

---

# TPU-MLIR Technical Reference Manual

Release 1.3.228

SOPHGO

Sep 21, 2023

# Table of Contents

<b>1 TPU-MLIR Introduction</b>	<b>3</b>
<b>2 Environment Setup</b>	<b>5</b>
2.1 Code Download . . . . .	5
2.2 Docker Configuration . . . . .	5
2.3 ModelZoo (Optional) . . . . .	6
2.4 Compilation . . . . .	6
<b>3 User Interface</b>	<b>7</b>
3.1 Introduction . . . . .	7
3.2 model_transform.py . . . . .	9
3.3 run_calibration.py . . . . .	11
3.4 model_deploy.py . . . . .	11
3.5 Other Tools . . . . .	12
3.5.1 model_runner.py . . . . .	12
3.5.2 npz_tool.py . . . . .	13
3.5.3 visual.py . . . . .	13
<b>4 Overall Design</b>	<b>15</b>
4.1 Layered . . . . .	15
4.2 Top Pass . . . . .	15
4.3 Tpu Pass . . . . .	17
4.4 Other Passes . . . . .	18
<b>5 Front-end Conversion</b>	<b>19</b>
5.1 Main Work . . . . .	19
5.2 Workflow . . . . .	19
5.3 Example . . . . .	21
<b>6 Quantization</b>	<b>25</b>
6.1 Basic Concepts . . . . .	25
6.1.1 Asymmetric Quantization . . . . .	25
6.1.2 Symmetric Quantization . . . . .	26
6.2 Scale Conversion . . . . .	27
6.3 Quantization derivation . . . . .	27
6.3.1 Convolution . . . . .	27
6.3.2 InnerProduct . . . . .	28
6.3.3 Add . . . . .	28

6.3.4	AvgPool . . . . .	28
6.3.5	LeakyReLU . . . . .	29
6.3.6	Pad . . . . .	29
6.3.7	PReLU . . . . .	30
<b>7</b>	<b>Calibration</b>	<b>31</b>
7.1	General introduction . . . . .	31
7.2	Calibration data screening and preprocessing . . . . .	33
7.2.1	Screening Principles . . . . .	33
7.2.2	Input format and preprocessing . . . . .	34
7.3	Algorithm Implementation . . . . .	35
7.3.1	KLD Algorithm . . . . .	35
7.3.2	Auto-tune Algorithm . . . . .	35
7.4	Example: yolov5s calibration . . . . .	37
7.5	visual tool introduction . . . . .	38
<b>8</b>	<b>Lowering</b>	<b>42</b>
8.1	Basic Process . . . . .	42
8.2	Mixed precision . . . . .	43
<b>9</b>	<b>SubNet</b>	<b>44</b>
<b>10</b>	<b>LayerGroup</b>	<b>45</b>
10.1	Basic Concepts . . . . .	45
10.2	BackwardH . . . . .	46
10.3	Dividing the Mem Cycle . . . . .	46
10.4	LMEM Allocation . . . . .	48
10.5	Divide the optimal Group . . . . .	50
<b>11</b>	<b>GMEM Allocation</b>	<b>51</b>
11.1	1. Purpose . . . . .	51
11.2	1. Principle . . . . .	51
11.2.1	2.1. GMEM allocation in weight tensor . . . . .	51
11.2.2	2.2. GMEM allocation in global neuron tensors . . . . .	52
<b>12</b>	<b>CodeGen</b>	<b>54</b>
<b>13</b>	<b>MLIR Definition</b>	<b>55</b>
13.1	Top Dialect . . . . .	55
13.1.1	Operations . . . . .	55
<b>14</b>	<b>Accuracy Validation</b>	<b>74</b>
14.1	Introduction . . . . .	74
14.1.1	Objects . . . . .	74
14.1.2	Metrics . . . . .	74
14.1.3	Datasets . . . . .	75
14.2	Validation Interface . . . . .	76
14.3	Validation Example . . . . .	76

14.3.1	mobilenet_v2 . . . . .	76
14.3.2	yolov5s . . . . .	78
<b>15</b>	<b>quantization aware traing</b>	<b>80</b>
15.1	Basic Principles . . . . .	80
15.2	tpu-mlir QAT implementation scheme and characteristics . . . . .	80
15.2.1	Main body flow . . . . .	80
15.2.2	Features of the Scheme . . . . .	81
15.3	Installation Method . . . . .	81
15.3.1	Install from source . . . . .	81
15.3.2	Installing the wheel file . . . . .	82
15.4	Basic Steps . . . . .	82
15.4.1	Step 1: Interface import and model prepare . . . . .	82
15.4.2	Step 2: Calibration and quantization training . . . . .	83
15.4.3	Step 3: Export tuned fp32 model . . . . .	83
15.4.4	Step 4: Initiate the training . . . . .	83
15.4.5	Step 5: Transform deployment . . . . .	84
15.5	Use Examples-resnet18 . . . . .	84
15.6	Tpu-mlir QAT test environment . . . . .	86
15.6.1	Adding a cfg File . . . . .	86
15.6.2	Modify and execute run_eval.py . . . . .	86
15.7	Use Examples-yolov5s . . . . .	87
<b>16</b>	<b>TpuLang Interface</b>	<b>88</b>
16.1	Main Work . . . . .	88
16.2	Work Process . . . . .	88
16.3	Operator Conversion Example . . . . .	90
<b>17</b>	<b>Custom Operators</b>	<b>95</b>
17.1	Overview . . . . .	95
17.2	Custom Operator Addition Process . . . . .	96
17.2.1	Add TpuLang Custom Operator . . . . .	96
17.2.2	Add Caffe Custom Operator . . . . .	98
17.3	Custom Operator Example . . . . .	99
17.3.1	Example of TpuLang . . . . .	99
17.3.2	Example of Caffe . . . . .	103
<b>18</b>	<b>Appendix.01: Migrating from NNTC to tpu-mlir</b>	<b>106</b>
18.1	ONNX to MLIR . . . . .	106
18.2	Make a quantization calibration table . . . . .	107
18.3	Generating int8 models . . . . .	108

---

## TABLE OF CONTENTS

---



# SOPHON

### **Legal Notices**

Copyright © SOPHGO 2022. 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**

---

## TABLE OF CONTENTS

---

**+86-10-57590723**

+86-10-57590724

### Release Record

Version	Release date	Explanation
v0.6.0	2022.11.05	support mix precision
v0.5.0	2022.10.20	support test model_zoo models
v0.4.0	2022.09.20	support convert caffe model
v0.3.0	2022.08.24	Support TFLite. Add the chapter on TFLite model conversion.
v0.2.0	2022.08.02	Add the chapter on test samples in running SDK.
v0.1.0	2022.07.29	Initial release, supporting resnet/mobilenet/vgg/ssd/yolov5s and using yolov5s as the use case.

# CHAPTER 1

## TPU-MLIR Introduction

TPU-MLIR is the TPU compiler project for AI chips. 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 TPUs. The code has been open-sourced to github: <https://github.com/sophgo/tpu-mlir>.

The overall architecture of TPU-MLIR is as follows:

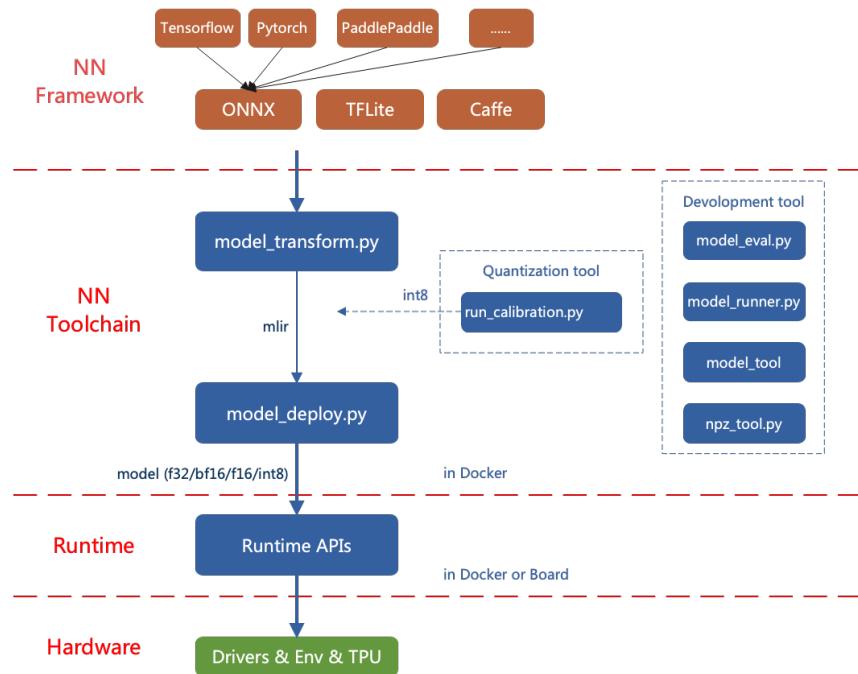


Fig. 1.1: TPU-MLIR overall architecture

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:v3.1
```

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
```

(continues on next page)

(continued from previous page)

```
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:v3.1  
# "myname" is just an example, you can use any name you want
```

