TPU-MLIR Quick Start

Release 1.3.228

SOPHGO

Sep 21, 2023

Table of contents

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

Tel

+86-10-57590723
+86-10-57590724

Release Record

Version	Release date	Explanation
v1.3.0	2023.07.27	Add the function to manually specify operations computing with floating-point; Add the list of supported front-end framework operators; Add a comparison between NNTC and TPU-MLIR quantization methods.
v1.2.0	2023.06.14	Adjusted the mixed quantization example
v1.1.0	2023.05.26	Added using TPU for post-processing
v1.0.0	2023.04.10	Support for PyTorch, added section introducing the conversion to PyTorch models
v0.8.0	2023.02.28	Added using TPU for pre-processing
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 shown in the figure (TPU-MLIR overall architecture).

The current directly supported frameworks are pytorch, onnx, tflite and caffe. 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/cvimodel 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 mainly introduces the process of this model conversion.

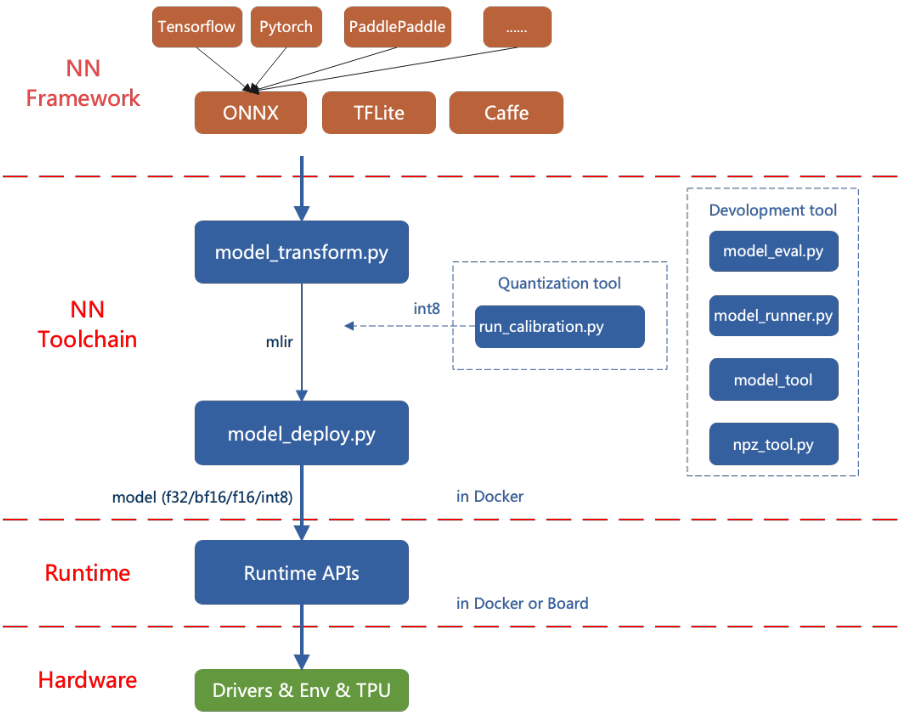


Fig. 1.1: TPU-MLIR overall architecture

CHAPTER 2

Environment Setup

Download the required image from DockerHub 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):

```
$ sudo apt install docker.io
$ sudo systemctl start docker
$ sudo systemctl enable docker
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ 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
```

Subsequent chapters assume that the user is already in the /workspace directory inside docker.

CHAPTER 3

Compile the ONNX model

This chapter takes yolov5s.onnx as an example to introduce how to compile and transfer an onnx model to run on the BM1684X TPU platform.

The model is from the official website of yolov5: https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

platform	file name	info
cv183x/cv182x/cv181x/cv180x	xxx.cvimodel	please refer to the CV18xx Guidance
其它	xxx.bmodel	please refer to the following

3.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 3.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

3.2 Prepare working directory

Create a model_yolov5s directory, note that it is the same level directory as tpu-mlir; and put both model files and image files into the model_yolov5s directory.

The operation is as follows:

```
$ mkdir model_yolov5s && cd model_yolov5s
$ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

3.3 ONNX to MLIR

If the input is image, we need to know the preprocessing of the model before transferring it. If the model uses preprocessed npz files as input, no preprocessing needs to be considered. The preprocessing process is formulated as follows (x represents the input):

$$y = (x - mean) \times scale$$

The image of the official yolov5 is rgb. Each value will be multiplied by 1/255, respectively corresponding to 0.0,0.0,0.0 and 0.0039216,0.0039216,0.0039216 when it is converted into mean and scale.

The model conversion command is as follows:

```
$ 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
```

The main parameters of model_transform.py are described as follows (for a complete introduction, please refer to the user interface chapter of the TPU-MLIR Technical Reference Manual):

Table 3.2: 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)
input_shapes	N	Shape of the inputs, 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
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
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.

3.4 MLIR to F16 bmodel

To convert the mlir file to the f16 bmodel, we need to run:

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize F16 \
    --chip bm1684x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.99,0.99 \
    --model yolov5s_1684x_f16.bmodel
```

The main parameters of model_deploy.py are as follows (for a complete introduction, please refer to the user interface chapter of the TPU-MLIR Technical Reference Manual):

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 calibration table path. Required when it is INT8 quantization
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
compare_all	N	Compare all tensors, if set.
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)
num_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
skip_validation	N	Skip bmodel correctness verification to boost deployment efficiency; bmodel verification is on by default.

After compilation, a file named yolov5s_1684x_f16.bmodel is generated.

3.5 MLIR to INT8 bmodel

3.5.1 Calibration table generation

Before converting to the INT8 model, you need to run calibration to get the calibration table. The number of input data is about 100 to 1000 according to the situation.

Then use the calibration table to generate a symmetric or asymmetric bmodel. It is generally not recommended to use the asymmetric one if the symmetric one already meets the requirements, because the performance of the asymmetric model will be slightly worse than the symmetric model.

Here is an example of the existing 100 images from COCO2017 to perform calibration:

```
$ run_calibration.py yolov5s.mlir \
    --dataset ../COCO2017 \
    --input_num 100 \
    -o yolov5s_cali_table
```

After running the command above, a file named yolov5s_cali_table will be generated, which is used as the input file for subsequent compilation of the INT8 model.

3.5.2 Compile to INT8 symmetric quantized model

Execute the following command to convert to the INT8 symmetric quantized model:

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --chip bm1684x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --model yolov5s_1684x_int8_sym.bmodel
```

After compilation, a file named yolov5s_1684x_int8_sym.bmodel is generated.

3.6 Effect comparison

There is a yolov5 use case written in python in this release package for object detection on images. The source code path is $TPUC_ROOT/python/samples/detect_yolov5.py. It can be learned how the model is used by reading the code. Firstly, preprocess to get the model's input, then do inference to get the output, and finally do post-processing. Use the following codes to validate the inference results of onnx/f16/int8 respectively.

The onnx model is run as follows to get dog_onnx.jpg:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model ../yolov5s.onnx \
    --output dog_onnx.jpg
```

The f16 bmodel is run as follows to get `dog_f16.jpg` :

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_1684x_f16.bmodel \
    --output dog_f16.jpg
```

The int8 symmetric bmodel is run as follows to get `dog_int8_sym.jpg`:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_1684x_int8_sym.bmodel \
    --output dog_int8_sym.jpg
```

The result images are compared as shown in the figure (Comparison of TPU-MLIR for YOLOv5s' compilation effect).

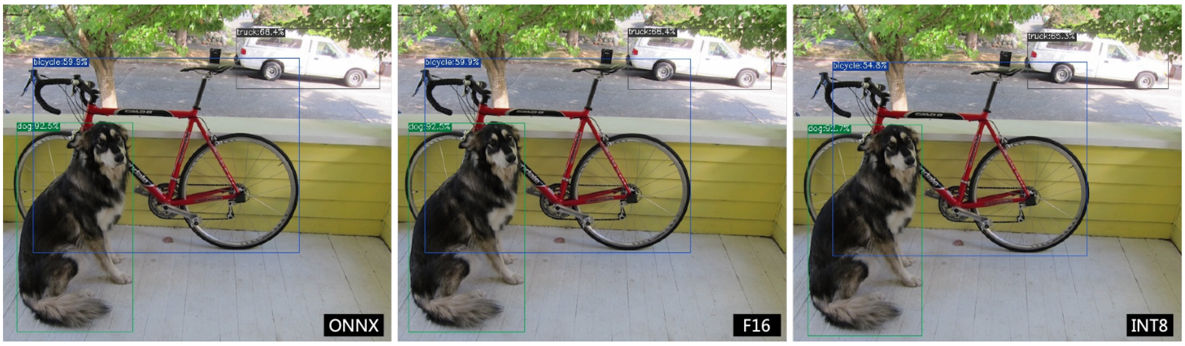


Fig. 3.1: Comparison of TPU-MLIR for YOLOv5s' compilation effect

Due to different operating environments, the final performance will be somewhat different from Fig. 3.1.

3.7 Model performance test

The following operations need to be performed outside of Docker,

3.7.1 Install the libsophon

Please refer to the **libsophon** manual to install **libsophon**.

3.7.2 Check the performance of BModel

After installing **libsophon**, you can use **bmrt_test** to test the accuracy and performance of the **bmodel**. You can choose a suitable model by estimating the maximum fps of the model based on the output of **bmrt_test**.

```
# Test the bmodel compiled above
# --bmodel parameter followed by bmodel file,

$ cd $TPUC_ROOT/../model_yolov5s/workspace
$ bmrt_test --bmodel yolov5s_1684x_f16.bmodel
$ bmrt_test --bmodel yolov5s_1684x_int8_sym.bmodel
```

Take the output of the last command as an example (the log is partially truncated here):

```
[BMRT][load_bmodel:983] INFO:pre net num: 0, load net num: 1
[BMRT][show_net_info:1358] INFO: ########################
[BMRT][show_net_info:1359] INFO: NetName: yolov5s, Index=0
[BMRT][show_net_info:1361] INFO: ---- stage 0 ----
[BMRT][show_net_info:1369] INFO:   Input 0) 'images' shape=[ 1 3 640 640 ] dtype=FLOAT32
[BMRT][show_net_info:1378] INFO:   Output 0) '350_Transpose_f32' shape=[ 1 3 80 80 85 ] ...
[BMRT][show_net_info:1378] INFO:   Output 1) '498_Transpose_f32' shape=[ 1 3 40 40 85 ] ...
[BMRT][show_net_info:1378] INFO:   Output 2) '646_Transpose_f32' shape=[ 1 3 20 20 85 ] ...
[BMRT][show_net_info:1381] INFO: ########################
[BMRT][bmrt_test:770] INFO:==> running network #0, name: yolov5s, loop: 0
[BMRT][bmrt_test:834] INFO:reading input #0, bytesize=4915200
[BMRT][print_array:702] INFO:  --> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
[BMRT][bmrt_test:982] INFO:reading output #0, bytesize=6528000
[BMRT][print_array:702] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
[BMRT][bmrt_test:982] INFO:reading output #1, bytesize=1632000
[BMRT][print_array:702] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
[BMRT][bmrt_test:982] INFO:reading output #2, bytesize=408000
[BMRT][print_array:702] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
[BMRT][bmrt_test:1014] INFO:net[yolov5s] stage[0], launch total time is 4122 us (npu 4009 us, cpu 113 us)
[BMRT][bmrt_test:1017] INFO:+++ The network[yolov5s] stage[0] output_data +++
[BMRT][print_array:702] INFO:output data #0 shape: [1 3 80 80 85 ] < 0.301003    ...
[BMRT][print_array:702] INFO:output data #1 shape: [1 3 40 40 85 ] < 0 0.228689  ...
[BMRT][print_array:702] INFO:output data #2 shape: [1 3 20 20 85 ] < 1.00135     ...
[BMRT][bmrt_test:1058] INFO:load input time(s): 0.008914
```

(continues on next page)

(continued from previous page)

```
25 [BMRT][bmrt_test:1059] INFO:calculate  time(s): 0.004132
26 [BMRT][bmrt_test:1060] INFO:get output time(s): 0.012603
27 [BMRT][bmrt_test:1061] INFO:compare    time(s): 0.006514
```

The following information can be learned from the output above:

1. Lines 05-08: the input and output information of bmodel
2. Line 19: running time on the TPU, of which the TPU takes 4009us and the CPU takes 113us. The CPU time here mainly refers to the waiting time of calling at HOST
3. Line 24: the time to load data into the NPU’s DDR
4. Line 25: the total time of Line 19
5. Line 26: the output data retrieval time

CHAPTER 4

Compile the Torch Model

This chapter takes yolov5s.pt as an example to introduce how to compile and transfer an pytorch model to run on the BM1684X TPU platform.

