+bn layer is fused (conv+bn is true fusion when train is used in theF
→front), and the parameters in the pseudo-quantization node are saved to the parameter file, andF
→then removed.
convert_deploy(model_quantized, net_type, input_shape)
```

### 15.4.4 Step 4: Initiate the training

The transformation deployment to sophg-tpu hardware was completed using the model\_transform.py and model\_deploy.py scripts of tpu-mlir;

By introducing tpu-mlir in SOPHGO-mq, user can use tpu-mlir inference interface to simulate the running of model on ASIC. By using this interface, model is generated and while training. User can replace traditional evaluation module with tpu-mlir inference, input and output to this interface are in numpy format, example code is as following:

```
import tpu_mlir
from tools.model_runner import mlir_inference
...
for i, (images, target) in enumerate(bmodel_test_loader):
    images = images.cpu()
    target = target.cpu()
    inputs['data'] = images.numpy()
    output = mlir_inference(inputs, mlir_model_path, dump_all = False)
    output = torch.from_numpy(list(output.values())[0])
    loss = criterion(output, target)
```

## 15.5 Use Examples-resnet18

Run application/imagenet\_example/main.py to qat train resent18 as follows:

```
CUDA_VISIBLE_DEVICES=0 python application/imagenet_example/main.py \
--arch=resnet18 \
--batch-size=128 \
--lr=1e-4 \
--epochs=1 \
--optim=sgd \
--cuda=0 \
--pretrained \
--evaluate \
--train_data=/home/data/imagenet \
--val_data=/home/data/imagenet \
--chip=BM1690 \
--quantmode=weight_activation \
--deploy_batch_size=10 \
--pre_eval_and_export \
--output_path=./
```

The command output log above contains the following(Original onnx model accuracy) accuracy information of the original model (it can be compared with the accuracy on the official webpage to confirm the correct training environment, such as the official nominal name:Acc@1 69.76 Acc@5 89.08,The link is:[https://pytorch.apachecn.org/#/docs/1.0/torchvision\\_models](https://pytorch.apachecn.org/#/docs/1.0/torchvision_models)) :

```
原始onnx模型精度
Test: [ 0/391] Time 4.935 ( 4.935) Loss 6.1858e-01 (6.1858e-01) Acc@1 82.03 ( 82.03) Acc@5 96.09 ( 96.09)
Test: [100/391] Time 0.070 ( 1.506) Loss 7.8789e-01 (9.1255e-01) Acc@1 78.91 ( 75.84) Acc@5 93.75 ( 93.23)
Test: [200/391] Time 0.070 ( 1.462) Loss 1.1644e+00 (1.0469e+00) Acc@1 66.41 ( 73.56) Acc@5 90.62 ( 91.71)
Test: [300/391] Time 0.070 ( 1.432) Loss 1.2773e+00 (1.1938e+00) Acc@1 76.56 ( 70.79) Acc@5 87.50 ( 89.75)
| * Acc@1 69.758 Acc@5 89.078
```

Fig. 15.1: Original onnx model accuracy

After completing the qat training, the eval accuracy of the running band quantization node, theoretically the int8 accuracy of the tpu-mlir should be exactly aligned with this, as shown in the figure(resnet18 qat training accuracy) below:

```
Test: [ 0/391] Time 4.280 ( 4.280) Loss 5.8762e-01 (5.8762e-01) Acc@1 85.94 ( 85.94) Acc@5 95.31 ( 95.31)
Test: [100/391] Time 0.140 ( 1.426) Loss 7.6487e-01 (8.9926e-01) Acc@1 78.91 ( 76.37) Acc@5 95.31 ( 93.49)
Test: [200/391] Time 2.282 ( 1.439) Loss 1.1717e+00 (1.0323e+00) Acc@1 67.97 ( 73.92) Acc@5 91.41 ( 92.04)
Test: [300/391] Time 3.115 ( 1.367) Loss 1.2757e+00 (1.1739e+00) Acc@1 74.22 ( 71.16) Acc@5 88.28 ( 90.08)
| * Acc@1 70.040 Acc@5 89.298
```

Fig. 15.2: resnet18 qat training accuracy

The final output directory is as follows(resnet18 qat training output model directory):

The resnet18\_ori.onnx in the figure above is the original pytorch model transferred onnx file. This resnet18\_ori.onnx is quantified by PTQ with the tpu-mlir tool chain, and its symmetry and asymmetry quantization accuracy are measured as the baseline and

```
≡ resnet18_calib_table_from_sophgo_mq
{ } resnet18_clip_ranges.json
≡ resnet18_deploy_model.onnx
≡ resnet18_ori.onnx
≡ resnet18_q_table_from_sophgo_mq_sophgo_tpu
≡ resnet18.onnx
```

Fig. 15.3: resnet18 qat training output model directory

`resnet18_cali_table_from_sophgo_mq` is the exported quantization parameter file with the following contents([resnet18 Sample qat quantization parameter table](#)):

```
# work_mode:QAT_all_int8 //Automatically generated, do not modify, work_mode choice:[QAT_all_int8, QAT_mix_prec]
# op_name      threshold    min      max
data 4.7558303 -2.1247120 2.6311185
472_Relu_weight 64 0.0015318460064008832 0.0004977746866643429 1.1920928955078125e-07 0.00042553359526209533 1.1920928955078125e-07 0.0006113630370236933 0.0016739938873797655 0.000381655654632486403 0.0031490952242165804 0.0002667098306119442 0.0019505330128595233 0.0006429849308915436 0.00078482684491302818 1.1920928955078125e-07 0.0003549025859683752 1.1920928955078125e-07 0.0003844442811086774 0.002309327016998806 1.1920928955078125e-07 0.002218325389549136 0.0025145262479782104 0.0012810979969799519 0.0008265025680884772 0.0010380028979852796 0.00019447231898084283 0.00037283531855791807 0.0010219684336334467 64 0 0 0 0 0 0 0 0 0 0 0 0 472_Relu 4.0415673 -0.0000000 4.0415673
476_MaxPool 4.0466290 0.0000000 4.0466290
```

Fig. 15.4: resnet18 Sample qat quantization parameter table

- a、In the red box of the first row in the figure above, work\_mode is QAT\_all\_int8, indicating int8 quantization of the whole network. It can be selected from [QAT\_all\_int8, QAT\_mix\_prec], and quantization parameters such as symmetry and asymmetry will also be included.
  - b、In the figure above, 472\_Relu\_weight represents the QAT-tuned scale and zp parameters of conv weight. The first 64 represents the scale followed by 64, and the second 64 represents the zp followed by 64.tpu-mlir imports the weight\_scale attribute of the top weight. If this attribute exists in the int8 lowering time, it is directly used. When it does not, it is recalculated according to the maximum lowering value.
  - c、In the case of asymmetric quantization, min and max above are calculated according to the scale, zp, qmin and qmax tuned by the activated qat. threshold is calculated according to the activated scale in the case of symmetric quantization, and both are not valid at the same time.

## 15.6 Tpu-mlir QAT test environment

QAT model is targeted to SOPHGO ASIC, accuracy of the model can be verified with end to end verification program, usually it is deployed on processor. Within development environment, accuracy can be evaluated by tpu-mlir inference interface for convinence, sample code as following:

### 15.6.1 Adding a cfg File

Go to the tpu-mlir/regression/eval directory and add {model\_name}\_qat.cfg to the qat\_config subdirectory. For example, the contents of the resnet18\_qat.cfg file are as follows:

```
dataset=${REGRESSION_PATH}/dataset/ILSVRC2012
test_input=${REGRESSION_PATH}/image/cat.jpg
input_shapes=[[1,3,224,224]] #Modified according to the actual shape
#The following is the image preprocessing parameters, fill in according to the actual situation
resize_dims=256,256
mean=123.675,116.28,103.53
scale=0.0171,0.0175,0.0174
pixel_format=rgb
int8_sym_tolerance=0.97,0.80
int8_asym_tolerance=0.98,0.80
debug_cmd=use_pil_resize
```

You can also add {model\_name}\_qat\_ori.cfg file: Quantify the original pytorch model as baseline, which can be exactly the same as {model\_name}\_qat.cfg above;

### 15.6.2 Modify and execute run\_eval.py

In the following figure, fill in more command strings of different precision evaluation methods in postprocess\_type\_all, such as the existing imagenet classification and coco detection precision calculation strings in the figure;In the following figure, model\_list\_all fills in the mapping of the model name to the parameter, for example:resnet18\_qat's [0,0], where the first parameter represents the first command string in postprocess\_type\_all, and the second parameter represents the first directory in qat\_model\_path (separated by commas):

```
postprocess_type_all=[
    "--count 0 --dataset_type imagenet --postprocess_type topx --dataset /workspace/datasets/ILSVRC2012_img_val_with_subdir",
    "--count 0 --dataset_type coco --postprocess_type coco_mAP --dataset /workspace/datasets/coco_for_mlir_test/val2017 --coco_annotation_file /workspace/datasets/coco2017/annotations/instances_val2017.json"
]
model_list_all={
    # object detection
    # "yolov5s_qat_ori": [1,1],
    # "yolov5s_qat": [1,1],
    # classification
    "resnet18_qat_ori": [0,0],
    "resnet18_qat": [0,0],
}
```

After configuring the postprocess\_type\_all and model\_list\_all arrays as needed, execute the following run\_eval.py command:

```
python3 run_eval.py
--qat_eval      #In qat validation mode, the default is to perform regular model accuracy[F]
→testing using the configuration in the tpu-mlir/regression/config
--fast_test     #Quick test before the official test (only test the accuracy of 30 graphs) to[F]
→confirm that all cases can run
--pool_size 20  #By default, 10 processes run. If the machine has many idle resources, you[F]
→can configure more
--batch_size 10 #qat exports the batch-size of the model. The default is 1
--qat_model_path '/workspace/classify_models./workspace/yolov5/qat_models' #Directory[F]
→of the qat model,For example, the value of model_list_all['resnet18_qat'][1] is 0, indicating[F]
→the first directory address of the model target in the qat_model_path:/workspace/classify_
→models/
--debug_cmd use_pil_resize    #Use pil resize
```

After or during the test, view the model\_eval script output log file starting with log\_ in the subdirectory named {model\_name}\_qat, For example, log\_resnet18\_qat.mlir indicates the log of testing resnet18\_qat.mlir in the directory.log\_resnet18\_qat\_bm1684x\_tpu\_int8\_sym.mlir Indicates the test log of resnet18\_qat\_bm1684x\_tpu\_int8\_sym.mlir in this directory.

## 15.7 Use Examples-yolov5s

Execute the following command in application/yolov5\_example to start QAT Training:

```
CUDA_VISIBLE_DEVICES=0 python train.py \
--cfg=yolov5s.yaml \
--weights=yolov5s.pt \
--data=coco.yaml \
--epochs=5 \
--output_path=./ \
--batch-size=8 \
--quantize \
```

After the training is completed, the same test and transformation deployment process as resnet18 before can be adopted.

## 15.8 Use Examples-bert

Execute the following command in application/nlp\_example to start QAT Training:

```
CUDA_VISIBLE_DEVICES=0 python qat_bertbase_questionanswer.py
```

# CHAPTER 16

---

## TpuLang Interface

---

This chapter mainly introduces the process of converting models using TpuLang.

### 16.1 Main Work

TpuLang provides mlir external interface functions. Users can directly build their own network through Tpulang, and convert the model to the Top layer (hardware-independent layer) mlir model (the Canonicalize part is not included, so the generated file name is “\*\_origin.mlir” ). This process will create and add operators (Op) one by one according to the input interface functions. Finally, a mlir model file and a corresponding weight npz file will be generated.

### 16.2 Work Process

1. Initialization: Set up the platform and create the graph.
  2. Add OPs: cyclically add OPs of the model
    - The input parameters are converted to dict format;
    - Create output tensor;
    - Set the quantization parameters of the tensor (scale, zero\_point);
    - Create op(op\_type, inputs, outputs, params) and insert it into the graph.
  3. Set the input and output tensor of the model. Get all model information.
  4. Initialize TpuLangConverter (initMLIRImporter)
-

## 5. generate\_mlir

- Create the input op, the nodes op in the middle of the model and the return op in turn, and add them to the mlir text (if the op has weight, an additional weight op will be created)

## 6. Output

- Convert the generated text to str and save it as “.mlir” file
- Save model weights (tensors) as “.npz” files

## 7. End: Release the graph.

The workflow of TpuLang conversion is shown in the figure ([TpuLang conversion process](#)).

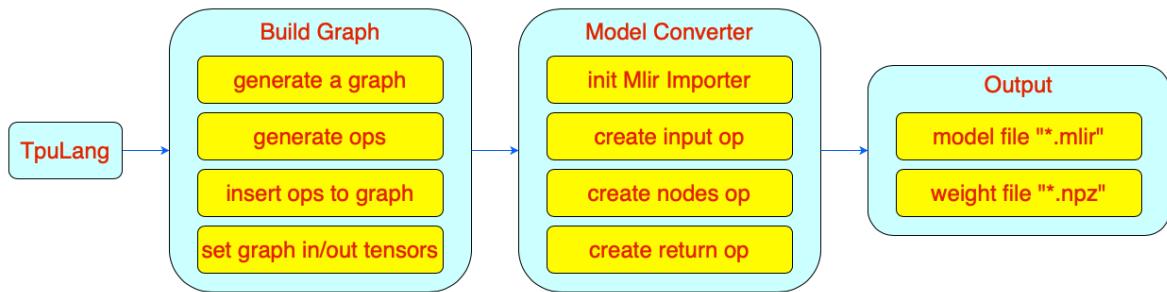


Fig. 16.1: TpuLang conversion process

**Supplementary Note:**

- The op interface requires:
  - The input tensor of the op (i.e., the output tensor of the previous operator or the graph input tensor and coeff);
  - attrs extracted from the interface. Attrs will be set by MLIRImporter as attributes corresponding to the ones defined in TopOps.td;
  - If the interface includes quantization parameters (i.e., scale and zero\_point), the tensor corresponding to this parameter needs to set (or check) the quantization parameters.
  - Return the output tensor(tensors) of the op.
- After all operators are inserted into the graph and the input/output tensors of the graph are set, the conversion to mlir text will start. This part is implemented by TpuLangConverter.
- The conversion process of TpuLang Converter is the same as onnx front-end part. Please refer to ([Front-end Conversion](#)).

### 16.3 Operator Conversion Example

This section takes the Conv operator as an example to convert a single Conv operator model to Top mlir

```
import numpy as np

def model_def(in_shape):
    tpul.init("BM1684X")
    in_shape = [1,3,173,141]
    k_shape =[64,1,7,7]
    x = tpul.Tensor(dtype='float32', shape=in_shape)
    weight_data = np.random.random(k_shape).astype(np.float32)
    weight = tpul.Tensor(dtype='float32', shape=k_shape, data=weight_data, is_
    ↵const=True)
    bias_data = np.random.random(k_shape[0]).astype(np.float32)
    bias = tpul.Tensor(dtype='float32', shape=k_shape[0], data=bias_data, is_
    ↵const=True)
    conv = tpul.conv(x, weight, bias=bias, stride=[2,2], pad=[0,0,1,1], out_dtype=
    ↵"float32")
    tpul.compile("model_def", inputs=[x],outputs=[conv], cmp=True)
    tpul.deinit()
```

Single Conv Model

The conversion process:

1. Interface definition

The conv interface is defined as follows:

```
def conv(input: Tensor,
         weight: Tensor,
         bias: Tensor = None,
         stride: List[int] = None,
         dilation: List[int] = None,
         pad: List[int] = None,
         group: int = 1,
         out_dtype: str = None,
         out_name: str = None):
    # pass
```

Parameter Description

- input: Tensor type, indicating the input Tensor with 4-dimensional NCHW format.
- weight: Tensor type, representing the convolution kernel Tensor with 4-dimensional [oc, ic, kh, kw] format. oc indicates the number of output channels, ic indicates the number of input channels, kh is kernel\_h, and kw is kernel\_w.
- bias: Tensor type, indicating the bias Tensor. There is no bias when it is None. Otherwise, the shape is required to be [1, oc, 1, 1].

- dilation: List[int], indicating the size of holes. None means dilation equals [1,1]. Otherwise, the length is required to be 2 and the order of List is [length, width].
- pad: List[int], indicating the padding size, if it is None, no padding is applied. Otherwise, the length is required to be 4. The order in the List is [Up, Down, Left, Right].
- stride: List[int], indicating the step size, [1,1] when it is None. Otherwise, the length is required to be 2 and the order in the List is [length, width].
- groups: int type, indicating the number of groups in the convolutional layer. If ic=oc=groups, the convolution is depthwise conv
- out\_dtype: string type or None, indicating the type of the output Tensor. When the input tensor type is float16/float32, None indicates that the output tensor type is consistent with the input. Otherwise, None means int32. Value range: /int32/uint32/float32/float16.
- out\_name: string type or None, indicating the name of the output Tensor. When it is None, the name will be automatically generated.

Define the Top.Conv operator in TopOps.td, the operator definition is as shown in the figure ([Conv Operator Definition](#))

## 2. Build Graph

- Initialize the model: create an empty Graph.
- Model input: Create input tensor x given shape and data type. A tensor name can also be specified here.
- conv interface:
  - Call the conv interface with specified input tensor and input parameters.
  - attributes, pack the input parameters into attributes defined by ([Conv Operator Definition](#))

```
attr = {
    "kernel_shape": ArrayAttr(weight.shape[2:]),
    "strides": ArrayAttr(stride),
    "dilations": ArrayAttr(dilation),
    "pads": ArrayAttr(pad),
    "do_relu": Attr(False, "bool"),
    "group": Attr(group)
}
```

- Define output tensor
  - Insert conv op. Insert Top.ConvOp into Graph.
  - return the output tensor
- Set the input of Graph and output tensors.
3. init\_MLIRImporter:

```
include > tpu_mlir > Dialect > Top > IR > ≡ TopOps.td
157  def Top_ConvOp: Top_Op<"Conv", [SupportFuseRelu]> {
158    let summary = "Convolution operator";
159
160    let description = [
161      In the simplest case, the output value of the layer with input size
162      .....
163    ];
164
165    let arguments = (ins
166      AnyTensor:$input,
167      AnyTensor:$filter,
168      AnyTensorOrNone:$bias,
169      I64ArrayAttr:$kernel_shape,
170      I64ArrayAttr:$strides,
171      I64ArrayAttr:$pads, // top,left,bottom,right
172      DefaultValuedAttr<I64Attr, "1">:$group,
173      OptionalAttr<I64ArrayAttr>:$dilations,
174      OptionalAttr<I64ArrayAttr>:$inserts,
175      DefaultValuedAttr<BoolAttr, "false">:$do_relu,
176      OptionalAttr<F64Attr>:$upper_limit,
177      StrAttr:$name
178    );
179
180    let results = (outs AnyTensor:$output);
181    let extraClassDeclaration = [
182      void parseParam(int64_t &n, int64_t &ic, int64_t &ih, int64_t &iw, int64_t &oc,
183      int64_t &oh, int64_t &ow, int64_t &g, int64_t &kh, int64_t &kw, int64_t &
184      ins_h,
185      int64_t &ins_w, int64_t &sh, int64_t &sw, int64_t &pt, int64_t &pb,
186      int64_t &pl,
187      int64_t &pr, int64_t &dh, int64_t &dw, bool &is_dw, bool &with_bias, bool &
188      do_relu,
189      float &relu_upper_limit);
190    ];
191  }
```

Fig. 16.2: Conv Operator Definition

Get the corresponding input\_shape and output\_shape from shapes according to input\_names and output\_names. Add model\_name, and generate the initial mlir text MLIRImporter.mlir\_module, as shown in the figure ([Initial mlir text](#)).

```
module attributes {module.chip = "ALL", module.name = "Conv2d", module.state = "TOP_F32", module.weight_file = "conv2d_top_f32_all_weight.npz"} {
  func.func @main(%arg0: tensor<1x16x100x100xf32>) -> tensor<1x32x100x100xf32> {
    %0 = "top.None"() : () -> none
  }
}
```

Fig. 16.3: Initial Mlir Text

#### 4. generate\_mlir

- Build input op, the generated Top.inputOp will be inserted into MLIRImporter.mlir\_module.
- Call Operation.create to create Top.ConvOp, and the parameters required by the create function are:
  - Input op: According to the interface definition, the inputs of the Conv operator include input, weight and bias. The inputOp has been created, and the op of weight and bias is created through getWeightOp().
  - output\_shape: get output shape from the output tensor stored in the Operator.
  - Attributes: Get attributes from Operator, and convert attributes to Attributes that can be recognized by MLIRImporter

After Top.ConvOp is created, it will be inserted into the mlir text

- Get the corresponding op from operands according to output\_names, create return\_op and insert it into the mlir text. By this point, the generated mlir text is as shown ([Full Mlir Text](#)).

```
#loc = loc(unknown)
module attributes {module.FLOPs = 109428480 : i64, module.chip = "ALL", module.name = "model_def", module.state = "TOP_F32", module.weight_file = "model_def_top_f32_all_weight.npz"} {
  func.func @main(%arg0: tensor<1x3x173x141xf32> loc(unknown)) -> tensor<1x64x84x69xf32> {
    %0 = "top.Input"(%arg0) : (tensor<1x3x173x141xf32>) -> tensor<1x3x173x141xf32> loc(#loc1)
    %1 = "top.Weight"() : () -> tensor<64x1x7x7xf32> loc(#loc2)
    %2 = "top.Weight"() : () -> tensor<64xf32> loc(#loc3)
    %3 = "top.Conv"(%0, %1, %2) {dilations = [1, 1], do_relu = false, group = 1 : i64, kernel_shape = [7, 7], pads = [0, 0, 1, 1], relu_l
imit = -1.00000e+00 : f64, strides = [2, 2]} : (tensor<1x3x173x141xf32>, tensor<64x1x7x7xf32>, tensor<64xf32>) -> tensor<1x64x84x69xf32>
    loc(#loc4)
    return %3 : tensor<1x64x84x69xf32> loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("BMTensor0")
#loc2 = loc("BMTensor1")
#loc3 = loc("BMTensor2")
#loc4 = loc("BMTensor3")
```

Fig. 16.4: Full Mlir Text

#### 5. Output

Save the mlir text as Conv\_origin.mlir and the weights in tensors as Conv\_TOP\_F32\_all\_weight.npz.

## 16.4 Tpulang API usage method

TpuLang is currently only applicable to the inference portion of inference frameworks. For static graph frameworks like TensorFlow, when integrating the network with TpuLang, users need to first initialize with `tpul.init('processor')` (where 'processor' can be BM1684X or BM1688). Next, prepare the tensors, use operators to build the network, and finally, call the `tpul.compile` interface to compile and generate bmodel. The detailed steps for each of these processes are explained below. You can find detailed information on various interfaces used (such as `tpul.init`, `deinit`, `Tensor`, and operator interfaces) in appx02 (Appendix 02: Basic Elements of TpuLang).

The following steps assume that the loading of the `tpu-mlir` release package has been completed.

### 16.4.1 Initialization

The specific definition of the initialization function can be found in the documentation under the section titled ([Initialization Function](#)).

```
import transform.TpuLang as tpul
import numpy as np

tpul.init('BM1684X')
```

### 16.4.2 Prepare the tensors

The specific definition of the initialization function can be found in the documentation under the section titled ([tensor](#))

```
shape = [1, 1, 28, 28]
x_data = np.random.randn(*shape).astype(np.float32)
x = tpul.Tensor(dtype='float32', shape=shape, data=x_data)
```

### 16.4.3 Build the graph

Continuing with the utilization of existing operators ([Operator](#)) and the tensors prepared earlier, here is a simple model construction example:

```
def conv_op(x,
            kshape,
            stride,
            pad=None,
            group=1,
            dilation=[1, 1],
            bias=False,
```

(continues on next page)

(continued from previous page)

```
        dtype="float32"):  
    oc = kshape[0]  
    weight_data = np.random.randn(*kshape).astype(np.float32)  
    weight = tpul.Tensor(dtype=dtype, shape=kshape, data=weight_data, ttype="coeff"  
    ↪")  
    bias_data = np.random.randn(oc).astype(np.float32)  
    bias = tpul.Tensor(dtype=dtype, shape=[oc], data=bias_data, ttype="coeff")  
    conv = tpul.conv(x,  
                     weight,  
                     bias=bias,  
                     stride=stride,  
                     pad=pad,  
                     dilation=dilation,  
                     group=group)  
    return conv  
  
def model_def(x):  
    conv0 = conv_op(x, kshape=[32, 1, 5, 5], stride=[1,1], pad=[2, 2, 2, 2], dtype=  
    ↪'float32')  
    relu1 = tpul.relu(conv0)  
    maxpool2 = tpul.maxpool(relu1, kernel=[2, 2], stride=[2, 2], pad=[0, 0, 0, 0])  
    conv3 = conv_op(maxpool2, kshape=[64, 32, 5, 5], stride=[1,1], pad=[2, 2, 2, 2],  
    ↪dtype='float32')  
    relu4 = tpul.relu(conv3)  
    maxpool5 = tpul.maxpool(relu4, kernel=[2, 2], stride=[2, 2], pad=[0, 0, 0, 0])  
    conv6 = conv_op(maxpool5, kshape=[1024, 64, 7, 7], stride=[1,1], dtype='float32')  
    relu7 = tpul.relu(conv6)  
    softmax8 = tpul.softmax(relu7, axis=1)  
    return softmax8  
  
y = model_def(x)
```

#### 16.4.4 compile

Call the tpul.compile function ([compile](#)). After compilation, you will get example\_f32.bmodel:

```
tpul.compile("example", [x], [y], mode="f32")
```

#### 16.4.5 deinit

The specific definition can be found in the documentation under the section titled ([Deinitialization Function](#))

```
tpul.deinit()
```

#### 16.4.6 deploy

Finally, use `model_deploy.py` to complete the model deployment. Refer to the documentation for specific usage instructions. ([model\\_deploy](#)).

# CHAPTER 17

---

## Custom Operators

---

### 17.1 Overview

TPU-MLIR already includes a rich library of operators that can fulfill the needs of most neural network models. However, in certain scenarios, there may be a requirement for users to define their own custom operators to perform computations on tensors. This need arises when:

1. TPU-MLIR does not support a specific operator, and it cannot be achieved by combining existing operators.
2. The operator is private.
3. Combining multiple operator APIs does not yield optimal computational performance, and custom operations at the TPU-Kernel level can improve execution efficiency.

The functionality of custom operators allows users to freely use the interfaces in TPU-Kernel to compute tensors on the TPU, and encapsulate this computation process as backend operators (refer to the TPU-KERNEL Technical Reference Manual for backend operator development). The backend operator calculation involves operations related to the global layer and local layer:

- a. The operator must implement the global layer. The input and output data of the global layer are stored in DDR. The data needs to be transferred from global memory to local memory for execution and then transferred back to global memory. The advantage is that local memory can be used flexibly, but it has the disadvantage of generating a considerable number of GDMA transfers, resulting in lower the Tensor Competing Processor utilization.
- b. The operator can optionally implement the local layer. The input and output data of the local layer are stored in local memory. It can be combined with other layers for

LayerGroup optimization, avoiding the need to transfer data to and from global memory during the calculation of this layer. The advantage is that it saves GDMA transfers and achieves higher computational efficiency. However, it is more complex to implement. The local memory needs to be allocated in advance during model deployment, limiting its usage and making it impractical for certain operators.

- c. The operator also need to implement some additional functions for correctness verification, shape inference and more complex local layer during the compilation phase.

The frontend can build models containing custom operators using tpulang or Caffe, and finally deploy the models through the model conversion interface of TPU-MLIR. This chapter primarily introduces the process of using custom operators in the TPU-MLIR release package.

## 17.2 Custom Operator Addition Process

Notice: in the following context, {op\_name} represent the name of operator, whose length is limited to 20. {processor\_arch} represents architecture of processor, whose optional values are BM1684X or BM1688.

### 17.2.1 Add TpuLang Custom Operator

1. Load TPU-MLIR

The following operations need to be in a Docker container. For the use of Docker, please refer to Setup Docker Container.

```
1 $ tar zxf tpu-mlir_xxxx.tar.gz  
2 $ source tpu-mlir_xxxx/envsetup.sh
```

envsetup.sh adds the following environment variables:

Table 17.1: Environment variables

Name	Value	Explanation
TPUC_ROOT	tpu-mlir_xxx	The location of the SDK package after decompression
MODEL_ZOO_PATH	\${TPUC_ROOT}/../model-zoo	The location of the model-zoo folder, at the same level as the SDK
REGRESSION_PATH	\${TPUC_ROOT}/regression	The location of the regression folder

envsetup.sh modifies the environment variables as follows:

```

1  export PATH=${TPUC_ROOT}/bin:$PATH
2  export PATH=${TPUC_ROOT}/python/tools:$PATH
3  export PATH=${TPUC_ROOT}/python/utils:$PATH
4  export PATH=${TPUC_ROOT}/python/test:$PATH
5  export PATH=${TPUC_ROOT}/python/samples:$PATH
6  export PATH=${TPUC_ROOT}/customlayer/python:$PATH
7  export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
8  export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
9  export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
10 export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
11 export REGRESSION_PATH=${TPUC_ROOT}/regression

```

2. Define the structure of parameter and parse function

- a. Define the structure of the operator parameters in \$TPUC\_ROOT/customlayer/include/backend\_custom\_param.h. This structure will be used in functions in subsequent steps. An example of the structure is as follows:

```

typedef struct {op_name}_param {
...
} {op_name}_param_t;

```

- b. One should implement corresponding functions to parse the parameters passed from the frontend of toolchain based on the parameters required by the operator. Parameters are passed through a pointer to a custom\_param\_t array. Starting from the second element of the array, a custom\_param\_t structure contains information about a parameter, and the parameter value is stored in the corresponding member variables in custom\_param\_t (which includes integer, floating-point number, integer array, and floating-point array variables). The order of the parameters is the same as the order in which the user provides them when calling the TpuLang interface. The definition of the custom\_param\_t is as follows:

```

typedef union {
    int int_t;
    float float_t;
    // max size of int and float array is set as 16
    int int_arr_t[16];
    float float_arr_t[16];
} custom_param_t;

```

An example of a parse function is as follows:

```

static {op_name}_param_t {op_name}_parse_param(const void* param) {
...
}

```

3. Plugins for compiler

In some cases, small modifications are needed for controlling the behaviour of

compiler for different type of custom operator with different parameters. Recently some Plugins are provided to realize the aims (please define them in file in directory ./plugin):

- a. [Required] Inference function. This plugin is used for comparation of output data between TOP and TPU dialect. The form of plugin function is as follows:

```
void inference_absadd(void* param, int param_size, const int (*input_shapes)[MAX_SHAPE_DIMS],
                      const int* input_dims, const float** inputs, float** outputs);
```

where input\_shapes and input\_dims is array of input shapes and dims respectively. inputs and output is pointer of data of inputs and outputs data outputs.

- b. [Optional] Shape infer function. This plugin is used for shape inference in TOP dialect. If not implmemnt, the default is that there is one input and one output while output shape is equal to input shape. The form of plugin function is as follows:

```
void shape_inference_absadd(void* param, int param_size, const int (*input_shapes)[MAX_SHAPE_DIMS],
                           const int* input_dims, int (*output_shapes)[MAX_SHAPE_DIMS], int* output_dims);
```

- c. [Optional] Force dynamic run. The shape of some operators changes dynamically (e.g., NonZero) and needs to be forced to run dynamically even under static compilation. The form of plugin function is as follows:

```
bool force_dynamic_run_{op_name}(void* param, int param_size);
```

Not provided this plugin, that custom operator if static by default.

- d. [Optional] Try to group with other operators. The form of plugin function is as follows:

```
bool local_gen_support_{op_name}(void* param, int param_size);
```

Not provided this plugin, the default is that custom operator corresponding to {op\_name} is not supported to group with other operators. Otherwise, one should implement the backend apis like api\_xxx\_local and `api\_xxx\_local\_bfsz` (optional) in cases when this plugin function returns true.

- e. [Optional] Try to split axis when group with other operators. The form of plugin function is as follows:

```
bool allow_data_split_{op_name}(void* param, int param_size, int axis, group_type_t group_type);
```

Not provided this plugin, the default is that when fusing with other operators, custom operator corresponding to {op\_name} is supported to try to split axis.

- f. [Optional] Backward slice derivation. Also applies to local layers (see the LayerGroup chapter for details). The form of plugin function is as follows:

```
bool backwardh_{op_name}(void* param, int param_size, int* in_idx, int* in_slice,
→ int out_idx, int out_slice);

bool backwardw_{op_name}(void* param, int param_size, int* in_idx, int* in_slice,
→ int out_idx, int out_slice);
```

where respectively in\_idx and in\_slice are pointers to index and size of slice of input tensor while out\_idx and out\_slice are index and size of slice of output tensor. Not provided this plugin, the default is that when fusing with other operators, out\_idx is equal to the value that in\_idx is pointing to and out\_slice is equal to the value that in\_slice is pointing to.

#### 4. Develop backend operators

We can develop backend operators based on TPU-Kernel(4.1) or based on ppl(4.2)

##### 4.1 Based on TPU-Kernel

Assuming the current path is \$TPUC\_ROOT/customlayer

- a. Declare the custom operator functions for the global layer and local layer in ./include/tpu\_impl\_custom\_ops.h

```
void tpu_impl_{op_name}_global // Required
void tpu_impl_{op_name}_local // Optional
```

- b. Add the tpu\_impl\_{op\_name}.c file in the ./src directory and invoke the TPU-Kernel interfaces to implement the corresponding functions.
- c. Add the interface\_{op\_name}.c file in the ./src directory and implement the backend api.

```
void api_{op_name}_global // Required. Calling void tpu_impl_{op_name}_global
void api_{op_name}_local // Optional. Calling void tpu_impl_{op_name}_local
```

##### 4.1 Based on ppl

Assuming the current path is \$TPUC\_ROOT/customlayer

- a. Add the {op\_name}.pl file in the ./PplBackend/src directory, where .pl is an implementation of the kernel function using ppl syntax.
- b. Add the {op\_name}\_tile.cpp file in the ./PplBackend/src directory and implement the tiling func and specifies the kernel implementation corresponding to the dtype.

```
// The kernelFunc definition is the same as the function name {op_name}.pl
using KernelFunc = int (*)(global_addr_t, global_addr_t,
```

(continues on next page)

(continued from previous page)

```

        float, int, int, int, int, int, bool);

int {op_name}_tiling/{op_name}(...) { // Required.
    KernelFunc func;
    if (dtype == SG_DTYPE_FP32) {
        func = {op_name}_f32;
    } else if (dtype == SG_DTYPE_FP16) {
        func = {op_name}_f16;
    } else if (dtype == SG_DTYPE_BFP16) {
        func = {op_name}_bf16;
    }
    ...
} else {
    assert(0 && "unsupported dtype");
}
// Optional. Tiling func
...
}

```

- c. Add the {op\_name}\_api.c file in the ./PplBackend/src directory and implement the backend api.