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
```

# CHAPTER 3

---

## User Interface

---

This chapter introduces the user interface.

### 3.1 Introduction

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`. Calibration is required if you need to get the INT8 model. The general process is shown in the figure ([User interface 1](#)).

Other complex cases such as image input with preprocessing and multiple inputs are also supported, as shown in the figure ([User interface 2](#)).

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 \
```

(continues on next page)

(continued from previous page)

```
--chip 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
```

Supporting the conversion of Caffe models, the commands are as follows:

```
# 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
# The call of model_deploy is consistent with onnx
# .....
```

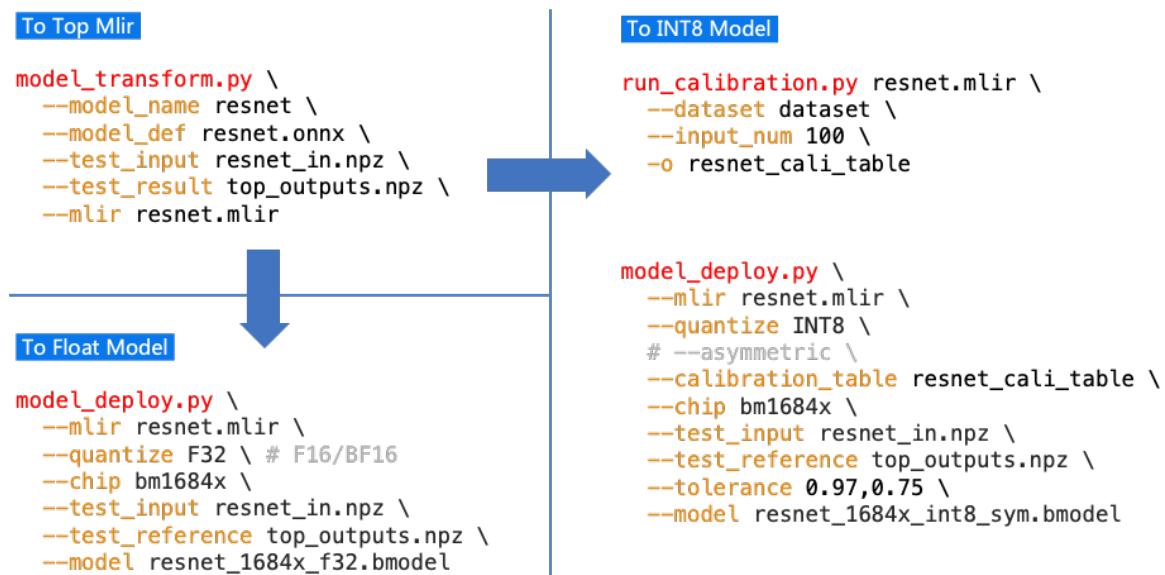


Fig. 3.1: User interface 1

**Support Picture Input**

```
model_transform.py \
--model_name resnet \
--model_def resnet.onnx \
--input_shapes [[1,3,224,224]] \
--resize_dims 256,256 \
--mean 123.675,116.28,103.53 \
--scale 0.0171,0.0175,0.0174 \
--pixel_format rgb \
--test_input cat.jpg \
--test_result top_outputs.npz \
--mlir resnet.mlir
```

**Support List Cali**

```
run_calibration.py resnet.mlir \
--data_list files.txt \
--input_num 100 \
-o resnet_cali_table
```

**Support Multiple Inputs****Method One:**

```
# all inputs in one npz file
model_transform.py \
--model_name somenet \
--model_def somenet.onnx \
--test_input some_in.npz \
--test_result top_outputs.npz \
--mlir somenet.mlir
```

**Method Two:**

```
# inputs in npy files
model_transform.py \
--model_name somenet \
--model_def somenet.onnx \
--test_input a.npy,b.npy,c.npy \
--test_result top_outputs.npz \
--mlir somenet.mlir
```

Fig. 3.2: User interface 2

## 3.2 model\_transform.py

Used to convert various neural network models into MLIR files, the supported parameters are shown below:

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)
model_data	N	Specify the model weight file, required when it is caffe model (corresponding to the ‘.caffemodel’ file)
input_shapes	N	The shape of the input, such as [[1,3,640,640]] (a two-dimensional array), which can support multiple inputs
input_types	N	Type of the inputs, such int32; separate by ‘,’ for multi inputs; float32 as default
resize_dims	N	The size of the original image to be adjusted to. If not specified, it will be resized to the input size of the model
keep_aspect_ratio	N	Whether to maintain the aspect ratio when resize. False by default. It will pad 0 to the insufficient part when setting
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 use the specified names as the output
add_postprocess	N	add postprocess op into bmodel, set the type of post handle op such as yolov3/yolov3_tiny/yolov5/ssd
test_input	N	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	N	Output file to save validation result
excepts	N	Names of network layers that need to be excluded from validation. Separated by comma
onnx_sim	N	option for onnx-sim, currently only support ‘skip_fuse_bn’ args
mlir	Y	The output mlir file name (including path)

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.

### 3.3 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
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
tune_num	N	The number of fine-tuning samples. 10 by default
histogram_bin_num	N	The number of histogram bins. 2048 by default
o	Y	Name of output calibration table file

### 3.4 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
quantize	Y	Quantization type (F32/F16/BF16/INT8)
chip	Y	The platform that the model will use. Support bm1686/bm1684x/bm1684/cv186x/cv183x/cv182x/cv181x/cv180x.
calibration_table	N	The quantization table path. Required when it is INT8 quantization
ig-nore_f16_overflow	N	Operators with F16 overflow risk are still implemented according to F16; otherwise, F32 will be implemented by default, such as LayerNorm
tolerance	N	Tolerance for the minimum similarity between MLIR quantized and MLIR fp32 inference results
test_input	N	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_reference	N	Reference data for validating mlir tolerance (in npz format). It is the result of each operator
excepts	N	Names of network layers that need to be excluded from validation. Separated by comma
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
core	N	When the target is selected as bm1686 or cv186x, it is used to select the number of tpu cores for parallel computing, and the default setting is 1 tpu core

## 3.5 Other Tools

### 3.5.1 model\_runner.py

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

Example:

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

Supported parameters:

Table 3.4: Function of model\_runner parameters

Name	Required?	Explanation
input	Y	Input npz file
model	Y	Model file (bmodel/mlir/onnx/tflite)
dump_all_tensors	N	Export all the results, including intermediate ones, when specified

### 3.5.2 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 3.5: 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

### 3.5.3 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 3.6: visual 功能

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
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](#))

# CHAPTER 4

---

## Overall Design

---

### 4.1 Layered

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

#### Top Dialect

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

#### Tpu Dialect

Chip-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.