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

4.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 4.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

4.2 Prepare working directory

Create a model_yolov5s_pt directory, note that it is the same level directory as tpu-mlir; and put both model files and image files into the model_yolov5s_pt directory.

The operation is as follows:

```
$ mkdir model_yolov5s_pt && cd model_yolov5s_pt
$ wget https://github.com/sophgo/model-zoo/raw/main/vision/detection/yolov5/yolov5s-5.0.pt
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

4.3 TORCH to MLIR

The model in this example has a RGB input with mean and scale of 0.0,0.0,0.0 and 0.0039216, 0.0039216,0.0039216 respectively.

The model conversion command:

```
$ model_transform.py \
    --model_name yolov5s_pt \
    --model_def ../yolov5s-5.0.pt \
    --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 \
    --test_input ../image/dog.jpg \
    --test_result yolov5s_pt_top_outputs.npz \
    --mlir yolov5s_pt.mlir
```

After converting to mlir file, a ${model_name}_in_f32.npz file will be generated, which is the input file of the model.

4.4 MLIR to F16 bmodel

Convert the mlir file to the bmodel of f16, the operation method is as follows:

```
$ model_deploy.py \
    --mlir yolov5s_pt.mlir \
    --quantize F16 \
    --chip bm1684x \
    --test_input yolov5s_pt_in_f32.npz \
    --test_reference yolov5s_pt_top_outputs.npz \
    --tolerance 0.99,0.99 \
    --model yolov5s_pt_1684x_f16.bmodel
```

After comiplation, a file named yolov5s_pt_1684x_f16.bmodel will be generated.

4.5 MLIR to INT8 bmodel

4.5.1 Calibration table generation

Before converting to the INT8 model, you need to run calibration to get the calibration table. Here is an example of the existing 100 images from COCO2017 to perform calibration:

```
$ run_calibration.py yolov5s_pt.mlir \
    --dataset ../COCO2017 \
```

(continues on next page)

(continued from previous page)

```
    --input_num 100 \
    -o yolov5s_pt_cali_table
```

After running the command above, a file named **yolov5s_pt_cali_table** will be generated, which is used as the input file for subsequent compilation of the INT8 model.

4.5.2 Compile to INT8 symmetric quantized model

Execute the following command to convert to the INT8 symmetric quantized model:

```
$ model_deploy.py \
    --mlir yolov5s_pt.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_pt_cali_table \
    --chip bm1684x \
    --test_input yolov5s_pt_in_f32.npz \
    --test_reference yolov5s_pt_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --model yolov5s_pt_1684x_int8_sym.bmodel
```

After compilation, a file named **yolov5s_pt_1684x_int8_sym.bmodel** will be generated.

4.6 Effect comparison

Use the source code under the **$TPUC_ROOT/python/samples/detect_yolov5.py** path to perform object detection on the image. Use the following codes to verify the execution results of pytorch/ f16/ int8 respectively.

The pytorch model is run as follows to get **dog_torch.jpg**:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model ../yolov5s.pt \
    --output dog_torch.jpg
```

The f16 bmodel is run as follows to get **dog_f16.jpg** :

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_pt_1684x_f16.bmodel \
    --output dog_f16.jpg
```

The int8 asymmetric bmodel is run as follows to get **dog_int8_sym.jpg** :

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
```

(continues on next page)

(continued from previous page)

```
--model yolov5s_pt_1684x_int8_sym.bmodel \
--output dog_int8_sym.jpg
```

The result images are compared as shown in the figure (Comparison of TPU-MLIR for YOLOv5s compilation effect).

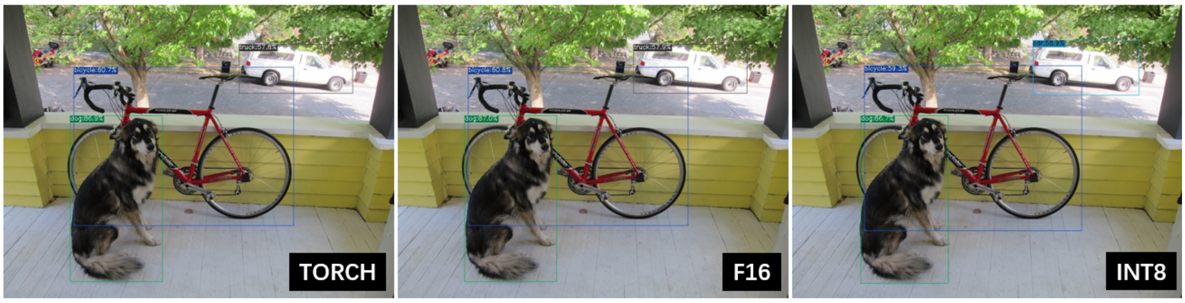


Fig. 4.1: Comparison of TPU-MLIR for YOLOv5s compilation effect

Due to different operating environments, the final performance will be somewhat different from Fig. 4.1.

CHAPTER 5

Compile the Caffe model

This chapter takes mobilenet_v2_deploy.prototxt and mobilenet_v2.caffemodel as examples to introduce how to compile and transfer a caffe model to run on the BM1684X TPU platform.

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

5.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 5.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

5.2 Prepare working directory

Create a **mobilenet_v2** directory, note that it is the same level as tpu-mlir, and put both model files and image files into the **mobilenet_v2** directory.

The operation is as follows:

```
$ mkdir mobilenet_v2 && cd mobilenet_v2
$ wget https://raw.githubusercontent.com/shicai/MobileNet-Caffe/master/mobilenet_v2_deploy.
↪prototxt
$ wget https://github.com/shicai/MobileNet-Caffe/raw/master/mobilenet_v2.caffemodel
$ cp -rf $TPUC_ROOT/regression/dataset/ILSVRC2012 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

5.3 Caffe to MLIR

The model in this example has a BGR input with mean and scale of 103.94, 116.78, 123.68 and 0.017, 0.017, 0.017 respectively.

The model conversion command:

```
$ model_transform.py \
    --model_name mobilenet_v2 \
    --model_def ../mobilenet_v2_deploy.prototxt \
    --model_data ../mobilenet_v2.caffemodel \
    --input_shapes [[1,3,224,224]] \
    --resize_dims=256,256 \
    --mean 103.94,116.78,123.68 \
    --scale 0.017,0.017,0.017 \
    --pixel_format bgr \
    --test_input ../image/cat.jpg \
    --test_result mobilenet_v2_top_outputs.npz \
    --mlir mobilenet_v2.mlir
```

After converting to mlir file, a `${model_name}_in_f32.npz` file will be generated, which is the input file of the model.

5.4 MLIR to F32 bmodel

Convert the mlir file to the bmodel of f32, the operation method is as follows:

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize F32 \
    --chip bm1684x \
    --test_input mobilenet_v2_in_f32.npz \
    --test_reference mobilenet_v2_top_outputs.npz \
    --tolerance 0.99,0.99 \
    --model mobilenet_v2_1684x_f32.bmodel
```

After compilation, a file named `mobilenet_v2_1684x_f32.bmodel` is generated.

5.5 MLIR to INT8 bmodel

5.5.1 Calibration table generation

Before converting to the INT8 model, you need to run calibration to get the calibration table. The number of input data is about 100 to 1000 according to the situation.

Then use the calibration table to generate a symmetric or asymmetric bmodel. It is generally not recommended to use the asymmetric one if the symmetric one already meets the requirements, because the performance of the asymmetric model will be slightly worse than the symmetric model.

Here is an example of the existing 100 images from ILSVRC2012 to perform calibration:

```
$ run_calibration.py mobilenet_v2.mlir \
    --dataset ../ILSVRC2012 \
    --input_num 100 \
    -o mobilenet_v2_cali_table
```

After running the command above, a file named mobilenet_v2_cali_table will be generated, which is used as the input file for subsequent compilation of the INT8 model.

5.5.2 Compile to INT8 symmetric quantized model

Execute the following command to convert to the INT8 symmetric quantized 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 0.96,0.70 \
    --model mobilenet_v2_1684x_int8_sym.bmodel
```

After compilation, a file named mobilenet_v2_1684x_int8_sym.bmodel is generated.

CHAPTER 6

Compile the TFLite model

This chapter takes the lite-model_mobilebert_int8_1.tflite model as an example to introduce how to compile and transfer a TFLite model to run on the BM1684X TPU platform.

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

6.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 6.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

6.2 Prepare working directory

Create a mobilebert_tf directory, note that it is the same level as tpu-mlir, and put the test image file into the mobilebert_tf directory.

The operation is as follows:

```
$ mkdir mobilebert_tf && cd mobilebert_tf
$ wget -O lite-model_mobilebert_int8_1.tflite https://storage.googleapis.com/tfhub-lite-models/
↪iree/lite-model/mobilebert/int8/1.tflite
$ cp ${REGRESSION_PATH}/npz_input/squad_data.npz .
$ mkdir workspace && cd workspace
```

REGRESSION_PATH is an environment variable, corresponding to the tpu-mlir_xxxx/regression directory.

6.3 TFLite to MLIR

The model conversion command:

```
$ model_transform.py \
    --model_name mobilebert_tf \
    --mlir mobilebert_tf.mlir \
    --model_def ../lite-model_mobilebert_int8_1.tflite \
    --test_input ../squad_data.npz \
    --test_result mobilebert_tf_top_outputs.npz \
    --input_shapes [[1,384],[1,384],[1,384]] \
    --channel_format none
```

After converting to mlir file, a mobilebert_tf_in_f32.npz file will be generated, which is the input file of the model.

6.4 MLIR to bmodel

This model is a tflite asymmetric quantized model, which can be converted into a bmodel according to the following parameters:

```
$ model_deploy.py \
    --mlir mobilebert_tf.mlir \
    --quantize INT8 \
    --chip bm1684x \
    --test_input mobilebert_tf_in_f32.npz \
    --test_reference mobilebert_tf_top_outputs.npz \
    --model mobilebert_tf_bm1684x_int8.bmodel
```

Once compiled, a file named mobilebert_tf_bm1684x_int8.bmodel is generated.

CHAPTER 7

Quantization and optimization

In deploying neuron network, the accuracy and throughput (inference speed) are critical targets. To achieve high accuracy and high speed, for some networks, mix precision inference is essential. In this chapter, with yolo as examples, method of setting mix precision inference is demonstrated, and two useful tools 'sensitive layer search' and 'local non-quantization' are illustrated.

7.1 Mix Precision

This chapter takes `yolov3 tiny` as examples to introduce how to use mix precision。This model is from <https://github.com/onnx/models/tree/main/vision/object_detection_segmentation/tiny-yolov3>。

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

7.1.1 Load tpu-mlir

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

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

`envsetup.sh` adds the following environment variables:

Table 7.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

7.1.2 Prepare working directory

Create a yolov3_tiny directory, note that it is the same level as tpu-mlir, and put both model files and image files into the yolov3_tiny directory.

The operation is as follows:

```
$ mkdir yolov3_tiny && cd yolov3_tiny
$ wget https://github.com/onnx/models/raw/main/vision/object_detection_segmentation/tiny-
↪yolov3/model/tiny-yolov3-11.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

7.1.3 Sample for onnx

`detect_yolov3.py` is a python program, to run `yolov3_tiny` model.