```

extern int {op_name}_tiling/{op_name}(...);           // Required.

void api_addconst_global/local(..., const void *param) { // Required.
    PARSE_PARAM({op_name}, {op_name}_param, param);
    {op_name}_tiling/{op_name}(...);
}

```

5. Define the operator's parameter structure and write the operator's general interface

Add the interface\_{op\_name}.c file in the ./src directory and implement the corresponding interfaces:

```

int64_t api_{op_name}_global_bfsz (Optional. Calculate global
buffer size)

int api_{op_name}_local_bfsz (Optional. Calculate local buffer size)

void type_infer_{op_name} (Optional. For dynamic run. Infer shape
and dtype of outputs from those of inputs)

void slice_infer_{op_name} (Optional. For dynamic run. Infer the
output slice from the input slice, if not implemented, there is only one
input and one output by default, and the output slice is the same as the
input slice)

```

6. Register the operator

In file register\_ops.cmake, add op name for registering your operator:

```
register_custom_op({op_name}) // 4.1 Based on TPU-Kernel
```

(continues on next page)

(continued from previous page)

```
// OR  
  
register_custom_ppl_op({op_name}) // 4.2 Based on ppl
```

Once local layer could be implemented, register it:

```
register_custom_local_op({op_name}) // 4.1 Based on TPU-Kernel  
  
// OR  
  
register_custom_ppl_local_op({op_name}) // 4.2 Based on ppl
```

Once global layer needs buffer, register it:

```
register_custom_global_bfsz({op_name})
```

Once local layer needs buffer, register it:

```
register_custom_local_bfsz({op_name})
```

## 7. Compile and install the dynamic library

Firstly, initialize your environment by running the shell command:

```
source $TPUC_ROOT/customlayer/envsetup.sh
```

Then compile the plugin for custom operators:

```
rebuild_custom_plugin
```

and compile the backend apis (target: libbackend\_custom.so):

```
rebuild_custom_backend
```

After that, compile the corresponding firmware according to the actual usage scenario (For dynamic run):

a. CMODEL mode (target: libfirmware\_custom\_xxx.so):

```
rebuild_custom_firmware_cmodel {processor_arch}
```

b. SoC mode (target: libxxx\_kernel\_module\_custom\_soc.so):

```
rebuild_custom_firmware_soc {processor_arch}
```

c. PCIe mode (target: libxxx\_kernel\_module\_custom\_PCIE.so, note that BM1688 does not support PCIe mode):

```
rebuild_custom_firmware_PCIE {processor_arch}
```

At this point we have completed the work on the backend part of the custom operator.

#### 8. Invoke TpuLang to build the model

Refer to the TPULang Interface section for instructions on how to use TpuLang.

TpuLang provides the TpuLang.custom interface to build custom operators in the frontend of toolchain (please ensure that the op\_name matches the name of the backend operator): Note that, params should be dictionary in python, whose key should be a string representing the name of parameter and value should be a integer or floating-point number, or a list of integer or floating-point number (the length of list should be no greater than 16). When building the neural network, the number and order of keys should keep the same for the same custom operator and for the same key, if its value is a list, the length should keep the same.

```
def custom(tensors_in: List[TpuLang.Tensor],
           op_name: str,
           out_dtypes: List[str],
           out_names: List[str] = None,
           params: dict = None)
           -> List[TpuLang.Tensor]
...
The custom op
Arguments:
    tensors_in: list of input tensors (including weight tensors).
    op_name: name of the custom operator.
    out_dtypes: list of data type of outputs.
    out_names: list of name of outputs.
    params: parameters of the custom op.

Return:
    tensors_out: list of output tensors.
...  
...
```

##### a. Define the tpulang interface of custom operator

For convenient, one could standardize custom operator in file \$TPUC\_ROOT/customlayer/python/my\_tpulang\_layer.py :

```
import transform.TpuLang as tpu

class xxx:
    @staticmethod
    def native(...):
        ...
        return ...
    @staticmethod
    def tpulang(inputs, ...):
        params = dict(...)
        outputs = tpu.custom(
            tensors_in=inputs,
```

(continues on next page)

(continued from previous page)

```
op_name={op_name},  
params=params,  
out_dtypes=...)  
return outputs
```

where native function is used to calculate the reference output data of custom layer. tpulang function constructs the custom layer using TpuLang.custom function.

#### b. Unit test

After defining the custom operator, one should test whether this interface is reliable. In the directory \$TPUC\_ROOT/customlayer/test\_if/unittest, create a python file named “test\_{op\_name}.py”. In this file, create a class, which is derived from class TestTPULangCustom and create test functions.

The shell command below would tries to automatically perform the unit tests:

```
run_custom_unittest {processor_arch}
```

#### 9. On-Processor test

When at least a dynamic subnet exists in the network, the firmware containing in bmodel might be not useful since shell command bmrt\_test does not work. In this case, one might need the following shell command to replace the old firmware with new one:

```
tpu_model --kernel_update xxx.bmodel libxxx_kernel_module_custom_soc.so #F  
↳ SoC mode  
  
tpu_model --kernel_update xxx.bmodel libxxx_kernel_module_custom_pcie.so  
↳ #PCIe mode
```

### 17.2.2 Add Caffe Custom Operator

#### 1. Defining custom operators in Caffe

To define custom operators in Caffe, you need to define a class in the file \$TPUC\_ROOT/customlayer/python/my\_caffe\_layer.py that inherits from caffe.Layer and override the setup, reshape, forward, and backward functions as needed.

#### 2. Implementing the frontend conversion function

Provided that the caffe layer type of custom operators is “Python”. One needs to implement a corresponding conversion function of MyCaffeConverter class defined in the file \$TPUC\_ROOT/customlayer/python/my\_converter.py.

After the definition, you can call my\_converter.py interface for Top MLIR conversion:

```
my_converter.py \
--model_name # the model name \
--model_def # .prototxt file \
--model_data # .caffemodel file \
--input_shapes # list of input shapes (e.g., [[1,2,3],[3,4,5]]) \
--mlir # output mlir file
```

The next steps are the same as compile and install steps in “Add TpuLang Custom Operator” section.

## 17.3 Custom Operator Example

This section assumes that the tpu-mlir release package has been loaded.

### 17.3.1 Example of TpuLang

This subsection provides a sample of swapchannel operator implementation and application through TpuLang interface.

#### 1. Parameter Parser

The definition of swapchannel\_param\_t in

`\${TPUC\_ROOT}/customlayer/include/backend\_custom\_param.h` is as follows:

```
typedef struct swapchannel_param {
    int order[3];
} swapchannel_param_t;
```

where the field order is corresponding to the frontend attribute named “order” .

Note that there is an array (here alias A) passing from compiler. Starting from the second element of the array, each element is corresponding to an frontend attribute of frontend. For convenient, file `\${TPUC\_ROOT}/customlayer/include/api\_common.h` provides a macro PARSE\_PARAM(swapchannel, sc\_param, param) to transform param` (pointer to array A) into `sc\_param` (pointer to backend custom param). One should define a parser function, whose parameter is a pointer of array B (constructing by dropping the first element from array A) in file `\${TPUC\_ROOT}/customlayer/include/param\_parser.h` and output type is `swapchannel\_parse\_param`.

```
static swapchannel_param_t swapchannel_parse_param(const void* param) {
    swapchannel_param_t sc_param = {0};
    for (int i = 0; i < 3; i++) {
```

(continues on next page)

(continued from previous page)

```

    sc_param.order[i] = ((custom_param_t *)param)[0].int_arr_t[i];
}
return sc_param;
}

```

## 2. Plugin Functions

In source file \${TPUC\_ROOT}/customlayer/plugin/plugin\_swapchannel.c:

```

#include <string.h>
#include <assert.h>
#include "param_parser.h"

void inference_swapchannel(void* param, int param_size, const int (*input_
    ↪shapes)[MAX_SHAPE_DIMS],
    const int* input_dims, const float** inputs, float** outputs) {
    PARSE_PARAM(swapchannel, sc_param, param);
    int in_num = 1;
    for (int i = 2; i < input_dims[0]; ++i) {
        in_num *= input_shapes[0][i];
    }
    int N = input_shapes[0][0];
    int C = input_shapes[0][1];
    assert(C == 3);
    for (int n = 0; n < N; ++n) {
        for (int c = 0; c < 3; ++c) {
            for (int x = 0; x < in_num; ++x) {
                memcpy(outputs[0] + n * C * in_num + sc_param.order[c] * in_num,
                    inputs[0] + n * C * in_num + c * in_num, in_num * sizeof(float));
            }
        }
    }
}

```

## 3. Backend Operator Implementation

The following is the declaration in the header file

`\${TPUC\_ROOT}/customlayer/include/backend\_swapchannel.h`:

```

void tpu_impl_swapchannel_global(
    global_addr_t input_global_addr,
    global_addr_t output_global_addr,
    const int *shape,
    const int *order,
    data_type_t dtype);

```

The code of `\${TPUC\_ROOT}/customlayer/src/tpu\_impl\_swapchannel.c`:

```
#include "tpu_impl_custom_ops.h"
```

(continues on next page)

(continued from previous page)

```

void tpu_impl_swapchannel_global(
    global_addr_t input_global_addr,
    global_addr_t output_global_addr,
    const int *shape,
    const int *order,
    data_type_t dtype)
{
    dim4 channel_shape = {.n = shape[0], .c = 1, .h = shape[2], .w = shape[3]};
    int data_size = tpu_data_type_size(dtype);
    int offset = channel_shape.w * channel_shape.h * data_size;
    for (int i = 0; i < 3; i++) {
        tpu_gdma_cpy_S2S(
            output_global_addr + i * offset,
            input_global_addr + order[i] * offset,
            &channel_shape,
            NULL,
            NULL,
            dtype);
    }
}

```

#### 4. Backend Interface

In file  `${TPUC_ROOT}/customlayer/src/interface_swapchannel.c`, one should define two functions: `void type_infer_swapchannel` and `void api_swapchannel_global`:

```

#include <string.h>
#include "tpu_utils.h"
#include "tpu_impl_custom_ops.h"
#include "param_parser.h"

// type infer function
void type_infer_swapchannel(
    const global_tensor_spec_t *input,
    global_tensor_spec_t *output,
    const void *param) {
    output->dtype = input->dtype;
    output->dims = input->dims;
    memcpy(output->shape, input->shape, output->dims);
    output->elem_num = input->elem_num;
}

// global api function
void api_swapchannel_global(
    const global_tensor_spec_t *input,
    global_tensor_spec_t *output,
    const void *param) {
    PARSE_PARAM(swapchannel, sc_param, param);
    tpu_impl_swapchannel_global(
        input->addr,

```

(continues on next page)

(continued from previous page)

```
    output->addr,  
    input->shape,  
    sc_param.order,  
    tpu_type_convert(input->dtype));  
}
```

## 5. Register the Custom Operator

Add the code in \${TPUC\_ROOT}/customlayer/register\_ops.cmake:

```
register_custom_op(swapchannel)
```

After completion, you can refer to the section “Add TpuLang Custom Operator” to compile and install dynamic libraries.

## 6. TpuLang Interface Invocation

In file \${TPUC\_ROOT}/customlayer/python/my\_tpulang\_layer.py, by using function TpuLang.custom, one could construct a custom operator named swapChannel, which has one input, one output, and an attribute whose value is an integer list of length 3:

```
import transform.TpuLang as tpul  
  
class swapChannel:  
    @staticmethod  
    def native(data):  
        return data[:, [2, 1, 0], :, :]  
    @staticmethod  
    def tpulang(inputs, dtype="float32"):  
        def shape_func(tensors_in:list):  
            return [tensors_in[0].shape]  
        params = {"order": [2, 1, 0]}  
        outs = tpul.custom(  
            tensors_in=inputs,  
            shape_func=shape_func,  
            # op_name should be consistent with the backend  
            op_name="swapchannel",  
            params=params,  
            out_dtypes=[dtype])  
        return outs
```

In file \${TPUC\_ROOT}/customlayer/test\_if/unittest/test\_swapchannel.py, one could create a unittest on custom operator named “swapChannel” :

```
import numpy as np  
import unittest  
from tpulang_custom_test_base import TestTPULangCustom  
import transform.TpuLang as tpul  
import my_tpulang_layer
```

(continues on next page)

(continued from previous page)

```

class TestSwapChannel(TestTPULangCustom):
    def __test(self, dtype):
        shape = [4, 32, 36, 36]
        self.data_in = np.random.random(shape).astype(dtype)
        x = tpul.Tensor(name="in", dtype=dtype, shape=shape, data=self.
        ↪data_in)
        y = my_tpulang_layer.swapChannel.tpulang(inputs=[x],
            dtype=dtype)[0]
        self.compile('SwapChannel', [x], [y], dtype)
    def test_fp32(self):
        self.__test('float32')
    def test_fp16(self):
        self.__test('float16')

if __name__ == '__main__':
    unittest.main()

```

### 17.3.2 Example of Caffe

This subsection provides application examples of custom operators absadd and ceiladd in Caffe.

#### 1. Defining Caffe custom operators

The definition of absadd and ceiladd in Caffe can be found in \$TPUC\_ROOT/customlayer/python/my\_caffe\_layer.py as follows:

```

import caffe
import numpy as np

# Define the custom layer
class AbsAdd(caffe.Layer):

    def setup(self, bottom, top):
        params = eval(self.param_str)
        self.b_val = params['b_val']

    def reshape(self, bottom, top):
        top[0].reshape(*bottom[0].data.shape)

    def forward(self, bottom, top):
        top[0].data [...] = np.abs(np.copy(bottom[0].data)) + self.b_val

    def backward(self, top, propagate_down, bottom):
        pass

class CeilAdd(caffe.Layer):

    def setup(self, bottom, top):

```

(continues on next page)

(continued from previous page)

```
params = eval(self.param_str)
self.b_val = params['b_val']

def reshape(self, bottom, top):
    top[0].reshape(*bottom[0].data.shape)

def forward(self, bottom, top):
    top[0].data[...] = np.ceil(np.copy(bottom[0].data)) + self.b_val

def backward(self, top, propagate_down, bottom):
    pass
```

The expression of corresponding operators in Caffe prototxt is as follows:

```
layer {
    name: "myabsadd"
    type: "Python"
    bottom: "input0_bn"
    top: "myabsadd"
    python_param {
        module: "my_caffe_layer"
        layer: "AbsAdd"
        param_str: "{ 'b_val': 1.2}"
    }
}

layer {
    name: "myceiladd"
    type: "Python"
    bottom: "input1_bn"
    top: "myceiladd"
    python_param {
        module: "my_caffe_layer"
        layer: "CeilAdd"
        param_str: "{ 'b_val': 1.5}"
    }
}
```

## 2. Implement operator front-end conversion functions

Define a convert\_python\_op function of the MyCaffeConverter class in \$TPUC\_ROOT/customlayer/python/my\_converter.py, the code is as follows:

```
def convert_python_op(self, layer):
    assert (self.layerType(layer) == "Python")
    in_op = self.getOperand(layer.bottom[0])
    p = layer.python_param

    dict_attr = dict(eval(p.param_str))
    params = dict_attr_convert(dict_attr)
```

(continues on next page)

(continued from previous page)

```
# p.layer.lower() to keep the consistency with the backend op name
attrs = {"name": p.layer.lower(), "params": params, 'loc': self.get_loc(layer.top[0])}

# The output shape compute based on reshape function in my_caffe_layer
out_shape = self.getShape(layer.top[0])
outs = top.CustomOp([self.mlir.get_tensor_type(out_shape)], [in_op],
                    attrs,
                    ip=self.mlir.insert_point).output
# add the op result to self.operands
self.addOperand(layer.top[0], outs[0])
```

### 3. Caffe front-end conversion

Complete the conversion of Caffe model in the \$TPUC\_ROOT/customlayer/test directory (i.e., my\_model.prototxt and my\_model.caffemodel, which contain absadd and ceiladd operators) by calling the my\_converter.py interface, the command is as follows:

```
my_converter.py \
--model_name caffe_test_net \
--model_def $TPUC_ROOT/customlayer/test/my_model.prototxt \
--model_data $TPUC_ROOT/customlayer/test/my_model.caffemodel \
--input_shapes [[1,3,14,14],[1,3,24,26]] \
--mlir caffe_test_net.mlir
```

So far, the Top MLIR file caffe\_test\_net.mlir has been obtained. For the subsequent model deployment process, please refer to the user interface chapter.

### 4. Backend operator and interface implementation

The implementation of absadd and ceiladd is similar to the swapchannel operator and can be found in \$TPUC\_ROOT/customlayer/include and \$TPUC\_ROOT/customlayer/src.

## 17.4 Custom AP(Application Processor) Operator Adding Process

### 17.4.1 TpuLang Custom AP Operator Adding

#### 1. Load TPU-MLIR

The process is consistent with when loading tpu-mlir for TPU custom operators.

#### 2. Write Processor Operator Implementation

Assuming you are currently in the \$TPUC\_ROOT/customlayer path, declare a custom derived class layer that inherits from the ap\_layer class in the header file ./include/custom\_ap/ap\_impl\_{op\_name}.h (where “forward()” declares the specific implementation method, “shape\_infer()” declares the method for inferring tensor shape changes before and after, “dtype\_infer()” declares the method

for inferring data type changes before and after, “get\_param()” declares the parameter parsing method). Also, add ap\_impl\_{op\_name}.cpp in the ./ap\_src directory, where you implement the corresponding functions, define new member variables, and override the member functions.

### 3. Register Custom Operator

- a. Add the operator’s name in ap\_impl\_{op\_name}.cpp to register the custom operator:

```
REGISTER_APLAYER_CLASS(AP_CUSTOM, {op_name});
```

- b. And define the member AP\_CUSTOM\_{OP\_NAME} in the enumeration type AP\_CUSTOM\_LAYER\_TYPE\_T in ./customlayer/include/custommap\_common.h, where OP\_NAME is uppercase.

```
typedef enum {
    AP_CUSTOM = 10001,
    AP_CUSTOM_TOPK = 10002,
    AP_CUSTOM_XXXX = 10003,
    AP_CUSTOM_LAYER_NUM ,
    AP_CUSTOM_LAYER_UNKNOW = AP_CUSTOM_LAYER_NUM,
} AP_CUSTOM_LAYER_TYPE_T;
```

- c. Define the instantiation method in customlayer/ap\_src/ap\_layer.cpp

```
bmap::ap_layer* create{OP_NAME}Layer() {
    return new bmap::ap_{op_name}layer();
}

void registerFactoryFunctions() {
    getFactoryMap()[std::string("{OP_NAME}")] = createTopkLayer;
    // Register other class creators
    // ...
}
```

### 4. Compiler Patch

Sometimes, it is necessary to modify the compiler to control the compilation behavior of different custom operators under different parameters, and this requires adding some patches. The following patch functions are currently supported (defined in the ./plugin folder):

- a. [Required] You need to implement the operator parameter parsing function yourself,

which is used to obtain the key parameters required by the operator, and override the get\_param() method of the custom layer:

```
int ap_mylayer::get_param(void *param, int param_size);
```

- b. [Required] Inference function, i.e., the C++ implementation of the operator. Override

the custom layer's forward() method:

```
int ap_mylayer::forward(void *raw_param, int param_size);
```

- c. [Optional] Shape inference function. This patch function is used for compiler shape

inference. If not implemented, by default, there is only one input and one output, and the output shape is the same as the input shape. The patch function is as follows:

```
int ap_mylayer::shape_infer(void *param, int param_size,
                           const vector<vector<int>> &input_shapes,
                           vector<vector<int>> &output_shapes);
```

Where input\_shapes/output\_shapes are arrays of input/output tensor shapes, and input\_dims/output\_dims are arrays of input/output tensor dimensions.

## 5. Compile and Install the Dynamic Library

First, initialize the environment:

```
source $TPUC_ROOT/customlayer/envsetup.sh
```

Then, complete the compilation of the patch (to obtain libplugin\_custom.so):

```
rebuild_custom_plugin
```

Compile the custom operator library file according to the processor architecture (to obtain libcustomapop.so). It is important to note that the environment for compiling the custom processor operator must be compatible with the glibc version in the bmodel runtime environment. The commands are as follows:

- a. x86 architecture

```
rebuild_custom_apop_x86
```

- b. ARM architecture

```
rebuild_custom_apop_aarch64
```

With this, we have completed the backend part of the custom processor operator.

## 6. Build Custom AP Operators with TpuLang

For how to use TpuLang, please refer to the TpuLang interface section.

TpuLang provides the TpuLang.custom interface which can also be used for custom processor operators. The method of use is basically the same as that

for custom TPU operators. The difference is that when defining the “TpuLang.custom” object, the “op\_name” parameter must start with the “ap.” prefix to distinguish it, for example, “ap.topk” :

```
class xxx:  
    @staticmethod  
    def native(...):  
        ...  
        return ...  
    @staticmethod  
    def tpulang(inputs, ...):  
        def shape_func(tensors_in:list, ...):  
            ...  
            return ...  
        params = dict(...)  
        outputs = tpul.lang.  
            custom(  
                tensors_in=inputs,  
                shape_func=shape_func,  
                op_name="ap.topk",  
                params=params,  
                out_dtypes=...)  
        return outputs
```

## 7. On-Processor Testing

When the network contains custom processor operators, the bmodel needs to include operator information. Use the command to write libcustomapop.so into the bmodel file, which is used for all host processor architectures:

```
tpu_model --custom_ap_update xxx.bmodel libcustomapop.so
```

Note: It is especially important that the environment for compiling the custom processor operator is compatible with the glibc version in the bmodel runtime environment.

## 17.5 Custom AP(Application Processor) Operator Example

This section assumes that the tpu-mlir release package has been loaded.

### 17.5.1 TpuLang Example

This subsection provides an example of a swapchannel operator implementation and its application through the TpuLang interface.

#### 1. Custom Operator Derived Class

Here, the field “order” corresponds to the “order” attribute on the frontend.

Define member variables in the custom class in {TPUC\_ROOT}/customlayer/ap\_src/ap\_impl\_{op\_name}.cpp:

```
private:
    int axis_;
    int K_;
```

Override the `get_param()` interface in the custom class in `{TPUC_ROOT}/customlayer/ap_src/ap_impl_{op_name}.cpp`. It is worth noting that what is passed from the compiler to the backend is an array A of `custom_param_t`, the first element of which is reserved, and from the second element onwards, each element corresponds to an attribute on the frontend:

```
int ap_toplayer::get_param(void *param, int param_size) {
    axis_ = ((custom_param_t *)param)[1].int_t;
    K_ = ((custom_param_t *)param)[2].int_t;
    return 0;
}
```

Override the `shape_infer()` interface in the custom class in `{TPUC_ROOT}/customlayer/ap_src/ap_impl_{op_name}.cpp`:

```
int ap_toplayer::shape_infer(const vector<vector<int>> &input_shapes,
                             vector<vector<int>> &output_shapes) {
    get_param(param, param_size);
    for (const auto& array : input_shapes) {
        output_shapes.emplace_back(array);
    }
    output_shapes[0][axis_] = std::min(K_, input_shapes[0][axis_]);
    return 0;
}
```

## 2. Processor Operator Implementation

Override the `forward()` interface in the custom class in `{TPUC_ROOT}/customlayer/ap_src/ap_impl_{op_name}.cpp`:

```
int ap_toplayer::forward(void *raw_param, int param_size) {
    // implementation code right here
    return 0;
}
```

## 3. Processor Operator Registration

- Add the operator's name in `ap_impl_{op_name}.cpp` to register the custom operator:

```
REGISTER_APLAYER_CLASS(AP_CUSTOM_TOPK, ap_topk);
```

- And define the member `AP_CUSTOM_TOPK` in the enumeration type `AP_CUSTOM_LAYER_TYPE_T` in `./customlayer/include/customap_common.h`.

```
typedef enum {
    AP_CUSTOM = 10001,
    AP_CUSTOM_TOPK = 10002,
    AP_CUSTOM_LAYER_NUM,
    AP_CUSTOM_LAYER_UNKNOW = AP_CUSTOM_LAYER_NUM,
} AP_CUSTOM_LAYER_TYPE_T;
```

- c. Define the instantiation method in customlayer/ap\_src/ap\_layer.cpp

```
bmap::ap_layer* createTopkLayer() {
    return new bmap::ap_topklayer();
}

void registerFactoryFunctions() {
    getFactoryMap()[std::string("TOPK")] = createTopkLayer;
    // Register other class creators
    // ...
}
```

#### 4. Frontend Preparation

The process of building a custom Processor operator using the TpuLang interface is basically the same as for a TPU custom operator. The difference is that when defining the “TpuLang.custom” object, the “op\_name” parameter must start with the “ap.” prefix to distinguish it, for example, “ap.topk”

# CHAPTER 18

---

## Implementing Backend Operators with PPL

---

PPL (Programming Language for TPUs) is a domain-specific programming language (DSL) based on C/C++ syntax extensions, designed for programming Tensor Processing Units (TPUs). This chapter demonstrates how to implement backend operators in PPL using the `add_const_fp` operator as an example, illustrating the compilation and utilization of PPL code within TPU-MLIR.

The implementation of PPL backend operators can be found in the `tpu-mlir/lib/PplBackend/src` directory. For release packages, it will be located in the `PplBackend/src` directory of the TPU-MLIR release package. For detailed instructions on writing PPL source code, refer to the documentation in `tpu-mlir/third_party/ppl/doc`.

### 18.1 How to Write and Call Backend Operators

Step 1: Implement Three Source Files

You need to create three source files: one for the device-side `pl` code, one for the host-side `cpp` code, and another for the host-side tiling function `cpp` code. . For the `add_const_fp` example, these files are:

- `add_const_fp.pl`: Implements the `add_const_f32` , `add_const_f16` and `add_const_bf16`, etc kernel interfaces.
- `add_const_fp_tile.cpp`: Implements the `add_tiling` function to call these kernel interfaces.
- `add_const_fp_api.cpp`: Implements the `api_add_const_fp_global` function to call these `add_tiling` interfaces.

**tiling.cpp File Example**

---

```

// Include the automatically generated header file from the pl file
#include "add_const_fp.h"
// Include the header file for MLIR data types and structures
#include "tpu_mlir/Backend/BM168x/Param.h"

// The entry function must be defined using extern "C"
extern "C" {
// If the pl file provides multiple operators, you can define function pointers in[F]
// advance.
// This can help reduce repetitive code. Note that the pointer type in the pl file
// needs to be defined using `gaddr_t`.
using KernelFunc = int (*)(gaddr_t, gaddr_t, float, int, int, int, int, bool);

// Add the entry function with user-defined input parameters
int add_tiling(gaddr_t ptr_dst, gaddr_t ptr_src, float rhs, int N, int C, int H,
               int W, bool relu, int dtype) {
KernelFunc func;
// Select the appropriate operator based on the input data type
if (dtype == DTTYPE_FP32) {
    func = add_const_f32;
} else if (dtype == DTTYPE_FP16) {
    func = add_const_f16;
} else if (dtype == DTTYPE_BFP16) {
    func = add_const_bf16;
} else {
    assert(0 && "unsupported dtype");
}

// Calculate the block size. Align the block size to `EU_NUM` to reduce memory[F]
// allocation failures.
// Since most of the memory on the TPU is aligned to `EU_NUM`, this alignment[F]
// will not affect memory allocation.
int block_w = align_up(N * C * H * W, EU_NUM);
int ret = -1;
while (block_w > 1) {
    ret = func(ptr_dst, ptr_src, rhs, N, C, H, W, block_w, relu);
    if (ret == 0) {
        return 0;
    } else if (ret == PplLocalAddrAssignErr) {
        // If the error type is `PplLocalAddrAssignErr`, it means the block size is too large,
        // and the local memory cannot accommodate it. The block size needs to be[F]
// reduced.
        block_w = align_up(block_w / 2, EU_NUM);
        continue;
    } else if (ret == PplL2AddrAssignErr) {
        // If the error type is `PplL2AddrAssignErr`, it means the block size is too large,
        // and the L2 memory cannot accommodate it. The block size needs to be reduced.
        // In this example, L2 memory is not allocated, so this error will not occur.
        assert(0);
    } else {
        // Other errors require debugging
}
}
}

```

(continues on next page)

(continued from previous page)

```

    assert(0);
    return ret;
}
}

return ret;
}
}

```

**Notes**

- The add\_const\_fp.h header file contains some error codes and chip-related parameter definitions.
- The pointers in the pl file need to be defined using the gaddr\_t type.

Table 18.1: Built-in Error Codes

Parameter Name	Description
PplLocalAddrAssignErr	Local memory allocation failed
FileErr	
LlvmFeErr	
PplFeErr	AST to IR conversion failed
PplOpt1Err	Optimization pass opt1 failed
PplOpt2Err	Optimization pass opt2 failed
PplFinalErr	Optimization pass final failed
PplTransErr	Code generation failed
EnvErr	Environment variable exception
PplL2AddrAssignErr	L2 memory allocation failed
PplShapeInferErr	Shape inference failed
PplSetMemRefShapeErr	
ToPplErr	
PplTensorConvErr	
PplDynBlockErr	

Table 18.2: Built-in Chip Parameters

Parameter Name	Description
EU_NUM	Number of EUs
LANE_NUM	Number of lanes

## Step 2: Call the Kernel Interface

In the function void tpu::AddConstOp::codegen\_global\_bm1684x() within lib/Dialect/Tpu/Interfaces/BM1684X/AddConst.cpp, call api\_add\_const\_fp\_global as follows:

```
BM168x::call_ppl_global_func("api_add_const_fp_global", &param,
    sizeof(param), input_spec->data(),
    output_spec->data());
```

If the operator supports local execution, implement `api_xxxxOp_local` and call it using `BM168x::call_ppl_local_func`.

```
BM168x::call_ppl_local_func("api_xxxx_local", &spec, sizeof(spec),
    &sec_info, input_spec->data(),
    output_spec->data());
```

This completes the implementation of the backend operator.

## 18.2 PPL Workflow in TPU-MLIR

1. Place the PPL compiler in the `third_party/ppl` directory and update it by referring to the `README.md` file in this directory.
2. Integrate the PPL source code compilation in `model_deploy.py`. The process is illustrated in the following diagram:

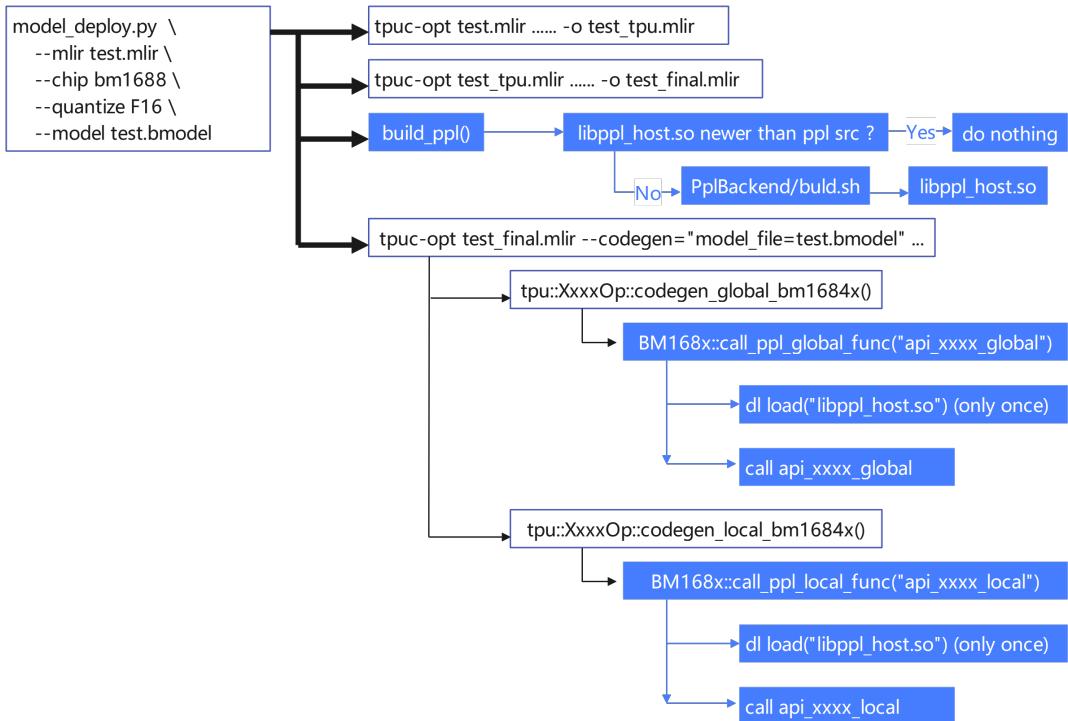


Fig. 18.1: PPL Workflow

# CHAPTER 19

---

## final.mlir Truncation Method

---

final.mlir, as the input file of codegen, is the final intermediate expression (IR) generated by the model after all hardware-independent and hardware-related optimizations. Because it contains hardware-related information, the structure is much more complicated than the previous IR.

When adapting the model, sometimes the MLIR file of the Tpu layer is inconsistent with the cmodel inference result of bmodel. In order to quickly locate the problem, in addition to using the bmodel\_checker.py tool to compare the output of each layer, you can also manually truncate the final.mlir file to generate a truncated model.

Therefore, this chapter will analyze the structure of final.mlir and explain how to truncate the model based on final.mlir to facilitate subsequent problem location.

- Recommended IDE: VSCode.
- Plugin: MLIR.

### 19.1 final.mlir structure introduction

The components of a single operator in final.mlir are as follows:

```
%out_value:out_num = "tpu.OpName"(%in0_value, %in1_value) {Attributes ...} :  
(tensor<in0_shapexin0_dtype, in0_addr : i64>, tensor<in1_shapexin1_dtype,  
in1_addr : i64>) -> (tensor<out0_shapexout0_dtype, out0_addr : i64>,  
tensor<out1_shapexout1_dtype, out1_addr : i64>, ...) loc(#loc2)
```

Fig. 19.1: final.mlir single operator example

Note:

- value represents the input/output of the operator in SSA form
- out\_num: represents the number of outputs. If it is a single-output operator, :out\_num will not be displayed.
- For the value of a multi-output operator, the user will refer to it in the %value#in\_index way (index starts from 0)
- Each input/output value has a corresponding Tensor type.
- A complete Tensor type contains shape, data type, and global memory address (Gmem addr).

In addition to the single operator, final.mlir also contains tpu.Group operators generated after LayerGroup, which contain multiple intermediate operators. These operators all complete calculations on Lmem. tpu.Group controls input data loading and output data storage through tpu.Load and tpu.Store, so the Tensor type of the intermediate operator does not have Gmem addr:

```
%out_value:out_num = "tpu.Group"(%in0_value, %in1_value) ({
  %12 = "tpu.Load" (%in0_value) {load info ...} (in0_type) -> %12_local_type loc(#loc32)
  ...
  %16 = "tpu.OpName"(%12) {Attributes ...} : (%12_local_type)-> (%16_local_type)
  ...
  %17 = "tpu.Store" (%16, %0) {store info ...} (%16_local_type) -> out0_type loc(#loc37)
  ...
  "Yield" (%17, ...) : (out0_type, out1_type ...) -> () loc(#920)
}) {GroupOp info ...} : (in0_type, in1_type) -> (out1_type, out2_type ...) loc(#loc920)
```

Fig. 19.2: tpu.Group example

- local\_type refers to Tensor type without Gmem addr.
- loc(#loc32) at the end of the operator refers to the location of a layer output of the model, that is, the number of the output. The corresponding output name can be found at the end of the final.mlir file according to the number.
- Yield represents the output set of tpu.Group.

The structure of the complete final.mlir file is as follows:

- The double-layer module contains mainfunc and subfunc, and mainfunc and subfunc have a calling relationship.
- arg0 in mainfunc refers to the input of the host side, so host\_in\_type does not have Gmem addr.
- The location of multiple outputs will be added to the end of the final.mlir file, and the inclusion relationship with each specific output location will be expressed, such as #loc950 = loc(fused[#loc2, #loc3]).