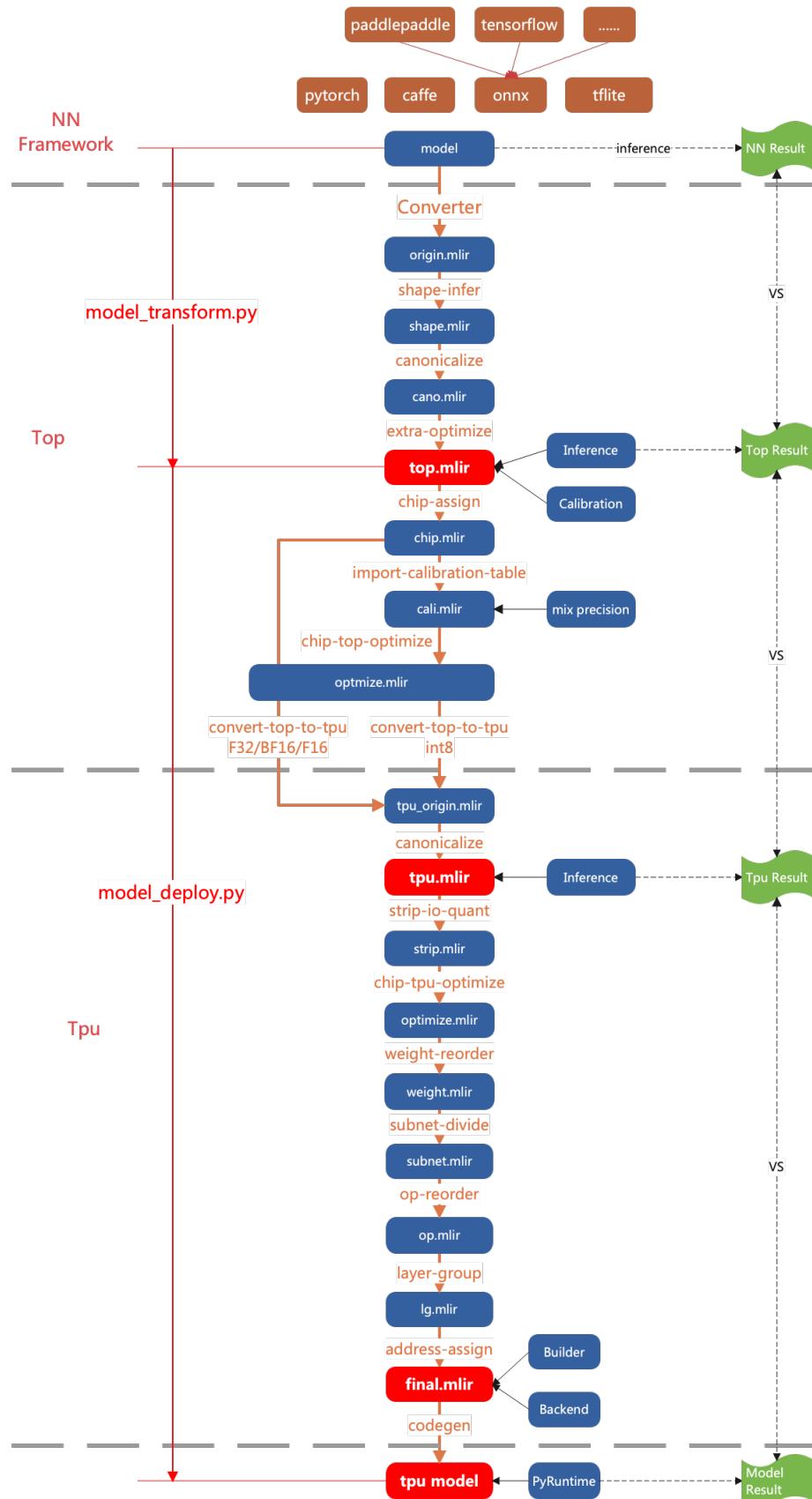


Fig. 4.1: TPU-MLIR overall process  
Copyright © SOPHGO

**chip-assign**

Assign chip, such as bm1684x, cv183x, etc; and adjust top mlir by chip, 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.

**chip-top-optimize**

Do top ops optimization by chip.

**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

**chip-tpu-optimize**

Do tpu ops optimization by chip.

**weight-reorder**

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

**subnet-divide**

Split the network into different subnets according to TPU/CPU, if all operators are TPU, there is only one 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 (chip-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).