The operation is as follows:

```
$ detect_yolov3.py \
    --model ../tiny-yolov3-11.onnx \
    --input ../COCO2017/000000366711.jpg \
    --output yolov3_onnx.jpg
```

The print result as follows:

```
person:60.7%
orange:77.5%
```

And get result image `yolov3_onnx.jpg`, as below(yolov3_tiny ONNX):

Fig. 7.1: yolov3_tiny ONNX

7.1.4 To INT8 symmetric model

Step 1: To F32 mlir

```
$ model_transform.py \
    --model_name yolov3_tiny \
    --model_def ../tiny-yolov3-11.onnx \
    --input_shapes [[1,3,416,416]] \
    --scale 0.0039216,0.0039216,0.0039216 \
    --pixel_format rgb \
    --keep_aspect_ratio \
    --pad_value 128 \
    --output_names=convolution_output1,convolution_output \
    --mlir yolov3_tiny.mlir
```

Step 2: Gen calibartion table

```
$ run_calibration.py yolov3_tiny.mlir \
    --dataset ../COCO2017 \
    --input_num 100 \
    -o yolov3_cali_table
```

Step 3: To model

```
$ model_deploy.py \
    --mlir yolov3_tiny.mlir \
    --quantize INT8 \
    --calibration_table yolov3_cali_table \
    --chip bm1684x \
    --model yolov3_int8.bmodel
```

Step 4: Run model

```
$ detect_yolov3.py \
     --model yolov3_int8.bmodel \
     --input ../COCO2017/000000366711.jpg \
     --output yolov3_int8.jpg
```

The print result as follows, indicates that one target is detected:

```
orange:72.9.0%
```

And get image **yolov3_int8.jpg**, as below(yolov3_tiny int8 symmetric):

It can be seen that the int8 symmetric quantization model performs poorly compared to the original model on this image and only detects one target.

Fig. 7.2: yolov3_tiny int8 symmetric

7.1.5 To Mix Precision Model

After int8 conversion, do these commands as beflow.

Step 1: Gen quantization table

Use run_qtable.py to gen qtable, parameters as below:

Table 7.2: run_qtable.py 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”)
calibration_table	Y	Name of calibration table file
chip	Y	The platform that the model will use. Support bm1686/bm1684x/bm1684/cv186x/cv183x/cv182x/cv181x/cv180x.
fp_type	N	Specifies the type of float used for mixing precision. Support auto,F16,F32,BF16. Default is auto, indicating that it is automatically selected by program
input_num	N	The number of sample, default 10
expected_cos	N	Specify the minimum cos value for the expected final output layer of the network. The default is 0.99. The smaller the value, the more layers may be set to floating-point
min_layer_cos	N	Specify the minimum cos expected per layer, below which an attempt is made to set the fp32 calculation. The default is 0.99
debug_cmd	N	Specifies a debug command string for development. It is empty by default
o	Y	output quantization table
global_compare_layers	N	global compare layers, for example:'layer1,layer2' or 'layer1:0.3,layer2:0.7'
fp_type	N	float type of mix precision
loss_table	N	output all loss of layers if each layer is quantized to f16

The operation is as follows:

```
$ run_qtable.py yolov3_tiny.mlir \
    --dataset ../COCO2017 \
    --calibration_table yolov3_cali_table \
    --min_layer_cos 0.999 \ #If the default 0.99 is used here, the program detects that the
↪original int8 model already meets the cos of 0.99 and simply stops searching
```

(continues on next page)

(continued from previous page)

```
    --expected_cos 0.9999 \
    --chip bm1684x \
    -o yolov3_qtable
```

The final output after execution is printed as follows:

```
int8 outputs_cos:0.999115 old
mix model outputs_cos:0.999517
Output mix quantization table to yolov3_qtable
total time:44 second
```

Above, int8 outputs_cos represents the cos similarity between original network output of int8 model and fp32; mix model outputs_cos represents the cos similarity of network output after mixing precision is used in some layers; total time represents the search time of 44 seconds. In addition, get quantization table yolov3_qtable, context as below:

```
# op_name   quantize_mode
model_1/leaky_re_lu_2/LeakyRelu:0_pooling0_MaxPool F16
convolution_output10_Conv F16
model_1/leaky_re_lu_3/LeakyRelu:0_LeakyRelu F16
model_1/leaky_re_lu_3/LeakyRelu:0_pooling0_MaxPool F16
model_1/leaky_re_lu_4/LeakyRelu:0_LeakyRelu F16
model_1/leaky_re_lu_4/LeakyRelu:0_pooling0_MaxPool F16
model_1/leaky_re_lu_5/LeakyRelu:0_LeakyRelu F16
model_1/leaky_re_lu_5/LeakyRelu:0_pooling0_MaxPool F16
model_1/concatenate_1/concat:0_Concat F16
```

In the table, first col is layer name, second is quantization type. Also full_loss_table.txt is generated, context as blow:

```
# chip: bm1684x  mix_mode: F16
###
No.0   : Layer: model_1/leaky_re_lu_3/LeakyRelu:0_LeakyRelu                Cos: 0.994063
No.1   : Layer: model_1/leaky_re_lu_2/LeakyRelu:0_LeakyRelu                Cos: 0.997447
No.2   : Layer: model_1/leaky_re_lu_5/LeakyRelu:0_LeakyRelu                Cos: 0.997450
No.3   : Layer: model_1/leaky_re_lu_4/LeakyRelu:0_LeakyRelu                Cos: 0.997982
No.4   : Layer: model_1/leaky_re_lu_2/LeakyRelu:0_pooling0_MaxPool         Cos: 0.998163
No.5   : Layer: convolution_output11_Conv                                  Cos: 0.998300
No.6   : Layer: convolution_output9_Conv                                   Cos: 0.999302
No.7   : Layer: model_1/leaky_re_lu_1/LeakyRelu:0_LeakyRelu                Cos: 0.999371
No.8   : Layer: convolution_output8_Conv                                   Cos: 0.999424
No.9   : Layer: model_1/leaky_re_lu_1/LeakyRelu:0_pooling0_MaxPool         Cos: 0.999574
No.10  : Layer: convolution_output12_Conv                                  Cos: 0.999784
```

This table is arranged smoothly according to the cos from small to large, indicating the cos calculated by this Layer after the precursor layer of this layer has been changed to the corresponding floating-point mode. If the cos is still smaller than the previous parameter min_layer_cos, this layer and its immediate successor layer will be set to floating-point calculation。 run_qtable.py calculates the output cos of the whole network every time the neighboring two layers are set to floating point. If the cos is larger than the specified expected_cos, the search is withdrawn. Therefore, if you set a larger expected_cos value, you will try to set more layers to floating point。

Step 2: Gen mix precision model

```
$ model_deploy.py \
   --mlir yolov3_tiny.mlir \
   --quantize INT8 \
   --quantize_table yolov3_qtable \
   --calibration_table yolov3_cali_table \
   --chip bm1684x \
   --model yolov3_mix.bmodel
```

Step 3: run mix precision model

```
$ detect_yolov3.py \
    --model yolov3_mix.bmodel \
    --input ../COCO2017/000000366711.jpg \
    --output yolov3_mix.jpg
```

The print result as follows:

```
person:63.9%
orange:72.9%
```

And get image **yolov3_mix.jpg** , as below(yolov3_tiny mix):

It can be seen that targets that cannot be detected in int8 model can be detected again with the use of mixing precision.

7.2 Sensitive Layer Search

This chapter takes **mobilenet-v2** as example to introduce how to use sensitive layer search. This model is from <nnmodels/pytorch_models/accuracy_test/classification/mobilenet_v2.pt>.

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

Fig. 7.3: yolov3_tiny mix

7.2.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 7.3: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

7.2.2 Prepare working directory

Create a mobilenet-v2 directory, note that it is the same level as tpu-mlir, and put both model files and image files into the mobilenet-v2 directory.

The operation is as follows:

```
$ mkdir mobilenet-v2 && cd mobilenet-v2
$ cp -rf $TPUC_ROOT/regression/dataset/ILSVRC2012 .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory. Note that mobilenet-v2.pt needs to be downloaded from nnmodels and then be placed in the mobilenet-v2 directory.

7.2.3 Accuracy test of float anf int8 models

Step 1: To F32 mlir

```
$ model_transform.py \
    --model_name mobilenet_v2 \
    --model_def ../mobilenet_v2.pt \
    --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 \
    --mlir mobilenet_v2.mlir
```

Step 2: Gen calibartion table

```
$ run_calibration.py mobilenet_v2.mlir \
    --dataset ../ILSVRC2012 \
    --input_num 100 \
    -o mobilenet_v2_cali_table
```

Step 3: To F32 bmodel

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize F32 \
    --chip bm1684 \
    --model mobilenet_v2_1684_f32.bmodel
```

Step 4: To INT8 model

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize INT8 \
    --chip bm1684 \
    --calibration_table mobilenet_v2_cali_table \
    --model mobilenet_v2_bm1684_int8_sym.bmodel
```

Step 5: Accuracy test

classify_mobilenet_v2.py is a python program, to run mobilenet-v2 model.

Test the fp32 model:

```
$ classify_mobilenet_v2.py \
    --model_def mobilenet_v2_bm1684_f32.bmodel \
    --input ../ILSVRC2012/n01440764_9572.JPEG \
    --output mobilenet_v2_fp32_bmodel.JPEG \
    --category_file ../ILSVRC2012/synset_words.txt
```

The classification information is displayed on the output image. The right label tench, Tinca tinca ranks first.

```
Top-5
n01440764 tench, Tinca tinca
n02536864 coho, cohoe, coho salmon, blue jack, silver salmon, Oncorhynchus kisutch
n02422106 hartebeest
n02749479 assault rifle, assault gun
n02916936 bulletproof vest
```

Test the INT8 model:

```
$ classify_mobilenet_v2.py \
    --model_def mobilenet_v2_bm1684_int8_sym.bmodel \
    --input ../ILSVRC2012/n01440764_9572.JPEG \
    --output mobilenet_v2_INT8_sym_bmodel.JPEG \
    --category_file ../ILSVRC2012/synset_words.txt
```

The right label tench, Tinca tinca ranks second.