```

module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    func.func @main(%arg0: host_in_type) -> (out1_type, out2_type, out3_type) {
      %0 = "top.Input"(%arg0) {...} : (host_in_type) -> in_type loc(#loc1)
      %1:3 = call @subfunc_0(%0) : (in_type) -> (out1_type, out2_type, out3_type) loc(#loc)
      return %1#0, %1#1, %1#2 : out1_type, out2_type, out3_type loc(#loc)
    } loc(#loc)

    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out1_type, out2_type, out3_type)
  attributes {...} {
    ...
    return %1, %2, %3 : out1_type, out2_type, out3_type loc(#loc)
  } loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("layer_name_1")
#loc2 = loc("layer_name_2")
#loc3 = loc("layer_name_3")
...
#loc950 = loc(fused[#loc2, #loc3])
...

```

Fig. 19.3: final.mlir structure example

## 19.2 final.mlir Truncation Process

1. Modify subfunc. Delete the internal structure of subfunc, and match the return value value with the corresponding type:

```

module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    func.func @main(%arg0: host_in_type) -> (out1_type, out2_type, out3_type) {
      ...
    } loc(#loc)
    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out4_type) attributes {...} {
      ...
      Sync the out type
      return %1, %2, %3 : out1_type, out2_type, out3_type loc(#loc)
      return %4 : out4_type loc(#loc)
    } loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("layer_name_1")
...

```

Sync the out type

Remove the unneeded parts and change return values

Fig. 19.4: Truncation process Step1

2. Synchronize the calling method of subfunc in mainfunc (value and type):
3. Check whether bmodel is modified successfully. First, you can execute the codegen step to see if bmodel can be generated normally (please replace <...> with the actual file or parameter):

```

module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    Sync the out type
    func.func @main(%arg0: host_in_type) -> (out4_type) {
      %0 = "top.Input"(%arg0) {...} : (host_in_type) -> in_type loc(#loc1)
      %1:3 = call @subfunc_0(%0) : (in_type) -> (out4_type, out2_type, out3_type) loc(#loc)
      %1 = call @subfunc_0(%0) : (in_type) -> (out4_type) loc(#loc)
      return %1#0, %1#1, %1#2 : out1_type, out2_type, out3_type loc(#loc)
      return %1: out4_type loc(#loc) Update mainfunc according to the modified subfunc
    } loc(#loc)

    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out4_type) attributes {...} {
      ...
    } loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("layer_name_1")
...

```

Fig. 19.5: Truncation process Step2

```
$ tpu-opt <final.mlir> --codegen="model_file=<bmodel_file> embed_debug_info=<true/false>
→ model_version=latest" -o /dev/null
```

When profile is needed for performance analysis, embed\_debug\_info is set to true.

4. Use model\_tool to check whether the input and output information of the bmodel meets expectations:

```
$ model_tool --info <bmodel_file>
```

Note:

1. When truncating, the model structure is deleted in units of operators, and each tpu.Group should be regarded as an operator.
2. Modifying only the function return value without deleting the redundant model structure may cause the output result to be wrong. This is because each activated Gmem addr allocation will be reused according to the activation life cycle. Once the life cycle ends, it will be allocated to the next appropriate activation, causing the data at the address to be overwritten by subsequent operations.
3. It is necessary to ensure that each output of tpu.Group has user, otherwise the codegen step may report an error. If you do not want to output a certain result of tpu.Group and it is inconvenient to delete it completely, you can add a meaningless tpu.Reshape operator to the output without user, and match it with the same Gmem addr and location, for example:
4. After truncating the model, you can update the module.coeff\_size information in the module module to reduce the size of the bmodel generated after truncation. The formula is as follows:

$$\text{CoeffSize} = \text{NumElement}_{\text{weight}} * \text{DtypeBytes}_{\text{weight}} + \text{Addr}_{\text{weight}} - \text{CoeffAddr}$$

```
...
%2:2 = "tpu.Group"(%1) ({
  ...
  %10 = "tpu.Store"(%8, %0) {...} : (%8_local_type, none) -> out0_type loc(#loc2)
  %11 = "tpu.Store"(%9, %0) {...} : (%9_local_type, none) -> out1_type loc(#loc3)
  "tpu.Yield"(%10, %11) : (out0_type, out1_type) -> () loc(#loc950)
}) {...} : (%1_type) -> (out0_type, out1_type) loc(#loc950)

# only returns %2#0, so add a meaningless reshape as the user of %2#1
%3 = "tpu.Reshape"(%2#1) {shape = []} : (out1_type) -> out1_type loc(#loc3)
return %2#0 : out0_type loc(#loc)
...
}
```

Fig. 19.6: reshape example

In the above formula, weight refers to the last top.Weight in final.mlir after truncation. neuron (i.e., activation) is not recommended to modify because the address will be reused.

# CHAPTER 20

---

## Superior MaskRCNN Interface Guidance

---

### 20.1 MaskRCNN Basic

A two-stage MaskRCNN is comprised of two parts:

- **3 weighted blocks:** includes backbone.pt and 2 bbox/mask intermediate layers (namely torch\_bbox/mask.pt ).
- **5 dynamic non-weight blocks:** includes RPN head, bbox pooler, bbox head, mask pooler and mask head.

Thus, the MaskRCNN is expressed by the following procedures:

- **bbox detector:** backbone.pt => RPN head => bbox pooler => torch\_bbox.pt => bbox head.
- **mask detector:** backbone.pt => RPN head => mask pooler => torch\_mask.pt => mask head.

#### 20.1.1 Fast Block Segmentation Methods

Due to compatibility issues between MaskRCNN and the original framework, users may be unable to trace each part. This chapter uses the mask head as an example of a part that cannot be traced.

The segmentation points for two types of MaskRCNN blocks, are precisely the first entry points to the first layer of next weight blocks, when searching the MaskRCNN graph topologically.

## 20.2 Superior MaskRCNN

As the fine-grained operation-based deployment towards cloning MaskRCNN encounters challenges with high complexity in dynamic ir transformation, the following superior MaskRCNN solution is proposed:

**Coarse-grained:**

1. **Built-In MaskRCNN-Exclusive Backend:** now mlir-backend directly supports dynamic non-weight blocks, currently including the RPN head, box head, bbox pooler, and mask pooler. Thus most heavy workloads related to frontend inference graph parser and optimization are saved. This allows for the avoidance of numerous dynamic shape inference or variant Op support.
2. **Model Reconstruction:** users only need 4 structural information to reconstruct the complete MaskRCNN:
  - **io\_map:** describes the blocks' interfaces and always maintains the same topology as MaskRCNN. Defined as  $\{(destination\_block\_id, operand\_id):(source\_block\_id, operand\_id)\}$ .
  - **Backbone:** from top to RPN typically, split from the original MaskRCNN in advance.
  - **Weighted blocks:** bbox/mask intermediate layers, split from the original MaskRCNN in advance.
  - **config.yaml:** a yaml file to store hyper-parameters, provided in advance.

## 20.3 Quick Start

Before dive into new MaskRCNN features, please first explore the new yaml file and new unit tests for MaskRCNN.

### 20.3.1 Prepare Your Yaml

A default yaml is prepared at `regression/dataset/MaskRCNN/CONFIG_MaskRCNN.yaml`, whose struct is:

- **Compile Parameters for model\_transform:** structural infomations to reconstruct MaskRCNN.
  - **io\_map:** defined as  $\{(destination\_block\_id, operand\_id):(source\_block\_id, operand\_id)\}$ ; here -1 represents the complete model's top inputs, -2 represents the complete model's top outputs, and 0, 1, 2... represents the id of MaskRCNN blocks.  
For example,  $\{(0,0):(-1,0),(1,0):(0,0),(-2,0):(1,0)\}$  means block[0]'s input[0] comes from input[0] of the complete model, block[1]'s input[0] comes from block[0]'s output[0], and output[0] of the complete model comes from block[1]'s output[0].

- **maskrcnn\_output\_num**: number of final output operands for the complete MaskRCNN.
  - **maskrcnn\_structure**: the list describing the order of MaskRCNN blocks. 1 means a torch.pt model, while 0 means PPLOp. For example, [1, 0, 1] means the 1st launches a torch model, the 2nd launches a PPLOp, and the 3rd launches another torch model.
  - **maskrcnn\_ppl\_op**: names of MaskRCNN operands implemented by PPL at the backend.
  - **numPPLOp\_InWithoutWeight\_MaskRCNN**: number of input operands for each PPLOp; remember not to count weight operands.
- **Hyper-parameters for MaskRCNN**: necessary MaskRCNN parameters, decided by the original MaskRCNN framework.

### 20.3.2 Block Unit Test

Use --case to test 4 dynamic non-weight blocks: RPN head, bbox pooler, bbox head, mask pooler.

More guidance will be found at `python/test/test_MaskRCNN.py`.

```
$ test_MaskRCNN.py --case MaskRCNN_Utest_RPNGetBboxes --debug
```

## 20.4 New Frontend Interface API

### 20.4.1 [Step 1] Run `model_transform`

Used to convert MaskRCNN into MLIR files.

- **Skip Inference**: please be aware that no input/reference data .npz files are needed at this step, but a config.yaml is required in advance.
- **Skip Preprocess**: please note that in this step, no preprocessing is applied by default.
- **New Enable Flag**: please remember enable\_maskrcnn.

```
$ model_transform.py \
--model_def backbone.pt \
--model_extern torch_bbox.pt,torch_mask.pt \
--model_name MaskRCNN \
--input_shapes [[1,3,800,1216],[1,1,4741,4],[1,1,20000,4],[1,1,20000,4],[1,1,100,4]] \
--mlir MaskRCNN.mlir \
--enable_maskrcnn \
--path_yaml regression/dataset/MaskRCNN/CONFIG_MaskRCNN.yaml
```

### 20.4.2 [Step 2] Generate Input Data

#### MaskRCNN Input Format

The MaskRCNN implemented by the proposed method requires 5 inputs:

- **preprocessed image**: image after preprocessing.
- **max\_shape\_RPN/max\_shape\_GetBboxB**: if input image is resized to shape S1 and original shape is S0 ,then max shape is  $\text{int}(S0 * S1 / S0)$ , and expanded to a constant weight tensor.
- **scale\_factor\_GetBboxB/scale\_factor\_MaskPooler**: if input image is resized to shape S1 and original shape is S0 ,then scale factor is  $\text{float}(S1 / S0)$ , and expanded to a constant weight tensor.

#### Input Formats Reorganizing Tools

A tool is offered at tpu-mlir/python/tools/tool\_maskrcnn.py to assist you in generating data satisfied with the above requirements.

- **Skip Preprocess**: input image shoud be after preprocess, as preprocess procedure for MaskRCNN is usually complex and relies on specific functions from original framework.

Besides path\_yaml, 3 more parameters need to be specifc:

- **path\_input\_image**: image after preprocessing, saved as npz.
- **basic\_max\_shape\_inverse**: the height and width after preprocessing.
- **basic\_scalar\_factor**: precisely the above  $\text{float}(S1 / S0)$ , basic\_max\_shape\_inverse divide orginal shape reordered in height, width.

The result data will be stored at same path of path\_input\_image, but suffixed by Superior-MaskRCNNInputPreprocessed.

Please explore tool\_maskrcnn.py for more guidance.

```
$ tool_maskrcnn.py \
--path_yaml      ./regression/dataset/MaskRCNN/CONFIG_MaskRCNN.yaml \
--path_input_image    ./regression/dataset/MaskRCNN/Superior_IMG_BackBone.npz \
--basic_max_shape_inverse 1216,800 \
--basic_scalar_factor   1.8734375,1.8735363 \
--debug
```

### 20.4.3 [Step 3] Run model\_deploy

- **Inference Skip:** quant compare and simulation compare are skipped here.
- **Mandatory Parameters:** --quantize mode is forced to be F32 and --processor is forced to be BM1684X.
- **New Enable Flag:** please remember enable\_maskrcnn.

```
$ model_deploy.py \
  --mlir MaskRCNN.mlir \
  --quantize F32 \
  --processor BM1684X \
  --model MaskRCNN.bmodel \
  --debug \
  --enable_maskrcnn
```

## 20.5 IO\_MAP Guidance

Manually generate io\_map in two steps:

- **Well-SuppliedDefinition of Block Interfaces:** precisely collect input and output operand shapes, and block connection patterns.
- **Create Corresponding io\_map:** it should precisely and uniquely reconstruct the complete MaskRCNN.

### 20.5.1 [Step-1] Describe Block Interface

A complete MaskRCNN is truncated into multiple blocks as discussed above.

Please describe following information for each block:

- **Input:** input operands or constant weights
- **shapes:** in 4-dims format.
- **dtypes:** only support fp32 or int32.
- **connections:** the corresponding output of the upper block which each input is sourced from and specifying which operand it originates from.

Note that -1 represents the inputs of the complete MaskRCNN, while -2 the outputs of the complete model. And bs represents batch\_size.

#### [1] Top\_In

In- put Num- ber	Name	Shape	Dtype
Input 0)	'img.1'	[1,3,800,1216]	
Input 1)	'max_shape_R'	[bs,1,max_filter_num,4]	int32
Input 2)	'max_shape_G'	[1,bs*20000,1,4]	int32
Input 3)	'scale_factor_C'	[1, bs,20000,4]	FP32
Input 4)	'scale_factor_M'	[bs,1,roi_slice,4]	FP32

**[Torch] SubBlock-0: BackBone.pt**

IO-Type	Name	Shape	Dtype	Connection Info[From]
Input 0)	'img.1'	[1,3,800,1216]	FP32	[TOP_IN]Input-0
Output 0)	'11'	[1,256,200,304]	FP32	
Output 1)	'12'	[1,256,100,152]	FP32	
Output 2)	'13'	[1,256,50,76]	FP32	
Output 3)	'16'	[1,256,25,38]	FP32	
Output 4)	'15'	[1,256,13,19]	FP32	
Output 5)	'18'	[1,3,200,304]	FP32	
Output 6)	'19'	[1,3,100,152]	FP32	
Output 7)	'20'	[1,3,50,76]	FP32	
Output 8)	'21'	[1,3,25,38]	FP32	
Output 9)	'22'	[1,3,13,19]	FP32	
Output 10)	'23'	[1,12,200,304]	FP32	
Output 11)	'24'	[1,12,100,152]	FP32	
Output 12)	'25'	[1,12,50,76]	FP32	
Output 13)	'26'	[1,12,25,38]	FP32	
Output 14)	'27'	[1,12,13,19]	FP32	

**[PPL] SubBlock-1: ppl::RPN\_get\_bboxes**

IO-Type	Name	Shape	Connection Info[From]
Out-put	0 result_list	[bs,1,max_per_img,num_levels]	
Input	1 cls_scores_0	[bs,3,200,304]	[Torch][SubBlock-0]Output(5)
Input	2 cls_scores_1	[bs,3,100,152]	[Torch][SubBlock-0]Output(6)
Input	3 cls_scores_2	[bs,3,50,76]	[Torch][SubBlock-0]Output(7)
Input	4 cls_scores_3	[bs,3,25,38]	[Torch][SubBlock-0]Output(8)
Input	5 cls_scores_4	[bs,3,13,19]	[Torch][SubBlock-0]Output(9)
Input	6 bbox_preds_0	[bs,12,200,304]	[Torch][SubBlock-0]Output(10)
Input	7 bbox_preds_1	[bs,12,100,152]	[Torch][SubBlock-0]Output(11)
Input	8 bbox_preds_2	[bs,12,50,76]	[Torch][SubBlock-0]Output(12)
Input	9 bbox_preds_3	[bs,12,25,38]	[Torch][SubBlock-0]Output(13)
Input	10 bbox_preds_4	[bs,12,13,19]	[Torch][SubBlock-0]Output(14)
Input	11 max_shape	[bs,1,max_filter_num,5]	[TOP_IN]Input-1
Input	12 mlvl_anchors_0	[bs,1,3*200*304,5]	[mlir][Weight]
Input	13 mlvl_anchors_1	[bs,1,3*100*152,5]	[mlir][Weight]
Input	14 mlvl_anchors_2	[bs,1,3*50*76,5]	[mlir][Weight]
Input	15 mlvl_anchors_3	[bs,1,3*25*38,5]	[mlir][Weight]
Input	16 mlvl_anchors_4	[bs,1,3*13*19,5]	[mlir][Weight]

**[PPL] SubBlock-2: ppl::Bbox\_Pooler**

IO-Type	Name	Shape	Connection Info[From]
Output	0 result_res	[bs*250,256,PH,PW]	
Output	1 result_rois	[bs,max_per_img,1,roi_len]	
Input	2 feat0	{bs,256,H,W}	[Torch][SubBlock-0]Output 0)
Input	3 feat1	[bs,256,H/2,W/2]	[Torch][SubBlock-0]Output 1)
Input	4 feat2	[bs,256,H/4,W/4]	[Torch][SubBlock-0]Output 2)
Input	5 feat3	[bs,256,H/8,W/8]	[Torch][SubBlock-0]Output 3)
Input	6 rois_multi_batch	[bs,roi_slice,1,roi_len]	[PPL][SubBlock-1]result_list

### [Torch] SubBlock-3: torch\_bbox.pt

Batch	IO-Type	Name	Shape	Dtype	Connection Info[From]
Batch-1	Input	0	[250,256,7,7]	FP32	[PPL][SubBlock-2]result_res
	Output	0	[250,81]	FP32	
	Output	1	[250,320]	FP32	

### [PPL] SubBlock-4: ppl::get\_bboxes\_B

Batch	IO-Type	Name	Shape	Connection Info[From]
Batch 1	Output	re-sult_det_bboxes	[bs,1,100,5]	
	Output	result_det_labels	[bs,1,100,1]	
	Input	rois	[1,bs*250,1,5]	[PPL][SubBlock-2]1-result_rois
	Input	bbox_pred	[1,bs*250,1,320]	[Torch][SubBlock-3]Output 1
	Input	cls_score	[1,bs*250,1,81]	[Torch][SubBlock-3]Output 0
	Input	max_val	[1,bs*20000,1,4]	[TOP_IN]Input-2
	Input	scale_factor	[1,bs,20000,4]	[TOP_IN]Input-3

### [PPL] SubBlock-5: ppl::Mask\_Pooler

IO-Type	In-index	Name	Shape	Connection Info[From]
Out-put	0	result_res	[roi_num,C,PH,PV]	
Input	1	x0	[bs,256,H,W]	[Torch][SubBlock-0]Output 0
Input	2	x1	[bs,C,H/2,W/2]	[Torch][SubBlock-0]Output 1
Input	3	x2	[bs,C,H/4,W/4]	[Torch][SubBlock-0]Output 2
Input	4	x3	[bs,C,H/8,W/8]	[Torch][SubBlock-0]Output 3
Input	5	det_bboxes_multi_b	[bs,1,roi_slice,roi_]	[PPL][SubBlock-4]0-
				result_det_bboxes
Input	6	det_labels_multi_ba	[bs,1,roi_slice,1]	[PPL][SubBlock-4]1-
				result_det_labels
Input	7	scale_factor	[bs,1,roi_slice,4]	[TOP_IN]Input-4

### [Torch] SubBlock-6: torch\_mask.pt

Batch	IO-Type	In-index	Name	Shape	Dtype	Connection Info[From]
Batch-1	Input	0	in-put.2	[100,256,14,14]	FP32	[PPL][SubBlock-5]0-result_res
	Output	0	75	[100,80,28,28]	FP32	
Batch-4	Input	0	in-put.2	[400,256,14,14]	FP32	
	Output	0	75	[400,80,28,28]	FP32	

### [2] TSubBlock-7: TOP\_OUT

IO-Type	Index	Shape	Dtype	Connection Info[From]
Output	0	[bs,1,100,5]	FP32	[PPL][SubBlock-5]0-result_det_bboxes
Output	1	[bs,1,100,1]	FP32	[PPL][SubBlock-5]1-result_det_labels
Output	2	[100,80,28,28]	FP32	[Torch][SubBlock-6]

#### 20.5.2 [Step-2] Describe IO\_MAP

Reorganize above block interfaces in the following format:

- **Block Name:** Name and serial index of one block.
- **Inputs:** the corresponding output of the upper block from which each input is sourced and specifying which operand it originates from.
- **Connections:** record total amounts of input operands

- **Mapping:** use the definition, (destination\_block\_index, operand\_index):(source\_block\_index:operand\_index)

Note that -1 represents the inputs of the complete MaskRCNN, while -2 the outputs of the complete model.

### [0]TORCH\_0-rpn

- **Inputs:**
  - $\leftarrow [-1]\text{TOP\_IN}[0]$
- **Connections:** 1
- **Mapping:**
  - (0,0):(-1,0)

### [1]PPL-RPNGetBboxes

- **Inputs:**
  - $\leftarrow [0]\text{TORCH\_0-rpn}[5:15]$
  - $\leftarrow [-1]\text{TOP\_IN}[1]$
- **Connections:** 10
- **Mapping:**
  - (1,0):(0,5)
  - (1,1):(0,6)
  - (1,2):(0,7)
  - (1,3):(0,8)
  - (1,4):(0,9)
  - (1,5):(0,10)
  - (1,6):(0,11)
  - (1,7):(0,12)
  - (1,8):(0,13)
  - (1,9):(0,14)
  - (1,10):(-1,1)

### [2]PPL-Bbox\_Pooler

- **Inputs:**
  - $\leftarrow [0]\text{TORCH\_0-rpn}[0:4]$
  - $\leftarrow [1]\text{PPL-RPNGetBboxes}[0]$
- **Connections:** 4 + 1

- **Mapping:**
  - (2,0):(0,0)
  - (2,1):(0,1)
  - (2,2):(0,2)
  - (2,3):(0,3)
  - (2,4):(1,0)

### [3]Torch-2

- **Inputs:**
  - $\leftarrow [2]\text{PPL-Bbox\_Pooler}$
- **Connections:** 1
- **Mapping:**
  - (3,0):(2,0)

### [4]PPL-GetBboxB

- **Inputs:**
  - $\leftarrow [2]\text{PPL-Bbox\_Pooler}[1]$
  - $\leftarrow [3]\text{Torch-2}[0:2]_{\text{inverse}}$
  - $\leftarrow [-1]\text{TOP\_IN}[2:4]$
- **Connections:** 1 + 2 (inverse) + 2
- **Mapping:**
  - (4,0):(2,1)
  - (4,1):(3,1)
  - (4,2):(3,0)
  - (4,3):(-1,2)
  - (4,4):(-1,3)

### [5]ppl-MaskPooler

- **Inputs:**
  - $\leftarrow [0]\text{Torch-RPN}[0:4]$
  - $\leftarrow [4]\text{PPL-GetBboxB}[0:2]$
  - $\leftarrow [-1]\text{TOP\_IN}[4]$
- **Connections:** 4 + 2
- **Mapping:**

- (5,0):(0,0)
- (5,1):(0,1)
- (5,2):(0,2)
- (5,3):(0,3)
- (5,4):(4,0)
- (5,5):(4,1)
- (5,6):(-1,4)

### [6]Torch-3

- **Inputs:**  $\leftarrow [5]\text{ppl-MaskPooler}$
- **Connections:** 1
- **Mapping:** (6,0):(5,0)

### [2]TOP\_OUT

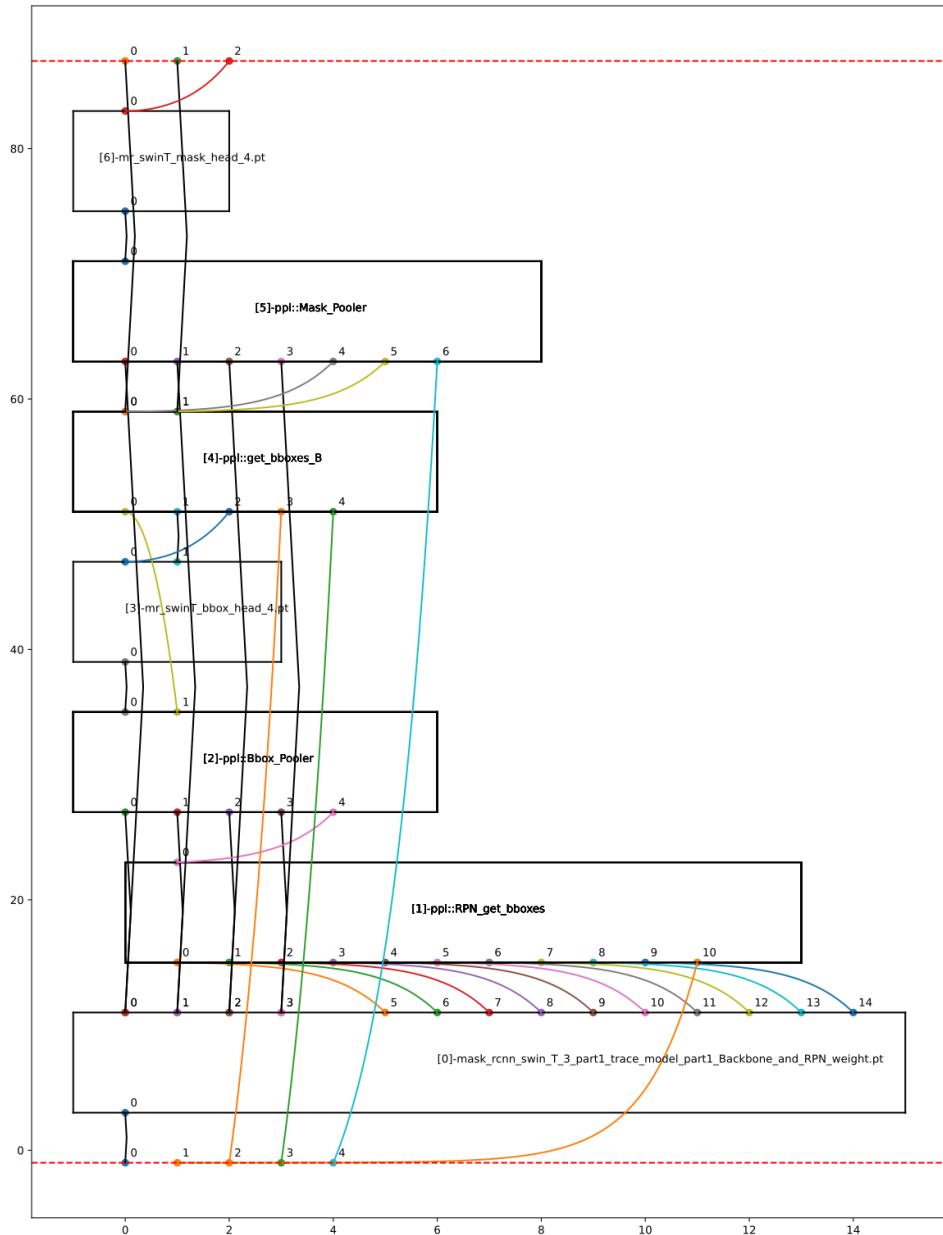
- **Inputs:**
  - $\leftarrow [4]\text{PPL-GetBboxB}[0:2]$
  - $\leftarrow [6]\text{Torch-3}$
- **Connections:** 2 + 11
- **Mapping:**
  - (-2,0):(4,0)
  - (-2,1):(4,1)
  - (-2,2):(6,0)

## 20.6 Generate IO\_MAP

After collecting all mapping information above, an io\_map dictionary is generated:

- **io\_map:** {(0,0):(-1,0),(1,0):(0,5),(1,1):(0,6),(1,2):(0,7),(1,3):(0,8),(1,4):(0,9),(1,5):(0,10),(1,6):(0,11),(1,7):1,1),(2,0):(0,0),(2,1):(0,1),(2,2):(0,2),(2,3):(0,3),(2,4):(1,0),(3,0):(2,0),(4,0):(2,1),(4,1):(3,1),(4,2):(3,0),(4,1,2),(4,4):(-1,3),(5,0):(0,0),(5,1):(0,1),(5,2):(0,2),(5,3):(0,3),(5,4):(4,0),(5,5):(4,1),(5,6):(-1,4),(6,0):(5,0),(-2,0):(4,0),(-2,1):(4,1),(-2,2):(6,0)}

Now directly use it at model\_transform, which will further dump a revised\_io\_map\_\${model\_name}.svg image to assist you in visualizing the io\_map.



## 20.7 mAP Inference

Transform and deploy such a coarse-grained MaskRCNN is not enough, to apply the mAP inference on the COCO2017 dataset, a careful intersection into the original framework is required.

Please refer to our model-zoo project for more inference details.

# CHAPTER 21

---

## LLMC Guidance

---

### 21.1 TPU-MLIR weight-only quantization

TPU-MLIR supports weight-only quantization for large models, utilizing the RTN (round to nearest) quantization algorithm with a quantization granularity of per-channel or per-group. The specific quantization configurations are as follows:

Table 21.1: weight-only quantization parameters

bit	symmetric	granularity	group_size
4	False	per-channel or per-group	-1 or 64(default)
8	True	per-channel	-1

The RTN quantization algorithm is straightforward and efficient, but it also has some limitations. In scenarios that require higher model accuracy, models quantized using the RTN algorithm may not meet the precision requirements. In such cases, it is necessary to utilize the large model quantization tool llmc\_tpu to further enhance accuracy.

## 21.2 llmc\_tpu

This project originates from [ModelTC/llmc](https://github.com/ModelTC/llmc). ModelTC/llmc is an excellent project specifically designed for compressing Large Language Models (LLMs). It leverages state-of-the-art compression algorithms to enhance efficiency and reduce model size without compromising prediction accuracy. If you want to learn more about the llmc project, please visit <https://github.com/ModelTC/llmc>.

This project is based on ModelTC/llmc with some customized modifications to support the Sophgo processor.

### 21.2.1 Environment Setup

#### 1. Download This Project

```
1 git clone git@github.com:sophgo/llmc-tpu.git
```

#### 2. Prepare the LLM or VLM Model for Quantization, Place the model you need to quantize in the same-level directory as llmc-tpu

For Example: Download Qwen2-VL-2B-Instruct from Huggingface

```
1 git lfs install  
2 git clone git@hf.co:Qwen/Qwen2-VL-2B-Instruct
```

#### 3. Download Docker and Set Up a Docker Container

pull docker images

```
1 docker pull registry.cn-hangzhou.aliyuncs.com/yongyang/llmcompression:pure-latest
```

create container. llmc\_test is just a name, and you can set your own name

```
1 docker run --privileged --name llmc_test -it --shm-size 64G --gpus all -v $PWD:/workspace F  
→registry.cn-hangzhou.aliyuncs.com/yongyang/llmcompression:pure-latest
```

#### 4. Enter llmc-tpu Directory and Install Dependencies

Note that you are already in a Docker container.