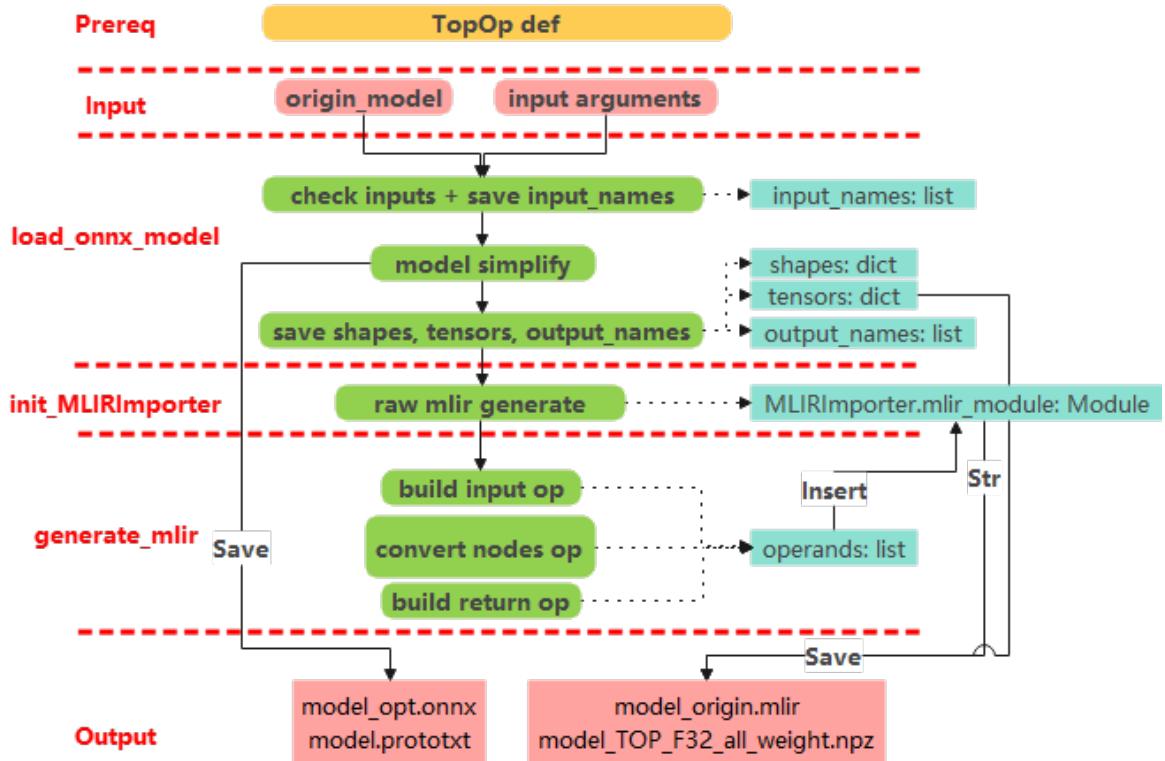


Fig. 5.1: Front-end conversion workflow

### 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.

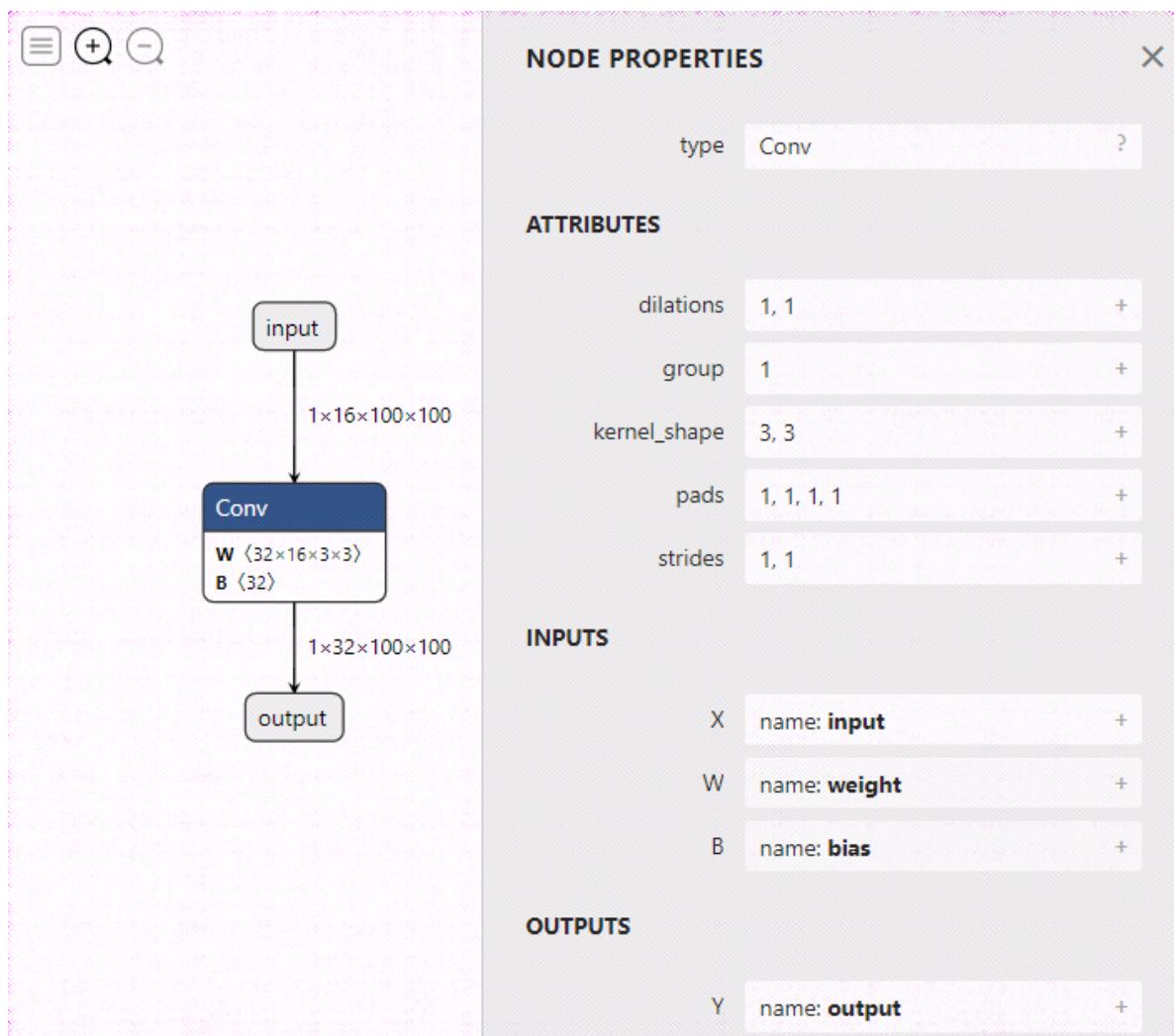


Fig. 5.2: Conv onnx model

```
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.

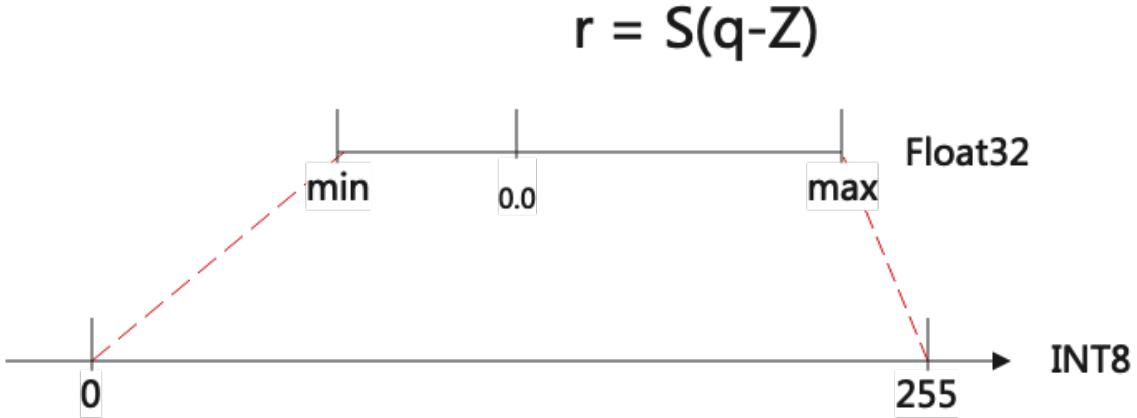


Fig. 6.1: Asymmetric quantization

$S$  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 chip 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:

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

The way to implement Add with TPU is related to specific TPU 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:

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

### 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  $S_x/S_y = 1$ .

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

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  $S_x/S_y = 1$ .

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

In the symmetric case, both Zx and Zy are 0, so the padded value is round(value/Sy). When asymmetric quantization, the padded value is round(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:

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

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

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

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. Then aggregate them to obtain the overall minmax and histogram of the data point distribution. The appropriate symmetric quantization threshold is obtained based on algorithms such as KLD. Finally, the auto-tune algorithm will be enabled to tune the quantization threshold of the input activation of a certain int8 layer by making use of the Euclidean distance between the output activation of int8 and fp32 layers. The above processes are integrated together and executed in unison. The optimized threshold and min/max values for each op are saved in a text file for quantization parameters. Int8 quantization can be achieved by using this text file in `model_deploy.py`. The overall process is shown in the figure ([Overall process of quantization](#)).

The following figure ([Example of quantization parameters file](#)) shows the final output of the calibration quantization parameters file

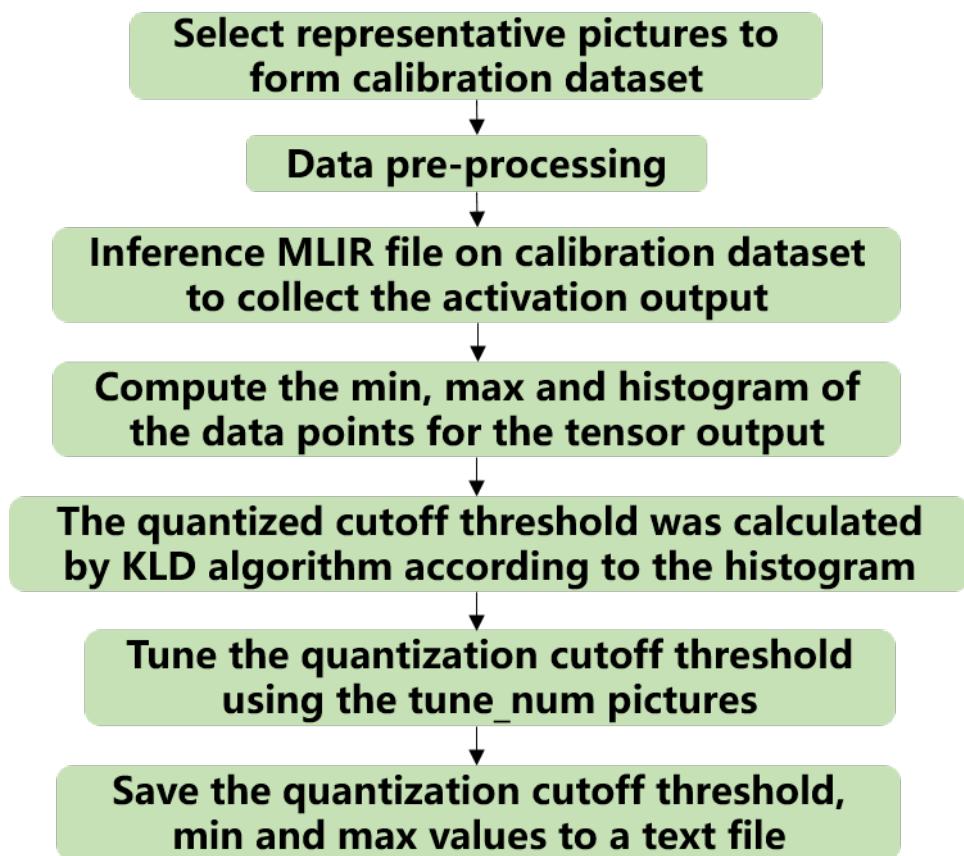


Fig. 7.1: Overall process of quantization

```
# 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. 7.2: Example of quantization parameters file

## 7.2 Calibration data screening and preprocessing

### 7.2.1 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.

### 7.2.2 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. 7.3: Example of `data_list` required format

## 7.3 Algorithm Implementation

### 7.3.1 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 waveform of abs (fp32\_tensor) (represented by the histogram of 2048 fp32 bins) to get the fp32 reference probability distribution P. This fp32 waveform 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)
14  end for
15
16  find i which kld[i] is minimal
17  int8 quantize threshold = (i + 0.5)*fp32 absmax/2048

```

### 7.3.2 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 waveform 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 10 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](#)).

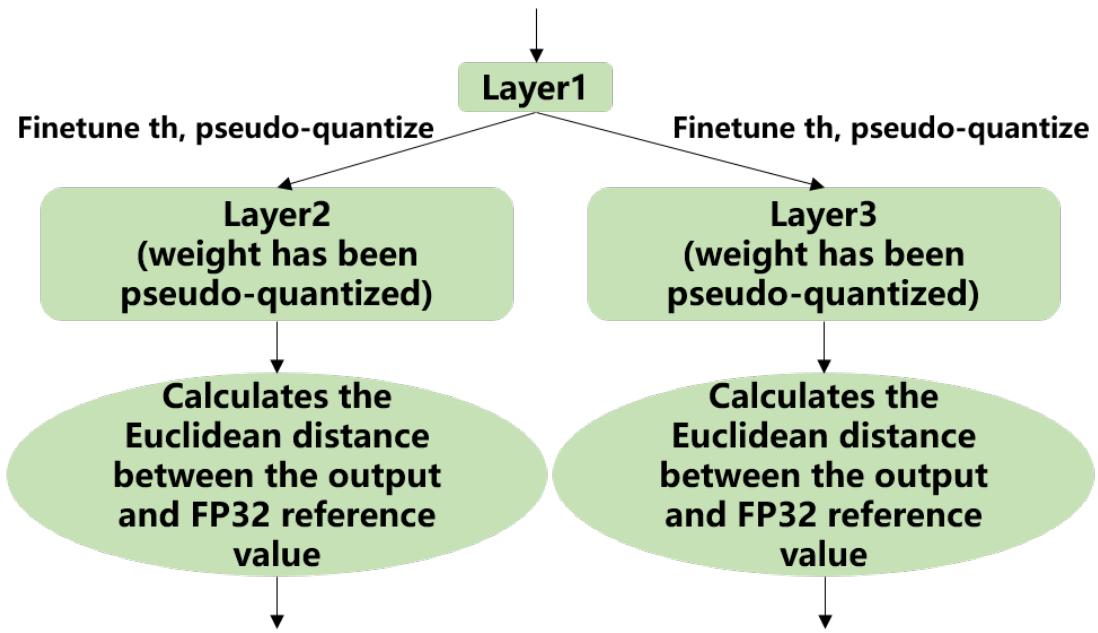


Fig. 7.4: Implementation of auto-tune

## 7.4 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 \
    --mlir yolov5s.mlir

```

Table 7.3: The arguments of `model_transform.py`

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

```

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

```

Table 7.4: The arguments of run\_calibration.py

Argument	Description
mlir_file	mlir file
-dataset	dataset for calibration
-data_list	Input list file contain all input
-input_num	num of images for calibration
-tune_list	Tune list file contain all input for tune
-tune_num	num of images for tune
-	Specify histogram bin numer for kld calculate
histogram_bin_num	
-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.

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

## 7.5 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. 7.5: 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:**

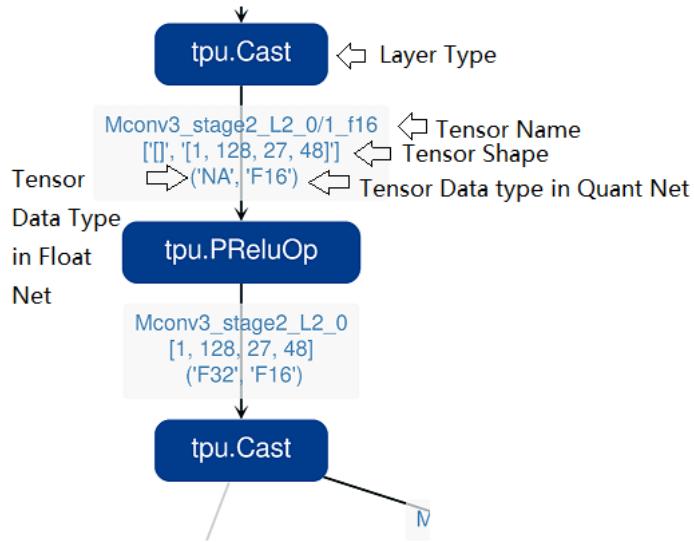
1. working directory and net file indication
2. accuracy summary area
3. layer information area

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 chips, 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 chip-independent layer (TOP), to the chip-related layer (TPU).