```
Top-5
n02408429 water buffalo, water ox, Asiatic buffalo, Bubalus bubalis
n01440764 tench, Tinca tinca
n01871265 tusker
n02396427 wild boar, boar, Sus scrofa
n02074367 dugong, Dugong dugon
```


7.2.4 To Mix Precision Model

After int8 conversion, do these commands as beflow.

Step 1: Search sensitive layers

Use run_sensitive_layer.py and bad cases to search sensitive layers, parameters as below:

Table 7.4: run_sensitive_layer.py 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")
calibration_table	Y	Name of calibration table file
chip	Y	The platform that the model will use. Support bm1686/bm1684x/bm1684/cv186x/cv183x/cv182x/cv181x/cv180x.
fp_type	N	Specifies the type of float used for mixing precision. Support auto,F16,F32,BF16. Default is auto, indicating that it is automatically selected by program
input_num	N	The number of samples used for calibration, default 10
inference_num	N	The number of samples used for inference, default 10
max_float_layers	N	The number of layers set to float, default 5
tune_list	N	The sample list for tune threshold
tune_num	N	The number of samples for tune threshold, default 5
histogram_bin_num	N	The number of bins used in kld calibration, default 2048
post_process	N	The user defined prost process program path, default None
expected_cos	N	Specify the minimum cos value for the expected final output layer of the network. The default is 0.99. The smaller the value, the more layers may be set to floating-point
debug_cmd	N	Specifies a debug command string for development. It is empty by default
o	Y	output quantization table
global_compare_layers	N	global compare layers, for example:'layer1,layer2' or 'layer1:0.3,layer2:0.7'
fp_type	N	float type of mix precision

In this example, 100 images are used for calibration and 30 images are used for inference, and the command is as follows (for the chip of CV18xx series, set the chip to the corresponding chip name) :

The operation is as follows:

```
$ run_sensitive_layer.py mobilenet_v2.mlir \
    --dataset ../ILSVRC2012 \
    --input_num 100 \
    --inference_num 30 \
    --calibration_table mobilenet_v2_cali_table \
    --chip bm1684 \
    --post_process post_process_func.py \
    -o mobilenet_v2_qtable
```

Sensitive layer program supports user defined post process programs post_process_func.py. The post process function must be named PostProcess.

```
$ def PostProcess(data):
    print("in post process")
    return data
```

The final output after execution is printed as follows:

```
the layer input3.1 is 0 sensitive layer, loss is 0.008808857469573828, type is top.Conv
the layer input11.1 is 1 sensitive layer, loss is 0.0016958347875666302, type is top.Conv
the layer input128.1 is 2 sensitive layer, loss is 0.0015641432811860367, type is top.Conv
the layer input130.1 is 3 sensitive layer, loss is 0.0014325751094084183, type is top.Scale
the layer input127.1 is 4 sensitive layer, loss is 0.0011817314259702227, type is top.Add
the layer input13.1 is 5 sensitive layer, loss is 0.001018420214596527, type is top.Scale
the layer 787 is 6 sensitive layer, loss is 0.0008603856180608993, type is top.Scale
the layer input2.1 is 7 sensitive layer, loss is 0.0007558935451825732, type is top.Scale
the layer input119.1 is 8 sensitive layer, loss is 0.000727441637624282, type is top.Add
the layer input0.1 is 9 sensitive layer, loss is 0.0007138056757098887, type is top.Conv
the layer input110.1 is 10 sensitive layer, loss is 0.000662179506136229, type is top.Conv
......
run result:
int8 outputs_cos:0.978847 old
mix model outputs_cos:0.989741
Output mix quantization table to mobilenet_v2_qtable
total time:402.15848112106323
success sensitive layer search
```

Above, int8 outputs_cos represents the cosine similarity between network outputs of int8 model and float model; mix model outputs_cos represents the cosine similarity between network outputs of mix model and float model; total time represents the search time is 402 seconds. In addition, this program generates a quantization table mobilenet_v2_qtable, the context is as below:

```
# op_name   quantize_mode
input3.1 F32
input11.1 F32
input128.1 F32
input130.1 F32
input127.1 F32
```

The first column in the table is layer name, and the second one is quantization type. Also a log file named **SensitiveLayerSearch** is generated, its context is as blow:

```
INFO:root:start to handle layer: input3.1, type: top.Conv
INFO:root:adjust layer input3.1 th, with method MAX, and threshlod 5.5119305
INFO:root:run int8 mode: mobilenet_v2.mlir
INFO:root:outputs_cos_los = 0.014830573787862011
INFO:root:adjust layer input3.1 th, with method Percentile9999, and threshlod 4.1202815
INFO:root:run int8 mode: mobilenet_v2.mlir
INFO:root:outputs_cos_los = 0.011843443367980822
INFO:root:adjust layer input3.1 th, with method KL, and threshlod 2.6186381997094728
INFO:root:run int8 mode: mobilenet_v2.mlir
INFO:root:outputs_cos_los = 0.008808857469573828
INFO:root:layer input3.1, layer type is top.Conv, best_th = 2.6186381997094728, best_method = KL, best_cos_loss = 0.008808857469573828
```

This log file records the cosine losses between the outputs of mix model and float model when setting each op to int8 with different quantize methods(MAX/Percentile9999/KL). It also contains the loss information printed in the screen and the cosine similarity of mix model and float model. The qtable generated by this program can be modified according to the loss information. The best thresholds of each op are recorded in a new cali table named new_cali_table. This table is restored in current workspace and need to be used when generating mix model. In this example, the loss of input3.1 is larger than other ops, thus you can only set input3.1 as float in qtable.

Step 2: Gen mix precision model

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize INT8 \
    --chip bm1684 \
    --calibration_table new_cali_table \
    --quantize_table mobilenet_v2_qtable \
    --model mobilenet_v2_bm1684_int8_mix.bmodel
```

Step 3: Test accuracy of mix model

```
$ classify_mobilenet_v2.py \
    --model_def mobilenet_v2_bm1684_mix.bmodel \
    --input ../ILSVRC2012/n01440764_9572.JPEG \
    --output mobilenet_v2_INT8_sym_bmodel.JPEG \
    --category_file ../ILSVRC2012/synset_words.txt
```

The classification results are as follows. The right label **tench, Tinca tinca** ranks first again.

```
Top-5
n01440764 tench, Tinca tinca
```

(continues on next page)

(continued from previous page)

```
n02749479 assault rifle, assault gun
n02916936 bulletproof vest
n02536864 coho, cohoe, coho salmon, blue jack, silver salmon, Oncorhynchus kisutch
n04090263 rifle
```

7.3 Local Non-Quantization

For specific neural networks, some layers may not be suitable for quantization due to significant differences in data distribution. The "Local Non-Quantization" allows you to add certain layers before, after, or between other layers to a mixed-precision table. These layers will not be quantized when generating a mixed-precision model.

In this chapter, we will continue using the example of the YOLOv5s network mentioned in Chapter 3 and demonstrate how to use the Local Non-Quantization to quickly generate a mix-precision model.

The process of generating FP32 and INT8 models is the same as in Chapter 3. Here, we focus on generating mix-precision model and the accuracy testing.

For YOLO series models, the last three convolutional layers often have significantly different data distributions, and adding them manually to the mixed-precision table can improve accuracy. With the Local Non-Quantization feature, you can search for the corresponding layers from the FP32 MLIR file and quickly add them to the mixed-precision table using the following command:

```
$ fp_forward.py \
    yolov5s.mlir \
    --quantize INT8 \
    --chip bm1684x \
    --fpfwd_outputs 474_Conv,326_Conv,622_Conv\
    --chip bm1684x \
    -o yolov5s_qtable
```

Opening the file "yolov5s_qtable" will reveal that the relevant layers have been added to the qtable.

Generating the Mixed-Precision Model

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --quantize_table yolov5s_qtable \
    --chip bm1684x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --model yolov5s_1684x_mix.bmodel
```

Validating the Accuracy of FP32 and Mixed-Precision Models In the model-zoo, there is a program called "yolo" used for accuracy validation of object detection models. You can use the "harness" field in the mlir.config.yaml file to invoke "yolo" as follows:

Modify the relevant fields as follows:

```
$ dataset:
   imagedir: $(coco2017_val_set)
   anno: $(coco2017_anno)/instances_val2017.json

harness:
   type: yolo
   args:
      - name: FP32
      bmodel: $(workdir)/$(name)_bm1684_f32.bmodel
      - name: INT8
      bmodel: $(workdir)/$(name)_bm1684_int8_sym.bmodel
      - name: mix
      bmodel: $(workdir)/$(name)_bm1684_mix.bmodel
```

Switch to the top-level directory of model-zoo and use tpu_perf.precision_benchmark for accuracy testing, as shown in the following command: .. code-block:: shell

> $ python3 -m tpu_perf.precision_benchmark yolov5s_path –mlir –target BM1684X –devices 0

The accuracy test results will be stored in output/yolo.csv:

mAP for the FP32 model: mAP for the mixed-precision model using the default mixed-precision table:

Performance Testing

mAP for the mixed-precision model using the manually added mixed-precision table:

Parameter Description

Table 7.5: fp_forward.py parameters

Name	Required?	Explanation
(None)	Y	mlir file
chip	Y	The platform that the model will use. Support bm1686/bm1684x/bm1684/cv186x/cv183x/cv182x/cv181x/cv180x.
fpfwd_inputs	N	Specify layers (including this layer) to skip quantization before them. Multiple inputs are separated by commas.
fpfwd_outputs	N	Specify layers (including this layer) to skip quantization after them. Multiple inputs are separated by commas.
fpfwd_blocks	N	Specify the start and end layers between which quantization will be skipped. Start and end layers are separated by space, and multiple blocks are separated by spaces.
fp_type	N	Specifies the type of float used for mixing precision. Support auto,F16,F32,BF16. Default is auto, indicating that it is automatically selected by program
o	Y	output quantization table

CHAPTER 8

Use TPU for Preprocessing

At present, the two main series of chips supported by TPU-MLIR are BM168x and CV18xx. Both of them support common image preprocessing fusion. The developer can pass the preprocessing arguments during the compilation process, and the compiler will directly insert the corresponding preprocessing operators into the generated model. The generated bmodel or cvimodel can directly use the unpreprocessed image as input and use TPU to do the preprocessing.

Table 8.1: Supported Preprocessing Type

Preprocessing Type	BM168x	CV18xx
Crop	True	True
Normalization	True	True
NHWC to NCHW	True	True
BGR/ RGB Conversion	True	True

The image cropping will first adjust the image to the size specified by the "–resize_dims" argument of the model_transform tool, and then crop it to the size of the model input. The normalization supports directly converting unpreprocessed image data.

To integrate preprocessing into the model, you need to speficy the "–fuse_preprocess" argument when using the model_deploy tool, and the test_input should be an image of the original format (i.e., jpg, jpeg and png format). There will be a preprocessed npz file of input named `${model_name}_in_ori.npz` generated. In addition, there is a "–customization_format" argument to specify the original image format input to the model. The supported image formats are described as follows:

Table 8.2: Types of customization_format and Description

customization_format	Description	BM168x	CV18xx
None	same with model format, do nothing, as default	True	True
RGB_PLANAR	rgb color order and nchw tensor format	True	True
RGB_PACKED	rgb color order and nhwc tensor format	True	True
BGR_PLANAR	bgr color order and nchw tensor format	True	True
BGR_PACKED	bgr color order and nhwc tensor format	True	True
GRAYSCALE	one color channel only and nchw tensor format	True	True
YUV420_PLANAR	yuv420 planner format, from vpss input	False	True
YUV_NV21	NV21 format of yuv420, from vpss input	False	True
YUV_NV12	NV12 format of yuv420, from vpss input	False	True
RGBA_PLANAR	rgba format and nchw tensor format	False	True

The "YUV*" type format is the special input format of CV18xx series chips. When the order of the color channels in the customization_format is different from the model input, a channel conversion operation will be performed. If the customization_format argument is not specified, the corresponding customization_format will be automatically set according to the pixel_format and channel_format arguments defined when using the model_transform tool.

8.1 Model Deployment Example

Take the mobilenet_v2 model as an example, use the model_transform tool to generate the original mlir, and the run_calibration tool to generate the calibration table (refer to the chapter "Compiling the Caffe Model" for more details).

8.1.1 Deploy to BM168x

The command to generate the preprocess-fused symmetric INT8 quantized bmodel model is as follows:

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize INT8 \
    --calibration_table mobilenet_v2_cali_table \
    --chip bm1684x \
    --test_input ../image/cat.jpg \
    --test_reference mobilenet_v2_top_outputs.npz \
    --tolerance 0.96,0.70 \
    --fuse_preprocess \
    --model mobilenet_v2_bm1684x_int8_sym_fuse_preprocess.bmodel
```