```
1 cd /workspace/llmc-tpu  
2 pip3 install -r requirements.txt
```

### 21.2.2 tpu Directory

```
|---- README.md |---- data |----LLM |----cali #Calibration Dataset |
|----eval #Eval Dataset |----VLM |----cali |----eval |---- config |
|----LLM #LLM quant config |---- Awq.yml #Awq config |---- GPTQ.yml
#GPTQ config |----VLM #VLM quant config |---- Awq.yml #Awq config |
example.yml #Quantization Parameters Reference Example |---- lm_quant.py #Quantization Main Program |
run_llmc.sh #Quantization Run Script
```

### 21.2.3 Operating Steps

#### [Phase 1] Prepare Calibration and Eval Datasets

- Note 1: **Calibration Dataset** can be an open-source dataset or a business dataset. If the model has been fine-tuned on downstream business datasets, then a business dataset needs to be selected for calibration.
- Note 2: **Eval Dataset** is primarily used to evaluate the accuracy performance of the current model, including the accuracy of pre-trained (pretrain) models or quantized (fake\_quant) models.

You can choose to use an open-source dataset or a business dataset.

#### open-source dataset

If a business dataset is available, it is preferable. If not, you can use an open-source dataset as follows:

Table 21.2: Dataset Selection

Model Type	Quantization Algorithm	Algo-	Calibration Dataset (Open-source)	Eval Dataset (Open-source)
LLM	Awq		pileval	wikitext2
LLM	GPTQ		wikitext2	wikitext2
VLM	Awq		MME	MME

The selection of the calibration dataset depends on the model type and quantization algorithm. For example, if the model being quantized is an LLM and uses the Awq algorithm, it is typically recommended to use the Pileval dataset as the calibration set. For these open-source datasets, this document provides the corresponding download commands, which can be executed to download the respective datasets. The specific steps are as follows: open the llmc-tpu/tools directory, where you will find two Python scripts, download\_calib\_dataset.py and download\_eval\_dataset.py, which are used to download the calibration and eval datasets, respectively.

If it is a VLM model, it is recommended to use the Awq algorithm. The command to download the dataset is as follows:

```
1 cd /workspace/llmc-tpu
      · Calibration Dataset
1 python3 tools/download_calib_dataset.py --dataset_name MME --save_path tpu/data/VLM/cali
      · Eval Dataset
1 python3 tools/download_eval_dataset.py --dataset_name MME --save_path tpu/data/VLM/eval
```

If it is an LLM model, it is recommended to use the Awq algorithm. The command to download the dataset is as follows:

```
1 cd /workspace/llmc-tpu
      · Calibration Dataset
1 python3 tools/download_calib_dataset.py --dataset_name pileval --save_path tpu/data/LLM/cali
      · Eval Dataset
1 python3 tools/download_eval_dataset.py --dataset_name wikitext2 --save_path tpu/data/LLM/
  ↪ eval
```

## business dataset

### 1. business calibration dataset

If the model has been fine-tuned on downstream business datasets, it is generally recommended to select the business dataset when choosing the calibration set. \* If it is an LLM, simply place the business dataset in the aforementioned LLM/cali directory. Regarding the specific format of the dataset, users can write each data entry as separate lines in a .txt file, with each line representing a single text data entry. By using the above configuration, you can perform calibration with a custom dataset. \* If it is a VLM, simply place the business dataset in the aforementioned VLM/cali directory. Regarding the specific format of the dataset, you can refer to the format in VLM/cali/general\_custom\_data and choose the format that meets your needs. It is important to note that the final JSON file should be named samples.json.

### 2. business eval dataset

If the model has been calibrated with downstream business datasets, it is generally recommended to use a business dataset for eval when selecting the eval set. \* If it is an LLM, simply place the business dataset in the aforementioned LLM/eval directory. Regarding the specific format of the dataset, users can write each data entry as a separate line of text in a .txt file, with each line representing one text data entry. Using the above configuration, custom dataset testing can be achieved. \* If it is a VLM, simply place the business dataset

in the aforementioned VLM/eval directory. Regarding the specific format of the dataset, you can refer to the format in VLM/cali/general\_custom\_data and choose the format that meets your needs. It is important to note that the final JSON file should be named samples.json.

### Phase Two: Configure the Quantization Configuration File

- Note: The quantization configuration file includes the settings required for the quantization process. Users can select configurations according to their needs. Additionally, to align with the TPU hardware configuration, certain parameters may have restrictions. Please refer to the detailed explanation below for more information.

#### Configuration File Parameter Description

```

1  base:
2      seed: &seed 42
3  model:
4      type: Qwen2VL # Set the model name. For specific supported models, refer to the llmc/
5      ↪models directory.
6      path: /workspace/Qwen2-VL-2B-Instruct    # Set the model weights path, please change to [F]
7      ↪your desired model
8      torch_dtype: auto
9  calib:
10     name: mme   # Set to the actual calibration dataset name, such as mme, pileval, etc.
11     download: False
12     path: /workspace/llmc-tpu/tpu/data/VLM/cali/MME # Set the calibration dataset path
13     n_samples: 128
14     bs: 1
15     seq_len: 512
16     preproc: pileval_awq
17     seed: *seed
18 eval:
19     eval_pos: [pretrain, fake_quant]
20     name: mme # Set to the actual eval dataset name, such as mme, wikitext2, etc.
21     download: False
22     path: /workspace/llmc-tpu/tpu/data/VLM/eval/MME # Set the eval dataset path
23     bs: 1
24     seq_len: 2048
25 quant:
26     method: Awq
27     quant_objects: [language] # By default, only quantize the LLM part. If you want to quantize [F]
28     ↪the VIT part, set it to [vision, language].
29 weight:
30     bit: 4 # Set to the desired quantization bit, supports 4 or 8
31     symmetric: False # Set to False for 4-bit and True for 8-bit
32     granularity: per_group # Set to per_group for 4-bit and per_channel for 8-bit.
      group_size: 64 # Set to 64 for 4-bit (corresponding to TPU-MLIR); set to -1 for 8-bit.
special:
      trans: True

```

(continues on next page)

(continued from previous page)

```

33     trans_version: v2
34     weight_clip: True
35     clip_sym: True
36     save:
37         save_trans: True      # When set to True, you can save the adjusted floating-point weights.
38         save_path: ./save_path # Set the path to save the weights
39     run:
40         task_name: awq_w_only
41         task_type: VLM      # Set to VLM or LLM

```

The above is a complete config file constructed using the Awq algorithm as an example. To simplify user operations, users can directly copy the above into their own config and then modify the parameters that are annotated.

Below are detailed explanations of some important parameters:

Table 21.3: Introduction of Relevant Parameters

Parameter	Description
model	model name. the supported models are in the llmc/models directory. You can add new models by including llmc/models/xxxx.py.
calib	calib class parameters mainly specify parameters related to the calibration set
eval	eval class parameters mainly specify parameters related to the eval set.
quant	specify the quantization parameters. It is generally recommended to use the Awq algorithm. For quant_objects, typically select language. For weight quantization parameters, refer to the table below.

To align with TPU-MLIR, the configuration of weight quantization related parameters is as follows:

Table 21.4: weight-only quantization parameters

bit	symmetric	granularity	group_size
4	False	per-channel or per-group	-1 or 64(default)
8	True	per-channel	-1

**Stage 3: Execute the Quantization Algorithm**

```
1 cd /workspace/llmc-tpu  
2 python3 tpu/llm_quant.py --llmc_tpu_path . --config_path ./tpu/example.yml
```

- config\_path refers to the path of the quantization configuration file, and llmc\_tpu\_path refers to the current llmc\_tpu directory path.

# CHAPTER 22

---

## Analyse TPU Performance with TPU Profile

---

### 22.1 1. Software and Hardware Framework of TPU

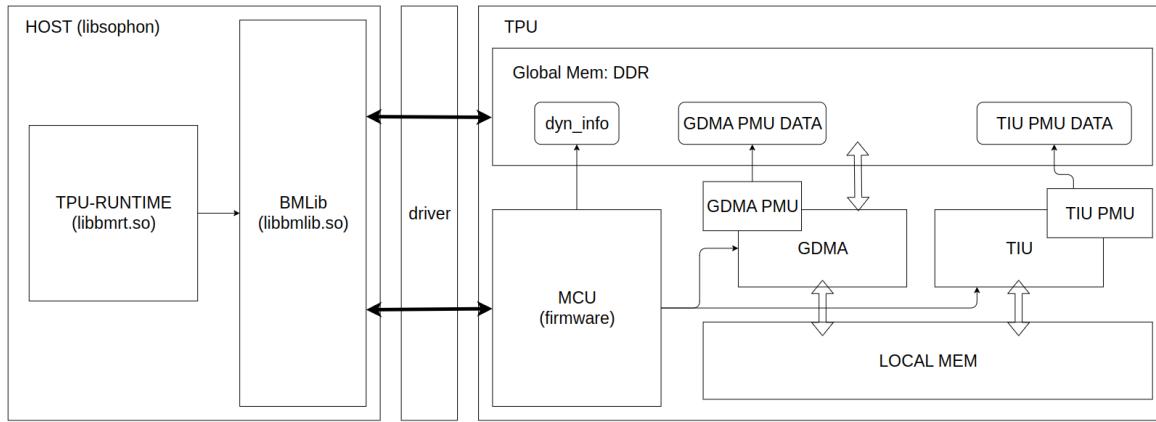
As the following figure shows, a whole TPU application depends on the cooperation of software and hardware:

**Software** Host provides **libsophon**, driver software packs. Driver abstracts the mechanism of basic communication and resource management, defines function interfaces. **Libsophon** implements various concrete functions for TPU inference, such as BMLib and TPU-RUNTIME.

- **BMLib** (`libbmlib.so`) wraps the driver interfaces for compatibility and portability of applications, improving performance and simplicity of programming.
- **TPU-RUNTIME** (`libbmrt.so`) implements loading, management, execution, and so on.

**Hardware** TPU mainly consists of three engines: MCU, GDMA, TIU.

- An A53 processor is used as MCU on BM1684X, which implements concrete operators by sending commands, communicating with driver, simple computation with firmware program.
- **GDMA** engine is used for transmitting data between Global Mem and Local Mem, moving data as 1D, matrix, 4D formats.
- **TIU** engine performs computing operations, including Convolution, Matrix Multiplication, Arithmetic Operations.



TPU Profile is a tool for visualizing the profile binary data in HTML formats, which comes from the time information recorded by the hardware modules called GDMA PMU and TPU PMU, the running time of key functions and the metadata in the bmodel. The profile data generation is done during compiling the bmodel and executing the inference application; it is disabled by default and can be enabled by setting environment variables.

This article uses Profile data and TPU Profile tools to visualize the complete running process of the model, providing an intuitive understanding of the internal TPU.

## 22.2 2. Compile Bmodel

(This operation and the following operations will use [TPU MLIR](#))

Due to the fact that the profile data will save some layer information during compilation to the BModel (resulting in a larger BModel size), it is disabled by default. To enable it, call `model_deploy.py` with the `--debug` option. If this option is not used during compilation, some data obtained by enabling profile during visualization may be missing.

Below, we demonstrate the `yolov5s` model in the TPU MLIR project.

Generate top MLIR:

```
model_transform.py \
--model_name yolov5s \
--model_def ./yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 350,498,646 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s.mlir
```

Convert top MLIR to BModel with FP16 precision:

```
model_deploy.py \
--mlir yolov5s.mlir \
--quantize F16 \
--chip bm1684x \
--test_input yolov5s_in_f32.npz \
--test_reference yolov5s_top_outputs.npz \
--model yolov5s_1684x_f16.bmodel \
--debug # Record profile data
```

Using the above commands, compile yolov5s.onnx into yolov5s\_bm1684x\_f16.bmodel. For more usage, please refer to the [TPU MLIR](#) repository.

### 22.3 3. Generate Profile Binary Data

Similar to the compilation process, the profile function at runtime is disabled by default to prevent additional time consumption during profile saving and transmission. To enable it, set the environment variable:

```
export BMRUNTIME_ENABLE_PROFILE=1
```

Below, use the model testing tool provided in libsophon, `bmrt_test`, as an application to generate profile data:

```
BMRUNTIME_ENABLE_PROFILE=1 bmrt_test --bmodel resnet50_fix8b.bmodel
```

The following is the output log after enabling profile mode:

```
[BMRT][load_bmodel:1084] INFO:Loading bmodel from [yolov5s_1684x_f16.bmodel]. Thanks for your patience...
[BMRT][load_bmodel:1023] INFO:pre net num: 0, load net num: 1
[BMRT][show_net_info:1520] INFO: #####
[BMRT][show_net_info:1521] INFO: NetName: yolov5s, Index=0
[BMRT][show_net_info:1523] INFO: ... stage 0 ...
[BMRT][show_net_info:1532] INFO: Input 0) 'images' shape=[ 1 3 640 640 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 0) '350_Transpose_f32' shape=[ 1 3 80 80 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 1) '498_Transpose_f32' shape=[ 1 3 40 40 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 2) '646_Transpose_f32' shape=[ 1 3 20 20 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1545] INFO: #####
[BMRT][bmrt_test:782] INFO:>>> running network #0, name: yolov5s, loop: 0
[BMRT][bmrt_test:868] INFO:reading input #0, bytesize=4915200
[BMRT][print_array:706] INFO: -> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1228800
[BMRT][write_block:295] INFO:write block: type=1, len=36
[BMRT][write_block:295] INFO:write block: type=3, len=44
[BMRT][end:76] INFO:bdc record_num=1048576 * The saved profile data information
[BMRT][write_block:295] INFO:write block: type=3, len=58816
[BMRT][end:89] INFO:gdn record_num=196, max_record_num=1048576
[BMRT][write_block:295] INFO:write block: type=4, len=37632
[BMRT][write_block:295] INFO:write block: type=5, len=256
[BMRT][write_block:295] INFO:write block: type=6, len=1072
[BMRT][print_note:94] INFO:*****
[BMRT][print_note:95] INFO:/* PROFILE MODE due to BMRUNTIME_ENABLE_PROFILE=1 */
[BMRT][print_note:96] INFO:/* Note: BMRUNTIME will collect time data during running * */
[BMRT][print_note:97] INFO:/* that will cost extra time. * */
[BMRT][print_note:98] INFO:/* Close PROFILE Mode by "unset BMRUNTIME_ENABLE_PROFILE" */
[BMRT][print_note:99] INFO:*****
[BMRT][bmrt_test:1065] INFO:reading output #0, bytesize=6528000
[BMRT][print_array:706] INFO: -> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1632000
[BMRT][bmrt_test:1065] INFO:reading output #1, bytesize=1632000
[BMRT][print_array:706] INFO: -> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=408000
[BMRT][print_array:706] INFO: -> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=102000
[BMRT][bmrt_test:1039] INFO:net[yolov5s] stage[0], launch total time is 375609 us (npu 5650 us, cpu 369959 us) CPU Time is not accuracy on Profile Mode
[BMRT][bmrt_test:1042] INFO:+++ The network[yolov5s] stage[0] output_data ++
[BMRT][print_array:706] INFO:output data #0 shape: [1 3 80 80 85 ] < 0.30957 -0.289551 0.0744629 -0.203003 -11.9375 -1.54297 -5.25391 -3.05859 -5.39453 -5.64844 -95 -6.26172 ... > len=1632000
[BMRT][print_array:706] INFO:output data #1 shape: [1 3 40 40 85 ] < -0.0398254 0.253174 -0.383057 -0.520996 -11.0391 -1.3125 -5.11328 -3.32031 -5.46484 -5.97656 812 -6.29688 ... > len=408000
[BMRT][print_array:706] INFO:output data #2 shape: [1 3 20 20 85 ] < 0.713379 0.654297 -0.534668 -0.241577 -9.82031 -1.23047 -5.77344 -3.35547 -5.92578 -5.12109 -44 -6.63672 ... > len=102000
[BMRT][bmrt_test:1083] INFO:load input time(s): 0.003650
[BMRT][bmrt_test:1084] INFO:calculate time(s): 0.375613
[BMRT][bmrt_test:1085] INFO:get output time(s): 0.003838
[BMRT][bmrt_test:1086] INFO:compare time(s): 0.000827
```

Meanwhile, a directory named `bmprofile_data-1` will be generated, containing all the profile data.

## 22.4 4. Visualize Profile Data

TPU-MLIR provides the script `tpu_profile.py`, which converts the raw profile data to a web page for visualization.

Convert the data in `bmprofile_data-1` to HTML in `bmprofile_out`:

```
tpu_profile.py bmprofile_data-1 bmprofile_out
```

Then list the output:

```
ls bmprofile_out  
# echarts.min.js profile_data.js result.html
```

Open `bmprofile_out/result.html` in a web browser to view the profile chart.

For more options:

```
tpu_profile.py --help
```

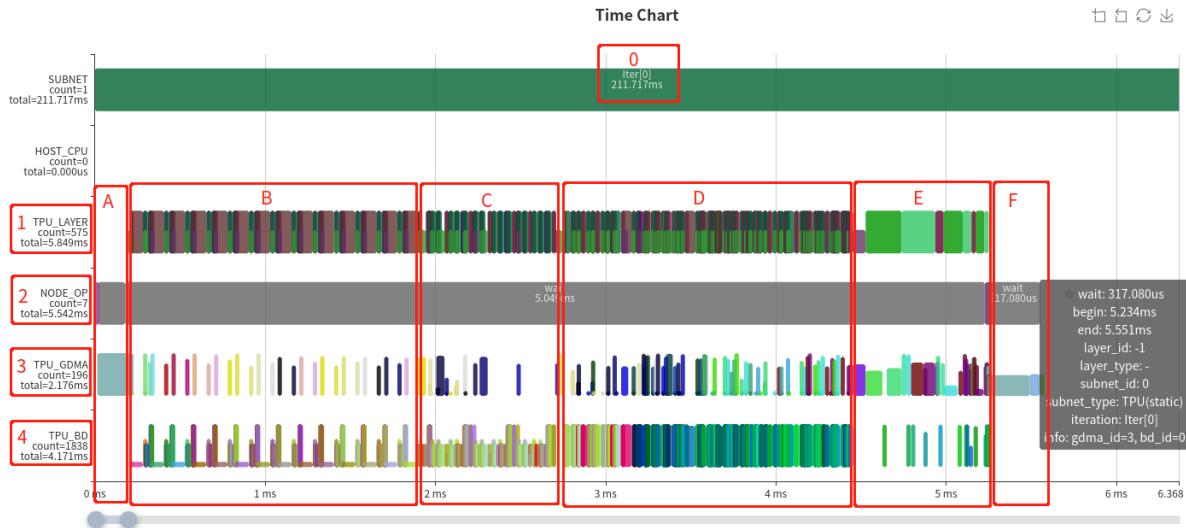
## 22.5 5. Analyse the Result

### 22.5.1 5.1 Overall Introduction

The complete result page is divided into:

- **Instruction timing chart**
- **Memory space-timing chart** (collapsed by default; expand via the “Show LOCALMEM” and “Show GLOBAL MEM” checkboxes)

### Profile: yolov5s on BM1684X



The instruction timing chart components:

0. Host time marker (may not be accurate; used as subnet separation markers)
1. **NODE\_OP**: Timing of each layer in the network, derived from TPU\_GDMA and TPU\_BD (TIU) operations. A **Layer Group** divides an operator into data transmission (half-height blocks) and computation (full-height blocks) that run in parallel.
2. **MCU**: Key functions recorded include setting GDMA, TIU instructions, and waiting for completion. The sum time equals the actual run time of the host-side management.
3. **GDMA**: Timing of GDMA operations on TPU; block height indicates actual data transmission bandwidth.
4. **TIU**: Timing of TIU operations on TPU; block height indicates effective compute utilization.

In the NODE\_OP line, the statistic **total=5.542ms** indicates the entire network runs in 5.542 ms. Instruction configuration time is short; most time is spent waiting during the network execution.

The overall operation splits into three parts: section A, sections B–E, and section F:

- **A**: MCU moves input data from user space to computational instruction space.
- **B–E**: Fused Layer Groups (multiple layers fused, sliced, and pipelined).
- **F**: MCU moves output data from instruction space back to user space.

Layers in B, C, and D form three fused Layer Groups (periodic half-height loads/saves). Section E contains unfused global layers.



The memory space–timing chart shows LOCAL MEM (top) and GLOBAL MEM (bottom):

- **X axis:** Time (aligned with the instruction timing chart)
- **Y axis:** Memory address space
- Green blocks: Occupied space (width = time, height = size)
- Red: GDMA write or TIU output
- Green: GDMA read or TIU input

LOCAL MEM is the TPU's internal compute space. On BM1684X, TIU has 64 lanes  $\times$  128 KB each, divided into 16 banks. Only Lane 0 is shown. Inputs and outputs must avoid the same bank to prevent conflicts.

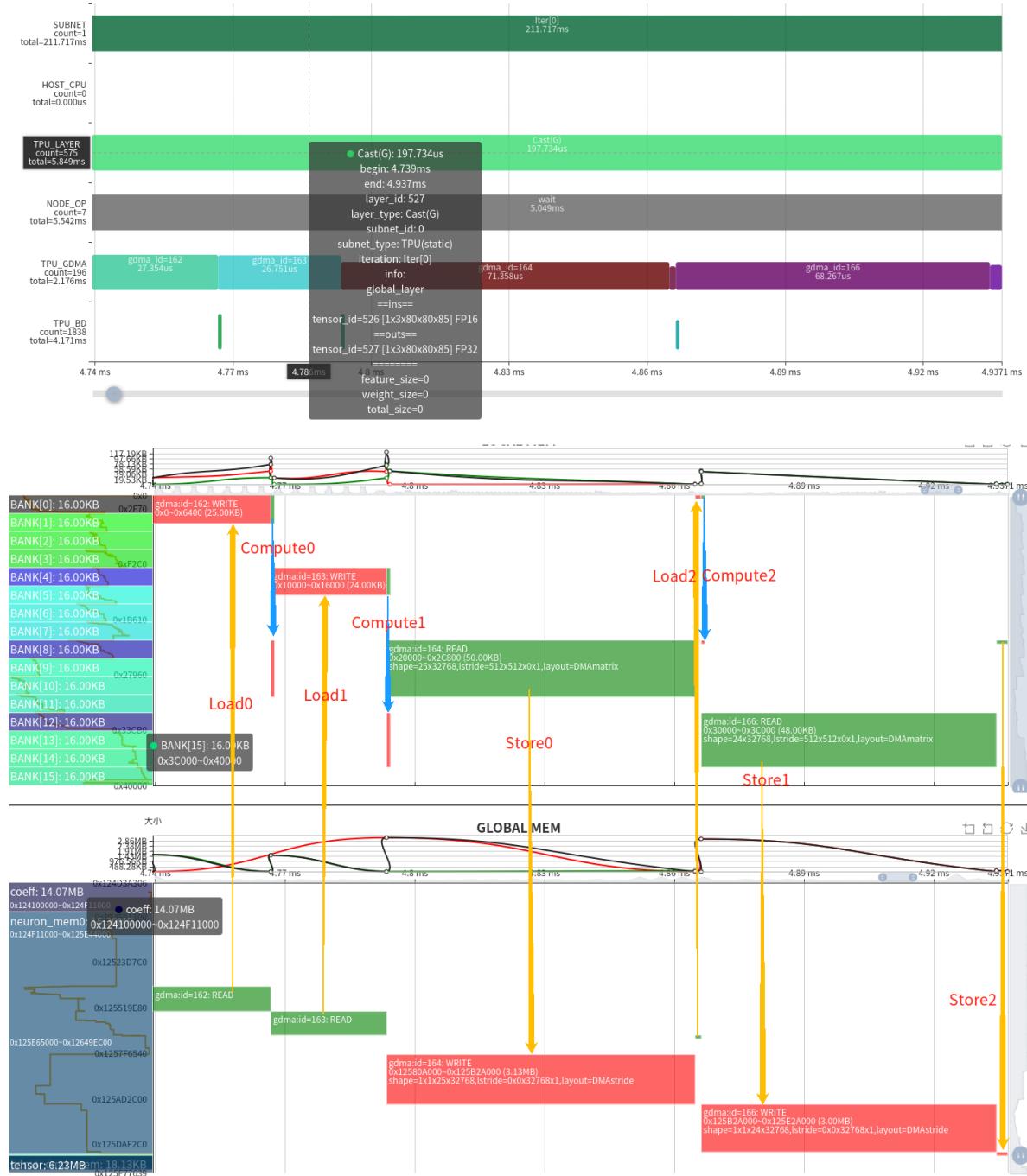
GLOBAL MEM is larger (4 GB–12 GB); only used regions are displayed. Green = GDMA read; Red = GDMA write.

Observations:

- Layer Groups use LOCAL MEM periodically; TIU I/O aligns with bank boundaries.
- GLOBAL MEM shows fewer writes during fused groups and alternating write/read during global layers.
- Total GLOBAL MEM usage: 14.07 MB (Coeff), 15.20 MB (Runtime), 6.23 MB (Tensor).

### 22.5.2 5.2 Global Layer

Analyze a simple global layer (previous Cast layer cannot fuse due to a Permute).



The layer casts a float16 tensor of shape  $1 \times 3 \times 80 \times 80 \times 85$  to float32.

Execution timeline:

time →

**Load0 | Compute0 | Store0 | |**

Load1 | Compute1 | Store1 |  
| Load2 | Compute2 | Store2

Since only one GDMA device handles loads and stores serially, the pipeline becomes:

time ——————>

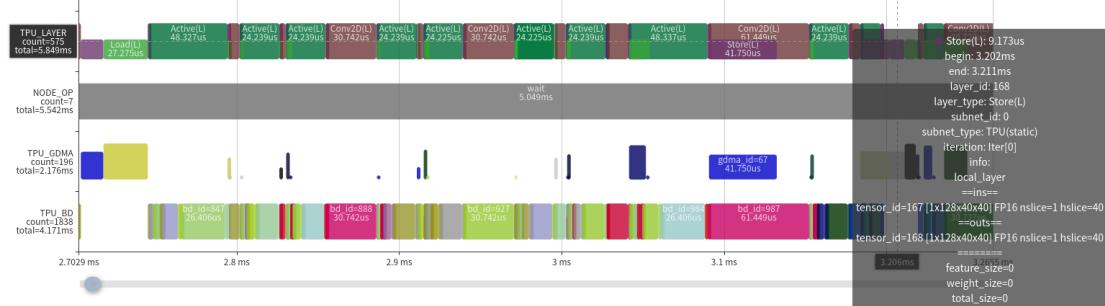
GDMA: Load0 | Load1 | Store0, Load2 | Store1 | Store2 TIU : | Compute0 | Compute1 | Compute2 |

The memory chart confirms this flow. Casting doubles memory, so transmission time doubles. The run time is bandwidth-bound, illustrating the necessity of layer fusion.

### 22.5.3 5.3 Local Layer Group

Two cases:

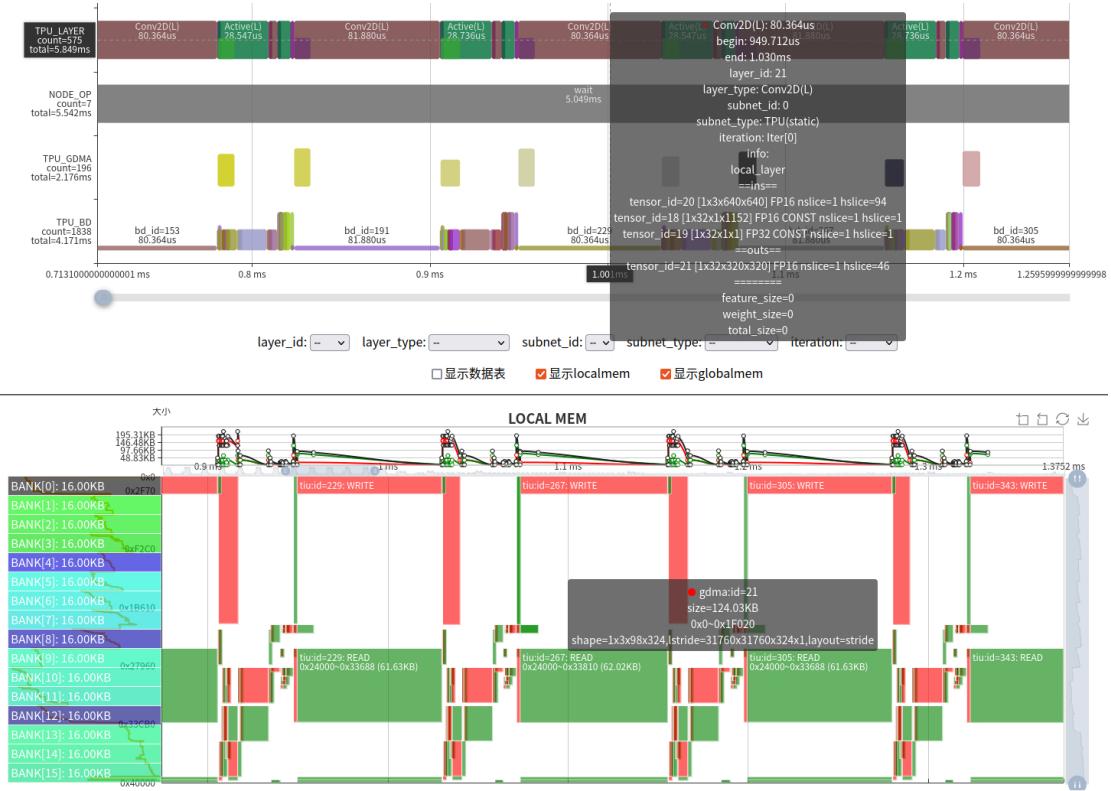
#### 1. High efficiency:



- Few GDMA ops in the middle, reducing data movement.
- High TIU efficiency; compute power is fully utilized.
- No gaps between TIU ops.

Optimization space is limited; improvements must come from network structure or other aspects.

#### 2. Low compute utilization:



Caused by unfriendly operator parameters. BM1684X TIU has 64 lanes; ideal input channels (IC) are multiples of 64. Here IC=3 → only 3/64 utilization.

Solutions:

- Use LOCAL MEM to increase slices and reduce instructions.
- Apply data transformations (e.g., permutations). For the first layer IC=3, we introduced a “3IC” technique.
- Modify the network or adjust computations.

Some inefficiencies are unavoidable without changes to TPU architecture or instructions.

## 22.6 6. Summary

This article demonstrates the complete TPU profiling process and how to use the visualization charts to analyze TPU runtime behavior and bottlenecks.

The Profile tool is essential for AI compiler development, providing deep insights for software and hardware design. It also aids debugging by visually detecting errors such as memory overlaps and synchronization issues.

TPU Profile’s display capabilities are continuously improving.

# CHAPTER 23

## Appendix.01: Migrating from NNTC to tpu-mlir

NNTC is using docker version sophgo/tpuc\_dev:v2.1, for MLIR docker version reference and environment initialization please refer to [Environment Setup](#).

In the following, we will use yolov5s as an example to explain the similarities and differences between nntc and mlir in terms of quantization, and for compiling floating-point models, please refer to <TPU-MLIR\_Quick\_Start> Compile the ONNX model.

First, refer to the section [Compile the ONNX model](#) to prepare the yolov5s model.

### 23.1 ONNX to MLIR

To quantize a model in mlir, first convert the original model to a top-level mlir file, this step can be compared to generating a fp32umodel in step-by-step quantization in nntc.

#### 1. MLIR's model conversion command

```
$ model_transform.py \
--model_name yolov5s \
--model_def ..../yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 350,498,646 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s.mlir
```

TPU-MLIR can directly encode image preprocessing into the converted MLIR file.

## 2. Model transformation commands for NNTC

```
$ python3 -m ufw.tools.on_to_umodel \
-m ..../yolov5s.onnx \
-s '(1,3,640,640)' \
-d 'compilation' \
--cmp
```

When importing a model with NNTC, you cannot specify the preprocessing method.

## 23.2 Make a quantization calibration table

If you want to generate a fixed-point model, you need a quantization tool to quantize the model, nntc uses calibration\_use\_pb for step-by-step quantization, and mlir uses run\_calibration.py for step-by-step quantization.

The number of input data is about 100~1000 depending on the situation, using the existing 100 images from COCO2017 as an example, perform calibration.

To use stepwise quantization in nntc, you need to make your own mdb quantization dataset using the image quantization dataset, and modify fp32\_prototxt to point the data input to the lmdb file.

---

**Note:** For the NNTC quantization dataset, please refer to the “Model Quantization” chapter in the <TPU-NNTC Development Reference Manual>, and note that the lmdb dataset is not compatible with TPU-MLIR. TPU-MLIR can directly use raw images as input for quantization tools. If it is voice, text or other non-image data, it needs to be converted to npz file.

---

### 1. MLIR Quantization Model

```
$ run_calibration.py yolov5s.mlir \
--dataset ..../COCO2017 \
--input_num 100 \
-o yolov5s_cali_table
```

After quantization you will get the quantization table yolov5s\_cali\_table

### 2. NNTC Quantization Model

```
$ calibration_use_pb quantize \
--model=../compilation/yolov5s_bmneto_test_fp32.prototxt \
--weights=../compilation/yolov5s_bmneto.fp32umodel \
-save_test_proto=True --bitwidth=TO_INT8
```

In nntc, after quantization, you get int8umodel and prototxt.

### 23.3 Generating int8 models

To convert to an INT8 symmetric quantized model, execute the following command.

#### 1. MLIR:

```
$ model_deploy.py \
  --mlir yolov5s.mlir \
  --quantize INT8 \
  --calibration_table yolov5s_cali_table \
  --processor bm1684 \
  --test_input yolov5s_in_f32.npz \
  --test_reference yolov5s_top_outputs.npz \
  --tolerance 0.85,0.45 \
  --model yolov5s_1684_int8_sym.bmodel
```

At the end of the run you get `yolov5s_1684_int8_sym.bmodel`.

#### 2. NNTC:

In nntc, the int8 bmodel is generated using `int8umodel` and `prototxt` using the `bmnetu` tool.

```
$ bmnetu --model=./compilation/yolov5s_bmneto_deploy_int8_unique_top.prototxt \
  --weight=./compilation/yolov5s_bmneto.int8umodel
```

At the end of the run you get `compilation.bmodel`.

# CHAPTER 24

---

## Appendix 02: Basic Elements of TpuLang

---

This chapter will introduce the basic elements of TpuLang programs: Tensor, Scalar, Control Functions, and Operator.

### 24.1 Tensor

In TpuLang, the properties of a Tensor, including its name, data, data type, and tensor type, can only be declared or set at most once.

Generally, it is recommended to create a Tensor without specifying a name to avoid potential issues arising from identical names. Only when it is necessary to specify a name should you provide one during the creation of the Tensor.

For Tensors that serve as the output of an Operator, you can choose not to specify the shape since the Operator will deduce it automatically. Even if you do specify a shape, when the Tensor is the output of an Operator, the Operator itself will deduce and modify the shape accordingly.

The definition of Tensor in TpuLang is as follows:

```
class Tensor:

    def __init__(self,
                 shape: list = [],
                 name: str = None,
                 ttype="neuron",
                 data=None,
                 dtype: str = "float32",
                 scale: Union[float, List[float]] = None,
```

(continues on next page)

(continued from previous page)

```
zero_point: Union[int, List[int]] = None)
#pass
```

As shown above, a Tensor in TpuLang has five parameters:

- shape: The shape of the Tensor, a List[int]. For Tensors that serve as the output of an Operator, the shape can be left unspecified with a default value of [].
- Name: The name of the Tensor, a string or None. It is recommended to use the default value None to avoid potential issues arising from identical names.
- dtype: The type of the Tensor, which can be “neuron,” “coeff,” or None. The initial value is “neuron.”
- data: The input data for the Tensor.ndarray or None, the default value is None, the Tensor will be initialized with all zeros based on the specified shape. If `ttype == "coeff"`, data **must** be provided (cannot be None). If data is an ndarray, its shape and dtype must match the declared shape and dtype.
- dtype: The data type of the Tensor, with a default value of “float32.” Other possible values include “float32,” “float16,” “int32,” “uint32,” “int16,” “uint16,” “int8,” and “uint8.”
- scale: The quantization scale parameter of Tensor, float or List[float], default value is None;
- zero\_point: The quantization zero-point parameter, also known as the offset parameter of Tensor, int or List[int], default value is None;

Example of declaring a Tensor:

```
#activation
input = tpul.Tensor(name='x', shape=[2,3], dtype='int8')
#weight
weight = tpul.Tensor(dtype='float32', shape=[3,4], data=np.random.uniform(0,1,
˓→shape).astype('float32'), ttype="coeff")
```

## 24.2 Tensor Preprocessing (Tensor.preprocess)

In TpuLang, if a Tensor is an input and requires preprocessing, you can call this function.

The definition of `Tensor.preprocess` in TpuLang is as follows:

```
class Tensor:

    def preprocess(self,
                  mean : List[float] = [0, 0, 0],
                  scale : List[float] = [1.0, 1.0, 1.0],
                  pixel_format : str = 'bgr',
```

(continues on next page)

(continued from previous page)

```
channel_format : str = 'nchw',
resize_dims : List[int] = None,
keep_aspect_ratio : bool = False,
keep_ratio_mode : str = 'letterbox',
pad_value : int = 0,
pad_type : str = 'center',
white_level : float = 4095,
black_level : float = 112):
#pass
```

As shown above, Tensor.preprocess in TpuLang has the following parameters:

- mean: The average value of each channel of Tensor. Default = [0, 0, 0]
- scale: The scale value of each channel of the Tensor. Default = [1, 1, 1]
- pixel\_format: The pixel format of Tensor. Default = ‘bgr’ , Choices: ‘rgb’ , ‘bgr’ , ‘gray’ , ‘rgba’ , ‘gbrg’ , ‘grbg’ , ‘bggr’ , ‘rggb’ .
- channel\_format: The data format of Tensor, i.e. whether channel is first or last. Default = ‘nchw’ .Choices: ‘nchw’ , ‘nhwc’ .
- resize\_dims: [h, w] of the Tensor after resizing. The default value is None, which means taking the h and w of the Tensor.
- keep\_aspect\_ratio: Parameter of resize operation that determines whether to maintain the same scaling ratio, bool, default = False
- keep\_ratio\_mode: Parameter of resize operation that specifies the mode when keep\_aspect\_ratio is enabled, default = ‘letterbox’ . Choices: ‘letterbox’ , ‘short\_side\_scale’ .
- pad\_value:Parameter of resize operation that sets the value when padding, int, default = 0.
- pad\_type: The padding strategy when resizing, str, default = ‘center’ . Choices: ‘normal’ , ‘center’ .
- white\_level: The white-level parameter for raw image processing, str, default = 4095
- black\_level: The black-level parameter for raw image processing, str, default = 112

Example of declaring Tensor.preprocess:

```
#activation
input = tpul.Tensor(name='x', shape=[2,3], dtype='int8')
input.preprocess(mean=[123.675,116.28,103.53], scale=[0.017,0.017,0.017])
# pass
```

## 24.3 Scalar

Define a scalar Scalar. A Scalar is a constant specified during declaration and cannot be modified afterward.