### 8.1 Basic Process

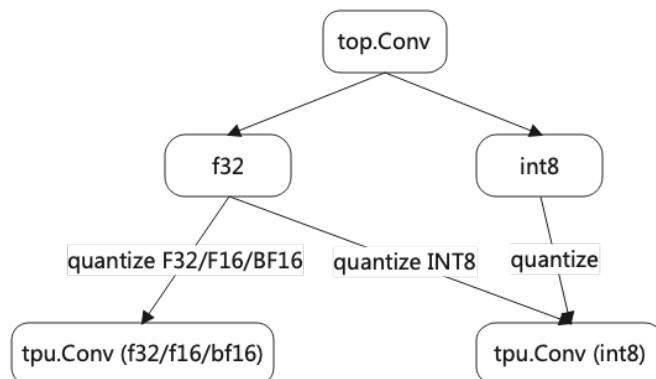


Fig. 8.1: Lowering process

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

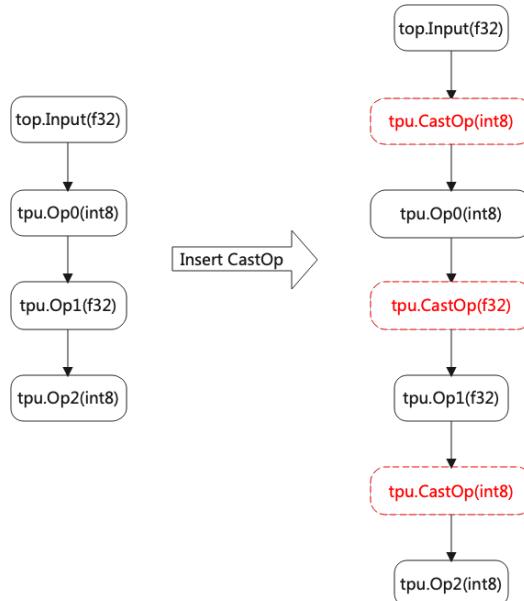


Fig. 8.2: Mixed precision

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

TPU chip memory can be divided into off-chip memory (i.e., Global Memory or GMEM) and on-chip memory (i.e., Local Memory or 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](#)).