8.1.2 Deploy to CV18xx

The command to generate the preprocess-fused symmetric INT8 quantized cvimodel model are as follows:

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize INT8 \
    --calibration_table mobilenet_v2_cali_table \
    --chip cv183x \
    --test_input ../image/cat.jpg \
    --test_reference mobilenet_v2_top_outputs.npz \
    --tolerance 0.96,0.70 \
    --fuse_preprocess \
    --customization_format RGB_PLANAR \
    --model mobilenet_v2_cv183x_int8_sym_fuse_preprocess.cvimodel
```

vpss input

When the input data comes from the video post-processing module VPSS provided by CV18xx (for details on how to use VPSS for preprocessing, please refer to "CV18xx Media Software Development Reference"), data alignment is required (e.g., 32-bit aligned width), fuse_preprocess and aligned_input need to be set at the same time. The command to generate the preprocessed-fused cvimodel model is as follows:

```
$ model_deploy.py \
    --mlir mobilenet_v2.mlir \
    --quantize INT8 \
    --calibration_table mobilenet_v2_cali_table \
    --chip cv183x \
    --test_input ../image/cat.jpg \
    --test_reference mobilenet_v2_top_outputs.npz \
    --tolerance 0.96,0.70 \
    --fuse_preprocess \
    --customization_format RGB_PLANAR \
    --aligned_input \
    --model mobilenet_v2_cv183x_int8_sym_fuse_preprocess_aligned.cvimodel
```

In the above command, aligned_input specifies the alignment that the model input needs to do.

Note that with vpss as input, runtime can use CVI_NN_SetTensorPhysicalAddr to reduce memory data copy.

CHAPTER 9

Use TPU for Postprocessing

Currently, TPU-MLIR supports integrating the post-processing of YOLO series and SSD network models into the model. The chips currently supporting this function include BM1684X, BM1686, and CV186X. This chapter will take the conversion of YOLOv5s to F16 model as an example to introduce how this function is used.

This chapter requires the following files (where xxxx corresponds to the actual version information):

tpu-mlir_xxxx.tar.gz (The release package of tpu-mlir)

9.1 Load tpu-mlir

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

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

envsetup.sh adds the following environment variables:

Table 9.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

9.2 Prepare working directory

Create a model_yolov5s directory, note that it is the same level directory as tpu-mlir; and put both model files and image files into the model_yolov5s directory.

The operation is as follows:

```
$ mkdir yolov5s_onnx && cd yolov5s_onnx
$ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

$TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

9.3 ONNX to MLIR

The model conversion command is as follows:

```
$ 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 326,474,622 \
    --add_postprocess yolov5 \
    --test_input ../image/dog.jpg \
    --test_result yolov5s_top_outputs.npz \
    --mlir yolov5s.mlir
```

There are two points to note here. The first is that the --add_postprocess argument needs to be included in the command. The second is that the specified --output_names should correspond to the final convolution operation.

The generated yolov5s.mlir file finally has a top.YoloDetection inserted at the end as follows:

```
%260 = "top.Weight"() : () -> tensor<255x512x1x1xf32> loc(#loc261)
%261 = "top.Weight"() : () -> tensor<255xf32> loc(#loc262)
%262 = "top.Conv"(%253, %260, %261) {dilations = [1, 1], do_relu = false, group = 1 : i64, kernel_shape = [1, 1], pads = [0, 0, 0, 0], relu_limit = -1.000000e+00 : f64, strides = [1, 1]} : (tensor<1x512x6x32xf32>, tensor<255x512x1x1xf32>, tensor<255xf32>) -> tensor<1x255x6x32xf32> loc(#loc263)
%263 = "top.YoloDetection"(%256, %259, %262) {anchors = [10, 13, 16, 30, 33, 23, 30, 61, 62, 45, 59, 119, 116, 90, 156, 198, 373, 326], class_num = 80 : i64, keep_topk = 200 : i64, net_input_h = 640 : i64, net_input_w = 640 : i64, nms_threshold = 5.000000e-01 : f64, num_boxes = 3 : i64, obj_threshold = 0.69999999999999996 : f64, version = "yolov5"} : (tensor<1x255x24x128xf32>, tensor<1x255x12x64xf32>, tensor<1x255x6x32xf32>) -> tensor<1x1x200x7xf32> loc(#loc264)
return %263 : tensor<1x1x200x7xf32> loc(#loc)
```

Here you can see that top.YoloDetection includes parameters such as anchors, num_boxes, and so on. If the post-processing is not standard YOLO, and needs to be changed to other parameters, these parameters in the MLIR file can be directly modified. Also, the output has been changed to one, with the shape of 1x1x200x7, where 200 represents the maximum number of detection boxes. When there are multiple batches, its value will change to batchx200. The 7 elements respectively represent [batch_number, class_id, score, center_x, center_y, width, height].

9.4 MLIR to Bmodel

To convert the MLIR file to an F16 bmodel, proceed as follows:

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize F16 \
    --chip bm1684x \
    --fuse_preprocess \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --model yolov5s_1684x_f16.bmodel
```

Here, the --fuse_preprocess parameter is added in order to integrate the preprocessing into the model as well. In this way, the converted model is a model that includes post-processing. The model information can be viewed with model_tool as follows:

```
$ model_tool --info yolov5s_1684x_f16.bmodel
```

```
bmodel version: B.2.2
chip: BM1684X
create time: Fri May 26 16:30:20 2023

kernel_module name: libbm1684x_kernel_module.so
kernel_module size: 2037536
==========================================
net 0: [yolov5s]  static
------------
stage 0:
subnet number: 2
input: images_raw, [1, 3, 640, 640], uint8, scale: 1, zero_point: 0
output: yolo_post, [1, 1, 200, 7], float32, scale: 1, zero_point: 0

device mem size: 24970588 (coeff: 14757888, instruct: 1372, runtime: 10211328)
host mem size: 0 (coeff: 0, runtime: 0)
```

Here, [1, 1, 200, 7] is the maximum shape, and the actual output varies depending on the number of detected boxes.

9.5 Bmodel Verification

In this release package, there is a YOLOv5 use case written in Python, with the source code located at $TPUC_ROOT/python/samples/detect_yolov5.py. It is used for object detection in images. By reading this code, you can understand how the final output result is transformed into bounding boxes.

The command execution is as follows:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_1684x_f16.bmodel \
    --net_input_dims 640,640 \
    --fuse_preprocess \
    --fuse_postprocess \
    --output dog_out.jpg
```

CHAPTER 10

Appendix.01: Reference for converting model to ONNX format

This chapter provides a reference for how to convert PyTorch, TensorFlow and PaddlePaddle models to ONNX format. You can also refer to the model conversion tutorial provided by ONNX official repository: https://github.com/onnx/tutorials. All the operations in this chapter are carried out in the Docker container. For the specific environment configuration method, please refer to the content of Chapter 2.

10.1 PyTorch model to ONNX

This section takes a self-built simple PyTorch model as an example to perform onnx conversion.

10.1.1 Step 0: Create a working directory

Create and enter the torch_model directory using the command line.

```
$ mkdir torch_model
$ cd torch_model
```

10.1.2 Step 1: Build and save the model

Create a script named simple_net.py in this directory and run it. The specific content of the script is as follows:

```
#!/usr/bin/env python3
import torch

# Build a simple nn model
class SimpleModel(torch.nn.Module):

    def __init__(self):
        super(SimpleModel, self).__init__()
        self.m1 = torch.nn.Conv2d(3, 8, 3, 1, 0)
        self.m2 = torch.nn.Conv2d(8, 8, 3, 1, 1)

    def forward(self, x):
        y0 = self.m1(x)
        y1 = self.m2(y0)
        y2 = y0 + y1
        return y2

# Create a SimpleModel and save its weight in the current directory
model = SimpleModel()
torch.save(model.state_dict(), "weight.pth")
```

After running the script, we will get a weight.pth weight file in the current directory.

10.1.3 Step 2: Export ONNX model

Create another script named export_onnx.py in the same directory and run it. The specific content of the script is as follows:

```
#!/usr/bin/env python3
import torch
from simple_net import SimpleModel

# Load the pretrained model and export it as onnx
model = SimpleModel()
model.eval()
checkpoint = torch.load("weight.pth", map_location="cpu")
model.load_state_dict(checkpoint)

# Prepare input tensor
input = torch.randn(1, 3, 16, 16, requires_grad=True)

# Export the torch model as onnx
torch.onnx.export(model,
                  input,
                  'model.onnx', # name of the exported onnx model
```

(continues on next page)

(continued from previous page)

```
                opset_version=13,
                export_params=True,
                do_constant_folding=True)
```

After running the script, we can get the onnx model named model.onnx in the current directory.

10.2 TensorFlow model to ONNX

In this section, we use the mobilenet_v1_0.25_224 model provided in the TensorFlow official repository as a conversion example.

10.2.1 Step 0: Create a working directory

Create and enter the tf_model directory using the command line.

```
$ mkdir tf_model
$ cd tf_model
```

10.2.2 Step 1: Prepare and convert the model

Download the model with the following commands and use the tf2onnx tool to export it as an ONNX model:

```
$ wget -nc http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_0.
↪25_224.tgz
# tar to get "*.pb" model def file
$ tar xzf mobilenet_v1_0.25_224.tgz
$ python -m tf2onnx.convert --graphdef mobilenet_v1_0.25_224_frozen.pb \
   --output mnet_25.onnx --inputs input:0 \
   --inputs-as-nchw input:0 \
   --outputs MobilenetV1/Predictions/Reshape_1:0
```

After running all commands, we can get the onnx model named mnet_25.onnx in the current directory.

10.3 PaddlePaddle model to ONNX

This section uses the SqueezeNet1_1 model provided in the official PaddlePaddle repository as a conversion example.

10.3.1 Step 0: Create a working directory

Create and enter the pp_model directory using the command line.

```
$ mkdir pp_model
$ cd pp_model
```

10.3.2 Step 1: Prepare the model

Download the model with the following commands:

```
$ wget https://bj.bcebos.com/paddlehub/fastdeploy/SqueezeNet1_1_infer.tgz
$ tar xzf SqueezeNet1_1_infer.tgz
$ cd SqueezeNet1_1_infer
```

In addition, use the paddle_infer_shape.py script from the PaddlePaddle project to perform shape inference on the model. The input shape is set to [1,3,224,224] in NCHW format here:

```
$ wget https://raw.githubusercontent.com/PaddlePaddle/Paddle2ONNX/develop/tools/paddle/
↪paddle_infer_shape.py
$ python paddle_infer_shape.py  --model_dir . \
                          --model_filename inference.pdmodel \
                          --params_filename inference.pdiparams \
                          --save_dir new_model \
                          --input_shape_dict="{'inputs':[1,3,224,224]}"
```

After running all commands, we will be in the SqueezeNet1_1_infer directory, and there will be a new_model directory under this directory.