```
class Scalar:  
  
    def __init__(self, value, dtype=None):  
        #pass
```

The Scalar constructor has two parameters:

- value: Variable type, i.e., int/float type, with no default value, and must be specified.
- dtype: The data type of the Scalar. If the default value None is used, it is equivalent to “float32.”

Otherwise, it can take values such as “float32,” “float16,” “int32,” “uint32,” “int16,” “uint16,” “int8,” and “uint8.”

Example of usage:

```
pad_val = tpul.Scalar(1.0)  
pad = tpul.pad(input, value=pad_val)
```

## 24.4 Control Functions

Control functions mainly involve controlling the initialization of TpuLang, starting the compilation process to generate target files, and other related operations.

Control functions are commonly used before and after the definition of Tensors and Operators in a TpuLang program. For example, initialization might be necessary before writing Tensors and Operators, and compilation and deinitialization might be performed after completing the definitions of Tensors and Operators.

### 24.4.1 Initialization Function

Initialization Function is used before constructing a network in a program.

The interface for the initialization function is as follows, where you choose the processor:

```
def init(device):  
    #pass
```

- The device parameter is of type string and can take values from the range “BM1684X” | “BM1688” | “CV183X” .

### 24.4.2 compile

#### The interface definition

```
def compile(name: str,
            inputs: List[Tensor],
            outputs: List[Tensor],
            cmp=True,
            refs=None,
            mode='f32',      # unused
            dynamic=False,
            asymmetric=False,
            no_save=False,
            opt=2,
            mlir_inference=True,
            bmodel_inference=True,
            log_level="normal",
            embed_debug_info=False):
    #pass
```

#### Description of the function

The function for compiling TpuLang model to bmodel.

#### Explanation of parameters

- name: A string. Model name.
- inputs: List of Tensors, representing all input Tensors for compiling the network.
- outputs: List of Tensors, representing all output Tensors for compiling the network.
- cmp: A boolean. True indicates result verification is needed, False indicates compilation only. ‘cmp’ parameter is useless when ‘mlir\_inference’ set to False.
- refs: List of Tensors, representing all Tensors requiring verification in the compiled network.
- mode: A string. Indicates the type of model, supporting “f32” and “int8” .
- dynamic: A boolean. Whether to do dynamic compilation.
- no\_save: A boolean. It indicates whether to temporarily store intermediate files in shared memory and release them along with the process. When this option is enabled, the compile function will return the generated ‘bmodel’ file as a bytes-like object, which the user needs to receive and do some further process, for example, by saving it using ‘f.write(bmodel\_bin).’ .
- asymmetric: A boolean. This parameter indicates whether it is for asymmetric quantization.

- opt: An integer type representing the compiler group optimization level. 0 indicates no need for layer group; 1 indicates grouping as much as possible; 2 indicates grouping based on dynamic programming.
- mlir\_inference: A boolean. Whether to do mlir inference. ‘cmp’ parameter is useless when ‘mlir\_inference’ set to False.
- bmodel\_inference: A boolean. Whether to do bmodel inference.
- **log\_level is used to control the log level. Currently it supports only-pass, only-layer-group, normal, and quiet:**
  - simple: Mainly prints graph to optimize pattern matching.
  - only-layer-group: mainly prints layer group information.
  - normal: The logs compiled and generated by bmodel will be printed out.
  - quiet: print nothing
- embed\_debug\_info: A boolean. Whether to enable profile.

#### 24.4.3 Deinitialization

After constructing the network, it is necessary to perform deinitialization to conclude the process. Only after deinitialization, the TPU executable target generated by the previously initiated compilation will be saved to the specified output directory.

```
def deinit():
    #pass
```

#### 24.4.4 Reset Default Graph

Before constructing a network, it is necessary to reset the default graph. If the input graph is None, after resetting the default graph, the current graph will be an empty graph. If a specific graph is provided, it will be set as the default graph. If there is only one subgraph, explicitly calling reset\_default\_graph is optional because the init function will invoke this method automatically.

```
def reset_default_graph(graph = None):
    #pass
```

#### 24.4.5 Get Current Default Graph

After building the network, if you need to obtain the default subgraph, call this function to retrieve the default graph.

```
def get_default_graph():
    #pass
```

#### 24.4.6 Reset Graph

To clear a graph and its stored Tensor information, call this function. If graph is None, it clears the information of the current default graph.

```
def reset_graph(graph = None):
    #pass
```

Note: If the Tensors in the graph are still used by other graphs, do not call this function to clear the graph's information.

#### 24.4.7 Rounding Mode

Rounding is the process of discarding extra digits beyond a certain point according to specific rules, yielding a shorter, unambiguous numerical representation. Given  $x$ , the rounded result is  $y$ . The following rounding modes are available:

Round to nearest; when the fractional part is 0.5, round to the nearest even number. Corresponds to `half_to_even`.

Round to nearest; positive values toward  $+\infty$ , negative values toward  $-\infty$ . Corresponds to `half_away_from_zero`. Formula:

$$y = \text{sign}(x) \lfloor |x| + 0.5 \rfloor = -\text{sign}(x) \lceil -|x| - 0.5 \rceil$$

Unconditional truncation toward zero. Corresponds to `towards_zero`. Formula:

$$y = \text{sign}(x) \lfloor |x| \rfloor = -\text{sign}(x) \lceil -|x| \rceil = \begin{cases} \lfloor x \rfloor & \text{if } x > 0, \\ \lceil x \rceil & \text{otherwise.} \end{cases}$$

Round toward  $-\infty$ . Corresponds to `down`. Formula:

$$y = \lfloor x \rfloor = -\lceil -x \rceil$$

Round toward  $+\infty$ . Corresponds to `up`. Formula:

$$y = \lceil x \rceil = -\lfloor -x \rfloor$$

Round to nearest; when the fractional part is 0.5, round toward  $+\infty$ . Corresponds to `half_up`. Formula:

$$y = \lceil x + 0.5 \rceil = -\lfloor -x - 0.5 \rfloor = \left\lceil \frac{\lfloor 2x \rfloor}{2} \right\rceil$$

Round to nearest; when the fractional part is 0.5, round toward  $-\infty$ . Corresponds to `half_down`. Formula:

$$y = \lfloor x - 0.5 \rfloor = -\lceil -x + 0.5 \rceil = \left\lfloor \frac{\lceil 2x \rceil}{2} \right\rfloor$$

The table below shows the mapping from  $x$  to  $y$  under different rounding modes.

	Half to Even	Half Away From Zero	Towards Zero	Down	Up	Half Up	Half Down
+1.8	+2	+2	+1	+1	+2	+2	+2
+1.5	+2	+2	+1	+1	+2	+2	+1
+1.2	+1	+1	+1	+1	+2	+1	+1
+0.8	+1	+1	0	0	+1	+1	+1
+0.5	0	+1	0	0	+1	+1	0
+0.2	0	0	0	0	+1	0	0
-0.2	0	0	0	-1	0	0	0
-0.5	0	-1	0	-1	0	0	-1
-0.8	-1	-1	0	-1	0	-1	-1
-1.2	-1	-1	-1	-2	-1	-1	-1
-1.5	-2	-2	-1	-2	-1	-1	-2
-1.8	-2	-2	-1	-2	-1	-2	-2

## 24.5 Operator

In order to optimize performance in TpuLang programming, operators are categorized into Local Operator, Limited Local Operator, and Global Operator.

- Local Operator: During compilation, local operators can be merged and optimized with other local operators, ensuring that the data between operations only exists in the local storage of the TPU.
- Limited Local Operator: Limited local operators can be merged and optimized with other local operators under certain conditions.
- Global Operator: Global operators cannot be merged and optimized with other operators. The input and output data of these operators need to be placed in the TPU's global storage.

Many of the following operations are element-wise operations, requiring input and output Tensors to have the same number of dimensions.

When an operation has two input Tensors, there are two categories based on whether shape broadcasting is supported or not. Support for shape broadcasting means that the shape values of `tensor_i0` (input 0) and `tensor_i1` (input 1) for the same dimension can be different. In this case, one of the tensor's shape values must be 1, and the data will be broadcasted to match the shape of the other tensor. Not supporting shape broadcasting requires the shape values of `tensor_i0` (input 0) and `tensor_i1` (input 1) to be identical.

### 24.5.1 NN/Matrix Operator

#### conv

##### The interface definition

```
def conv(input: Tensor,
         weight: Tensor,
         bias: Tensor = None,
         stride: List[int] = None,
         dilation: List[int] = None,
         pad: List[int] = None,
         group: int = 1,
         out_dtype: str = None,
         out_name: str = None):
    #pass
```

##### Description of the function

Two-dimensional convolution operation. You can refer to the definitions of 2D convolution in various frameworks. This operation belongs to **local operations**.

##### Explanation of parameters

- input: Tensor type, representing the input Tensor in 4D NCHW format.
- weight: Tensor type, representing the convolutional kernel Tensor in 4D NCHW format.
- bias: Tensor type, representing the bias Tensor. If None, it indicates no bias. Otherwise, it requires a shape of [1, oc, 1, 1], where oc represents the number of output channels.
- stride: List of integers, representing the stride size along each spatial axis. If None, it is [1, 1]. If not None, it requires a length of 2.
- dilation: List of integers, representing the dilation size along each spatial axis. If None, it is [1, 1]. If not None, it requires a length of 2.
- pad: List of integers, representing the padding size along each spatial axis, which follows the order of [x1\_begin, x2\_begin, ..., x1\_end, x2\_end, ...]. If None, it is [0, 0, 0, 0]. If not None, it requires a length of 4.
- groups: An integer, representing the number of groups in the convolution layer.
- out\_dtype: str or None. If None, the output tensor's data type matches the input's. Choices: "float32" or "float16".
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.
- BM1684X: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.

### `conv_int`

#### The interface definition

```
def conv_int(input: Tensor,
             weight: Tensor,
             bias: Tensor = None,
             stride: List[int] = None,
             dilation: List[int] = None,
             pad: List[int] = None,
             group: int = 1,
             input_zp: Union[int, List[int]] = None,
             weight_zp: Union[int, List[int]] = None,
             out_dtype: str = None,
             out_name: str = None):
    # pass
```

#### Description of the function

Two-dimensional convolution operation. You can refer to the definitions of 2D convolution in various frameworks.

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    output = (input - izp) Conv (weight - wzp) + bias[c];
```

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, the input tensor in 4-D NCHW format.
- **weight:** **Tensor type, the convolution kernel in 4-D [oc, ic, kh, kw] format, where**  
    oc = number of output channels ic = number of input channels kh = kernel height  
    kw = kernel width
- bias: Tensor type or None. If None, no bias is applied; otherwise shape must be [1, oc, 1, 1]. Data type is int32.
- stride: List[int] or None, the stride for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.
- dilation: List[int] or None, the dilation for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.
- pad: List[int] or None, the padding for each spatial dimension in [x1\_begin, x2\_begin, x1\_end, x2\_end] order. Defaults to [0, 0, 0, 0] if None; if provided, length must be 4.
- groups: int, number of convolution groups. If ic = oc = groups, performs depthwise convolution.
- input\_zp: int or List[int] or None, the zero-point for input. Defaults to 0 if None; if a list is provided its length must equal ic. (List mode not supported currently.)
- weight\_zp: int or List[int] or None, the zero-point for weight. Defaults to 0 if None; if a list is provided its length must equal ic (the number of input channels).
- out\_dtype: string or None, the output tensor's data type. Defaults to int32 if None. Valid values: “int32”, “uint32” .
- out\_name: string or None, the name of the output tensor. If None, a name is generated automatically.

### Return value

Returns a Tensor whose data type is determined by out\_dtype.

### Processor support

- BM1688: The input data type can be INT8 or UINT8. The bias data type must be INT32.
- BM1684X: The input data type can be INT8 or UINT8. The bias data type must be INT32.

## conv\_quant

### The interface definition

```
def conv_quant(input: Tensor,
               weight: Tensor,
               bias: Tensor = None,
               stride: List[int] = None,
               dilation: List[int] = None,
               pad: List[int] = None,
               group: int = 1,
               input_scale: Union[float, List[float]] = None,
               weight_scale: Union[float, List[float]] = None,
               output_scale: Union[float, List[float]] = None,
               input_zp: Union[int, List[int]] = None,
               weight_zp: Union[int, List[int]] = None,
               output_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
    # pass
```

### Description of the function

Two-dimensional convolution operation. You can refer to the definitions of 2D convolution in various frameworks.

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    conv_i32 = (input - izp) Conv (weight - wzp) + bias[c];
    output = requant_int(conv_i32, mul, shift) + ozp

    mul,shift are obtained from iscale, wscale, oscale
```

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, the input tensor in 4-D NCHW format.
- **weight: Tensor type, the convolution kernel in 4-D [oc, ic, kh, kw] format, where**  
oc = number of output channels ic = number of input channels kh = kernel height  
kw = kernel width
- bias: Tensor type or None. If None, no bias is applied; otherwise shape must be [1, oc, 1, 1]. Data type is int32.
- stride: List[int] or None, the stride for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.

- dilation: List[int] or None, the dilation for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.
- pad: List[int] or None, the padding for each spatial dimension in [x1\_begin, x2\_begin, x1\_end, x2\_end] order. Defaults to [0, 0, 0, 0] if None; if provided, length must be 4.
- groups: int, number of convolution groups. If  $ic = oc = \text{groups}$ , performs depthwise convolution.
- input\_scale: float or List[float] or None, the input quantization scale(s). Defaults to the tensor's existing scale if None; if a list is provided its length must equal  $ic$ . (List mode not supported.)
- weight\_scale: float or List[float] or None, the kernel quantization scale(s). Defaults to the tensor's existing scale if None; if a list is provided its length must equal  $oc$ .
- output\_scale: float or List[float], the output quantization scale(s). Must be provided; if a list is given its length must equal  $oc$ . (List mode not supported.)
- input\_zp: int or List[int] or None, the input zero-point(s). Defaults to 0 if None; if a list is provided its length must equal  $ic$ . (List mode not supported.)
- weight\_zp: int or List[int] or None, the kernel zero-point(s). Defaults to 0 if None; if a list is provided its length must equal  $oc$ .
- output\_zp: int or List[int] or None, the output zero-point(s). Defaults to 0 if None; if a list is provided its length must equal  $oc$ . (List mode not supported.)
- out\_dtype: string or None, the output tensor's data type. Defaults to int8 if None. Valid values: "int8", "uint8".
- out\_name: string or None, the name of the output tensor. If None, a name is generated automatically.

### Return value

Returns a Tensor whose data type is determined by out\_dtype.

### Processor support

- BM1688: The input data type can be INT8 or UINT8. The bias data type must be INT32.
- BM1684X: The input data type can be INT8 or UINT8. The bias data type must be INT32.

## deconv

### The interface definition

```
def deconv(input: Tensor,
           weight: Tensor,
           bias: Tensor = None,
           stride: List[int] = None,
           dilation: List[int] = None,
           pad: List[int] = None,
           output_padding: List[int] = None,
           group: int = 1,
           out_dtype: str = None,
           out_name: str = None):
    #pass
```

### Description of the function

Two-dimensional deconvolution operation. You can refer to the definitions of 2D deconvolution in various frameworks. This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing the input Tensor in 4D NCHW format.
- weight: Tensor type, representing the convolutional kernel Tensor in 4D NCHW format.
- bias: Tensor type, representing the bias Tensor. If None, it indicates no bias. Otherwise, it requires a shape of [1, oc, 1, 1], where oc represents the number of output channels.
- stride: List of integers, representing the stride size along each spatial axis. If None, it is [1, 1]. If not None, it requires a length of 2.
- dilation: List of integers, representing the dilation size along each spatial axis. If None, it is [1, 1]. If not None, it requires a length of 2.
- pad: List of integers, representing the padding size along each spatial axis. If None, it is [0, 0, 0, 0]. If not None, it requires a length of 4.
- output\_padding: List of integers, representing the output padding size along each spatial axis, which follows the order of [x1\_begin, x2\_begin…x1\_end, x2\_end,…]. If None, it is [0, 0, 0, 0]. If not None, it requires a length of 4.
- group: An integer, representing the number of group in the deconvolution layer.
- out\_dtype: str or None. If None, the output tensor's data type matches the input's. Choices: “float32” or “float16” .
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.
- BM1684X: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.

### deconv\_int

#### The interface definition

```
def deconv_int(input: Tensor,
               weight: Tensor,
               bias: Tensor = None,
               stride: List[int] = None,
               dilation: List[int] = None,
               pad: List[int] = None,
               output_padding: List[int] = None,
               group: int = 1,
               input_zp: Union[int, List[int]] = None,
               weight_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
    # pass
```

#### Description of the function

Two-dimensional convolution operation. You can refer to the definitions of 2D convolution in various frameworks.

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    output = (input - izp) Deconv (weight - wzp) + bias[c];
```

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor, the input tensor in 4-D NCHW format.
- **weight: Tensor, the deconvolution (transpose convolution) kernel in 4-D [oc, ic, kh, kw] format, where**  
    oc = number of output channels   ic = number of input channels   kh = kernel height  
    kw = kernel width
- bias: Tensor or None. If None, no bias is applied; otherwise its shape must be [1, oc, 1, 1]. Data type is int32.
- stride: List[int] or None, the stride for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.
- dilation: List[int] or None, the dilation for each spatial dimension. Defaults to [1, 1] if None; if provided, length must be 2.
- pad: List[int] or None, the padding for each spatial dimension in [x1\_begin, x2\_begin, x1\_end, x2\_end] order. Defaults to [0, 0, 0, 0] if None; if provided, length must be 4.
- output\_padding: List[int] or None, the additional size added to the output shape. Defaults to [0, 0] if None; if provided, length must be 1 or 2.
- groups: int, the number of deconvolution groups.
- input\_zp: int or List[int] or None, the zero-point for input quantization. Defaults to 0 if None; if a list is provided its length must equal ic. (List mode not supported currently.)
- weight\_zp: int or List[int] or None, the zero-point for kernel quantization. Defaults to 0 if None; if a list is provided its length must equal ic (the number of input channels).
- out\_dtype: string or None, the output tensor's data type. Defaults to int32 if None. Valid values: “int32”, “uint32” .
- out\_name: string or None, the name of the output tensor. If None, a name is generated automatically.

### Return value

Returns a Tensor whose data type is determined by out\_dtype.

### Processor support

- BM1688: The input data type can be INT8 or UINT8. The bias data type must be INT32.
- BM1684X: The input data type can be INT8 or UINT8. The bias data type must be INT32.

## conv3d

### The interface definition

```
def conv3d(input: Tensor,
           weight: Tensor,
           bias: Tensor = None,
           stride: List[int] = None,
           dilation: List[int] = None,
           pad: List[int] = None,
           group: int = 1,
           out_dtype: str = None,
           out_name: str = None):
    #pass
```

### Description of the function

Three-dimensional convolution operation. You can refer to the definitions of 3D convolution in various frameworks. This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing the input Tensor in 5D NCDHW format.
- weight: Tensor type, representing the convolutional kernel Tensor in 4D NCDHW format.
- bias: Tensor type, representing the bias Tensor. If None, it indicates no bias. Otherwise, it requires a shape of [1, oc, 1, 1, 1] or [oc], where oc represents the number of output channels.
- stride: List of integers, representing the stride size along each spatial axis. If None, it is [1, 1, 1]. If not None, it requires a length of 3.
- dilation: List of integers, representing the dilation size along each spatial axis. If None, it is [1, 1, 1]. If not None, it requires a length of 3.
- pad: List of integers, representing the padding size along each spatial axis, which follows the order of [x1\_begin, x2\_begin…x1\_end, x2\_end,…]. If None, it is [0, 0, 0, 0, 0, 0]. If not None, it requires a length of 6.
- groups: An integer, representing the number of groups in the convolution layer.
- out\_dtype: string or None, the output tensor's data type. If None, inherits the input tensor's data type. Valid values: “float32”, “float16” .
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.
- BM1684X: The input data type can be FLOAT32 or FLOAT16. The data types of input and weight must match. The bias data type must be FLOAT32.

## conv3d\_int

### The interface definition

```
def conv3d_int(input: Tensor,
               weight: Tensor,
               bias: Tensor = None,
               stride: List[int] = None,
               dilation: List[int] = None,
               pad: List[int] = None,
               group: int = 1,
               input_zp: Union[int, List[int]] = None,
               weight_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
```

### Description of the function

Fixed-point three-dimensional convolution operation. You can refer to the definitions of fixed-point 3D convolution in various frameworks.

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    kzp = is_kzp_const ? kzp_val : kzp_vec[c];
    output = (input - izp) Conv3d (weight - kzp) + bias[c];
```

Conv3d represents 3D convolution computation.

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input Tensor in 5D NCTHW format.
- weight: Tensor type, representing the convolutional kernel Tensor in 5D [oc, ic, kt, kh, kw] format. Here, oc represents the number of output channels, ic represents the number of input channels, kt is the kernel depth, kh is the kernel height, and kw is the kernel width.
- bias: Tensor type, representing the bias Tensor. If None, it indicates no bias. Otherwise, it requires a shape of [1, oc, 1, 1, 1].
- stride: List of integers, representing the stride size. If None, it is [1, 1, 1]. If not None, it requires a length of 3. The order in the list is [stride\_t, stride\_h, stride\_w].
- dilation: List of integers, representing the dilation size. If None, it is [1, 1, 1]. If not None, it requires a length of 2. The order in the list is [dilation\_t, dilation\_h, dilation\_w].
- pad: List of integers, representing the padding size. If None, it is [0, 0, 0, 0, 0, 0]. If not None, it requires a length of 6. The order in the list is [before, after, top, bottom, left, right].
- groups: An integer, representing the number of groups in the convolution layer. If ic=oc=groups, the convolution is depthwise conv3d.
- input\_zp: List of integers or an integer, representing the input offset. If None, it is 0. If a list is provided, it should have a length of ic.
- weight\_zp: List of integers or an integer, representing the kernel offset. If None, it is 0. If a list is provided, it should have a length of ic, where ic represents the number of input channels.
- out\_dtype: A string or None, representing the data type of the input Tensor. If None, it is int32. Possible values: int32/uint32.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the data type determined by out\_dtype.

### Processor support

BM1688: The data type of input and weight can be INT8/UINT8. The data type of bias is INT32. BM1684X: The data type of input and weight can be INT8/UINT8. The data type of bias is INT32.

### conv3d\_quant

#### The interface definition

```
def conv3d_quant(input: Tensor,
                 weight: Tensor,
                 bias: Tensor = None,
                 stride: List[int] = None,
                 dilation: List[int] = None,
                 pad: List[int] = None,
                 group: int = 1,
                 input_scale: Union[float, List[float]] = None,
                 weight_scale: Union[float, List[float]] = None,
                 output_scale: Union[float, List[float]] = None,
                 input_zp: Union[int, List[int]] = None,
                 weight_zp: Union[int, List[int]] = None,
                 output_zp: Union[int, List[int]] = None,
                 out_dtype: str = None,
                 out_name: str = None):
    # pass
```

#### Description of the function

Two-dimensional convolution operation. You can refer to the definitions of 2D convolution in various frameworks.

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    conv_i32 = (input - izp) Conv (weight - wzp) + bias[c];
    output = requant_int(conv_i32, mul, shift) + ozp
    mul,shift are obtained from iscale, wscale, oscale
```

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor, the input tensor in 5-D NCTHW format (N, C, T, H, W).
- weight: Tensor, the 3D convolution kernel in 5-D [oc, ic, kt, kh, kw] format, where - oc = number of output channels - ic = number of input channels - kt = kernel temporal depth - kh = kernel height - kw = kernel width
- bias: Tensor or None. If None, no bias is applied; otherwise its shape must be [1, oc, 1, 1, 1]. Data type is int32.
- stride: List[int] or None, the stride along each spatial/temporal dimension. Defaults to [1, 1, 1] if None; if provided, length must be 3.
- dilation: List[int] or None, the dilation along each spatial/temporal dimension. Defaults to [1, 1, 1] if None; if provided, length must be 3.
- pad: List[int] or None, the padding for each dimension in [t\_begin, h\_begin, w\_begin, t\_end, h\_end, w\_end] order. Defaults to [0, 0, 0, 0, 0, 0] if None; if provided, length must be 6.
- groups: int, the number of convolution groups. If ic == oc == groups, this is a depthwise 3D conv.
- input\_scale: float, List[float], or None, the quantization scale(s) for the input. If None, uses the scale in tensor\_i; if a list is provided, its length must be ic. (List mode not supported currently.)
- weight\_scale: float, List[float], or None, the quantization scale(s) for the kernel. If None, uses the scale in weight; if a list is provided, its length must be oc.
- output\_scale: float or List[float], the quantization scale(s) for the output. Cannot be None; if a list is provided, its length must be oc. (List mode not supported currently.)
- input\_zp: int, List[int], or None, the zero-point(s) for the input. Defaults to 0 if None; if a list is provided, its length must be ic. (List mode not supported currently.)
- weight\_zp: int, List[int], or None, the zero-point(s) for the kernel. Defaults to 0 if None; if a list is provided, its length must be oc.
- output\_zp: int, List[int], or None, the zero-point(s) for the output. Defaults to 0 if None; if a list is provided, its length must be oc. (List mode not supported currently.)
- out\_dtype: string or None, the output tensor's data type. If None, defaults to int8. Valid values: “int8”, “uint8” .
- out\_name: string or None, the name of the output tensor. If None, a name is generated automatically.

### Return value

Returns a Tensor with the data type determined by out\_dtype.

### Processor support

- BM1688: The data type of input and weight can be INT8/UINT8. The data type of bias is INT32.
- BM1684X: The data type of input and weight can be INT8/UINT8. The data type of bias is INT32.

## matmul

### The interface definition

```
def matmul(input: Tensor,
           right: Tensor,
           bias: Tensor = None,
           right_transpose: bool = False,
           left_transpose: bool = False,
           output_transpose: bool = False,
           keep_dims: bool = True,
           out_dtype: str = None,
           out_name: str = None):
    #pass
```

### Description of the function

Matrix multiplication operation. You can refer to the definitions of matrix multiplication in various frameworks. This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor, the left operand of the matmul. Must have rank  $\geq 2$ , with shape  $[\dots, m, k]$  where m and k are the last two dimensions.
- right: Tensor, the right operand of the matmul. Must have rank  $\geq 2$ , with shape  $[\dots, k, n]$  where k and n are the last two dimensions.
- bias: Tensor or None. If None, no bias is applied; otherwise its shape must be [n].
- left\_transpose: bool, default False. If True, transpose the last two dims of input before multiplication (i.e. swap m and k).

- right\_transpose: bool, default False. If True, transpose the last two dims of right before multiplication (i.e. swap k and n).
- output\_transpose: bool, default False. If True, transpose the last two dims of the result before returning (i.e. swap result's last two dims).
- keep\_dims: bool, default True. If True, the output retains the same rank as the broadcasted inputs; if False, the output is squeezed to a 2-D matrix of shape [M, N].
- out\_dtype: string or None. If None, inherits the data type of input. Valid values: “float32”, “float16” .
- out\_name: string or None. The name of the output tensor. If None, a name is generated automatically.

Notes on shapes and broadcasting: input and right must have the same rank. If rank = 2, a simple matrix-matrix multiply is performed. If rank > 2, a batched matmul is performed: The inner dimensions must match: input.shape[-1] == right.shape[-2]. The batch dims (input.shape[:-2] and right.shape[:-2]) must be broadcastable to a common shape.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16. The input and right data types must be consistent. The bias data type must be FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16. The input and right data types must be consistent.

### matmul\_int

#### The interface definition

```
def matmul_int(input: Tensor,
               right: Tensor,
               bias: Tensor = None,
               right_transpose: bool = False,
               left_transpose: bool = False,
               output_transpose: bool = False,
               keep_dims: bool = True,
               input_zp: Union[int, List[int]] = None,
               right_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
    #pass
```

### Description of the function

Matrix multiplication operation. You can refer to the definitions of matrix multiplication in various frameworks. This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor, the left operand of the matmul. Must have rank  $\geq 2$ , with shape  $[\dots, m, k]$  (i.e. its last two dims are  $[m, k]$ ).
- right: Tensor, the right operand of the matmul. Must have rank  $\geq 2$ , with shape  $[\dots, k, n]$  (i.e. its last two dims are  $[k, n]$ ).
- bias: Tensor or None. If None, no bias is applied; otherwise its shape must be  $[n]$ .
- left\_transpose: bool, default False. If True, transpose the last two dims of input before multiplication (swap m and k).
- right\_transpose: bool, default False. If True, transpose the last two dims of right before multiplication (swap k and n).
- output\_transpose: bool, default False. If True, transpose the last two dims of the result before returning.
- keep\_dims: bool, default True. If True, the output retains the same rank as the broadcasted inputs; if False, the output is squeezed to a 2-D matrix of shape  $[M, N]$ .
- input\_zp: int or List[int], the zero-point(s) for input. Defaults to 0 if None. (List mode not supported currently.)
- right\_zp: int or List[int], the zero-point(s) for right. Defaults to 0 if None. (List mode not supported currently.)
- out\_dtype: string or None. If None, defaults to int32. Valid values: “int32” , “uint32” .
- out\_name: string or None. The name of the output tensor. If None, a name is generated automatically.

Notes on shapes and broadcasting: input and right must have the same rank. If rank = 2, a simple matrix-matrix multiply is performed. If rank > 2, a batched matmul is performed: The inner dimensions must match:  $\text{input.shape}[-1] == \text{right.shape}[-2]$ . The batch dims ( $\text{input.shape}[:-2]$  and  $\text{right.shape}[:-2]$ ) must be broadcastable to a common shape.

### Return value

Returns a Tensor whose data type is specified by out\_dtype.

### Processor support

- BM1688: The input data type can be INT8/UINT8. The bias data type is INT32.
- BM1684X: The input data type can be INT8/UINT8. The bias data type is INT32.

### matmul\_quant

#### The interface definition

```
def matmul_quant(input: Tensor,
                  right: Tensor,
                  bias: Tensor = None,
                  right_transpose: bool = False,
                  keep_dims: bool = True,
                  input_scale: Union[float, List[float]] = None,
                  right_scale: Union[float, List[float]] = None,
                  output_scale: Union[float, List[float]] = None,
                  input_zp: Union[int, List[int]] = None,
                  right_zp: Union[int, List[int]] = None,
                  output_zp: Union[int, List[int]] = None,
                  out_dtype: str = None,
                  out_name: str = None):
    #pass
```

#### Description of the function

Matrix multiplication operation. You can refer to the definitions of matrix multiplication in various frameworks. This operation belongs to **local operations**.

#### Explanation of parameters

- input:Tensor type, representing the left operand; rank  $\geq 2$ , with its last two dims shaped [m, k].
- right:Tensor type, representing the right operand; rank  $\geq 2$ , with its last two dims shaped [k, n].
- bias:Tensor type, representing the bias tensor. If None, no bias is applied; otherwise its shape must be [n].

- right\_transpose:bool type, default False. Specifies whether to transpose the right matrix before computation.
- keep\_dims:bool type, default True. Specifies whether to retain the original number of dims; if False, the output shape is 2-D.
- input\_scale>List[float] or float, representing the quantization scale for input. If None, uses the input tensor's own scale. List[float] not supported.
- right\_scale>List[float] or float, representing the quantization scale for right. If None, uses the right tensor's own scale. List[float] not supported.
- output\_scale>List[float] or float, representing the quantization scale for output. Cannot be None. List[float] not supported.
- input\_zp>List[int] or int, representing the zero-point for input. If None, defaults to 0. List[int] not supported.
- right\_zp>List[int] or int, representing the zero-point for right. If None, defaults to 0. List[int] not supported.
- output\_zp>List[int] or int, representing the zero-point for output. If None, defaults to 0. List[int] not supported.
- out\_dtype:string type or None, representing the output tensor's data type; if None, defaults to int8/uint8. Valid values: int8/uint8.
- out\_name:string type or None, representing the output tensor's name; if None, an internal name is autogenerated.

The ranks of the left and right Tensors must match. If the rank of the Tensors is 2, a matrix-matrix multiplication is performed. If the rank of the Tensors is greater than 2, a batched matrix multiplication is performed. It requires `input.shape[-1] == right.shape[-2]`, and `input.shape[:-2]` and `right.shape[:-2]` must satisfy broadcasting rules.

#### Return value

Returns a Tensor whose data type is specified by `out_dtype`.

#### Processor support

- BM1688: The input data type can be INT8/UINT8. The bias data type is INT32.
- BM1684X: The input data type can be INT8/UINT8. The bias data type is INT32.

### 24.5.2 Base Element-wise Operator

add

#### The interface definition

```
def add(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

#### Description of the function

Element-wise addition operation between tensors.  $tensor_o = tensor_{i0} + tensor_{i1}$ . This operation supports broadcasting. This operation belongs to **local operations**.

#### Explanation of parameters

- tensor\_i0: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- tensor\_i1: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- scale: List[float] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the scales of tensor\_i0, tensor\_i1, and the output.
- zero\_point: List[int] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the zero-points of tensor\_i0, tensor\_i1, and the output.
- out\_dtype: A string or None, representing the data type of the output Tensor. If set to None, it will be consistent with the input data type. Optional values include float32/float16/int8(uint8/int16/int16/int32/uint32).
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor whose data type is specified by out\_dtype or is consistent with the input data type (when one of the inputs is int8, the output defaults to int8 type). When the input is float32/floating16, the output data type must be consistent with the input.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of tensor\_i0 and tensor\_i1 must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of tensor\_i0 and tensor\_i1 must be consistent.

## sub

### The interface definition

```
def sub(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

### Description of the function

Element-wise subtraction operation between tensors.  $tensor_o = tensor_{i0} - tensor_{i1}$ . This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- tensor\_i1: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- scale: List[float] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the scales of tensor\_i0, tensor\_i1, and the output.