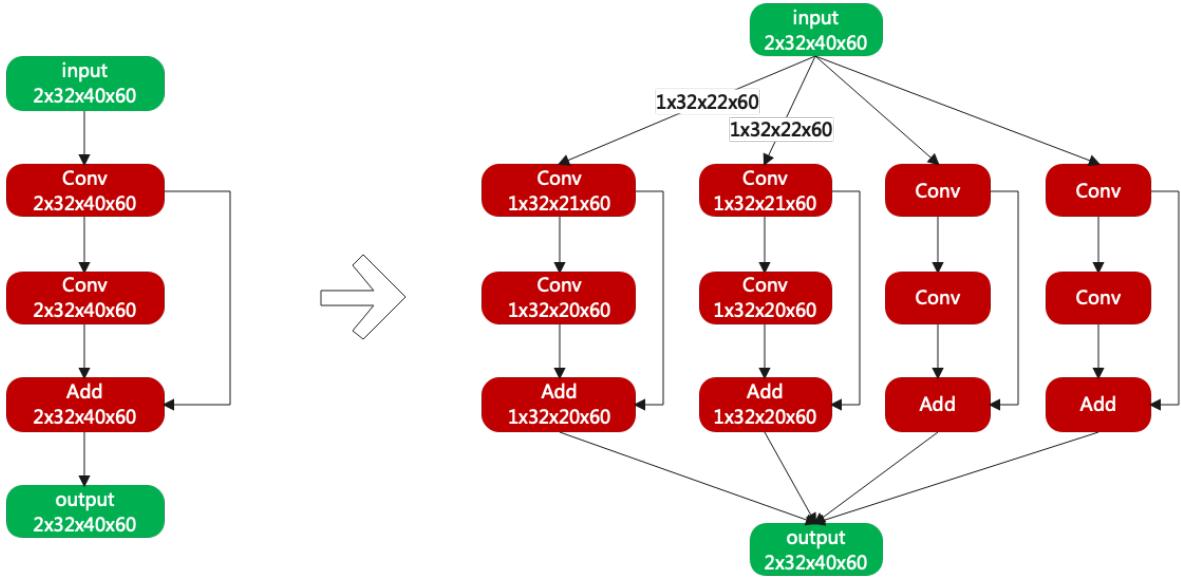


Fig. 10.1: Network slicing example

## 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.

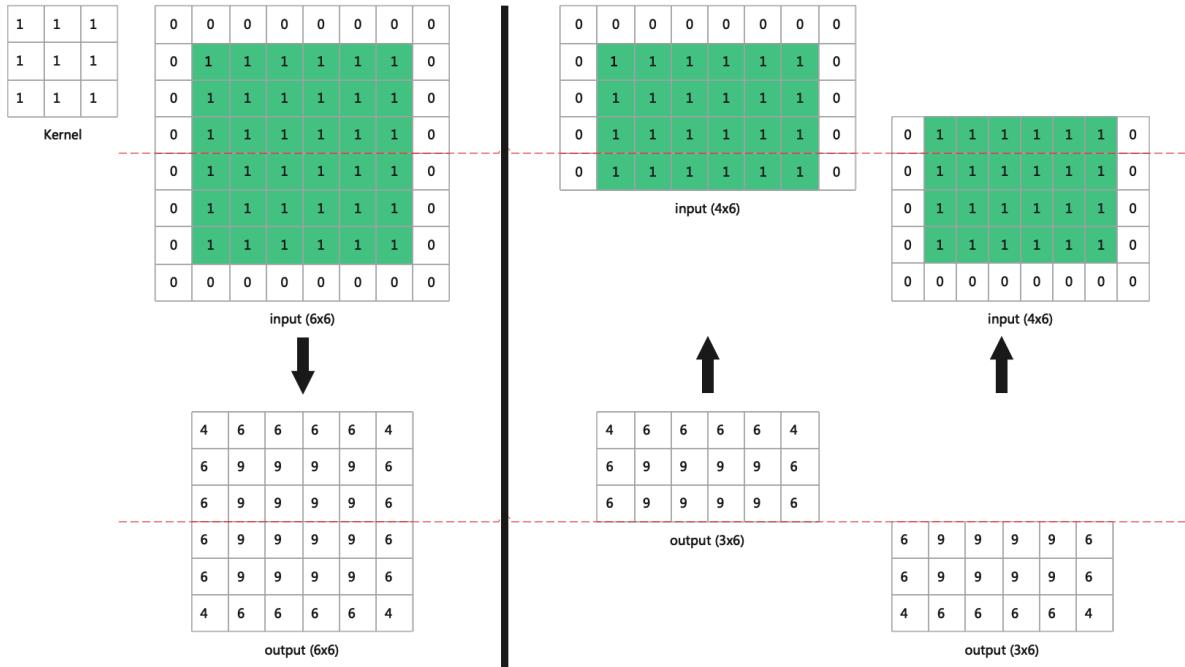


Fig. 10.2: Convolutional BackwardH example

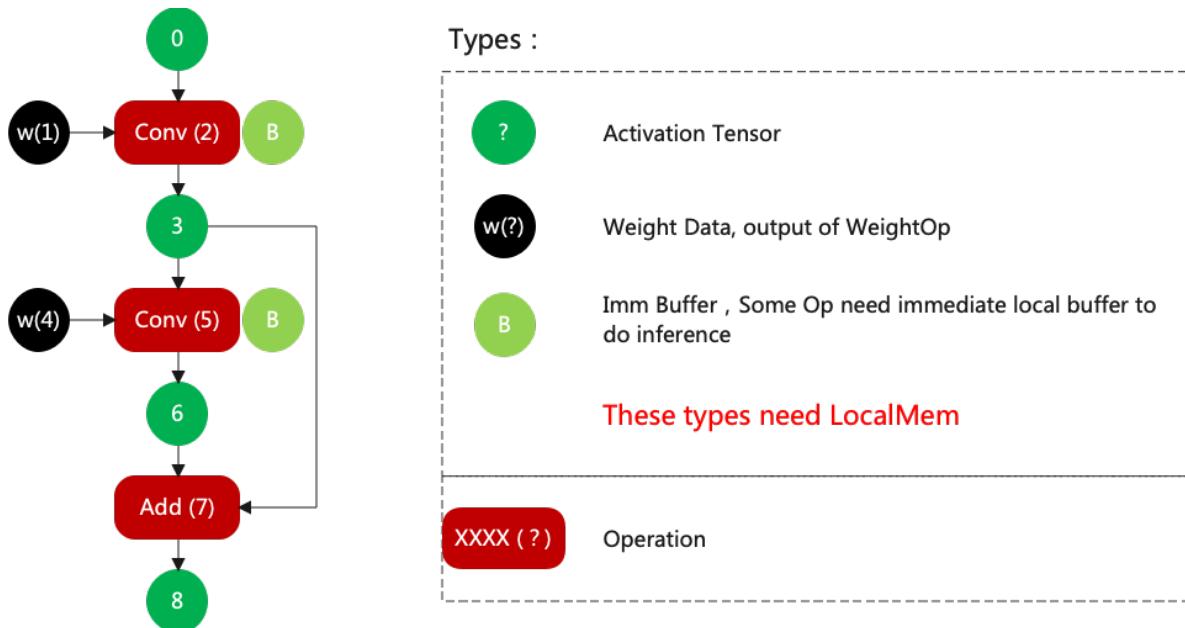


Fig. 10.3: LMEM's ID assignment

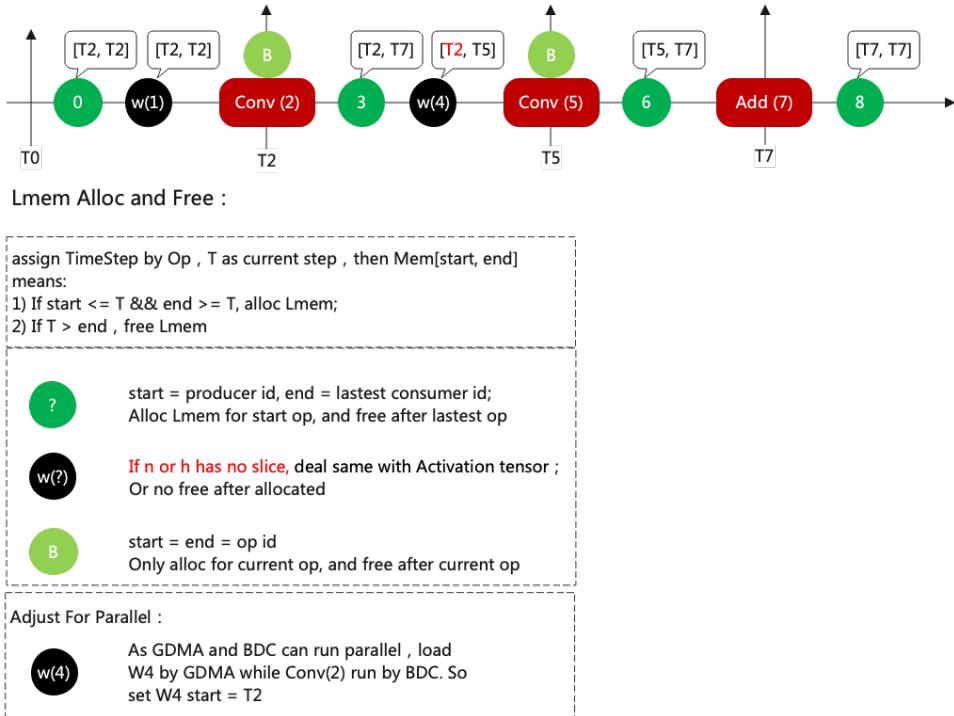


Fig. 10.4: TimeStep assignment

- 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.

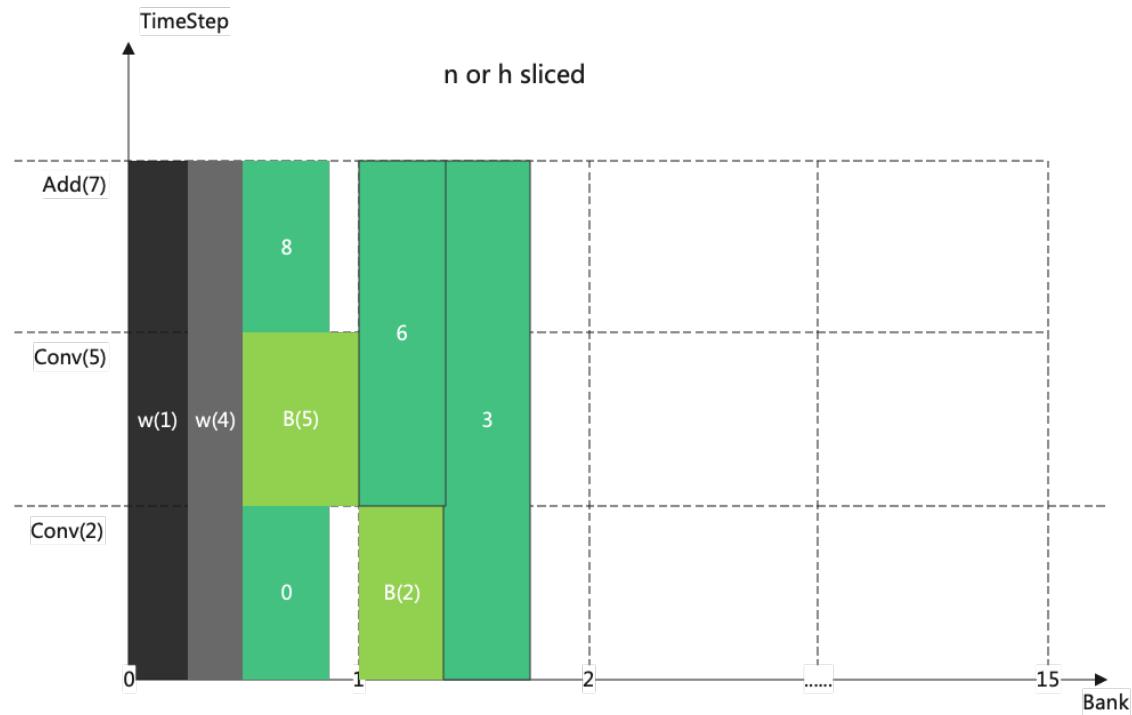


Fig. 10.5: Allocation in the case of slice

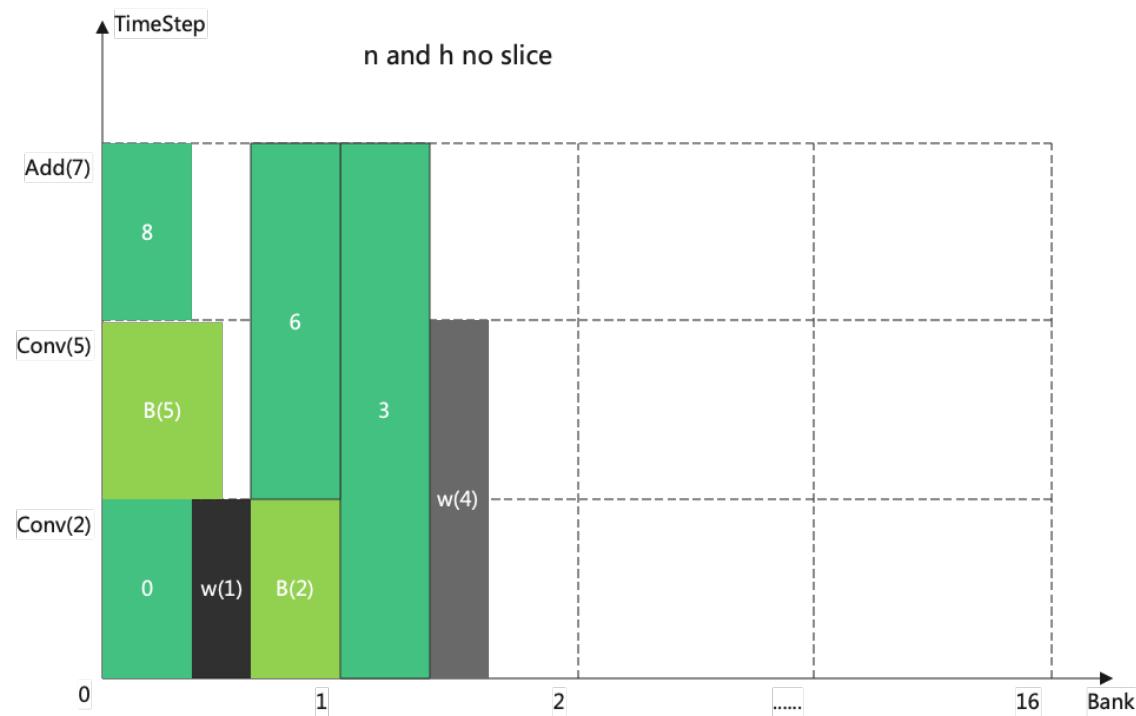


Fig. 10.6: Allocation in the case of no slice

## 10.5 Divide the optimal Group

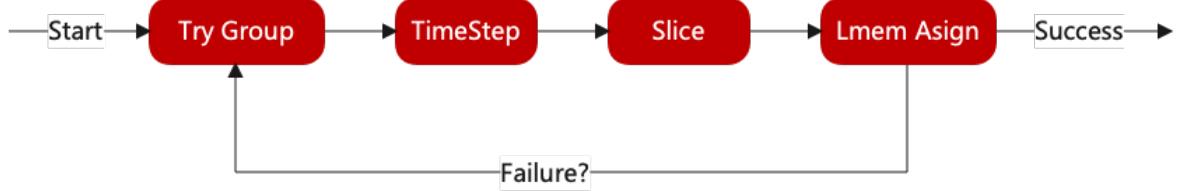


Fig. 10.7: Group process

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:

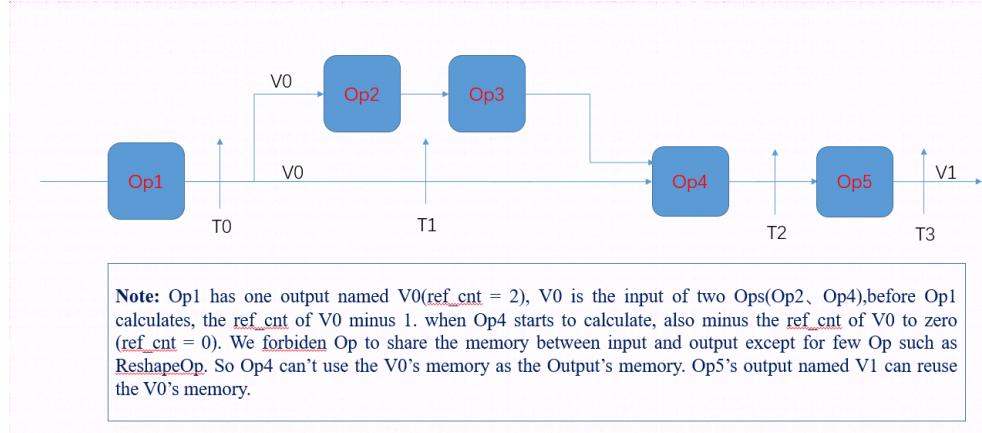
The corresponding tensor, address, size, ref\_cnt (how many OPs are using this tensor) are recorded in rec\_tbl at each allocation. The tensor and address are recorded in the auxiliary data structures hold\_edges,in\_using\_addr respectively.

```
//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

---

# 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 \
--chip 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 based on pytorch fx can automatically complete model trace, pseudo-quantization node insertion, custom module replacement and other operations. In most cases, users can complete model transformation with one click using the default configuration.

Feature 3:This scheme is based on Sensetime's open source mqbench qat training framework, which has a certain community foundation and is convenient for industry and academia to evaluate reasoning performance and accuracy on our tpu.

## 15.3 Installation Method

### 15.3.1 Install from source

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

2、Execute after entering the MQBench directory:

```
pip install -r requirements.txt #Note: torch version 1.10.0 is currently required  
python setup.py install
```

3、If python -c ‘import mqbench’ does not return any error, the installation is correct. If the installation is incorrect, run pip uninstall mqbench and try again.

### 15.3.2 Installing the wheel file

Download the python whl package from <https://MQBench-1.0.0-py3-none-any.whl> and run pip3 install MQBench-1.0.0-py3-none-any.whl to install it directly.

## 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:

```
#Initializing Interface
from mqbench.prepare_by_platform import prepare_by_platform, BackendType
#Calibrate and quantify switches
from mqbench.utils.state import enable_calibration, enable_quantization
#Transform Deployment interface
from mqbench.convert_deploy import convert_deploy
#Use the pre-trained resnet18 model in torchvision model zoo
model = torchvision.models.__dict__["resnet18"](pretrained=True)
Backend = BackendType.sophgo_tpu
#1.trace model and then add quantization nodes in a specific way based on the requirements of F
→sophgo_tpu hardware
model_quantized = prepare_by_platform(model, Backend)
```

When sophgo\_tpu backend is selected on the above interface, the third parameter prepare\_custom\_config\_dict of this interface is not configured by default. In this case, the default quantization configuration is shown as the following figure:

```
prepare_by_platform.py > ObserverDict
BackendType.Sophgo_TPU: dict(qtype='affine',      # noqa: E241
                            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)
```

In the above figure, items in the dict behind sophgo\_tpu in order of top to bottom meaning are:

- 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

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

### 15.4.4 Step 4: Initiate the training

**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;

### 15.4.5 Step 5: Transform deployment

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

## 15.5 Use Examples-resnet18

Run example/imagenet\_example/main.py to qat train resnet18 as follows:

```
python3 imagenet_example/main.py
--arch=resnet18
--batch-size=192
--epochs=1
--lr=1e-4
--gpu=0
--pretrained
--backend=sophgo_tpu
--optim=sgd
--deploy_batch_size=10
--train_data=/data/imagenet/for_train_val/
--val_data=/data/imagenet/for_train_val/
--output_path=/workspace/classify_models
```

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)) :

```
5 => using pre-trained model 'resnet18'
6 原始onnx模型精度
7 Test: [ 0/261] Time 3.447 ( 3.447)    Loss 6.8051e-01 (6.8051e-01)    Acc@1 80.21 ( 80.21)
8 Test: [100/261] Time 0.504 ( 0.483)   Loss 8.1262e-01 (9.0736e-01)    Acc@1 73.96 ( 76.23)
9 Test: [200/261] Time 0.099 ( 0.467)   Loss 1.0743e+00 (1.1929e+00)    Acc@1 80.21 ( 70.82)
0 * 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:

```
qat训练后的带量化节点的eval精度:
Test: [ 0/391] Time 2.255 ( 2.255)    Loss 6.4446e-01 (6.4446e-01)    Acc@1 82.81
Test: [100/391] Time 0.130 ( 0.427)   Loss 7.2533e-01 (8.9875e-01)    Acc@1 80.47
Test: [200/391] Time 0.129 ( 0.424)   Loss 1.1836e+00 (1.0291e+00)    Acc@1 66.41
Test: [300/391] Time 0.128 ( 0.421)   Loss 1.3070e+00 (1.1715e+00)    Acc@1 73.44
* Acc@1 69.986 Acc@5 89.256
```

Fig. 15.2: resnet18 qat training accuracy

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

```

46803476 Nov  7 22:14 resnet18_mqmobile.onnx
47005085 Nov  7 22:14 resnet18_mqmobile.pt
116387 Nov  7 22:14 resnet18_mqmobile_cali_table_from_mqbench_sophgo_tpu
286163 Nov  7 22:14 resnet18_mqmobile_clip_ranges.json
46744954 Nov  7 22:14 resnet18_mqmobile_deploy_model.onnx
46743449 Nov  7 19:20 resnet18_ori.onnx
46843277 Nov  7 19:20 resnet18_ori.pt

```

Fig. 15.3: resnet18 qat training output model directory

The one with \_ori in the figure above is the original pt of pytorch model zoo and the 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\_mqmobile\_cali\_table\_from\_mqbench\_sophgo\_tpu 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 1.1920
007888888940215111 0.0003651758888736367 1.1920928955078125e-07 0.0006113630370236933 0.0016739938873797655 0.00038
1655654632486403 0.0031490952242165804 0.0002667098306119442 0.0019505330128595233 0.0006429849308915436 0.00078482
684491302818 1.1920928955078125e-07 0.0003549025859683752 1.1920928955078125e-07 0.003084442811086774 0.00230932701
6998806 1.1920928955078125e-07 0.002218325389549136 0.0025145262479782104 0.0012810979959799519 0.00082650256808847
72 0.0010380028979852796 0.00019447231898084283 0.00037283531855791807 0.0010219684336334467 64 0 0 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

### 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_ann
]
```

```
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
```

(continues on next page)

(continued from previous page)

```
--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

Similar to resnet18, run the following command in example/yolov5\_example to start qat training:

```
python3 train.py
--cfg=yolov5s.yaml
--weights=yolov5s.pt
--data=coco.yaml
--epochs=5
--output_path=/workspace/yolov5/qat_models
--batch-size=8
--quantize
```

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

# 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 (chip-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;
    - Inference output shape, and 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](#)).

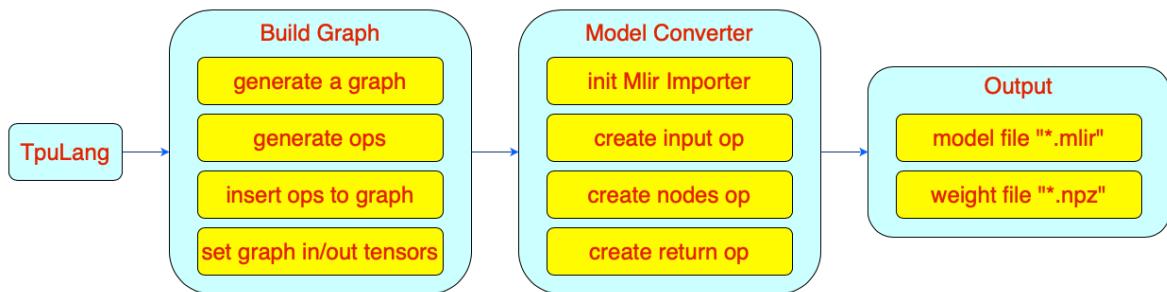


Fig. 16.1: TpuLang conversion process

### 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);
  - According to the parameters extracted by the interface, the output\_shape is obtained by inference (i.e., shape\_inference is required);
  - 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. The original model definition is shown in the figure ([Single Conv Model](#))

```
def model_de():
    bml.init("BM1684X", True)
    in_shape = [1,3,173,141]
    k_shape = [64,1,7,7]
    x = bml.Tensor(dtype='float32', shape=in_shape)      You, 现在 + Uncommitted changes
    weight_data = np.random.random(k_shape).astype(np.float32)
    weight = bml.Tensor(dtype='float32', shape=k_shape, data=weight_data, is_const=True)
    bias_data = np.random.random(k_shape[0]).astype(np.float32)
    bias = bml.Tensor(dtype='float32', shape=k_shape[0], data=bias_data, is_const=True)
    conv = bml.conv_v2(x, weight, bias=bias, stride=[2,2], pad=[0,0,1,1],
                      out_dtype="float32")
    bml.compile("model_def", inputs=[x], outputs=[conv], cmp=True)
    bml.deinit()
```