10.3.3 Step 2: Convert the model

Install the paddle2onnx tool through the following commands, and use this tool to convert the PaddlePaddle model to the ONNX format:

```
$ pip install paddle2onnx
$ paddle2onnx  --model_dir new_model \
          --model_filename inference.pdmodel \
          --params_filename inference.pdiparams \
          --opset_version 13 \
          --save_file squeezenet1_1.onnx
```

After running all the above commands we will get an onnx model named squeezenet1_1.onnx.

CHAPTER 11

Appendix.02: CV18xx Guidance

CV18xx series chip currently supports ONNX and Caffe models but not TFLite models. In terms of quantization, CV18xx supports BF16 and symmetric INT8 format. This chapter takes the CV183X as an example to introduce the compilation and runtime sample of the CV18xx series chip.

11.1 Compile yolov5 model

11.1.1 TPU-MLIR Setup

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

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

envsetup.sh adds the following environment variables:

Table 11.1: 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:

```
export PATH=${TPUC_ROOT}/bin:$PATH
export PATH=${TPUC_ROOT}/python/tools:$PATH
export PATH=${TPUC_ROOT}/python/utils:$PATH
export PATH=${TPUC_ROOT}/python/test:$PATH
export PATH=${TPUC_ROOT}/python/samples:$PATH
export PATH=${TPUC_ROOT}/customlayer/python:$PATH
export LD_LIBRARY_PATH=$TPUC_ROOT/lib:$LD_LIBRARY_PATH
export PYTHONPATH=${TPUC_ROOT}/python:$PYTHONPATH
export PYTHONPATH=${TPUC_ROOT}/customlayer/python:$PYTHONPATH
export MODEL_ZOO_PATH=${TPUC_ROOT}/../model-zoo
export REGRESSION_PATH=${TPUC_ROOT}/regression
```

11.1.2 Prepare working directory

Create the model_yolov5s directory in the same directory as tpu-mlir, and put the model and image files in this directory.

The operation is as follows:

```
$ mkdir model_yolov5s && cd model_yolov5s
$ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

Here $TPUC_ROOT is an environment variable, corresponding to the tpu-mlir_xxxx directory.

11.1.3 ONNX to MLIR

If the input is an image, we need to learn the preprocessing of the model before conversion. If the model uses the preprocessed npz file as input, there is no need to consider preprocessing. The preprocessing process is expressed as follows (x stands for input):

$$y = (x - mean) \times scale$$

The input of yolov5 on the official website is rgb image, each value of it will be multiplied by 1/255, and converted into mean and scale corresponding to 0.0,0.0,0.0 and 0.0039216,0.0039216,0.0039216.

The model conversion command is as follows:

```
$ 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 326,474,622 \
    --test_input ../image/dog.jpg \
    --test_result yolov5s_top_outputs.npz \
    --mlir yolov5s.mlir
```

For the argument description of model_transform, refer to the section The main parameters of model_transform.py .

11.1.4 MLIR to BF16 Model

Convert the mlir file to the cvimodel of bf16, the operation is as follows:

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize BF16 \
    --chip cv183x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --model yolov5s_cv183x_bf16.cvimodel
```

For the argument description of model_deploy.py, refer to the section The main parameters of model_deploy.py .

11.1.5 MLIR to INT8 Model

Before converting to the INT8 model, you need to do calibration to get the calibration table. The number of input data depends on the situation but is normally around 100 to 1000. Then use the calibration table to generate INT8 symmetric cvimodel.

Here we use the 100 images from COCO2017 as an example to perform calibration:

```
$ run_calibration.py yolov5s.mlir \
    --dataset ../COCO2017 \
    --input_num 100 \
    -o yolov5s_cali_table
```

After the operation is completed, a file named ${model_name}_cali_table will be generated, which is used as the input of the following compilation work.

To convert to symmetric INT8 cvimodel model, execute the following command:

```
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --chip cv183x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --model yolov5s_cv183x_int8_sym.cvimodel
```

After compiling, a file named ${model_name}_cv183x_int8_sym.cvimodel will be generated.

11.1.6 Result Comparison

The onnx model is run as follows to get dog_onnx.jpg:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model ../yolov5s.onnx \
    --output dog_onnx.jpg
```

The FP32 mlir model is run as follows to get dog_mlir.jpg:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s.mlir \
    --output dog_mlir.jpg
```

The BF16 cvimodel is run as follows to get dog_bf16.jpg:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_cv183x_bf16.cvimodel \
    --output dog_bf16.jpg
```

The INT8 cvimodel is run as follows to get dog_int8.jpg:

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_cv183x_int8_sym.cvimodel \
    --output dog_int8.jpg
```

The comparison of the four images is shown in Fig. 11.1, due to the different operating environments, the final effect and accuracy will be slightly different from Fig. 11.1.

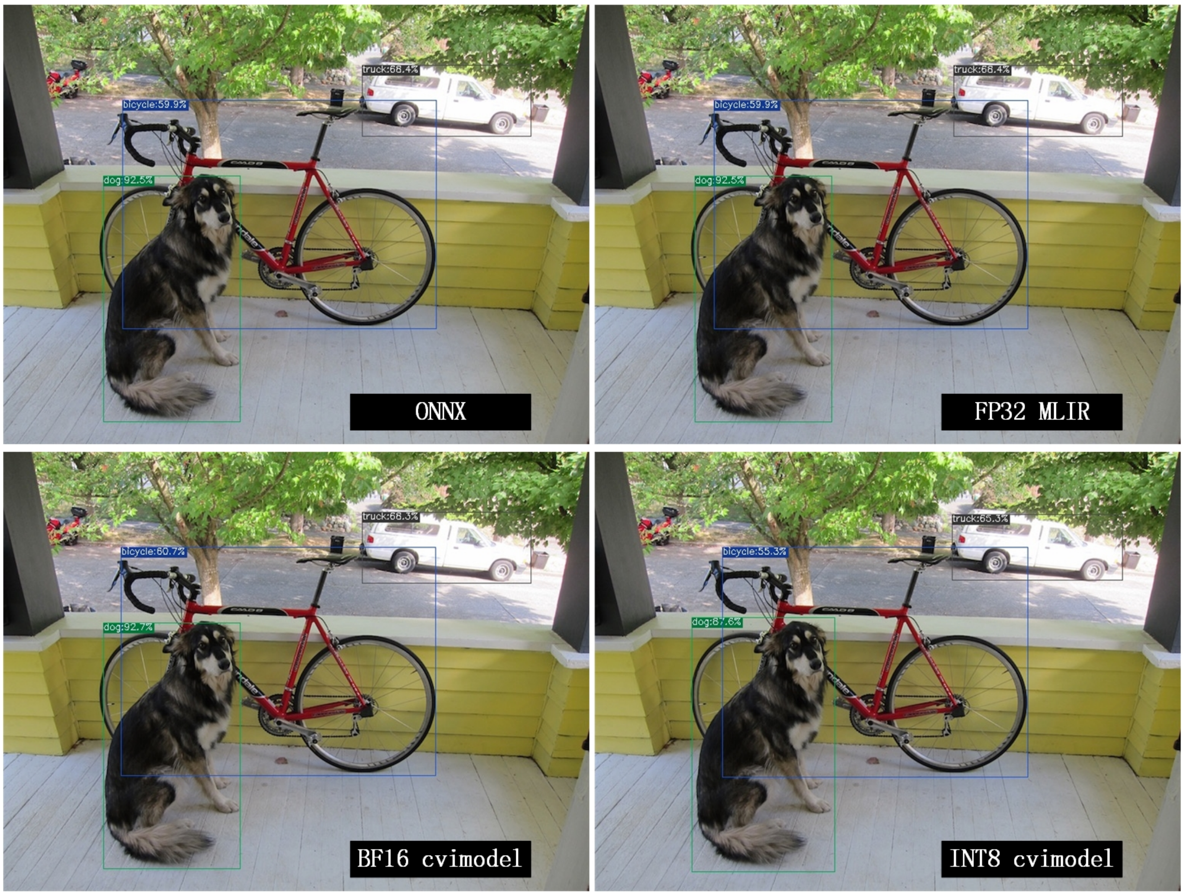


Fig. 11.1: Comparing the results of different models

The above tutorial introduces the process of TPU-MLIR deploying the ONNX model to the CV18xx series chip. For the conversion process of the Caffe model, please refer to the chapter "Compiling the Caffe Model" . You only need to replace the chip name with the specific CV18xx chip.

11.2 Merge cvimodel Files

For the same model, independent cvimodel files can be generated according to the input batch size and resolution(different H and W). However, in order to save storage, you can merge these related cvimodel files into one cvimodel file and share its weight part. The steps are as follows:

11.2.1 Step 0: generate the cvimodel for batch 1

Please refer to the previous section to create a new workspace directory and convert yolov5s to the mlir fp32 model by model_transform.py

---

Attention：

1.Use the same workspace directory for the cvimodels that need to be merged, and do not share the workspace with other cvimodes that do not need to be merged.

2.In Step 0, Step 1, –merge_weight is required

```
$ 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 326,474,622 \
    --test_input ../image/dog.jpg \
    --test_result yolov5s_top_outputs.npz \
    --mlir yolov5s_bs1.mlir
```

Use the yolov5s_cali_table generated in preceding sections, or generate calibration table by run_calibration.py.

```
# Add --merge_weight
 $ model_deploy.py \
    --mlir yolov5s_bs1.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --chip cv183x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --merge_weight \
    --model yolov5s_cv183x_int8_sym_bs1.cvimodel
```

11.2.2 Step 1: generate the cvimodel for batch 2

Generate mlir fp32 file in the same workspace:

```
$ model_transform.py \
    --model_name yolov5s \
    --model_def ../yolov5s.onnx \
    --input_shapes [[2,3,640,640]] \
    --mean 0.0,0.0,0.0 \
    --scale 0.0039216,0.0039216,0.0039216 \
    --keep_aspect_ratio \
    --pixel_format rgb \
    --output_names 326,474,622 \
    --test_input ../image/dog.jpg \
    --test_result yolov5s_top_outputs.npz \
    --mlir yolov5s_bs2.mlir
```

```
# Add --merge_weight
 $ model_deploy.py \
    --mlir yolov5s_bs2.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --chip cv183x \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --merge_weight \
    --model yolov5s_cv183x_int8_sym_bs2.cvimodel
```

11.2.3 Step 2: merge the cvimodel of batch 1 and batch 2

Use model_tool to mrege two cvimodel files:

```
model_tool \
  --combine \
    yolov5s_cv183x_int8_sym_bs1.cvimodel \
    yolov5s_cv183x_int8_sym_bs2.cvimodel \
    -o yolov5s_cv183x_int8_sym_bs1_bs2.cvimodel
```

11.2.4 Step 3: use the cvimodel through the runtime interface

Use model_tool to check the program id of bs1 and bs2.:

```
model_tool --info yolov5s_cv183x_int8_sym_bs1_bs2.cvimodel
```

At runtime, you can run different batch program in the following ways:

```
CVI_MODEL_HANDEL bs1_handle;
CVI_RC ret = CVI_NN_RegisterModel("yolov5s_cv183x_int8_sym_bs1_bs2.cvimodel", &bs1_
↪handle);
assert(ret == CVI_RC_SUCCESS);
// choice batch 1 program
CVI_NN_SetConfig(bs1_handle, OPTION_PROGRAM_INDEX, 0);
CVI_NN_GetInputOutputTensors(bs1_handle, ...);
....


CVI_MODEL_HANDLE bs2_handle;
// Reuse loaded cvimodel
CVI_RC ret = CVI_NN_CloneModel(bs1_handle, &bs2_handle);
assert(ret == CVI_RC_SUCCESS);
// choice batch 2 program
CVI_NN_SetConfig(bs2_handle, OPTION_PROGRAM_INDEX, 1);
CVI_NN_GetInputOutputTensors(bs2_handle, ...);
...

// clean up bs1_handle and bs2_handle
CVI_NN_CleanupModel(bs1_handle);
CVI_NN_CleanupModel(bs2_handle);
```

11.2.5 Overview:

Using the above command, you can merge either the same models or different models

The main steps are:

1. When generating a cvimodel through model_deploy.py, add the –merge_weight parameter.
2. The work directory of the model to be merged must be the same, and do not clean up any intermediate files before merging the models(Reuse the previous model’s weight is implemented through the intermediate file _weight_map.csv).
3. Use model_tool to merge cvimodels.