- zero\_point: List[int] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the zero-points of tensor\_i0, tensor\_i1, and the output.
- out\_dtype: A string type or None, representing the data type of the output tensor. If None, it is consistent with the input tensors' dtype. The optional parameters are float32/float16/int8/int16/int32.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

#### Return value

Returns a Tensor, and the data type of this Tensor is specified by out\_dtype or is consistent with the input data type. When the input is float32/float16, the output data type must be the same as the input. When the input is int8/uint8/int16/uint16/int32/uint32, the output data type is int8/int16/int32.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of tensor\_i0 and tensor\_i1 must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of tensor\_i0 and tensor\_i1 must be consistent.

### mul

#### The interface definition

```
def mul(tensor_i0: Union[Tensor, Scalar, int, float],  
        tensor_i1: Union[Scalar, int, float],  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_dtype: str = None,  
        out_name: str = None):  
    #pass
```

### Description of the function

Element-wise multiplication operation between tensors.  $tensor\_o = tensor\_i0 * tensor\_i1$ . This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- `tensor_i0`: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- `tensor_i1`: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of `tensor_i0` and `tensor_i1` must be a Tensor.
- `scale`: List[float] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the scales of `tensor_i0`, `tensor_i1`, and the output.
- `zero_point`: List[int] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the zero-points of `tensor_i0`, `tensor_i1`, and the output.
- `out_dtype`: A string or None, representing the data type of the output Tensor. If set to None, it will be consistent with the input data type. Optional values include float32/floating16/int8/uint8/int16/uint16/int32/uint32.
- `out_name`: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor whose data type is specified by `out_dtype` or is consistent with the input data type (when one of the inputs is int8, the output defaults to int8 type). When the input is float32/floating16, the output data type must be consistent with the input.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of `tensor_i0` and `tensor_i1` must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. When the data type is FLOAT16/FLOAT32, the data types of `tensor_i0` and `tensor_i1` must be consistent.

## div

### The interface definition

```
def div(tensor_i0: Union[Tensor, Scalar],  
        tensor_i1: Union[Tensor, Scalar],  
        out_name: str = None):  
    #pass
```

### Description of the function

Element-wise division operation between tensors.  $tensor_o = tensor_i0/tensor_i1$ . This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- tensor\_i1: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## max

### The interface definition

```
def max(tensor_i0: Union[Tensor, Scalar, int, float],  
        tensor_i1: Union[Tensor, Scalar, int, float],  
        scale: List[float]=None,
```

(continues on next page)

(continued from previous page)

```
zero_point: List[int]=None,  
out_dtype: str = None,  
out_name: str = None):  
#pass
```

### Description of the function

Element-wise maximum operation between tensors.  $tensor\_o = \max(tensor\_i0, tensor\_i1)$ . This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- tensor\_i1: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- scale: List[float] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the scales of tensor\_i0, tensor\_i1, and the output.
- zero\_point: List[int] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the zero-points of tensor\_i0, tensor\_i1, and the output.
- out\_dtype: A string or None, representing the data type of the output Tensor. If set to None, it will be consistent with the input data type. Optional values include float32/float16/int8/uint8/int16/uint16/int32/uint32.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor, and the data type of this Tensor is specified by out\_dtype or is consistent with the input data type. When the input is float32/float16, the output data type must be the same as the input. When the input is int8/uint8/int16/uint16/int32/uint32, the output can be any integer type.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT16/UINT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT16/UINT16/INT8/UINT8.

## min

### The interface definition

```
def min(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

### Description of the function

Element-wise minimum operation between tensors.  $tensor_o = \min(tensor_i0, tensor_i1)$ . This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type or Scalar, int, float. It represents the left operand Tensor or Scalar for the input.
- tensor\_i1: Tensor type or Scalar, int, float. It represents the right operand Tensor or Scalar for the input. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- scale: List[float] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the scales of tensor\_i0, tensor\_i1, and the output.
- zero\_point: List[int] type or None, representing the quantization parameters; if None, indicates non-quantized computation; if a List, its length must be 3, corresponding to the zero-points of tensor\_i0, tensor\_i1, and the output.
- out\_dtype: A string or None, representing the data type of the output Tensor. If set to None, it will be consistent with the input data type. Optional values include float32/float16/int8(uint8)/int16(uint16)/int32(uint32).
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor, and the data type of this Tensor is specified by out\_dtype or is consistent with the input data type. When the input is float32/float16, the output data type must be the same as the input. When the input is int8/uint8/int16/uint16/int32/uint32, the output can be any integer type.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT16/UINT16/INT32/UINT32/INT8/UINT8
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT16/UINT16/INT32/UINT32/INT8/UINT8

## add\_shift

### The interface definition

```
def add_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
              is_saturate: bool=True,
              out_name: str = None):
    #pass
```

### Description of the function

Operation Formula  $tensor_o = (tensor_{i0} - tensor_{i1}) \ll shift$ . After adding tensor\_i0 and tensor\_i1 element-wise, a rounded arithmetic shift by shift bits is applied. A positive shift denotes a left shift; a negative shift denotes a right shift. The rounding mode is determined by round\_mode. The sum is first stored in INT64 as an intermediate result, then the rounded arithmetic shift is performed on the INT64 value. The result supports saturation. If tensor\_i0 and tensor\_i1 are signed and tensor\_o is unsigned, saturation is mandatory. This operation supports broadcasting. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor, Scalar, or int type, representing the left-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- tensor\_i1: Tensor, Scalar, or int type, representing the right-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- shift: int type, specifying the number of bits to shift.
- round\_mode: String type, specifying the rounding mode; default is ‘half\_away\_from\_zero’ . Valid values are ‘half\_away\_from\_zero’ , ‘half\_to\_even’ , ‘towards\_zero’ , ‘down’ , and ‘up’ .
- is\_saturate: bool type, indicating whether to apply saturation; default is True.
- out\_dtype: String type or None, specifying the output Tensor data type; if None, defaults to the type of tensor\_i0. Optional values are int8, uint8, int16, uint16, int32, and uint32.
- out\_name: String type or None, specifying the name of the output Tensor; if None, a name is generated automatically.

### Return value

Returns a Tensor. The data type of the Tensor is specified by out\_dtype, or is consistent with the input data type.

### Processor support

- BM1688: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.
- BM1684X: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.

### sub\_shift

#### The interface definition

```
def sub_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
              is_saturate: bool=True,
              out_name: str = None):
    #pass
```

### Description of the function

Operation Formula  $tensor\_o = (tensor\_i0 - tensor\_i1) \ll shift$ . Element-wise subtraction between two tensors followed by a rounded arithmetic shift by shift bits. If shift > 0, performs a left shift; if shift < 0, performs a right shift. The rounding mode is determined by round\_mode. This operation supports broadcasting of input tensors. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor, Scalar, or int type, representing the left-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- tensor\_i1: Tensor, Scalar, or int type, representing the right-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- shift: int type, specifying the number of bits to shift.
- round\_mode: String type, specifying the rounding mode; default is ‘half\_away\_from\_zero’ . Valid values are ‘half\_away\_from\_zero’ , ‘half\_to\_even’ , ‘towards\_zero’ , ‘down’ , and ‘up’ .
- is\_saturate: bool type, indicating whether to apply saturation; default is True.
- out\_dtype: String type or None, specifying the output Tensor’s data type; if None, defaults to tensor\_i0’s type. Optional values are ‘int8’ , ‘int16’ , and ‘int32’ .
- out\_name: String type or None, specifying the name of the output Tensor; if None, a name is generated automatically.

### Return value

Returns a Tensor. The data type of the Tensor is specified by out\_dtype, or is consistent with the input data type.

### Processor support

- BM1688: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.
- BM1684X: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.

## mul\_shift

### The interface definition

```
def mul_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
              is_saturate: bool=True,
              out_name: str = None):
    #pass
```

### Description of the function

Operation Formula  $tensor\_o = (tensor\_i0 * tensor\_i1) << shift$  Subtract the tensors element-wise and then perform a rounded arithmetic shift for the result. When shift is positive, perform a left shift; when shift is negative, perform a right shift. The rounding mode is determined by round\_mode. After multiplying the data for mul\_shift, save the intermediate result as INT64, and then perform a rounded arithmetic shift operation based on INT64. The result supports saturation processing. When tensor\_i0 and tensor\_i1 are signed and tensor\_o is unsigned, the result must be saturated. This operation supports broadcasting of input tensors. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor, Scalar, or int type, representing the left-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- tensor\_i1: Tensor, Scalar, or int type, representing the right-hand input operand. At least one of tensor\_i0 and tensor\_i1 must be a Tensor.
- shift: int type, specifying the number of bits to shift.
- round\_mode: String type, specifying the rounding mode; default is 'half\_away\_from\_zero'. Valid values are 'half\_away\_from\_zero', 'half\_to\_even', 'towards\_zero', 'down', and 'up'.
- is\_saturate: bool type, indicating whether to apply saturation; default is True.
- out\_dtype: String type or None, specifying the output Tensor's data type; if None, defaults to tensor\_i0's type. Optional values are 'int8' /' uint8' /' int16' /' uint16' /' int32' /' uint32' .
- out\_name: String type or None, specifying the name of the output Tensor; if None, a name is generated automatically.

### Return value

Returns a Tensor. The data type of the Tensor is specified by `out_dtype`, or is consistent with the input data type.

### Processor support

- BM1688: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.
- BM1684X: The input data type can be INT32/UINT32/INT16/UINT6/INT8/UINT8.

### copy

#### The interface definition

```
def copy(tensor_i, out_name=None):
    #pass
```

#### Description of the function

The Copy function is applied to copy the input data into the output Tensor. This operation belongs to **global operations**.

#### Explanation of parameters

- `tensor`: A Tensor type, representing the input Tensor.
- `out_name`: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## clamp

### The interface definition

```
def clamp(tensor_i, min, max, out_name = None):  
    #pass
```

### Description of the function

Clipping operation for all elements in the input tensor, restricting values to a specified minimum and maximum range. Values greater than the maximum are truncated to the maximum, and values less than the minimum are truncated to the minimum. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor.
- min\_value: Scalar type, representing the lower bound of the range.
- max\_value: Scalar type, representing the upper bound of the range.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### 24.5.3 Element-wise Compare Operator

gt

#### The interface definition

```
def gt(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

#### Description of the function

Element-wise greater than comparison operation between tensors.  $tensor_o = tensor_{i0} > tensor_{i1} ? 1 : 0$ . This operation supports broadcasting.  $tensor_{i0}$  or  $tensor_{i1}$  can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

#### Explanation of parameters

- $tensor_{i0}$ : Tensor type, representing the left operand input Tensor.
- $tensor_{i1}$ : Tensor type, representing the right operand input Tensor.
- $scale$ : List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of  $tensor_{i0}$ ,  $tensor_{i1}$ , and the output; the scales of  $tensor_{i0}$  and  $tensor_{i1}$  must be identical.
- $zero\_point$ : List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of  $tensor_{i0}$ ,  $tensor_{i1}$ , and the output; the zero\_points of  $tensor_{i0}$  and  $tensor_{i1}$  must be identical.
- $out\_name$ : A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.

It

### The interface definition

```
def lt(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

### Description of the function

Element-wise less than comparison operation between tensors.  $tensor_o = tensor_{i0} < tensor_{i1} : 0$ . This operation supports broadcasting. tensor\_i0 or tensor\_i1 can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input Tensor.
- tensor\_i1: Tensor type, representing the right operand input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of tensor\_i0, tensor\_i1, and the output; the scales of tensor\_i0 and tensor\_i1 must be identical.
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of tensor\_i0, tensor\_i1, and the output; the zero\_points of tensor\_i0 and tensor\_i1 must be identical.

- `out_name`: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

#### Return value

Returns a Tensor with the same data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of `tensor_i0` and `tensor_i1` must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of `tensor_i0` and `tensor_i1` must be consistent.

ge

#### The interface definition

```
def ge(tensor_i0: Tensor,  
       tensor_i1: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,  
       out_name: str = None):  
    #pass
```

#### Description of the function

Element-wise greater than or equal to comparison operation between tensors.  $tensor_o = tensor_{i0} \geq tensor_{i1} : 0$ . This operation supports broadcasting. `tensor_i0` or `tensor_i1` can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

#### Explanation of parameters

- `tensor_i0`: Tensor type, representing the left operand input Tensor.
- `tensor_i1`: Tensor type, representing the right operand input Tensor.
- `scale`: `List[float]` type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of `tensor_i0`, `tensor_i1`, and the output; the scales of `tensor_i0` and `tensor_i1` must be identical.

- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of tensor\_i0, tensor\_i1, and the output; the zero\_points of tensor\_i0 and tensor\_i1 must be identical.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

#### Return value

Returns a Tensor with the same data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.

### le

#### The interface definition

```
def le(tensor_i0: Tensor,  
       tensor_i1: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,  
       out_name: str = None):  
    #pass
```

#### Description of the function

Element-wise less than or equal to comparison operation between tensors.  $tensor_o = tensor_i0 \leq tensor_i1 ? 1 : 0$ . This operation supports broadcasting. tensor\_i0 or tensor\_i1 can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input Tensor.
- tensor\_i1: Tensor type, representing the right operand input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of tensor\_i0, tensor\_i1, and the output; the scales of tensor\_i0 and tensor\_i1 must be identical.
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of tensor\_i0, tensor\_i1, and the output; the zero\_points of tensor\_i0 and tensor\_i1 must be identical.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of tensor\_i0 and tensor\_i1 must be consistent.

### eq

#### The interface definition

```
def eq(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

### Description of the function

Element-wise equality comparison operation between tensors.  $tensor\_o = tensor\_i0 == tensor\_i1 ? 1 : 0$ . This operation supports broadcasting.  $tensor\_i0$  or  $tensor\_i1$  can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

### Explanation of parameters

- $tensor\_i0$ : Tensor type, representing the left operand input Tensor.
- $tensor\_i1$ : Tensor type, representing the right operand input Tensor.
- $scale$ : List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of  $tensor\_i0$ ,  $tensor\_i1$ , and the output; the scales of  $tensor\_i0$  and  $tensor\_i1$  must be identical.
- $zero\_point$ : List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of  $tensor\_i0$ ,  $tensor\_i1$ , and the output; the zero\_points of  $tensor\_i0$  and  $tensor\_i1$  must be identical.
- $out\_name$ : A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of  $tensor\_i0$  and  $tensor\_i1$  must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of  $tensor\_i0$  and  $tensor\_i1$  must be consistent.

ne

### The interface definition

```
def ne(tensor_i0: Tensor,  
       tensor_i1: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,
```

(continues on next page)

(continued from previous page)

```
out_name: str = None):  
#pass
```

### Description of the function

Element-wise not equal to comparison operation between tensors.  $tensor\_o = tensor\_i0! = tensor\_i1?1 : 0$ . This operation supports broadcasting.  $tensor\_i0$  or  $tensor\_i1$  can be assigned as COEFF\_TENSOR. This operation belongs to **local operations**.

### Explanation of parameters

- $tensor\_i0$ : Tensor type, representing the left operand input Tensor.
- $tensor\_i1$ : Tensor type, representing the right operand input Tensor.
- $scale$ : List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three floats corresponding to the scales of  $tensor\_i0$ ,  $tensor\_i1$ , and the output; the scales of  $tensor\_i0$  and  $tensor\_i1$  must be identical.
- $zero\_point$ : List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of three integers corresponding to the zero\_points of  $tensor\_i0$ ,  $tensor\_i1$ , and the output; the zero\_points of  $tensor\_i0$  and  $tensor\_i1$  must be identical.
- $out\_name$ : A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of  $tensor\_i0$  and  $tensor\_i1$  must be consistent.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data types of  $tensor\_i0$  and  $tensor\_i1$  must be consistent.

## gts

### The interface definition

```
def gts(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### Description of the function

Element-wise greater-than comparison operation between tensors and scalars.  $tensor_o = tensor_i0 > scalar_i1?1 : 0$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

### lts

#### The interface definition

```
def lts(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

#### Description of the function

Element-wise less-than comparison between a tensor and a scalar.  $tensor_o = tensor_{i0} < scalar_{i1}$ ?1 : 0. This operation belongs to **local operations**.

#### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

ges

### The interface definition

```
def ges(tensor_i0: Tensor,
       scalar_i1: Union[Scalar, int, float],
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

### Description of the function

Element-wise greater-than-or-equal-to comparison between a tensor and a scalar.  $tensor\_o = tensor\_i0 \geq scalar\_i1?1 : 0$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

les

### The interface definition

```
def les(tensor_i0: Tensor,
       scalar_i1: Union[Scalar, int, float],
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

### Description of the function

Element-wise less-than-or-equal-to comparison between a tensor and a scalar.  $tensor_o = tensor_{i0} \leq scalar_{i1} ? 1 : 0$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

### eqs

#### The interface definition

```
def eqs(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

#### Description of the function

The element-wise equality comparison operation between a tensor and a scalar.  $tensor_o = tensor_i0 == scalar_i1?1 : 0$ . This operation belongs to **local operations**.

#### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

nes

### The interface definition

```
def nes(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### Description of the function

The element-wise inequality comparison operation between a tensor and a scalar.  $tensor\_o = tensor\_i0! = scalar\_i1?1 : 0$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i0: Tensor type, representing the left operand input.
- scalar\_i1: Tensor type, representing the right operand input.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8. The data type of scalar\_i1 is FLOAT32.

#### 24.5.4 Activation Operator

##### relu

###### The interface definition

```
def relu(input: Tensor, out_name: str = None):
    #pass
```

###### Description of the function

The ReLU activation function, implemented on an element-wise basis.  $y = \max(0, x)$ . This operation belongs to **local operations**.

###### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### prelu

#### The interface definition

```
def prelu(input: Tensor, slope : Tensor, out_name: str = None):  
    #pass
```

#### Description of the function

prelu activation function, implements function element by element  $y = \begin{cases} x & x > 0 \\ x * slope & x \leq 0 \end{cases}$ . This operation belongs to **local operations**.

#### Explanation of parameters

- input: Tensor type, representing the input Tensor.
- slope: Tensor type, representing the slope Tensor. Only supports slope as a coeff Tensor.
- out\_name: string type or None, representing the name of the output Tensor; if None, a name will be automatically generated internally.

#### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## leaky\_relu

### The interface definition

```
def leaky_relu(input: Tensor,
               negative_slope: float = 0.01,
               out_name: str = None,
               round_mode : str="half_away_from_zero",):
    #pass
```

### Description of the function

The leaky ReLU activation function, implemented on an element-wise basis.  $y = \begin{cases} x & x > 0 \\ x * params[0] & x \leq 0 \end{cases}$ . This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing the input tensor.
- negative\_slope: float type, representing the negative slope for inputs  $< 0$ ; default is 0.01.
- out\_name: string type or None, the name of the output tensor; if None, a name is auto-generated internally.
- round\_mode: string type, the rounding mode; default is “half\_away\_from\_zero”. Valid values are “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, and “up.”

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## abs

### The interface definition

```
def abs(input: Tensor, out_name: str = None):  
    #pass
```

### Description of the function

The abs absolute value activation function, implemented on an element-wise basis.  $y = |x|$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## In

### The interface definition

```
def ln(input: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,  
       out_name: str = None):  
    #pass
```

### Description of the function

The ln activation function, implemented on an element-wise basis.  $y = \log(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor.
- scale: List[float] type or None, quantization parameter(s). If None, indicates non-quantized computation. If a list, length must be 2, specifying the scales for tensor\_i0 and the output.
- zero\_point: List[int] type or None, quantization parameter(s). If None, indicates non-quantized computation. If a list, length must be 2, specifying the zero points for tensor\_i0 and the output.
- out\_name: string type or None, the name of the output tensor; if None, a name will be automatically generated internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## ceil

### The interface definition

```
def ceil(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### Description of the function

The ceil rounding up activation function, implemented on an element-wise basis.  $y = \lfloor x \rfloor$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor.
- scale: List[float] type or None, quantization parameter(s). · None indicates non-quantized computation. · If a list, it must have length 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameter(s). · None indicates non-quantized computation. · If a list, it must have length 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: string type or None, the name of the output tensor; if None, a name is automatically generated internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### floor

### The interface definition

```
def floor(input: Tensor,  
         scale: List[float]=None,  
         zero_point: List[int]=None,  
         out_name: str = None):  
    #pass
```

### Description of the function

The floor rounding down activation function, implemented on an element-wise basis.  $y = \lceil x \rceil$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale:List[float] type or None, quantization parameters. None indicates non-quantized computation. If a list, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point:List[int] type or None, quantization parameters. None indicates non-quantized computation. If a list, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### round

#### The interface definition

```
def round(input: Tensor, out_name: str = None):
    #pass
```

### Description of the function

The round activation function, which rounds to the nearest integer using the round half up (four-way tie-breaking) method, implemented on an element-wise basis.  $y = \text{round}(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## sin

### The interface definition

```
def sin(input: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,  
       out_name: str = None):  
    #pass
```

### Description of the function

The sin sine activation function, implemented on an element-wise basis.  $y = \sin(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### cos

#### The interface definition

```
def cos(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

#### Description of the function

The cos cosine activation function, implemented on an element-wise basis.  $y = \cos(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### exp

#### The interface definition

```
def exp(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

#### Description of the function

The exp exponential activation function, implemented on an element-wise basis.  $y = e^x$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## tanh

### The interface definition

```
def tanh(input: Tensor,  
         scale: List[float]=None,  
         zero_point: List[int]=None,  
         out_name: str = None,  
         round_mode : str="half_away_from_zero"):  
    #pass
```

### Description of the function

The tanh hyperbolic tangent activation function, implemented on an element-wise basis.  $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale:List[float] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point:List[int] type or None, quantization parameters. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.
- round\_mode:string type, rounding mode. Defaults to “half\_away\_from\_zero” . Allowed values: “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.

### sigmoid

#### The interface definition

```
def sigmoid(input: Tensor,
            scale: List[float]=None,
            zero_point: List[int]=None,
            out_name: str = None,
            round_mode : str="half_away_from_zero"):
    #pass
```

### Description of the function

The sigmoid activation function, implemented on an element-wise basis.  $y = 1/(1 + e^{-x})$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor.
- scale: List[float] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: string type or None, name of the output tensor. If None, a name is auto-generated internally.
- round\_mode: string type, rounding mode. Defaults to “half\_away\_from\_zero” . Allowed values: “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.

### log\_sigmoid

#### The interface definition

```
def log_sigmoid(input: Tensor,
                 scale: List[float]=None,
                 zero_point: List[int]=None,
                 out_name: str = None):
    #pass
```

### Description of the function

The log\_sigmoid activation function, implemented on an element-wise basis.  $y = \log(1/(1 + e^{-x}))$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor.
- scale: List[float] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: string type or None, name of the output tensor. If None, a name is auto-generated internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8.FLOAT16 data is automatically converted to FLOAT32.

### elu

#### The interface definition

```
def elu(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### Description of the function

The ELU (Exponential Linear Unit) activation function, implemented on an element-wise basis.  $y = \begin{cases} x & x \geq 0 \\ e^x - 1 & x < 0 \end{cases}$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

## square

### The interface definition

```
def square(input: Tensor,
           scale: List[float]=None,
           zero_point: List[int]=None,
           out_name: str = None):
    #pass
```

### Description of the function

The square function, implemented on an element-wise basis.  $y = \square x$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### sqrt

#### The interface definition

```
def sqrt(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### Description of the function

The `sqrt` square root activation function, implemented on an element-wise basis.  $y = \sqrt{x}$ . This operation belongs to **local operations**.

### Explanation of parameters

- `tensor`: A Tensor type, representing the input Tensor.
- `scale`: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [`tensor_i0_scale`, `output_scale`].
- `zero_point`: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [`tensor_i0_zero_point`, `output_zero_point`].
- `out_name`: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### rsqrt

#### The interface definition

```
def rsqrt(input: Tensor,  
          scale: List[float]=None,  
          zero_point: List[int]=None,  
          out_name: str = None):  
    #pass
```

### Description of the function

The rsqrt square root takes the deactivation function, implemented on an element-wise basis.  $y = 1/(sqrt{x})$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### silu

#### The interface definition

```
def silu(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

### Description of the function

The silu activation function, implemented on an element-wise basis.  $y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\eta^2} d\eta$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## swish

### The interface definition

```
def swish(input: Tensor,
          beta: float,
          scale: List[float]=None,
          zero_point: List[int]=None,
          round_mode: str = "half_away_from_zero",
          out_name: str = None):
    #pass
```

### Description of the function

The swish activation function, implemented on an element-wise basis.  $y = x * (1 / (1 + e^{-x * beta}))$ . This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing the input tensor.
- beta: Scalar or float type, representing the  $\beta$  value.
- scale: List[float] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [input\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization parameter. None indicates non-quantized computation. If a List, length must be 2, specifying [input\_zero\_point, output\_zero\_point].
- round\_mode: string type, rounding mode. Defaults to “half\_away\_from\_zero”. Allowed values: “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .
- out\_name: string type or None, name of the output tensor. If None, a name is auto-generated internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### erf

#### The interface definition

```
def erf(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### Description of the function

The erf activation function, for the corresponding elements  $x$  and  $y$  at the same positions in the input and output Tensors, is implemented on an element-wise basis.  $y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\eta^2} d\eta$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

## tan

### The interface definition

```
def tan(input: Tensor, out_name: str = None):
    #pass
```

### Description of the function

The tan tangent activation function, implemented on an element-wise basis.  $y = \tan(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.

## softmax

### The interface definition

```
def softmax(input: Tensor,
            axis: int,
            out_name: str = None):
    #pass
```

### Description of the function

The softmax activation function, which normalizes an input vector into a probability distribution consisting of probabilities proportional to the exponentials of the input numbers.  $tensor_o = \exp(tensor_i) / \sum(\exp(tensor_i), axis)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- axis: An int type, representing the axis along which the operation is performed.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### softmax\_int

#### The interface definition

```
def softmax_int(input: Tensor,
                axis: int,
                scale: List[float],
                zero_point: List[int] = None,
                out_name: str = None,
                round_mode : str="half_away_from_zero"):  
    #pass
```

#### Description of the function

Softmax fixed-point operation. Please refer to the softmax definition in each framework.

```
for i in range(256)
    table[i] = exp(scale[0] * i)

for n,h,w in N,H,W
    max_val = max(input[n,c,h,w] for c in C)
    sum_exp = sum(table[max_val - input[n,c,h,w]] for c in C)
    for c in C
        prob = table[max_val - input[n,c,h,w]] / sum_exp
        output[n,c,h,w] = saturate(int(round(prob * scale[1])) + zero_point[1]), F
    ↵其中saturate饱和到output数据类型
```

Among them, “table” represents table lookup.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor.
- axis: int type, axis along which the operation is performed.
- scale: List[float] type, quantization scales for input and output. Must be of length 2, specifying [input\_scale, output\_scale].
- zero\_point: List[int] type or None, quantization zero points for input and output. Must match the length of scale. If None, defaults to [0, 0].
- out\_name: string type or None, name of the output tensor. If None, a name is auto-generated internally.
- round\_mode: string type, rounding mode. Defaults to “half\_away\_from\_zero”. Allowed values: “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be INT8/UINT8.
- BM1684X: The input data type can be INT8/UINT8.

### mish

#### The interface definition

```
def mish(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### Description of the function

The Mish activation function, implemented on an element-wise basis.  $y = x * \tanh(\ln(1 + e^x))$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

### hswish

#### The interface definition

```
def hswish(input: Tensor,
           scale: List[float]=None,
           zero_point: List[int]=None,
           out_name: str = None):
    #pass
```

### Description of the function

The h-swish activation function, implemented on an element-wise basis.  $y = \begin{cases} 0 & x \leq -3 \\ x & x \geq 3 \\ x * ((x + 3) / 6) & -3 < x < 3 \end{cases}$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### arccos

#### The interface definition

```
def arccos(input: Tensor, out_name: str = None):
    #pass
```

### Description of the function

The arccosine (inverse cosine) activation function, implemented on an element-wise basis.  $y = \arccos(x)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.

## arctanh

### The interface definition

```
def arctanh(input: Tensor, out_name: str = None):
    #pass
```

### Description of the function

The arctanh (inverse hyperbolic tangent) activation function, implemented on an element-wise basis.  $y = \operatorname{arctanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32. FLOAT16 data is automatically converted to FLOAT32.

## sinh

### The interface definition

```
def sinh(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### Description of the function

The sinh (hyperbolic sine) activation function, implemented on an element-wise basis.  $y = \sinh(x) = \frac{e^x - e^{-x}}{2}$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].

- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

#### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32.FLOAT16 data is automatically converted to FLOAT32.

### cosh

#### The interface definition

```
def cosh(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

#### Description of the function

The cosh (hyperbolic cosine) activation function, implemented on an element-wise basis.  $y = \cosh(x) = \frac{e^x + e^{-x}}{2}$ . This operation belongs to **local operations**.

#### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].

- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32. FLOAT16 data is automatically converted to FLOAT32.

### sign

#### The interface definition

```
def sign(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

#### Description of the function

The sign activation function, implemented on an element-wise basis.  $y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$ .

This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## gelu

### The interface definition

```
def gelu(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None,
         round_mode : str="half_away_from_zero"):
    #pass
```

### Description of the function

The GELU (Gaussian Error Linear Unit) activation function, implemented on an element-wise basis.  $y = x * 0.5 * (1 + \operatorname{erf}(\frac{x}{\sqrt{2}}))$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.
- round\_mode: string type, rounding mode. Defaults to “half\_away\_from\_zero”. Allowed values: “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

## hsigmoid

### The interface definition

```
def hsigmoid(input: Tensor,  
            scale: List[float]=None,  
            zero_point: List[int]=None,  
            out_name: str = None):  
    #pass
```

### Description of the function

The hsigmoid (hard sigmoid) activation function, implemented on an element-wise basis.  $y = \min(1, \max(0, \frac{x}{6} + 0.5))$ . This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the input Tensor.
- scale: List[float] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two floats [tensor\_i0\_scale, output\_scale].
- zero\_point: List[int] type or None, specifying quantization parameters. A value of None indicates non-quantized computation. If provided, it must be a list of two integers [tensor\_i0\_zero\_point, output\_zero\_point].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same shape and data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.
- BM1684X: The input data type can be FLOAT32/INT8/UINT8. FLOAT16 data is automatically converted to FLOAT32.

## 24.5.5 Data Arrange Operator

### permute

#### The interface definition

```
def permute(input:tensor,  
           order:Union[List[int], Tuple[int]],  
           out_name:str=None):  
    #pass
```

### Description of the function

Permute the dimensions of the input Tensor according to the permutation parameter.

For example: Given an input shape of (6, 7, 8, 9) and a permutation parameter order of (1, 3, 2, 0), the output shape will be (7, 9, 8, 6). This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing input Tensor.
- order: List[int] or Tuple[int] type, representing permutation order. The length of order should be the same as the dimensions of input tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

### tile

#### The interface definition

```
def tile(tensor_i: Tensor,  
        reps: Union[List[int], Tuple[int]],  
        out_name: str = None):  
    #pass
```

### Description of the function

Repeat the data by copying it along the specified dimension(s). This operation is considered a **restricted local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- reps: A List[int] or Tuple[int] indicating the number of copies for each dimension. The length of reps must match the number of dimensions of the tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

### broadcast

#### The interface definition

```
def broadcast(input: Tensor,
             reps: Union[List[int], Tuple[int]],
             out_name: str = None):
    #pass
```

### Description of the function

Repeat the data by copying it along the specified dimension(s). This operation is considered a **restricted local operation**.

### Explanation of parameters

- input: Tensor type, representing the input tensor for the operation.
- reps: A List[int] or Tuple[int] indicating the number of copies for each dimension. The length of reps must match the number of dimensions of the tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

### concat

#### The interface definition

```
def concat(inputs: List[Tensor],  
          scales: Optional[Union[List[float],List[int]]] = None,  
          zero_points: Optional[List[int]] = None,  
          axis: int = 0,  
          out_name: str = None,  
          dtype="float32",  
          round_mode: str="half_away_from_zero"):  
    #pass
```

#### Description of the function

Concatenate multiple tensors along the specified axis.

This operation is considered a **restricted local operation**.

### Explanation of parameters

- inputs: A List[Tensor] type, containing multiple tensors. All tensors must have the same data type and the same number of shape dimensions.
- scales: An optional Union[List[float], List[int]] type, containing multiple input scales and one output scale, where the last element is the scale for the output.
- zero\_points: An optional List[int] type, containing multiple input zero points and one output zero point, with the last one being the zero point for the output.
- axis: An int type, indicating the axis along which the concatenation operation will be performed.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.
- dtype: A string type, defaulting to “float32” .
- round\_mode: String type, representing rounding type. default to “half\_away\_from\_zero” .

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

## split

### The interface definition

```
def split(input:tensor,
          axis:int=0,
          num:int=1,
          size:Union[List[int], Tuple[int]]=None,
          out_name:str=None):
    #pass
```

### Description of the function

**Split the input tensor into multiple tensors along the specified axis. If size is not empty, the dimensions of the split tensors are determined by size.**

Otherwise, the tensor is split into num equal parts along the specified axis, assuming the tensor's size along that axis is divisible by num.

This operation belongs to **local operations**.

### Explanation of parameters

- input: A Tensor type, indicating the tensor that is to be split.
- axis: An int type, indicating the axis along which the tensor will be split.
- num: An int type, indicating the number of parts to split the tensor into.
- size: A List[int] or Tuple[int] type. When not splitting evenly, this specifies the size of each part. For even splitting, it can be set to empty.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a List[Tensor], where each Tensor has the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

### pad

### The interface definition

```
def pad(input:tensor,
        method='constant',
        value:Union[Scalar, Variable, None]=None,
        padding:Union[List[int], Tuple[int], None]=None,
        out_name:str=None):
    #pass
```

### Description of the function

Padding the input tensor.

This operation belongs to **local operations**.

### Explanation of parameters

- input: A Tensor type, indicating the tensor that is to be padded.
- method: string type, representing the padding method. Optional values are “constant”, “reflect”, “symmetric” or “edge” .
- value: A Scalar, Variable type, or None, representing the value to be filled. The data type is consistent with that of the tensor.
- padding: A List[int], Tuple[int], or None. If padding is None, a zero-filled list of length  $2 * \text{len}(\text{tensor.shape})$  is used. For example, the padding of a hw 2D Tensor is [h\_top, w\_left, h\_bottom, w\_right].
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

### repeat

#### The interface definition

```
def repeat(tensor_i:Tensor,  
          reps:Union[List[int], Tuple[int]],  
          out_name:str=None):  
    #pass
```

### Description of the function

Duplicate data along a specified dimension. Functionally equivalent to tile. This operation is considered a **restricted local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- reps: A List[int] or Tuple[int] type, representing the number of replications for each dimension. The length of reps must be consistent with the number of dimensions of the tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

## extract

### Definition