Fig. 16.2: Single Conv Model

The conversion process:

1. Interface definition

The conv\_v2 interface is defined as follows:

```
def conv_v2(tensor_i,
            weight,
            bias = None,
            stride = None,
            dilation = None,
            pad = None,
            group = 1,
            input_zp = None,
            weight_zp = None,
            out_dtype = None,
            out_name = None):
    # pass
```

#### Parameter Description

- tensor\_i: 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
- input\_zp: List[int] type or int type, indicating the input offset. If None, input\_zp equals 0. Otherwise, the length of List is required to be ic.
- weight\_zp: List[int] type or int type, indicating the convolution kernel offset. If None, weight\_zp equals 0. Otherwise, the length of list is required to be ic, where ic represents the number of input channels.
- 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](#))

### 1. 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\_v2 interface:
  - Call the conv\_v2 interface with specified input tensor and input parameters.
  - Inference output shape, and generate output tensor

```
def _shape_inference():
    kh_ext = dilation[0] * (weight.shape[2] - 1) + 1
    kw_ext = dilation[1] * (weight.shape[3] - 1) + 1
    oh = (input.shape[2] + pad[0] + pad[1] - kh_ext) // stride[0] + 1
    ow = (input.shape[3] + pad[2] + pad[3] - kw_ext) // stride[1] + 1
    return [input.shape[0], weight.shape[0], oh, ow]
output = Tensor(_shape_inference(), dtype=out_dtype, name=out_name)
```

- attributes, pack the input parameters into attributes defined by ([Conv Operator Definition](#))

```
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. 16.3: 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)
}
    
```

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

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_F
32", module.weight_file = "conv2d_top_f32_all_weight.npz"} {
  func.func @main(%arg0: tensor<1x16x100x100xf32>) -> tensor<1x32x100x100xf32> {
    %0 = "top.None"() : () -> none
  }
}
    
```

Fig. 16.4: Initial Mlir Text

3. 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](#)).

#### 4. Output

```

#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<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. 16.5: Full Mlir Text

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

# 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 TPU 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 layer group 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.

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

### 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. Develop backend operators based on TPU-Kernel

Assuming the current path is \$TPUC\_ROOT/customlayer, add the nodechip\_{op\_name}.h header file in the ./include directory to declare the custom operator functions for the global layer and local layer (void nodechip\_{op\_name}\_global and void nodechip\_{op\_name}\_local, respectively). Then, add the nodechip\_{op\_name}.c file in the ./src directory and invoke the TPU-Kernel interfaces to implement the corresponding functions.

3. Define the operator's parameter structure and write the operator's interface

- a. Add the corresponding structure {op\_name}\_param\_t in the ./include/backend\_custom\_param.h header file to receive parameters from the frontend of toolchain, based on the parameters required by the operator.
- b. Add the api\_{op\_name}.h header file in the ./include directory to declare the interfaces for the custom operator functions (void api\_{op\_name}\_global and void api\_{op\_name}\_local). Then, add the api\_{op\_name}.c file in the ./src directory and implement the corresponding interfaces.
- c. Additionally, users need to 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, where each 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 struct {
    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;
```

4. Define the backend interface

In ./src/backend\_custom\_api.cpp, build the backend interface using macro definitions. This interface will be called during Codegen in the frontend of toolchain. The format is as follows:

```
IMPL_CUSTOM_API_GLB({op_name}, {op_name}_param_t)
IMPL_CUSTOM_API_LOC({op_name}, {op_name}_param_t)
```

5. Compile and install the dynamic library

By running the build.sh script in \$TPUC\_ROOT/customlayer, the compilation of the custom operator will be completed. It will generate the backend\_custom\_api.so dynamic library and install it in \$TPUC\_ROOT/lib.

#### 6. 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 (ensure that the op\_name part matches the name of the backend operator):

```
TpuLang.custom(tensors_in: list,
               shape_func,
               op_name: str,
               out_dtypes: list,
               out_names: list = None,
               params: dict = None)
...
The custom op
Arguments:
    tensors_in: list of input tensors (including weight tensors)
    shape_func: function for doing shape inference, taking tensors_in as the
                parameter, return is the list of output tensors shape
    op_name: name of the custom operator,
    out_dtypes: list of outputs' data type
    out_name: list of output names
    params: parameters of the custom op

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

#### 17.2.2 Add Caffe Custom Operator

Steps 1-5 are the same as in Add TpuLang Custom Operator section, and will not be repeated here.

#### 6. 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\_layer.py that inherits from caffe.Layer and override the setup, reshape, forward, and backward functions as needed.

#### 7. Implementing the frontend conversion function

The type of custom operators implemented in python is “Python”, so you need to implement a corresponding conversion function of MyCaffeConverter class defined in the file \$TPUC\_ROOT/customlayer/python/my\_converter.py, based on the definition in step 6.

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
```

## 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. Backend Operator Implementation

The following is the declaration in the header file

`${TPUC_ROOT}/customlayer/include/nodechip_swapchannel.h:`

```
#ifndef NODECHIP_ABSADD_H_
#define NODECHIP_ABSADD_H_

#include "tpu_kernel.h"

#ifndef __cplusplus
extern "C" {
#endif

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

#ifndef __cplusplus
}
#endif

#endif
```

The code of  `${TPUC_ROOT}/customlayer/src/nodechip_swapchannel.c:`

```
#include "nodechip_swapchannel.h"
#include "common.h"

void nodechip_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);
    }
}
```

## 2. Operator Parameter Structure and Implementation of the Operator Interface

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;
```

The following is the declaration in the header file

`${TPUC_ROOT}/customlayer/include/api_swapchannel.h`:

```
#pragma once
#include "api_common.h"
#include "backend_custom_param.h"

#ifndef __cplusplus
extern "C" {
#endif

void api_swapchannel_global(
    global_tensor_spec_t *input,
    global_tensor_spec_t *output,
    custom_param_t *param);

#ifndef __cplusplus
```

(continues on next page)

(continued from previous page)

```
}
```

```
#endif
```

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

```
#include "tpu_utils.h"
#include "api_swapchannel.h"
#include "nodechip_swapchannel.h"

// parse param function
swapchannel_param_t parsParam(custom_param_t* param) {
    swapchannel_param_t sc_param = {0};
    for (int i = 0; i < 3; i++) {
        sc_param.order[i] = param[0].int_arr_t[i];
    }
    return sc_param;
}

// global api function
void api_swapchannel_global(
    global_tensor_spec_t *input,
    global_tensor_spec_t *output,
    custom_param_t *param)
{
    swapchannel_param_t sc_param = parsParam(param);

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

### 3. Backend Interface

The code of \${TPUC\_ROOT}/customlayer/src/backend\_custom\_api.cpp:

```
#include "backend_helper.h"
#include "common_def.h"
#include "api_common.h"

// 1. include head file of api function
#include "api_swapchannel.h"

// 2. global backend api functions
IMPL_CUSTOM_API_GLB(swapchannel, swapchannel_param_t)
```

After completing the implementation of the backend interface, you can run \${TPUC\_ROOT}/customlayer/build.sh to compile and install the custom operator dynamic library.

#### 4. TpuLang Interface Invocation

Here is an example of Python code that utilizes the TpuLang interface to build a custom operator model:

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

# 1. initialize tpulang
tpul.init("BM1684X", True)

# 2. prepare the input
dtype = "float32"
input_shape = [1, 3, 14, 14]
x_data = np.random.random(input_shape).astype(np.float32)
x = tpul.Tensor(dtype=dtype, shape=input_shape, data=x_data)

# 3. build model
def shape_func(tensors_in):
    # the shape inference function
    return [tensors_in[0].shape]

out_names = ["out"]
params = {"order": [2, 1, 0]}

outs = tpul.custom(
    tensors_in=[x],
    shape_func=shape_func,
    # op_name should be consistent with the backend
    op_name="swapchannel",
    params=params,
    out_dtypes=[dtype],
    out_names=out_names)

# 4. compile to Top mlir file, the input will be saved in {top_mlir}_in_f32.
#→npz
top_mlir = "tpulang_test_net"
tpul.compile(top_mlir, [x], outs, False, 2, has_custom=True)
```

By running the above code, you can obtain the Top MLIR file `tpulang_test_net.mlir`. For the subsequent model deployment process, please refer to the User Interface chapter.

### 17.3.2 Example of Caffe

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

#### 1. 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.

#### 2. Defining Caffe custom operators

The definition of absadd and ceiladd in Caffe can be found in \$TPUC\_ROOT/customlayer/python/my\_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):
        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_layer"
        layer: "AbsAdd"
        param_str: "{ 'b_val': 1.2}"
    }
}

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

### 3. 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)

    # 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_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])
```

### 4. 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 ab-

sadd 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.

# CHAPTER 18

## 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.

### 18.1 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.

## 18.2 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.

It is worth mentioning that mlir also has a run\_qtable tool to help generate mixed-precision models

### 18.3 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 \
  --chip 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.