11.3 Compile and Run the Runtime Sample

This part introduces how to compile and run the runtime samples, include how to cross-compile samples for EVB board and how to compile and run samples in docker. The following 4 samples are included:

- Sample-1 : classifier (mobilenet_v2)
- Sample-2 : classifier_bf16 (mobilenet_v2)
- Sample-3 : classifier fused preprocess (mobilenet_v2)

- Sample-4 : classifier multiple batch (mobilenet_v2)

11.3.1 1) Run the provided pre-build samples

The following files are required:

- cvitek_tpu_sdk_[cv186x|cv183x|cv182x|cv182x_uclibc|cv181x_glibc32|cv181x_musl_riscv64_rvv|cv
- cvimodel_samples_[cv186x|cv183x|cv182x|cv181x|cv180x].tar.gz

Select the required files according to the chip type and load them into the EVB file system. Execute them on the Linux console of EVB. Here, we take CV183x as an example.

Unzip the model file (delivered in cvimodel format) and the TPU_SDK used by samples. Enter into the samples directory to execute the test. The process is as follows:

```
#env
tar zxf cvimodel_samples_cv183x.tar.gz
export MODEL_PATH=$PWD/cvimodel_samples
tar zxf cvitek_tpu_sdk_cv183x.tar.gz
export TPU_ROOT=$PWD/cvitek_tpu_sdk
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh
# get cvimodel info
cd samples
./bin/cvi_sample_model_info $MODEL_PATH/mobilenet_v2.cvimodel

####################################
# sample-1 : classifier
###################################
./bin/cvi_sample_classifier \
   $MODEL_PATH/mobilenet_v2.cvimodel \
   ./data/cat.jpg \
   ./data/synset_words.txt

# TOP_K[5]:
#  0.326172, idx 282, n02123159 tiger cat
#  0.326172, idx 285, n02124075 Egyptian cat
#  0.099609, idx 281, n02123045 tabby, tabby cat
#  0.071777, idx 287, n02127052 lynx, catamount
#  0.041504, idx 331, n02326432 hare

####################################
# sample-2 : classifier_bf16
###################################
./bin/cvi_sample_classifier_bf16 \
   $MODEL_PATH/mobilenet_v2_bf16.cvimodel \
   ./data/cat.jpg \
   ./data/synset_words.txt

# TOP_K[5]:
#  0.314453, idx 285, n02124075 Egyptian cat
#  0.040039, idx 331, n02326432 hare
```

(continues on next page)

(continued from previous page)

```
#  0.018677, idx 330, n02325366 wood rabbit, cottontail, cottontail rabbit
#  0.010986, idx 463, n02909870 bucket, pail
#  0.010986, idx 852, n04409515 tennis ball


############################################
# sample-3 : classifier fused preprocess
############################################
./bin/cvi_sample_classifier_fused_preprocess \
   $MODEL_PATH/mobilenet_v2_fused_preprocess.cvimodel \
   ./data/cat.jpg \
   ./data/synset_words.txt

# TOP_K[5]:
#  0.326172, idx 282, n02123159 tiger cat
#  0.326172, idx 285, n02124075 Egyptian cat
#  0.099609, idx 281, n02123045 tabby, tabby cat
#  0.071777, idx 287, n02127052 lynx, catamount
#  0.041504, idx 331, n02326432 hare

############################################
# sample-4 : classifier multiple batch
############################################
./bin/cvi_sample_classifier_multi_batch \
   $MODEL_PATH/mobilenet_v2_bs1_bs4.cvimodel \
   ./data/cat.jpg \
   ./data/synset_words.txt

# TOP_K[5]:
#  0.326172, idx 282, n02123159 tiger cat
#  0.326172, idx 285, n02124075 Egyptian cat
#  0.099609, idx 281, n02123045 tabby, tabby cat
#  0.071777, idx 287, n02127052 lynx, catamount
#  0.041504, idx 331, n02326432 hare
```

At the same time, the script is provided as a reference, and the execution effect is the same as that of direct operation, as follows:

```
./run_classifier.sh
./run_classifier_bf16.sh
./run_classifier_fused_preprocess.sh
./run_classifier_multi_batch.sh
```

There are more samples can be refered in the cvitek_tpu_sdk/samples/samples_extra:

```
./bin/cvi_sample_detector_yolo_v3_fused_preprocess \
   $MODEL_PATH/yolo_v3_416_fused_preprocess_with_detection.cvimodel \
   ./data/dog.jpg \
   yolo_v3_out.jpg
```

(continues on next page)

(continued from previous page)

```
./bin/cvi_sample_detector_yolo_v5_fused_preprocess \
   $MODEL_PATH/yolov5s_fused_preprocess.cvimodel \
   ./data/dog.jpg \
   yolo_v5_out.jpg

./bin/cvi_sample_detector_yolox_s \
   $MODEL_PATH/yolox_s.cvimodel \
   ./data/dog.jpg \
   yolox_s_out.jpg

./bin/cvi_sample_alphapose_fused_preprocess \
   $MODEL_PATH/yolo_v3_416_fused_preprocess_with_detection.cvimodel \
   $MODEL_PATH/alphapose_fused_preprocess.cvimodel \
   ./data/pose_demo_2.jpg \
   alphapose_out.jpg

./bin/cvi_sample_fd_fr_fused_preprocess \
   $MODEL_PATH/retinaface_mnet25_600_fused_preprocess_with_detection.cvimodel \
   $MODEL_PATH/arcface_res50_fused_preprocess.cvimodel \
   ./data/obama1.jpg \
   ./data/obama2.jpg
```

11.3.2 2) Cross-compile samples

The source code is given in the released packages. You can cross-compile the samples' source code in the docker environment and run them on EVB board according to the following instructions.

The following files are required in this part:

- cvitek_tpu_sdk_[cv186x|cv183x|cv182x|cv182x_uclibc|cv181x_glibc32|cv181x_musl_riscv64_rvv|cv
- cvitek_tpu_samples.tar.gz

aarch 64-bit (such as cv183x aarch64-bit platform)

Prepare TPU sdk:

```
tar zxf host-tools.tar.gz
tar zxf cvitek_tpu_sdk_cv183x.tar.gz
export PATH=$PWD/host-tools/gcc/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin:
↪$PATH
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

Compile samples and install them into "install_samples" directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
cmake -G Ninja \
   -DCMAKE_BUILD_TYPE=RELEASE \
   -DCMAKE_C_FLAGS_RELEASE=-O3 \
   -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
   -DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-aarch64-linux.cmake \
   -DTPU_SDK_PATH=$TPU_SDK_PATH \
   -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
   -DCMAKE_INSTALL_PREFIX=../install_samples \
   ..
cmake --build . --target install
```

arm 32-bit (such as 32-bit cv183x/cv182x platform)

Prepare TPU sdk:

```
tar zxf host-tools.tar.gz
tar zxf cvitek_tpu_sdk_cv182x.tar.gz
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
export PATH=$PWD/host-tools/gcc/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf/bin:
↪$PATH
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

If docker version < 1.7, please update 32-bit system library(just once):

```
dpkg --add-architecture i386
apt-get update
apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Compile samples and install them into install_samples directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
cmake -G Ninja \
   -DCMAKE_BUILD_TYPE=RELEASE \
   -DCMAKE_C_FLAGS_RELEASE=-O3 \
   -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
   -DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-linux-gnueabihf.
↪cmake \
   -DTPU_SDK_PATH=$TPU_SDK_PATH \
   -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
   -DCMAKE_INSTALL_PREFIX=../install_samples \
   ..
cmake --build . --target install
```

uclibc 32-bit platform (such as cv182x uclibc platform)

Prepare TPU sdk:

```
tar zxf host-tools.tar.gz
tar zxf cvitek_tpu_sdk_cv182x_uclibc.tar.gz
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
export PATH=$PWD/host-tools/gcc/arm-cvitek-linux-uclibcgnueabihf/bin:$PATH
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

If docker version < 1.7, please update 32-bit system library(just once):

```
dpkg --add-architecture i386
apt-get update
apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Compile samples and install them into **install_samples** directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
cmake -G Ninja \
    -DCMAKE_BUILD_TYPE=RELEASE \
    -DCMAKE_C_FLAGS_RELEASE=-O3 \
    -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
    -DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-linux-uclibc.cmake \
    -DTPU_SDK_PATH=$TPU_SDK_PATH \
    -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
    -DCMAKE_INSTALL_PREFIX=../install_samples \
    ..
cmake --build . --target install
```

riscv 64-bit musl platform (such as cv180x/cv181x riscv 64-bit musl platform)

Prepare TPU sdk:

```
tar zxf host-tools.tar.gz
tar zxf cvitek_tpu_sdk_cv181x_musl_riscv64_rvv.tar.gz
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
export PATH=$PWD/host-tools/gcc/riscv64-linux-musl-x86_64/bin:$PATH
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

Compile samples and install them into **install_samples** directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
cmake -G Ninja \
```

(continues on next page)

(continued from previous page)

```
    -DCMAKE_BUILD_TYPE=RELEASE \
    -DCMAKE_C_FLAGS_RELEASE=-O3 \
    -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
    -DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-riscv64-linux-musl-
↪x86_64.cmake \
    -DTPU_SDK_PATH=$TPU_SDK_PATH \
    -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
    -DCMAKE_INSTALL_PREFIX=../install_samples \
    ..
cmake --build . --target install
```

riscv 64-bit glibc platform(such as cv180x/cv181x 64-bit glibc platform)

Prepare TPU sdk:

```
tar zxf host-tools.tar.gz
tar zxf cvitek_tpu_sdk_cv181x_glibc_riscv64.tar.gz
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
export PATH=$PWD/host-tools/gcc/riscv64-linux-x86_64/bin:$PATH
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

Compile samples and install them into install_samples directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
cmake -G Ninja \
    -DCMAKE_BUILD_TYPE=RELEASE \
    -DCMAKE_C_FLAGS_RELEASE=-O3 \
    -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
    -DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-riscv64-linux-x86_64.
↪cmake \
    -DTPU_SDK_PATH=$TPU_SDK_PATH \
    -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
    -DCMAKE_INSTALL_PREFIX=../install_samples \
    ..
cmake --build . --target install
```

11.3.3 3) Run samples in docker environment

The following files are required:

- cvitek_tpu_sdk_x86_64.tar.gz
- cvimodel_samples_[cv186x|cv183x|cv182x|cv181x|cv180x].tar.gz
- cvitek_tpu_samples.tar.gz

Prepare TPU sdk:

```
tar zxf cvitek_tpu_sdk_x86_64.tar.gz
export TPU_SDK_PATH=$PWD/cvitek_tpu_sdk
cd cvitek_tpu_sdk && source ./envs_tpu_sdk.sh && cd ..
```