```
def extract(input: Tensor,  
           start: Union[List[int], Tuple[int]] = None,  
           end: Union[List[int], Tuple[int]] = None,  
           stride: Union[List[int], Tuple[int]] = None,  
           out_name: str = None)
```

### Description

Extract slice of input tensor. This operation is considered a **restricted local operation**.

### Parameters

- input: Tensor type, representing input tensor.
- start: A list or tuple of int, or None, representing the start of slice. If set to None, start is filled all with 0.
- end: A list or tuple of int, or None, representing the end of slice. If set to None, end is given as shape of input.
- stride: A list or tuple of int, or None, representing the stride of slice. If set to None, stride is filled all with 1.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns a Tensor, whose data type is same of that of table.

### Processor Support

- BM1688: Data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: Data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

## roll

### Definition

```
def roll(input:Tensor,  
        shifts: Union[int, List[int], Tuple[int]],  
        dims: Union[int, List[int], Tuple[int]] = None,  
        out_name:str=None):  
    #pass
```

### Description

Roll the tensor input along the given dimension(s). Elements that are shifted beyond the last position are re-introduced at the first position. If dims is None, the tensor will be flattened before rolling and then restored to the original shape. This operation is considered a **restricted local operation**.

### Parameters

- input: Tensor type. the input tensor.
- shifts: int, a list or tuple of int. the number of places by which the elements of the tensor are shifted. If shifts is a tuple.
- dims: int, a list or tuple of int or None. Axis along which to roll.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns a Tensor with the same data type as the input Tensor.

### Processor Support

- BM1688: Data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.
- BM1684X: Data type can be FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16.

## 24.5.6 Sort Operator

### arg

#### The interface definition

```
def arg(input: Tensor,
        method: str = "max",
        axis: int = 0,
        keep_dims: bool = True,
        out_name: str = None):
    #pass
```

### Description of the function

Translate: For the input tensor, find the maximum or minimum values along the specified axis, output the corresponding indices, and set the dimension of that axis to 1. This operation is considered a **restricted local operation**.

### Explanation of parameters

- input: Tensor type, representing the Tensor to be operated on.
- method: A string type, indicating the method of operation, options include ‘max’ and ‘min’ .
- axis: An integer, indicating the specified axis. Default to 0.
- keep\_dims: A boolean, indicating whether to keep the specified axis after the operation. The default value is True, which means to keep it (in this case, the length of that axis is 1).
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns two Tensors, the first Tensor represents indices, of type int32; and the second Tensor represents values, the type of which will be the same as the type of the input.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## topk

### Definition

```
def topk(input: Tensor,  
         axis: int,  
         k: int,  
         out_name: str = None):
```

### Description

Find top k numbers after sorted

### Parameters

- input: Tensor type, representing the input tensor.
- axis: Int type, representing axis used in sorting.
- k: Int type, representing the number of top values along axis.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns two Tensors: the first one represents the values, whose data type is the same as that of the input tensor while the second one represents the indices in input tensor after sorted along axis.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## sort

### Definition

```
def sort(input: Tensor,  
        axis: int = -1,  
        descending : bool = True,  
        out_name = None)
```

### Description

Sort input tensor along axis then return the sorted tensor and corresponding indices.

### Parameters

- input: Tensor type, representing input.
- axis: Int type, representing the axis used in sorting. (Recently, only support axis == -1)
- descending: Bool type, representing whether it is sorted descending or not.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns two Tensors: data type of the first is the same of that of input, and data type of the second is INT32.

### Processor Support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### argsort

#### Definition

```
def argsort(input: Tensor,
            axis: int = -1,
            descending : bool = True,
            out_name : str = None)
```

### Description

Sort input tensor along axis then return the corresponding indices of sorted tensor.

### Parameters

- input: Tensor type, representing input.
- axis: Int type, representing the axis used in sorting. (Recently, only support axis == -1)
- descending: Bool type, representing whether it is sorted descending or not.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns one Tensor whose data type is INT32.

### Processor Support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### sort\_by\_key (TODO)

#### Definition

```
def sort_by_key(input: Tensor,  
               key: Tensor,  
               axis: int = -1,  
               descending : bool = True,  
               out_name = None)
```

### Description

Sort input tensor by key along axis then return the sorted tensor and corresponding keys.

### Parameters

- input: Tensor type, representing input.
- key: Tensor type, representing key.
- axis: Int type, representing the axis used in sorting.
- descending: Bool type, representing whether it is sorted descending or not.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns two Tensors: data type of the first is the same of that of input, and data type of the second is is the same of that of key.

### Processor Support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## 24.5.7 Shape About Operator

### squeeze

#### The interface definition

```
def squeeze(tensor_i: Tensor, axis: Union[Tuple[int], List[int]], out_name: str = None):  
    #pass
```

### Description of the function

The operation reduces dimensions by removing axes with a size of 1 from the shape of the input. If no axes (axis) are specified, it removes all axes that have a size of 1. This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- axis: A List[int] or Tuple[int] type, indicating the specified axes.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## reshape

### The interface definition

```
def reshape(tensor: Tensor, new_shape: Union[Tuple[int], List[int], Tensor], out_
           ↴name: str = None):
    #pass
```

### Description of the function

Translate: Perform a reshape operation on the input tensor. This operation belongs to **local operations**.

### Explanation of parameters

- tensor: A Tensor type, representing the tensor for the input operation.
- new\_shape: A List[int], Tuple[int], or Tensor type, representing the shape after transformation.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### shape\_fetch

#### The interface definition

```
def shape_fetch(tensor_i: Tensor,
               begin_axis: int = None,
               end_axis: int = None,
               step: int = 1,
               out_name: str = None):
    #pass
```

#### Description of the function

To extract the shape information of an input tensor between specified axes (axis). This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- begin\_axis: An int type, indicating the axis to start from.
- end\_axis: An int type, indicating the axis to end at.
- step: An int type, indicating the step size.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the data type INT32.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

## unsqueeze

### The interface definition

```
def unsqueeze(input: Tensor, axes: List[int] = [1,2], out_name: str = None):  
    #pass
```

### Description of the function

The operation adds dimensions by adding axes with a size of 1 from the shape of the input. This operation belongs to **local operations**.

### Explanation of parameters

- input: Tensor type, representing the input tensor for the operation.
- axis: A List[int] or Tuple[int] type, indicating the specified axes.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

**Return value**

Returns a Tensor with the same data type as the input Tensor.

**Processor support**

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### 24.5.8 Quant Operator

#### requant\_fp\_to\_int

**The interface definition**

```
def requant_fp_to_int(tensor_i,
                      scale,
                      offset,
                      requant_mode,
                      out_dtype,
                      out_name = None,
                      round_mode='half_away_from_zero'):
```

**Description of the function**

Quantizes the input tensor.

When requant\_mode equals 0, the corresponding calculation for this operation is:

```
output = saturate(int(round(input * scale)) + offset),
Where `saturate` refers to saturation to the data type of the output.
```

- For the BM1684X: The input data type can be FLOAT32, and the output data type can be INT16, UINT16, INT8, or UINT8.

When requant\_mode equals 1, the corresponding calculation formula for this operation is:

```
output = saturate(int(round(float(input) * scale + offset))),
Where `saturate` refers to saturation to the data type of the output.
```

- For the BM1684X: The input data type can be INT32, INT16, or UINT16, and the output data type can be INT16, UINT16, INT8, or UINT8.

This operation belongs to **local operations**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor with 3 to 5 dimensions.
- scale: Either a List[float] or float type, representing the quantization coefficient.
- offset: When requant\_mode == 0, either a List[int] or int type; when requant\_mode == 1, either a List[float] or float type. Represents the output offset.
- requant\_mode: An int type, representing the quantization mode.
- round\_mode: A string type, representing the rounding mode. The default is “half\_away\_from\_zero”. The possible values for round\_mode are “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .
- out\_dtype: A string type, representing the data type of the output tensor.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor. The data type of this Tensor is determined by out\_dtype.

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

### requant\_fp

#### The interface definition

```
def requant_fp(tensor_i: Tensor,  
               scale: Union[float, List[float]],  
               offset: Union[float, List[float]],  
               out_dtype: str,  
               out_name: str=None,  
               round_mode: str='half_away_from_zero',  
               first_round_mode: str='half_away_from_zero'):
```

### Description of the function

Quantizes the input tensor.

The calculation formula for this operation is:

```
output = saturate(int(round(float(input) * scale + offset))),  
where saturate saturates to the output data type.
```

This operation is a **local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor, with 3-5 dimensions.
- scale: List[float] or float, representing the quantization scale.
- offset: List[int] or int, representing the output offset.
- out\_dtype: String type, representing the data type of the input tensor. The data type can be “int16” /” uint16” /” int8” /” uint8” .
- out\_name: String type or None, representing the name of the output tensor. When set to None, the name will be automatically generated internally.
- round\_mode: String type, representing the rounding mode. Default is “half\_away\_from\_zero” . The round\_mode can take values of “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .
- first\_round\_mode: String type, representing the rounding mode used for quantizing tensor\_i previously. Default is “half\_away\_from\_zero” . The first\_round\_mode can take values of “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

### Return Value

Returns a Tensor. The data type of this Tensor is determined by out\_dtype.

### Processor support

- BM1688: Support input datatype: INT32/INT16/UINT16.
- BM1684X: Support input datatype: INT32/INT16/UINT16.

## requant\_int

### The interface definition

```
def requant_int(tensor_i: Tensor,
    mul: Union[int, List[int]],
    shift: Union[int, List[int]],
    offset: Union[int, List[int]],
    requant_mode: int,
    out_dtype: str="int8",
    out_name=None,
    round_mode='half_away_from_zero', rq_axis:int = 1, fuse_rq_to_matmul:bool = False):
    #pass
```

### Description of the function

Quantize the input tensor.

#### computation mode

When requant\_mode == 0, the corresponding computation is: output = shift > 0 ? (input << shift) : input output = saturate((output \* multiplier) >> 31), where >> is round\_half\_up, saturate to INT32 output = shift < 0 ? (output >> -shift) : output, where >> rounding mode is determined by round\_mode output = saturate(output + offset), where saturate to the output data type BM1684X: Input data type can be INT32, output data type can be INT32/INT16/INT8 BM1688: Input data type can be INT32, output data type can be INT32/INT16/INT8

When requant\_mode == 1, the corresponding computation is: output = saturate((input \* multiplier) >> 31), where >> is round\_half\_up, saturate to INT32 output = saturate(output >> -shift + offset), where >> rounding mode is determined by round\_mode, saturate to the output data type BM1684X: Input data type can be INT32, output data type can be INT32/INT16/INT8 BM1688: Input data type can be INT32, output data type can be INT32/INT16/INT8

When requant\_mode == 2 (recommended), the corresponding computation is: output = input \* multiplier output = shift > 0 ? (output << shift) : (output >> -shift), where >> rounding mode is determined by round\_mode output = saturate(output + offset), where saturate to the output data type BM1684X: Input data type can be INT32/INT16/UINT16, output data type can be INT16/UINT16/INT8/UINT8 BM1688: Input data type can be INT32/INT16/UINT16, output data type can be INT16/UINT16/INT8/UINT8

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor, 3-5 dimensions.
- mul: List[int] or int, representing the quantization multiplier coefficients.
- shift: List[int] or int, representing the quantization shift coefficients. Right shift is negative, left shift is positive.
- offset: List[int] or int, representing the output offset.
- requant\_mode: int, representing the quantization mode.
- round\_mode: string, representing the rounding mode. Default is “half\_up” .
- out\_dtype: string or None, representing the output tensor type. None means the output data type is “int8” .
- out\_name: string or None, representing the output tensor name. If None, the name will be generated automatically.
- rq\_axis: int, representing the axis on which to apply requant.
- fuse\_rq\_to\_matmul: bool, indicating whether to fuse requant into matmul. Default is False.

### Return value

Returns a tensor. The data type of this tensor is determined by out\_dtype.

### Processor support

- BM1684X
- BM1688

### dequant\_int\_to\_fp

#### The interface definition

```
def dequant_int_to_fp(tensor_i: Tensor,
                      scale: Union[float, List[float]], offset: Union[int, List[int]], float, List[float]],
                      out_dtype: str="float32", out_name: str=None, round_mode:
                      str='half_away_from_zero'):
```

### Description of the function

Dequantizes the input tensor.

The calculation formula for this operation is:

```
::  
    output = (input - offset) * scale
```

This operation is a **local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor with 3-5 dimensions.
- scale: List[float] or float, representing the quantization scale.
- offset: List[int] or int, representing the output offset.
- out\_dtype: String type, representing the output tensor type. Default output data type is “float32”. For input data types int8/uint8, the values can be “float16”, “float32”. For input types int16/uint16, the output type can only be “float32” .
- out\_name: String type or None, representing the name of the output tensor. If set to None, the name will be automatically generated internally.
- round\_mode: String type, representing the rounding mode. Default is “half\_away\_from\_zero”. The round\_mode can take values of “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .

### Return value

Returns a Tensor. The data type of this Tensor is specified by out\_dtype.

### Processor support

- BM1684X: Input data types can be INT16/UINT16/INT8/UINT8.

## dequant\_int

### The interface definition

```
def dequant_int(tensor_i: Tensor,  
    mul: Union[int, List[int]], shift: Union[int, List[int]], offset: Union[int, List[int]],  
    lshift: int, requant_mode: int, out_dtype: str="int8", out_name=None,  
    round_mode='half_up'):
```

### Description of the function

Dequantizes the input tensor.

When requant\_mode==0, the calculation formula for this operation is:

```
::  
    output = (input - offset) * multiplier output = saturate(output >> -shift)
```

BM1684X: Input data types can be INT16/UINT16/INT8/UINT8, output data types can be INT32/INT16/UINT16.

When requant\_mode==1, the calculation formula for this operation is:

```
::  
    output = ((input - offset) * multiplier) << lshift output = saturate(output  
>> 31) output = saturate(output >> -shift)
```

BM1684X: Input data types can be INT16/UINT16/INT8/UINT8, output data types can be INT32/INT16/INT8.

This operation is a **local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor with 3-5 dimensions.
- mul: List[int] or int, representing the quantization multiplier.
- shift: List[int] or int, representing the quantization shift. Negative for right shift, positive for left shift.
- offset: List[int] or int, representing the output offset.
- lshift: int, representing the left shift coefficient.
- requant\_mode: int, representing the quantization mode. Values can be 0 or 1, where 0 is “Normal” and 1 is “TFLite” .
- round\_mode: String type, representing the rounding mode. Default is “half\_up” , with options “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

- out\_dtype: String type, representing the input tensor type. Default is “int8” .
- out\_name: String type or None, representing the name of the output tensor. If set to None, the name will be automatically generated internally.

### Return value

Returns a Tensor. The data type of this Tensor is determined by out\_dtype.

### Processor support

- BM1684X

### cast

#### The interface definition

```
def cast(tensor_i: Tensor,  
        out_dtype: str = 'float32',  
        out_name: str = None,  
        round_mode: str = 'half_away_from_zero'):
```

#### Description of the function

Converts the input tensor tensor\_i to the specified data type out\_dtype, and rounds the data according to the specified rounding mode round\_mode. Note that this operator cannot be used alone and must be used in conjunction with other operators.

#### Explanation of parameters

- tensor\_i: Tensor type, representing the input Tensor.
- out\_dtype: str = ‘float32’ , the data type of the output tensor, default is float32.
- out\_name: str = None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.
- round\_mode: str = ‘half\_away\_from\_zero’ , the rounding mode, default is half\_away\_from\_zero. Possible values are “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” . Note that this function does not support the rounding modes “half\_up” and “half\_down” .

### Return value

Returns a Tensor whose data type is determined by the input out\_dtype.

### Processor Support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.

## 24.5.9 Up/Down Scaling Operator

### maxpool2d

#### The interface definition

```
def maxpool2d(input: Tensor,
               kernel: Union[List[int], Tuple[int], None] = None,
               stride: Union[List[int], Tuple[int], None] = None,
               pad: Union[List[int], Tuple[int], None] = None,
               ceil_mode: bool = False,
               scale: List[float] = None,
               zero_point: List[int] = None,
               out_name: str = None,
               round_mode: str = "half_away_from_zero"):
    #pass
```

#### Description of the function

Performs Max Pooling on the input Tensor. The Max Pooling 2d operation can refer to the maxpool2d operator of each framework. This operation is a **local operation**.

#### Explanation of parameters

- input: Tensor type, indicating the input operation Tensor.
- kernel: List[int] or Tuple[int] type or None. If None is entered, global\_pooling is used. If not None, the length of this parameter is required to be 2.
- stride: List[int] or Tuple[int] type or None, indicating the step size. If None is entered, the default value [1,1] is used. If not None, the length of this parameter is required to be 2.

- pad: List[int] or Tuple[int] type or None, indicating the padding size. If None is entered, the default value [0,0,0,0] is used. If not None, the length of this parameter is required to be 4.
- ceil: bool type, indicating whether to round up when calculating the output shape.
- scale: List[float] type or None, quantization parameter. None is used to represent non-quantized calculation. If it is a List, the length is 2, which are the scales of input and output respectively.
- zero\_point: List[int] type or None, offset parameter. None is used to represent non-quantized calculation. If it is a List, the length is 2, which are the zero\_points of input and output respectively.
- out\_name: string type or None, indicating the name of the output Tensor. If it is None, the name will be automatically generated internally.
- round\_mode: string type, indicates the rounding mode for the second time when the input and output Tensors are quantized. The default value is ‘half\_away\_from\_zero’. The value range of round\_mode is “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

#### Return value

Returns a Tensor with the same data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

#### maxpool2d\_with\_mask

##### The interface definition

```
def maxpool2d_with_mask(input: Tensor,
                        kernel: Union[List[int], Tuple[int], None] = None,
                        stride: Union[List[int], Tuple[int], None] = None,
                        pad: Union[List[int], Tuple[int], None] = None,
                        ceil_mode: bool = False,
                        out_name: str = None,
                        mask_name: str = None):
    #pass
```

### Description of the function

Perform Max pooling on the input Tensor and output its mask index. Please refer to the pooling operations under various frameworks. This operation belongs to **local operation**.

### Explanation of parameters

- input: Tensor type, indicating the input operation Tensor.
- kernel: List[int] or Tuple[int] type or None. If None is entered, global\_pooling is used. If not None, the length of this parameter is required to be 2.
- pad: List[int] or Tuple[int] type or None. Indicates the padding size. If None is entered, the default value [0,0,0,0] is used. If not None, the length of this parameter is required to be 4.
- stride: List[int] or Tuple[int] type or None. Indicates the stride size. If None is entered, the default value [1,1] is used. If not None, the length of this parameter is required to be 2.
- ceil\_mode: bool type, indicating whether to round up when calculating the output shape.
- out\_name: string type or None. Indicates the name of the output Tensor. If None, the name is automatically generated internally.
- mask\_name: string type or None. Indicates the name of the output Mask. If None, the name is automatically generated internally.

### Return value

Returns two Tensors, one of which has the same data type as the input Tensor and the other returns a coordinate Tensor, which records the coordinates selected when using comparison operation pooling.

### Processor support

- BM1688: The input data type can be FLOAT32
- BM1684X: The input data type can be FLOAT32

## maxpool3d

### The interface definition

```
def maxpool3d(input: Tensor,  
              kernel: Union[List[int],int,Tuple[int, ...]] = None,  
              stride: Union[List[int],int,Tuple[int, ...]] = None,  
              pad: Union[List[int],int,Tuple[int, ...]] = None,  
              ceil_mode: bool = False,  
              scale: List[float] = None,  
              zero_point: List[int] = None,  
              out_name: str = None,  
              round_mode : str="half_away_from_zero"):  
    #pass
```

### Description of the function

Performs Max Pooling on the input Tensor. The Max Pooling 3d operation can refer to the maxpool3d operator of each framework. This operation is a **local operation**.

### Explanation of parameters

- input: Tensor type, representing the input tensor for the operation.
- kernel: List[int] or Tuple[int] or int or None, if None, global pooling is used. If not None and a single integer is provided, it indicates the same kernel size in three dimensions. If a List or Tuple is provided, its length must be 3.
- pad: List[int] or Tuple[int] or int or None, represents the padding size. If None, the default value [0,0,0,0,0,0] is used. If not None and a single integer is provided, it indicates the same padding size in three dimensions. If a List or Tuple is provided, its length must be 6.
- stride: List[int] or Tuple[int] or int or None, represents the stride size. If None, the default value [1,1,1] is used. If not None and a single integer is provided, it indicates the same stride size in three dimensions. If a List or Tuple is provided, its length must be 3.
- ceil\_mode: bool type, indicates whether to round up when calculating the output shape.
- scale: List[float] type or None, quantization parameters. If None, non-quantized computation is performed. If a List is provided, its length must be 2, representing the scale for input and output respectively.
- zero\_point: List[int] type or None, offset parameters. If None, non-quantized computation is performed. If a List is provided, its length must be 2, representing the zero point for input and output respectively.

- out\_name: string type or None, represents the name of the output Tensor. If None, a name will be automatically generated internally.
- round\_mode: string type, indicates the rounding mode for the second time when the input and output Tensors are quantized. The default value is ‘half\_away\_from\_zero’. The value range of round\_mode is “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up” .

#### Return value

Returns a Tensor with the same data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8/UINT8.

### avgpool2d

#### The interface definition

```
def avgpool2d(input: Tensor,
              kernel: Union[List[int], Tuple[int, None]] = None,
              stride: Union[List[int], Tuple[int, None]] = None,
              pad: Union[List[int], Tuple[int, None]] = None,
              ceil_mode: bool = False,
              scale: List[float] = None,
              zero_point: List[int] = None,
              out_name: str = None,
              count_include_pad: bool = False,
              round_mode: str = "half_away_from_zero",
              first_round_mode: str = "half_away_from_zero"):  
    #pass
```

#### Description of the function

Performs Avg Pooling on the input Tensor. The Avg Pooling 2d operation can refer to the avgpool2d operator of each framework. This operation is a **local operation**.

### Explanation of parameters

- input: Tensor type, indicating the input operation Tensor.
- kernel: List[int] or Tuple[int] type or None. If None is entered, global\_pooling is used. If not None, the length of this parameter is required to be 2.
- stride: List[int] or Tuple[int] type or None, indicating the step size. If None is entered, the default value [1,1] is used. If not None, the length of this parameter is required to be 2.
- pad: List[int] or Tuple[int] type or None, indicating the padding size. If None is entered, the default value [0,0,0,0] is used. If not None, the length of this parameter is required to be 4.
- ceil\_mode: bool type, indicating whether to round up when calculating the output shape.
- scale: List[float] type or None, quantization parameter. None is used to represent non-quantized calculation. If it is a List, the length is 2, which are the scales of input and output respectively.
- zero\_point: List[int] type or None, offset parameter. None is used to represent non-quantized calculation. If it is a List, the length is 2, which are the zero\_points of input and output respectively.
- out\_name: string type or None, indicating the name of the output Tensor. If it is None, the name will be automatically generated internally.
- count\_include\_pad: Bool type, indicating whether the pad value is included when calculating the average value. The default value is False.
- round\_mode: String type, when the input and output Tensors are quantized, it indicates the second rounding mode. The default value is ‘half\_away\_from\_zero’ .The value range of round\_mode is “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .
- first\_round\_mode: String type, when the input and output Tensors are quantized, it indicates the first rounding mode. The default value is ‘half\_away\_from\_zero’ .The value range of round\_mode is “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.

## avgpool3d

### The interface definition

```
def avgpool3d(input: Tensor,
    kernel: Union[List[int],int,Tuple[int, ...]] = None,
    stride: Union[List[int],int,Tuple[int, ...]] = None,
    pad: Union[List[int],int,Tuple[int, ...]] = None,
    ceil_mode: bool = False,
    scale: List[float] = None,
    zero_point: List[int] = None,
    out_name: str = None,
    count_include_pad : bool = False,
    round_mode : str="half_away_from_zero",
    first_round_mode : str="half_away_from_zero"):
    #pass
```

### Description of the function

Performs Avg Pooling on the input Tensor. The Avg Pooling 3d operation can refer to the avgpool3d operator of each framework. This operation is a **local operation**.

### Explanation of parameters

- tensor: Tensor type, representing the input tensor for the operation.
- kernel: List[int] or Tuple[int] or int or None, if None, global pooling is used. If not None and a single integer is provided, it indicates the same kernel size in three dimensions. If a List or Tuple is provided, its length must be 3.
- pad: List[int] or Tuple[int] or int or None, represents the padding size. If None, the default value [0,0,0,0,0,0] is used. If not None and a single integer is provided, it indicates the same padding size in three dimensions. If a List or Tuple is provided, its length must be 6.
- stride: List[int] or Tuple[int] or int or None, represents the stride size. If None, the default value [1,1,1] is used. If not None and a single integer is provided, it indicates the same stride size in three dimensions. If a List or Tuple is provided, its length must be 3.
- ceil\_mode: bool type, indicates whether to round up when calculating the output shape.

- scale: List[float] type or None, quantization parameters. If None, non-quantized computation is performed. If a List is provided, its length must be 2, representing the scale for input and output respectively.
- zero\_point: List[int] type or None, offset parameters. If None, non-quantized computation is performed. If a List is provided, its length must be 2, representing the zero point for input and output respectively.
- out\_name: string type or None, represents the name of the output Tensor. If None, a name will be automatically generated internally.
- count\_include\_pad: bool type, specifies whether to include padded elements in the average calculation. Defaults to False.
- round\_mode: string type, indicates the rounding mode for the second time when the input and output Tensors are quantized. The default value is ‘half\_away\_from\_zero’ .The value range of round\_mode is “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .
- first\_round\_mode: String type, indicating the rounding mode for the first round when the input and output Tensors are quantized. The default value is ‘half\_away\_from\_zero’ .The value range of round\_mode is “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

#### Return value

Returns a Tensor with the same data type as the input Tensor.

#### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/UINT8/INT8.

#### upsample

##### The interface definition

```
def upsample(tensor_i: Tensor,  
            scale: int = 2,  
            out_name: str = None):  
    #pass
```

### Description of the function

The output is scaled repeatedly on the input tensor data in h and w dimensions. This operation is considered a **local operation**.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- scale: int type, representing the expansion multiple.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16/INT8.
- BM1684X: The input data type can be FLOAT32/FLOAT16/INT8.

## reduce

### The interface definition

```
def reduce(tensor_i: Tensor,
          method: str = 'ReduceSum',
          axis: Union[List[int], Tuple[int], int] = None,
          keep_dims: bool = False,
          out_name: str = None):
    #pass
```

### Description of the function

Perform reduce operations on the input tensor according to axis\_list. This operation is considered a **restricted local operation**. This operation is considered a **local operation** only when the input data type is FLOAT32.

### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- method: string type, representing the reduce method. The method can be “ReduceMin”, “ReduceMax”, “ReduceMean”, “ReduceProd”, “ReduceL2”, “ReduceL1”, “ReduceSum” .
- axis: A List[int] or Tuple[int] type, indicating the specified axes.
- keep\_dims: A boolean, indicating whether to keep the specified axis after the operation.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with the same data type as the input Tensor.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## 24.5.10 Normalization Operator

### batch\_norm

#### The interface definition

```
def batch_norm(input: Tensor,
               mean: Tensor,
               variance: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               out_name: str = None):
    #pass
```

### Description of the function

The batch\_norm op first completes batch normalization of the input values, and then scales and shifts them. The batch normalization operation can refer to the batch\_norm operator of each framework.

This operation belongs to **local operations**.

### Explanation of parameters

- input: \* input: A Tensor type, representing the input Tensor. The dimension of input is not limited, if  $x$  is only 1 dimension,  $c$  is 1, otherwise  $c$  is equal to the shape[1] of  $x$ .
- mean: A Tensor type, representing the mean value of the input, shape is [c].
- variance: A Tensor type, representing the variance value of the input, shape is [c].
- gamma: A Tensor type or None, representing the scaling after batch normalization. If the value is not None, shape is required to be [c]. If None is used, shape[1] is equivalent to all 1 Tensor.
- beta: A Tensor type or None, representing the translation after batch normalization and scaling. If the value is not None, shape is required to be [c]. If None is used, shape[1] is equivalent to all 0 Tensor.
- epsilon: FLOAT type, The epsilon value to use to avoid division by zero.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns the Tensor type with the same data type as the input Tensor., representing the normalized output.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## layer\_norm

### The interface definition

```
def layer_norm(input: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               axis: int,
               out_name: str = None):
    #pass
```

### Description of the function

The layer\_norm op first completes layer normalization of the input values, and then scales and shifts them. The layer normalization operation can refer to the layer\_norm operator of each framework.

This operation belongs to **local operations**.

### Explanation of parameters

- input: A Tensor type, representing the input Tensor. The dimension of input is not limited, if  $x$  is only 1 dimension,  $c$  is 1, otherwise  $c$  is equal to the  $\text{shape}[1]$  of  $x$ .
- gamma: A Tensor type or None, representing the scaling after layer normalization. If the value is not None, shape is required to be  $[c]$ . If None is used,  $\text{shape}[1]$  is equivalent to all 1 Tensor.
- beta: A Tensor type or None, representing the translation after layer normalization and scaling. If the value is not None, shape is required to be  $[c]$ . If None is used,  $\text{shape}[1]$  is equivalent to all 0 Tensor.
- epsilon: FLOAT type, The epsilon value to use to avoid division by zero.
- axis: int type, the first normalization dimension. If  $\text{rank}(X)$  is  $r$ , axis' allowed range is  $[-r, r]$ . Negative value means counting dimensions from the back.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns the Tensor type with the same data type as the input Tensor., representing the normalized output.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### group\_norm

#### The interface definition

```
def group_norm(input: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               num_groups: int,
               out_name: str = None):
    #pass
```

#### Description of the function

The group\_norm op first completes group normalization of the input values, and then scales and shifts them. The group normalization operation can refer to the group\_norm operator of each framework.

This operation belongs to **local operations**.

#### Explanation of parameters

- input: A Tensor type, representing the input Tensor. The dimension of input is not limited, if x is only 1 dimension, c is 1, otherwise c is equal to the shape[1] of x.
- gamma: A Tensor type or None, representing the scaling after group normalization. If the value is not None, shape is required to be [c]. If None is used, shape[1] is equivalent to all 1 Tensor.
- beta: A Tensor type or None, representing the translation after group normalization and scaling. If the value is not None, shape is required to be [c]. If None is used, shape[1] is equivalent to all 0 Tensor.
- epsilon: FLOAT type, The epsilon value to use to avoid division by zero.

- num\_groups:int type, The number of groups of channels. It should be a divisor of the number of channels C.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns the Tensor type with the same data type as the input Tensor., representing the normalized output.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### rms\_norm

#### The interface definition

```
def rms_norm(input: Tensor,
             gamma: Tensor = None,
             epsilon: float = 1e-5,
             axis: int = -1,
             out_name: str = None):
    #pass
```

#### Description of the function

The rms\_norm op first completes RMS normalization of the input values, and then scales them. The RMS normalization operation can refer to the RMSNorm operator of each framework.

This operation belongs to **local operations**.

### Explanation of parameters

- input: A Tensor type, representing the input Tensor. The dimension of input is not limited.
- gamma: A Tensor type or None, representing the scaling after RMS normalization. If the value is not None, shape is required to be equal with the last dimension of the input. If None is used, shape is equivalent to all 1 Tensor.
- epsilon: FLOAT type, The epsilon value to use to avoid division by zero.
- axis: int type, the first normalization dimension. If rank(X) is r, axis' allowed range is [-r, r). Negative value means counting dimensions from the back.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns the Tensor type with the same data type as the input Tensor., representing the normalized output.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### normalize

#### Definition

```
def normalize(input: Tensor,
              p: float = 2.0,
              axes: Union[List[int], int] = 1,
              eps : float = 1e-12,
              out_name: str = None):
```

### Description

Perfrom  $L_p$  normalization over specified dimension of input tensor. For a tensor input of sizes  $(n_0, \dots, n_{dim}, \dots, n_k)$ , each  $n_{dim}$ -element vector  $v$  along dimension axes is transformed as:

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}$$

With the default arguments, it uses the Euclidean norm over vectors along dimension (1) for normalization.

This operation belongs to **local operations**.

### Parameters

- input: Tensor type, representing the input Tensor. The dimension of input is not limited. Support data type included: float32, float16.
- p: float type, representing the exponent value in the norm operation. Default to 2.0 .
- axes: Union[list[int], int] type, representing the dimension need to normalized. Default to 1. If axes is list, all the values in the list must be continuous. Caution: axes = [0, -1] is not continuous.
- eps: float type, the epsilon value to use to avoid division by zero. Default to 1e-12.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns the Tensor type with the same data type as the input Tensor., representing the normalized output.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

### 24.5.11 Vision Operator

#### nms

##### Definition

```
def nms(boxes: Tensor,  
        scores: Tensor,  
        format: str = 'PYTORCH',  
        max_box_num_per_class: int = 1,  
        out_name: str = None)
```

##### Description

Perform non-maximum-suppression upon input tensor.

##### Parameters

- boxes: Tensor type, representing a tensor of 3 dimensions, where the first dimension is number of batch, the second dimension is number of box, the third dimension is 4 coordinates of boxes.
- scores: Tensor type, representing a tensor of 3 dimensions, where the first dimension is number of batch, the second dimension is number of classes, the third dimension is number of boxes.
- format: String type, where ‘TENSORFLOW’ representing Tensorflow format [y1, x1, y2, x2] and ‘PYTORCH’ 表示representing Pytorch format [x\_center, y\_center, width, height]. The default value is ‘PYTORCH’ .
- max\_box\_num\_per\_class: Int type, representing max number of boxes per class. It must be greater than 0. The default value is 1.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

##### Returns

Returns one Tensor, which is the selected indices from the boxes tensor of 2 dimensions:[num\_selected\_indices, 3], the selected index format is [batch\_index, class\_index, box\_index].

### Processor support

- BM1688: The input data type can be FLOAT32.
- BM1684X: The input data type can be FLOAT32.

## interpolate

### Definition

```
def interpolate(input: Tensor,
                scale_h: float,
                scale_w: float,
                method: str = 'nearest',
                coord_mode: str = "pytorch_half_pixel",
                out_name: str = None)
```

### Description

Perform interpolation upon input tensor.

### Parameters

- input: Tensor type, representing the input Tensor. Must be at least a 2-dimensional tensor.
- scale\_h: Float type, representing the resize scale along h-axis. Must be greater than 0.
- scale\_w: Float type, representing the resize scale along w-axis. Must be greater than 0.
- method: String type, representing the interpolation method. Optional values are “nearest” or “linear”. Default is “nearest” .
- coord\_mode: string type, representing the method used in inverse map of coordinates. Optional values are “align\_corners” , “pytorch\_half\_pixel” , “half\_pixel” or “asymmetric” . Default is “pytorch\_half\_pixel” .
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

Note that, parameter coord\_mode defined here is the same as the parameter coordinate\_transformation\_mode defined in onnx operator Resize. Supposed that resize scale along h/w-axis is scale, input coordinate is  $x_{in}$ , input size is  $l_{in}$ , output coordinate is  $x_{out}$ , output size is  $l_{out}$ , then the defintion of inverse map of coordinates is as follows: \* “half\_pixel” :

```
x_in = (x_out + 0.5) / scale - 0.5
```

- “pytorch\_half\_pixel” :

```
x_in = len > 1 ? (x_out + 0.5) / scale - 0.5 : 0
```

- “align\_corners” :

```
x_in = x_out * (l_in - 1) / (l_out - 1)
```

- “asymmetric” :

```
x_in = x_out / scale
```

## Returns

Returns a Tensor representing the interpolated result. The data type is the same as the input type, and the shape is adjusted based on the scaling factors.

## Processor support

- BM1688: Supports input data types FLOAT32/FLOAT16/INT8.
- BM1684X: Supports input data types FLOAT32/FLOAT16/INT8.

## yuv2rgb

### The interface definition

```
def yuv2rgb(  
    inputs: Tensor,  
    src_format: int,  
    dst_format: int,  
    ImageOutFormatAttr: str,  
    formula_mode: str,  
    round_mode: str,  
    out_name: str = None,  
):
```

### Description of the function

Transfer input tensor from yuv to rgb. Require tensor shape=[n,h\*3/2,w], n represents batch, h represents pixels height, w represents pixels width.

### Explanation of parameters

- inputs: Tensor type, representing the input yuv tensor. Its dims must be 3, 1st dim represents batch, 2nd dim represents pixels height, 3rd dim represents pixels width.
- src\_format: Int type, representing the input format.  
`FORMAT\_MAPPING\_YUV420P\_YU12`=0, `FORMAT\_MAPPING\_YUV420P\_YV12`=1, `FORMAT\_MAPPING\_NV12`=2, `FORMAT\_MAPPING\_NV21`=3.
- dst\_format: Int type, representing the output format. `FORMAT\_MAPPING\_RGB`=4, `FORMAT\_MAPPING\_BGR`=5.
- ImageOutFormatAttr: string type, representing the output dtype, currently only support UINT8.
- formula\_mode: string type, representing the formula to transfer from yuv to rgb, currently support \_601\_limited, \_601\_full.
- round\_mode: string type, currently support HalfAwayFromZero, HalfToEven.
- out\_name: string type, representing the name of output tensor, default= None.

### Return value

One rgb tensor will be output, with shape=[n,3,h,w], where n represents batch, h represents pixels height, w represents pixels width.

### Processor support

- BM1684X: The input data type must be UINT8/INT8. Output data type is UINT8.
- BM1688: The input data type must be UINT8/INT8. Output data type is UINT8.

## roiExtractor

### Definition

```
def roiExtractor(rois: Tensor,
                 target_lvls: Tensor,
                 feats: List[Tensor],
                 PH: int,
                 PW: int,
                 sampling_ratio: int,
                 list_spatial_scale: Union[int, List[int], Tuple[int]],
                 mode:str=None,
                 out_name:str=None)
```

### Description

Given 4 feature maps, extract the corresponding ROI from rois based on the target\_lvls indices and perform ROI Align with the corresponding feature maps to obtain the final output. This operation is considered a **restricted local operation**.

### Parameters

- rois: Tensor type, representing all the ROIs.
- target\_lvls: Tensor type, representing which level of feature map each ROI corresponds to.
- feats: List[Tensor], representing all feature maps.
- PH: Int type, representing the height of the output.
- PW: Int type, representing the width of the output.
- sampling\_ratio: Int type, representing the sample ratio for each level of the feature maps.
- **list\_spatial\_scale: List[int] or int, representing the spatial scale corresponding to each feature map level.**  
Please note that spatial scale follows mmdetection style, where one int value is initially given, and but its float reciprocal is adapted for roialign.
- **mode: string type, representing the implementation forms, now supporting two modes: DynNormal, or DynFuse.**

**Please note that in DynFuse mode, coordinates of rois can satisfy either mmdetection style, which is 5-length of [batch\_id, x0, y0 x1, y1],**

or customized style, which is 7-length of [a, b, x0, y0, x1, y1, c], please customize the position of batch\_id.

in DynNormal mode, a customized [a, b, x0, y0 x1, y1, c] coordinates style is adapted in case any customers desire to apply their models.

- out\_name: string type, representing the name of output tensor, default= None.

### Returns

Returns a Tensor with the same data type as the input rois.

### Processor support

- BM1688: Supports input data types FLOAT32/FLOAT16.
- BM1684X: Supports input data types FLOAT32/FLOAT16.

## 24.5.12 Select Operator

### nonzero

#### The interface definition

```
def nonzero(tensor_i:Tensor,  
           dtype: str = 'int32',  
           out_name: str = None):  
    #pass
```

#### Description of the function

Extract the corresponding location information when input Tensor data is true. This operation is considered a **global operation**.

#### Explanation of parameters

- tensor\_i: Tensor type, representing the input tensor for the operation.
- dtype: String type. The data type of the output tensor, with a default value of “int32.”
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Return value

Returns a Tensor with data type INT32.

### Processor support

- BM1688: The input data type can be FLOAT32/FLOAT16.
- BM1684X: The input data type can be FLOAT32/FLOAT16.

## lut

### Definition

```
def lut(input: Tensor,  
        table: Tensor,  
        out_name: str = None):  
    #pass
```

### Description

Use look-up table to transform values of input tensor.

### Parameters

- input: Tensor type, representing the input.
- table: Tensor type, representing the look-up table.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

### Returns

Returns one Tensor, whose data type is the same as that of the table tensor.

### Processor support

- BM1688: The data type of input can be INT8/UINT8. The data type of table an be INT8/UINT8.
- BM1684X: The data type of input can be INT8/UINT8. The data type of table an be INT8/UINT8.

## select

### Definition

```
def select(lhs: Tensor,
          rhs: Tensor,
          tbrn: Tensor,
          fbrn: Tensor,
          type: str,
          out_name = None):
    #pass
```

### Description

Select by the comparison result of lhs and rhs. If condition is True, select tbrn, otherwise select fbrn.

### Parameters

- lhs: Tensor type, representing the left-hand-side.
- rhs: Tensor type, representing the right-hand-side.
- tbrn: Tensor type, representing the true branch.
- fbrn: Tensor type, representing the false branch.
- type: String type, representing the comparison operator. Optional values are “Greater” /” Less” /” GreaterOrEqual” /” LessOrEqual” /” Equal” /” NotEqual” .
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

Constraint: The shape and data type of lhs and rhs should be the same. The shape and data type of tbrn and fbrn should be the same.

### Returns

Returns a Tensor whose data type is the same that of tbrn.

### Processor Support

- BM1688: Data type of lhs/ rhs/ tbrn/ fbrn can be FLOAT32/FLOAT16(TODO).
- BM1684X: Data type of lhs/ rhs/ tbrn/ fbrn can be FLOAT32/FLOAT16(TODO).

### cond\_select

#### Definition

```
def cond_select(cond: Tensor,
                tbrn: Union[Tensor, Scalar],
                fbrn: Union[Tensor, Scalar],
                out_name:str = None):
    #pass
```

#### Description

Select by condition representing by cond. If condition is True, select tbrn, otherwise select fbrn.

#### Parameters

- cond: Tensor type, representing condition.
- tbrn: Tensor type or Scalar type, representing true branch.
- fbrn: Tensor type or Scalar type, representing false branch.
- out\_name: A string or None, representing the name of the output Tensor. If set to None, the system will automatically generate a name internally.

Constraint: If tbrn and fbrn are all Tensors, then the shape and data type of tbrn and fbrn should be the same.

## Returns

Returns a Tensor whose data type is the same that of tbrn.

## Processor Support

- BM1688: Data type of cond/ tbrn/ fbrn can be FLOAT32/FLOAT16/INT8/UINT8.
- BM1684X: Data type of cond/ tbrn/ fbrn can be FLOAT32/FLOAT16/INT8/UINT8.

## bmodel\_inference\_combine

### Definition

```
def bmodel_inference_combine(
    bmodel_file: str,
    final_mlir_fn: str,
    input_data_fn: Union[str, dict],
    tensor_loc_file: str,
    reference_data_fn: str,
    dump_file: bool = True,
    save_path: str = "",
    out_fixed: bool = False,
    dump_cmd_info: bool = True,
    skip_check: bool = True, # disable data_check to increase processing speed
    run_by_op: bool = False, # enable to run_by_op, may cause timeout error[F]
    ↪when some OPs contain too many atomic cmds
    desire_op: list = [], # set ["A", "B", "C"] to only dump tensor A/B/C, dump all[F]
    ↪tensor as default
    is_soc: bool = False, # soc mode ONLY support {reference_data_fn=xxx.npz,[F]
    ↪dump_file=True}
    using_memory_opt: bool = False, # required when is_soc=True
    enable_soc_log: bool = False, # required when is_soc=True
    soc_tmp_path: str = "/tmp", # required when is_soc=True
    hostname: str = None, # required when is_soc=True
    port: int = None, # required when is_soc=True
    username: str = None, # required when is_soc=True
    password: str = None, # required when is_soc=True
):
```

### Description

Dump tensors layer by layer according to the bmodel, which help to verify the correctness of bmodel.

### Parameters

- bmodel\_file: String type, representing the abs path of bmodel.
- final\_mlir\_fn: String type, representing the abs path of final.mlir.
- input\_data\_fn: String type or Dict type, representing the input data, supporting Dict/.dat/.npz.
- tensor\_loc\_file: String type, representing the abs path of tensor\_location.json.
- reference\_data\_fn: String type, representing the abs path of .mlir/.npz with module.state = “TPU\_LOWERED” . Used to restore the shape during bmodel infer.
- dump\_file: Bool type, representing whether save results as file.
- save\_path: String type, representing the abs path of saving results on host.
- out\_fixed: Bool type, representing whether to get results in fixed number.
- dump\_cmd\_info: Bool type, enable to save atomic cmd info at save\_path.
- skip\_check: Bool type, set to True to disable data check to decrease time cost for CMODEL/PCIE mode.
- run\_by\_op: Bool type, enable to run\_by\_op, decrease time cost but may cause timeout error when some OPs contain too many atomic cmds.
- desire\_op: List type, specify this option to dump specific tensors, dump all tensor as default.
- is\_soc: Bool type, representing whether to use in soc mode.
- using\_memory\_opt: Bool type, enable to use memory opt, decrease memory usage at the expense of increasing time cost. Suggest to enable when running large model.
- enable\_soc\_log: Bool type, enable to print and save log at save\_path.
- soc\_tmp\_path: String type, representing the abs path of tmp files and tools on device in soc mode.
- hostname: String type, representing the ip address of device in soc mode.
- port: Int type, representing the port of device in soc mode.
- username: String type, representing the username of device in soc mode.
- password: String type, representing the password of device in soc mode.

Attention:

- When the funciton is called in cmodel/pcie mode, functions use\_cmodel/use\_chip from /tpu-mlir/envsetup.sh is required.
- When the funciton is called in soc mode, use use\_chip and reference\_data\_fn must be .npz.

### Returns

- cmodel/pcie mode: if dump\_file=True, then bmodel\_infer\_xxx.npz will be generated in save\_path, otherwise return python dict.
- soc mode: soc\_infer\_xxx.npz will be generated in save\_path.

### Processor Support

- BM1688: only cmodel mode.
- BM1684X: cmodel/pcie/soc mode.

## scatter

### Definition

```
def scatter(input: Tensor,
           index: Tensor,
           updates: Tensor,
           axis: int = 0,
           out_name: str = None):
    #pass
```

### Description

Based on the specified indices, write the input data to specific positions in the target Tensor. This operation allows the elements of the input Tensor to be scattered to the specified positions in the output Tensor. Refer to the ScatterElements operation in various frameworks for more details. This operation belongs to **local operation**.

### Parameters

- input: Tensor type, represents the input operation Tensor, i.e., the target Tensor that needs to be updated.
- index: Tensor type, represents the index Tensor that specifies the update positions.
- updates: Tensor type, represents the values to be written into the target Tensor.
- axis: int type, represents the axis along which to update.
- out\_name: string type or None, represents the name of the output Tensor. If None, a name will be automatically generated internally.

### Returns

Returns a new Tensor with updates applied at the specified positions, while other positions retain the original values from the input Tensor.

### Processor Support

- BM1684X: The input data type can be FLOAT32/UINT8/INT8.
- BM1688: The input data type can be FLOAT32/UINT8/INT8.

## scatterND

### Definition

```
def scatterND(input: Tensor,  
             indices: Tensor,  
             updates: Tensor,  
             out_name: str = None):  
    #pass
```

### Description

Based on the specified indices, write the input data to specific positions in the target Tensor. This operation allows the elements of the input Tensor to be scattered to the specified positions in the output Tensor. Refer to the scatterND operation in ONNX 11 for more details. This operation belongs to **local operation**.

## Parameters

- input: Tensor type, represents the input operation Tensor, i.e., the target Tensor that needs to be updated.
- indices: Tensor type, represents the index Tensor that specifies the update positions. The datatype must be uint32.
- updates: Tensor type, represents the values to be written into the target Tensor. Rank(updates) = Rank(input) + Rank(indices) - shape(indices)[-1] -1.
- out\_name: string type or None, represents the name of the output Tensor. If None, a name will be automatically generated internally.

## Returns

Returns a new Tensor with updates applied at the specified positions, while other positions retain the original values from the input Tensor. The shape and datatype are the same with the input tensor.

## Processor Support

- BM1684X: The input data type can be FLOAT32/UINT8/INT8.
- BM1688: The input data type can be FLOAT32/UINT8/INT8.

### 24.5.13 Preprocess Operator

#### mean\_std\_scale

##### The interface definition

```
def mean_std_scale(input: Tensor,
                   std: List[float],
                   mean: List[float],
                   scale: Optional[Union[List[float],List[int]]] = None,
                   zero_points: Optional[List[int]] = None,
                   out_name: str = None,
                   dtype="float16",
                   round_mode: str = "half_away_from_zero"):  
    #pass
```

### Description of the function

Preproces input Tensor data. This operation is considered a **global operation**.

### Explanation of parameters

- input: Tensor type, representing the input data.
- std: List[float], representing the standard deviation of the dataset. The dimensions of mean and std must match the channel dimension of the input, i.e., the second dimension of the input.
- mean: List[float], representing the mean of the dataset. The dimensions of mean and std must match the channel dimension of the input, i.e., the second dimension of the input.
- scale: Optional[Union[List[float],List[int]]] type or None, reprpesenting the scale factor.
- zero\_points: Optional[List[int]] type or None,representing the zero point.
- out\_name: string type or None, representing the name of Tensor, tpulang will auto generate name if out\_name is None.
- odtype: String, representing the data type of the output Tensor. Default is “float16” . Currently supports float16 and int8.
- round\_mode: String, representing the rounding method. Default is “half\_away\_from\_zero” , with options “half\_away\_from\_zero” , “half\_to\_even” , “towards\_zero” , “down” , “up” .

### Return value

Returns a Tensor with the type of odtype.

### Processor support

- BM1684X: The input data type can be FLOAT32/UINT8/INT8, the output data type can be INT8/FLOAT16.

#### 24.5.14 Transform Operator

rope

##### The interface definition

```
def rope( input: Tensor,
          weight0: Tensor,
          weight1: Tensor,
          is_permute_optimize: bool = False,    # unused
          mul1_round_mode: str = 'half_up',
          mul2_round_mode: str = 'half_up',
          add_round_mode: str = 'half_up',
          mul1_shift: int = None,
          mul2_shift: int = None,
          add_shift: int = None,
          mul1_saturation: bool = True,
          mul2_saturation: bool = True,
          add_saturation: bool = True,
          out_name: str = None):
    #pass
```

##### Description of the function

Perform a rotation encoding (RoPE) operation on the input Tensor. This operation belongs to **global operation**

##### Explanation of parameters

- input: Tensor type, indicating the input operation Tensor. It must be four-dimensional.
- weight0: Tensor, indicating the input operation Tensor.
- weight1: Tensor, indicating the input operation Tensor.
- is\_permute\_optimize: bool type, indicating whether to perform permute sinking and check the shape of permute sinking. # unused
- mul1\_round\_mode: Type String, representing the rounding method of mul1 in RoPE. The default value is “half\_away\_from\_zero”, and the range is “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”.
- mul2\_round\_mode: Type String, representing the rounding method of mul2 in RoPE. The default value is “half\_away\_from\_zero”, and the range is “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”.

- add\_round\_mode: Type String, representing the rounding method of add in RoPE. The default value is “half\_away\_from\_zero”, and the range is “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”.
- mul1\_shift: int type, representing the number of bits of the shift of mul1 in RoPE.
- mul2\_shift: int type, indicating the number of bits of the shift of mul2 in RoPE.
- add\_shift: int type, indicating the number of bits shifted by add in RoPE.
- mul1\_saturation: bool type, indicating whether the calculation result of mul1 in RoPE requires saturation processing. The default is True saturation processing, and no modification is needed unless necessary.
- mul2\_saturation: bool type, indicating whether the calculation result of mul2 in RoPE requires saturation processing. The default is True saturation processing, and no modification is needed unless necessary.
- add\_saturation: bool type, indicating whether the add calculation result in RoPE requires saturation processing. The default is True saturation processing, and no modification is needed unless necessary.
- out\_name: output name, type string, default to None.

#### Return value

Return a Tensor with the data type of odtype.

#### Processor support

- BM1684X: The input data types can be FLOAT32,FLOAT16 and INT types.

### multi\_scale\_deformable\_attention

#### The interface definition

```
def multi_scale_deformable_attention(  
    query: Tensor,  
    value: Tensor,  
    key_padding_mask: Tensor,  
    reference_points: Tensor,  
    sampling_offsets_weight: Tensor,  
    sampling_offsets_bias_ori: Tensor,  
    attention_weights_weight: Tensor,  
    attention_weights_bias_ori: Tensor,  
    value_proj_weight: Tensor,  
    value_proj_bias_ori: Tensor,
```

(continues on next page)

(continued from previous page)

```
    output_proj_weight: Tensor,  
    output_proj_bias_ori: Tensor,  
    spatial_shapes: List[List[int]],  
    embed_dims: int,  
    num_heads: int = 8,  
    num_levels: int = 4,  
    num_points: int = 4,  
    out_name: str = None):  
  
    #pass
```

### Description of the function

Perform multi-scale deformable attention on the input, and the specific function can refer to [https://github.com/open-mmlab/mmcv/blob/main/mmcv/ops/multi\\_scale\\_deform\\_attn.py](https://github.com/open-mmlab/mmcv/blob/main/mmcv/ops/multi_scale_deform_attn.py):`MultiScaleDeformableAttention:forward`, the implementation of this operation is different from the official one. This operation is considered a **global operation**.

### Explanation of parameters

- query: Tensor type, query of Transformer with shape (1, num\_query, embed\_dims).
- value: Tensor type, the value tensor with shape (1, num\_key, embed\_dims).
- key\_padding\_mask: Tensor type, the mask of the query tensor with shape (1, num\_key).
- reference\_points: Tensor type, normalized reference points with shape (1, num\_query, num\_levels, 2), all elements are in the range [0, 1], the upper left corner is (0,0), and the lower right corner is (1,1), including the padding area.
- sampling\_offsets\_weight: Tensor type, the weight of the fully connected layer for calculating the sampling offset with shape (embed\_dims, num\_heads\*num\_levels\*num\_points\*2).
- sampling\_offsets\_bias\_ori: Tensor type, the bias of the fully connected layer for calculating the sampling offset with shape (num\_heads\*num\_levels\*num\_points\*2).
- attention\_weights\_weight: Tensor type, the weight of the fully connected layer for calculating the attention weight with shape (embed\_dims, num\_heads\*num\_levels\*num\_points).
- attention\_weights\_bias\_ori: Tensor type, the bias of the fully connected layer for calculating the attention weight with shape (num\_heads\*num\_levels\*num\_points).
- value\_proj\_weight: Tensor type, the weight of the fully connected layer for calculating the value projection with shape (embed\_dims, embed\_dims).

- value\_proj\_bias\_ori: Tensor type, the bias of the fully connected layer for calculating the value projection with shape (embed\_dims).
- output\_proj\_weight: Tensor type, the weight of the fully connected layer for calculating the output projection with shape (embed\_dims, embed\_dims).
- output\_proj\_bias\_ori: Tensor type, the bias of the fully connected layer for calculating the output projection with shape (embed\_dims).
- spatial\_shapes: List[List[int]] type, the spatial shape of different level features with shape (num\_levels, 2), the last dimension represents (h, w).
- embed\_dims: int type, hidden\_size of query, key, and value.
- num\_heads: int type, the number of attention heads, default is 8.
- num\_levels: int type, the number of levels of multi-scale attention, default is 4.
- num\_points: int type, the number of sampling points at each level, default is 4.
- out\_name: string type or None, the name of the output Tensor, and the name will be automatically generated internally if it is None.

#### Return value

Returns a Tensor with the same data type as query.dtype.

#### Processor support

- BM1684X: The input data type can be FLOAT32/FLOAT16.
- BM1688: The input data type can be FLOAT32/FLOAT16.

### 24.5.15 Transform Operator

#### a16matmul

##### The interface definition

```
def a16matmul(input: Tensor,  
              weight: Tensor,  
              scale: Tensor,  
              zp: Tensor,  
              bias: Tensor = None,  
              right_transpose=True,  
              out_dtype: str = 'float16',  
              out_name: str = None,  
              group_size: int = 128,
```

(continues on next page)

(continued from previous page)

```
bits: int = 4,  
g_idx: Tensor = None,  
):  
  
#pass
```

### Description of the function

Perform W4A16/W8A16 MatMul on the input. This operation is considered a **global operation**.

### Explanation of parameters

- input: Tensor type, represents the input tensor.
- weight: Tensor type, represents the weight after 4-bit/8-bit quantization, stored as int32.
- scale: Tensor type, represents the quantization scaling factor for the weights, stored as float32.
- zp: Tensor type, represents the quantization zero point for the weights, stored as int32.
- bias: Tensor type, represents the bias, stored as float32.
- right\_transpose: Boolean type, indicates whether the weight matrix is transposed; currently only supports True.
- out\_dtype: String type, represents the data type of the output tensor.
- out\_name: String type or None, represents the name of the output Tensor; if None, a name will be automatically generated internally.
- group\_size: Integer type, indicates the group size for quantization.
- bits: Integer type, represents the quantization bit-width; only supports 4 bits/8 bits.
- g\_idx: Tensor type, represents the quantization reordering coefficient; currently not supported.

**Return value**

Returns a Tensor with the same data type as out\_dtype.

**Processor support**

- BM1684X: The input data type can be FLOAT32/FLOAT16.
- BM1688: The input data type can be FLOAT32/FLOAT16.

### 24.5.16 Transform Operator

#### qwen2\_block

##### The interface definition

```
def qwen2_block(hidden_states: Tensor,
                 position_ids: Tensor,
                 attention_mask: Tensor,
                 q_proj_weights: Tensor,
                 q_proj_scales: Tensor,
                 q_proj_zps: Tensor,
                 q_proj_bias: Tensor,
                 k_proj_weights: Tensor,
                 k_proj_scales: Tensor,
                 k_proj_zps: Tensor,
                 k_proj_bias: Tensor,
                 v_proj_weights: Tensor,
                 v_proj_scales: Tensor,
                 v_proj_zps: Tensor,
                 v_proj_bias: Tensor,
                 o_proj_weights: Tensor,
                 o_proj_scales: Tensor,
                 o_proj_zps: Tensor,
                 o_proj_bias: Tensor,
                 down_proj_weights: Tensor,
                 down_proj_scales: Tensor,
                 down_proj_zps: Tensor,
                 gate_proj_weights: Tensor,
                 gate_proj_scales: Tensor,
                 gate_proj_zps: Tensor,
                 up_proj_weights: Tensor,
                 up_proj_scales: Tensor,
                 up_proj_zps: Tensor,
                 input_layernorm_weight: Tensor,
                 post_attention_layernorm_weight: Tensor,
                 cos: List[Tensor],
                 sin: List[Tensor],
```

(continues on next page)

(continued from previous page)

```
out_dtype: str = 'float16',
group_size: int = 128,
weight_bits: int = 4,
hidden_size: int = 3584,
rms_norm_eps: float = 1e-06,
num_attention_heads: int = 28,
num_key_value_heads: int = 4,
mrope_section: List[int] = [16, 24, 24],
quant_method: str = "gptq",
out_name: str = None
):

#pass
```

### Description of the function

A block layer of qwen2 during the prefill stage. This operation is considered a **global operation**.

### Explanation of parameters

- hidden\_states: Tensor type, representing activation values, with shape (1, seq\_length, hidden\_size).
- position\_ids: Tensor type, representing positional indices, with shape (3, 1, seq\_length).
- attention\_mask: Tensor type, representing the attention mask, with shape (1, 1, seq\_length, seq\_length).
- q\_proj\_weights: Tensor type, representing the quantized query weights, stored as int32.
- q\_proj\_scales: Tensor type, representing the quantization scaling factors for the query, stored as float32.
- q\_proj\_zps: Tensor type, representing the quantization zero-points for the query, stored as int32.
- q\_proj\_bias: Tensor type, representing the query bias, stored as float32.
- k\_proj\_weights: Tensor type, representing the quantized key weights, stored as int32.
- k\_proj\_scales: Tensor type, representing the quantization scaling factors for the key, stored as float32.
- k\_proj\_zps: Tensor type, representing the quantization zero-points for the key, stored as int32.
- k\_proj\_bias: Tensor type, representing the key bias, stored as float32.

- v\_proj\_weights: Tensor type, representing the quantized value weights, stored as int32.
- v\_proj\_scales: Tensor type, representing the quantization scaling factors for the value, stored as float32.
- v\_proj\_zps: Tensor type, representing the quantization zero-points for the value, stored as int32.
- v\_proj\_bias: Tensor type, representing the value bias, stored as float32.
- o\_proj\_weights: Tensor type, representing the quantized output projection weights, stored as int32.
- o\_proj\_scales: Tensor type, representing the quantization scaling factors for the output projection, stored as float32.
- o\_proj\_zps: Tensor type, representing the quantization zero-points for the output projection, stored as int32.
- o\_proj\_bias: Tensor type, representing the output projection bias, stored as float32.
- down\_proj\_weights: Tensor type, representing the quantized down projection layer weights, stored as int32.
- down\_proj\_scales: Tensor type, representing the quantization scaling factors for the down projection layer, stored as float32.
- down\_proj\_zps: Tensor type, representing the quantization zero-points for the down projection layer, stored as int32.
- gate\_proj\_weights: Tensor type, representing the quantized gate projection layer weights, stored as int32.
- gate\_proj\_scales: Tensor type, representing the quantization scaling factors for the gate projection layer, stored as float32.
- gate\_proj\_zps: Tensor type, representing the quantization zero-points for the gate projection layer, stored as int32.
- up\_proj\_weights: Tensor type, representing the quantized up projection layer weights, stored as int32.
- up\_proj\_scales: Tensor type, representing the quantization scaling factors for the up projection layer, stored as float32.
- up\_proj\_zps: Tensor type, representing the quantization zero-points for the up projection layer, stored as int32.
- input\_layernorm\_weight: Tensor type, representing the weights for layer normalization on the input, stored as int32.
- post\_attention\_layernorm\_weight: Tensor type, representing the weights for layer normalization on the attention layer output, stored as int32.
- cos: List[Tensor] type, representing the cosine positional encodings.
- sin: List[Tensor] type, representing the sine positional encodings.

- out\_dtype: string type, representing the data type of the output tensor.
- group\_size: int type, representing the group size used for quantization.
- weight\_bits: int type, representing the quantization bit width, currently only supports 4 bits/8 bits.
- hidden\_size: int type, representing the hidden size for the query/key/value.
- rms\_norm\_eps: float type, representing the epsilon parameter in layer normalization.
- num\_attention\_heads: int type, representing the number of attention heads.
- num\_key\_value\_heads: int type, representing the number of key/value heads.
- mrope\_section: List[int] type, representing the sizes of the three dimensions for the positional encoding.
- quant\_method: str type, representing the quantization method, currently only GPTQ quantization is supported.
- out\_name: string type or None, representing the name of the output tensor; if None, the name will be automatically generated.

#### Return value

Returns 3 Tensors: the activation output, the key cache, and the value cache, all with the data type specified by out\_dtype.

#### Processor support

- BM1684X: The input data type can be FLOAT32/FLOAT16.
- BM1688: The input data type can be FLOAT32/FLOAT16.

### 24.5.17 Transform Operator

#### qwen2\_block\_cache

##### The interface definition

```
def qwen2_block_cache(hidden_states: Tensor,
                      position_ids: Tensor,
                      attention_mask: Tensor,
                      k_cache: Tensor,
                      v_cache: Tensor,
                      q_proj_weights: Tensor,
                      q_proj_scales: Tensor,
                      q_proj_zps: Tensor,
```

(continues on next page)

(continued from previous page)

```
q_proj_bias: Tensor,
k_proj_weights: Tensor,
k_proj_scales: Tensor,
k_proj_zps: Tensor,
k_proj_bias: Tensor,
v_proj_weights: Tensor,
v_proj_scales: Tensor,
v_proj_zps: Tensor,
v_proj_bias: Tensor,
o_proj_weights: Tensor,
o_proj_scales: Tensor,
o_proj_zps: Tensor,
o_proj_bias: Tensor,
down_proj_weights: Tensor,
down_proj_scales: Tensor,
down_proj_zps: Tensor,
gate_proj_weights: Tensor,
gate_proj_scales: Tensor,
gate_proj_zps: Tensor,
up_proj_weights: Tensor,
up_proj_scales: Tensor,
up_proj_zps: Tensor,
input_layernorm_weight: Tensor,
post_attention_layernorm_weight: Tensor,
cos: List[Tensor],
sin: List[Tensor],
out_dtype: str = 'float16',
group_size: int = 128,
weight_bits: int = 4,
hidden_size: int = 3584,
rms_norm_eps: float = 1e-06,
num_attention_heads: int = 28,
num_key_value_heads: int = 4,
mrope_section: List[int] = [16, 24, 24],
quant_method: str = "gptq",
out_name: str = None
):
```

#pass

### Description of the function

A block layer of qwen2 during the decode stage. This operation is considered a **global operation**.

### Explanation of parameters

- hidden\_states: Tensor type, representing activation values, with shape (1, 1, hidden\_size).
- position\_ids: Tensor type, representing positional indices, with shape (3, 1, 1).
- attention\_mask: Tensor type, representing the attention mask, with shape (1, 1, 1, seq\_length + 1).
- k\_cache: Tensor type, representing the key cache. Its shape is (1, seq\_length, num\_key\_value\_heads, head\_dim).
- v\_cache: Tensor type, representing the value cache. Its shape is (1, seq\_length, num\_key\_value\_heads, head\_dim).
- q\_proj\_weights: Tensor type, representing the quantized query weights, stored as int32.
- q\_proj\_scales: Tensor type, representing the quantization scaling factors for the query, stored as float32.
- q\_proj\_zps: Tensor type, representing the quantization zero-points for the query, stored as int32.
- q\_proj\_bias: Tensor type, representing the query bias, stored as float32.
- k\_proj\_weights: Tensor type, representing the quantized key weights, stored as int32.
- k\_proj\_scales: Tensor type, representing the quantization scaling factors for the key, stored as float32.
- k\_proj\_zps: Tensor type, representing the quantization zero-points for the key, stored as int32.
- k\_proj\_bias: Tensor type, representing the key bias, stored as float32.
- v\_proj\_weights: Tensor type, representing the quantized value weights, stored as int32.
- v\_proj\_scales: Tensor type, representing the quantization scaling factors for the value, stored as float32.
- v\_proj\_zps: Tensor type, representing the quantization zero-points for the value, stored as int32.
- v\_proj\_bias: Tensor type, representing the value bias, stored as float32.
- o\_proj\_weights: Tensor type, representing the quantized output projection weights, stored as int32.

- o\_proj\_scales: Tensor type, representing the quantization scaling factors for the output projection, stored as float32.
- o\_proj\_zps: Tensor type, representing the quantization zero-points for the output projection, stored as int32.
- o\_proj\_bias: Tensor type, representing the output projection bias, stored as float32.
- down\_proj\_weights: Tensor type, representing the quantized down projection layer weights, stored as int32.
- down\_proj\_scales: Tensor type, representing the quantization scaling factors for the down projection layer, stored as float32.
- down\_proj\_zps: Tensor type, representing the quantization zero-points for the down projection layer, stored as int32.
- gate\_proj\_weights: Tensor type, representing the quantized gate projection layer weights, stored as int32.
- gate\_proj\_scales: Tensor type, representing the quantization scaling factors for the gate projection layer, stored as float32.
- gate\_proj\_zps: Tensor type, representing the quantization zero-points for the gate projection layer, stored as int32.
- up\_proj\_weights: Tensor type, representing the quantized up projection layer weights, stored as int32.
- up\_proj\_scales: Tensor type, representing the quantization scaling factors for the up projection layer, stored as float32.
- up\_proj\_zps: Tensor type, representing the quantization zero-points for the up projection layer, stored as int32.
- input\_layernorm\_weight: Tensor type, representing the weights for layer normalization on the input, stored as int32.
- post\_attention\_layernorm\_weight: Tensor type, representing the weights for layer normalization on the attention layer output, stored as int32.
- cos: List[Tensor] type, representing the cosine positional encodings.
- sin: List[Tensor] type, representing the sine positional encodings.
- out\_dtype: string type, representing the data type of the output tensor.
- group\_size: int type, representing the group size used for quantization.
- weight\_bits: int type, representing the quantization bit width, currently only supports 4 bits/8 bits.
- hidden\_size: int type, representing the hidden size for the query/key/value.
- rms\_norm\_eps: float type, representing the epsilon parameter in layer normalization.
- num\_attention\_heads: int type, representing the number of attention heads.

- num\_key\_value\_heads: int type, representing the number of key/value heads.
- mrope\_section: List[int] type, representing the sizes of the three dimensions for the positional encoding.
- quant\_method: str type, representing the quantization method, currently only GPTQ quantization is supported.
- out\_name: string type or None, representing the name of the output tensor; if None, the name will be automatically generated.

#### **Return value**

Returns 3 Tensors: the activation output, the key cache, and the value cache, all with the data type specified by out\_dtype.

#### **Processor support**

- BM1684X: The input data type can be FLOAT32/FLOAT16.
- BM1688: The input data type can be FLOAT32/FLOAT16.