Compile samples and install them into `install_samples` directory:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build
cd build
cmake -G Ninja \
  -DCMAKE_BUILD_TYPE=RELEASE \
  -DCMAKE_C_FLAGS_RELEASE=-O3 \
  -DCMAKE_CXX_FLAGS_RELEASE=-O3 \
  -DTPU_SDK_PATH=$TPU_SDK_PATH \
  -DCNPY_PATH=$TPU_SDK_PATH/cnpy \
  -DOPENCV_PATH=$TPU_SDK_PATH/opencv \
  -DCMAKE_INSTALL_PREFIX=../install_samples \
  ..
cmake --build . --target install
```

Run samples:

```
# envs
tar zxf cvimodel_samples_cv183x.tar.gz
export MODEL_PATH=$PWD/cvimodel_samples
source cvitek_mlir/cvitek_envs.sh

# get cvimodel info
cd ../install_samples
./bin/cvi_sample_model_info $MODEL_PATH/mobilenet_v2.cvimodel
```

Other samples are samely to the instructions of running on EVB board.

11.4 FAQ

11.4.1 Model transformation FAQ

1 Related to model transformation

1.1 Whether pytorch,tensorflow, etc. can be converted directly to cvimodel?

pytorch: Supports the .pt model statically via `jit.trace(torch_model.eval(), inputs).save('model_name.pt')`.

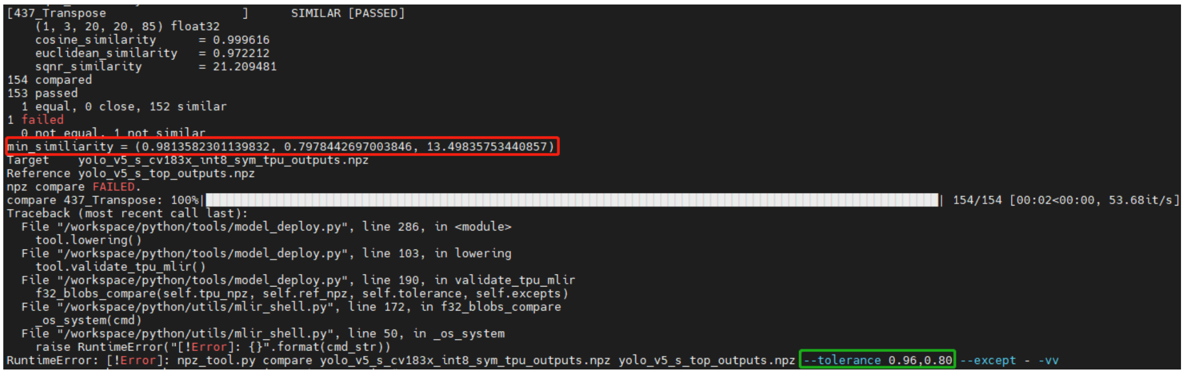
tensorflow / others: It is not supported yet and can be supported indirectly through onnx.

1.2 An error occurs when model transform.py is executed

model_transform.py This script convert the onnx,caffe model into the fp32 mlir. The high probability of error here is that there are unsupported operators or incompatible operator attributes, which can be fed back to the tpu team to solve.

1.3 An error occurs when model deploy.py is executed

model_deploy.py This script quantizes fp32 mlir to int8/bf16mlir, and then converts int8/bf16mlir to cvimodel. In the process of conversion, two similarity comparisons will be involved: one is the quantitative comparison between fp32 mlir and int8/bf16mlir, and the other is the similarity comparison between int8/bf16mlir and the final converted cvimodel. If the similarity comparison fails, the following err will occur:



Solution: The tolerance parameter is incorrect. During the model conversion process, similarity will be calculated for the output of int8/bf16 mlir and fp32 mlir, and tolerance is to limit the minimum value of similarity. If the calculated minimum value of similarity is lower than the corresponding preset tolerance value, the program will stop execution. Consider making adjustments to tolerance. (If the minimum similarity value is too low, please report it to the tpu team.)

1.4 What is the difference between the pixel_format parameter of model_transform.py and the customization_format parameter of model_deploy.py?

Channel_order is the input image type of the original model (only gray/rgb planar/bgr planar is supported),customization_format is the input image type of cvimodel, which is determined by the customer and must be used together with fuse_preprocess. (If the input is a YUV image obtained through VPSS or VI, set customization_format to YUV format.) If pixel_format is inconsistent with customization_format,cvimodel will automatically converts the input to the type specified by pixel_format.

1.5 Whether the multi-input model is supported and how to preprocess it?

Models with multiple input images using different preprocessing methods are not supported.

2 Related to model quantization

2.1 run run_calibration.py raise KeyError: 'images'

Please check that the path of the data set is correct.

2.2 How to deal with multiple input problems by running quantization?

When running run_calibration.py, you can store multiple inputs using .npz, or using the –data_list argument, and the multiple inputs in each row of the data_list are separated by ",".

2.3 Is the input preprocessed when quantization is performed?

Yes, according to the preprocessing parameters stored in the mlir file, the quantization process is preprocessed by loading the preprocessing parameters.

2.4 The program is killed by the system or the memory allocation fails when run calibration

It is necessary to check whether the memory of the host is enough, and the common model requires about 8G memory. If memory is insufficient, try adding the following parameters when running run_calibration.py to reduce memory requirements.

```
--tune_num 2                    # default is 5
```

2.5 Does the calibration table support manual modification?

Supported, but it is not recommended.

3 Others

3.1 Does the converted model support encryption?

Not supported for now.

3.2 What is the difference in inference speed between bf16 model and int8 model?

The theoretical difference is about 3-4 times, and there will be differences for different models, which need to be verified in practice.

3.3 Is dynamic shape supported?

Cvimodel does not support dynamic shape. If several shapes are fixed, independent cvimodel files can be generated through the form of shared weights. See Merge cvimodel Files for details.

11.4.2 Model performance evaluation FAQ

1 Evaluation process

First converted to bf16 model, through the `model_tool --info xxxx.cvimodel` command to obtain the ION memory and the storage space required by the model , and then execute `model_runner` on the EVB board to evaluate the performance, and then evaluate the accuracy in the business scenario according to the provided sample. After the accuracy of the model output meets the expectation, the same evaluation is performed on the int8 model.

2 After quantization, the accuracy does not match the original model, how to debug?

2.1 Ensure `--test_input`, `--test_reference`, `--compare_all` , `--tolerance` parameters are set up correctly.

2.2 Compare the results of the original model and the bf16 model. If the error is large, check whether the pre-processing and post-processing are correct.

2.3 If int8 model accuracy is poor:

1) Verify that the data set used by run_calibration.py is the validation set used when training the model;
2) A business scenario data set (typically 100-1000 images) can be added for run_calibration.

2.4 Confirm the input type of cvimodel:

1) If the `--fuse_preprocess` argument is specified, the input type of cvimodel is uint8;
2) If `--quant_input` is specified,in general,bf16_cvimoel input type is fp32,int8_cvimodel input type is int8;
3) The input type can also be obtained with `model_tool --info xxx.cvimodel`

3 bf16 model speed is relatively slow,int8 model accuracy does not meet expectations how to do?

Try using a mixed-precision quantization method. See mix precision for details.

11.4.3 Common problems of model deployment

1 The The CVI_NN_Forward interface encounters an error after being invoked for many times or is stuck for a long time

There may be driver or hardware issues that need to be reported to the tpu team for resolution.

2 Is the model preprocessing slow?

2.1 Add the --fuse_preprocess parameter when running model_deploy.py, which will put the preprocessing inside the TPU for processing.

2.2 If the image is obtained from vpss or vi, you can use --fuse_preprocess, --aligned_input when converting to the model. Then use an interface such as CVI_NN_SetTensorPhysicalAddr to set the input tensor address directly to the physical address of the image, reducing the data copy time.

3 Are floating-point and fixed-point results the same when comparing the inference results of docker and evb ?

Fixed point has no difference, floating point has difference, but the difference can be ignored.

4 Support multi-model inference parallel?

Multithreading is supported, but models are inferred on TPU in serial.

5 Fill input tensor related interface

CVI_NN_SetTensorPtr : Set the virtual address of input tensor, and the original tensor memory will not be freed. Inference **copies data** from a user-set virtual address to the original tensor memory.

CVI_NN_SetTensorPhysicalAddr : Set the physical address of input tensor, and the original tensor memory will be freed. Inference directly reads data from the newly set physical address, **data copy is not required** . A Frame obtained from VPSS can call this interface by passing in the Frame' s first address. Note that model_deploy.py must be set --fused_preprocess and --aligned_input .

CVI_NN_SetTensorWithVideoFrame : Fill the Input Tensor with the VideoFrame structure. Note The address of VideoFrame is a physical address. If the model is fused preprocess and aligned_input, it is equivalent to CVI_NN_SetTensorPhysicalAddr, otherwise the VideoFrame data will be copied to the Input Tensor.

CVI_NN_SetTensorWithAlignedFrames : Support multi-batch, similar to CVI_NN_SetTensorWithVideoFrame .

CVI_NN_FeedTensorWithFrames : similar to CVI_NN_SetTensorWithVideoFrame .

6 How is ion memory allocated after model loading

6.1 Calling CVI_NN_RegisterModel allocates ion memory for weight and cmdbuf (you can see the weight and cmdbuf sizes by using model_tool).

6.2 Calling CVI_NN_GetInputOutputTensors allocates ion memory for tensor(including private_gmem, shared_gmem, io_mem).

6.3 Calling CVI_NN_CloneModel can share weight and cmdbuf memory.

6.4 Other interfaces do not apply for ion memory.

6.5 Shared_gmem of different models can be shared (including multithreading), so initializing shared_gmem of the largest model first will saves ion memory.

7 The model inference time becomes longer after loading the business program

Generally, after services are loaded, the tdma_exe_ms becomes longer, but the tiu_exe_ms remains unchanged. This is because tdma_exe_ms takes time to carry data in memory. If the memory bandwidth is insufficient, the tdma time will increase.

suggestion:

1) vpss/venc optimize chn and reduce resolution

2) Reduces memory copy

3) Fill input tensor by using copy-free mode

11.4.4 Others

1 In the cv182x/cv181x/cv180x on-board environment, the taz:invalid option –z decompression fails

Decompress the sdk in other linux environments and then use it on the board. windows does not support soft links. Therefore, decompressing the SDK in Windows may cause the soft links to fail and an error may be reported

2 If tensorflow model is pb form of saved_model, how to convert it to pb form of frozen_model

```
import tensorflow as tf
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input, decode_
↪predictions
import numpy as np
import tf2onnx
import onnxruntime as rt

img_path = "./cat.jpg"
# pb model and variables should in model dir
pb_file_path = "your model dir"
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
# Or set your preprocess here
x = preprocess_input(x)

model = tf.keras.models.load_model(pb_file_path)
preds = model.predict(x)

# different model input shape and name will differently
spec = (tf.TensorSpec((1, 224, 224, 3), tf.float32, name="input"), )
output_path = model.name + ".onnx"

model_proto, _ = tf2onnx.convert.from_keras(model, input_signature=spec,
↪opset=13, output_path=output_path)
```

CHAPTER 12

Appendix.03: Test SDK release package with TPU-PERF

12.1 Configure the system environment

If you are using Docker for the first time, use the methods in Environment Setup to install and configure Docker. At the same time, **git-lfs** will be used in this chapter. If you use **git-lfs** for the first time, you can execute the following commands for installation and configuration (only for the first time, and the configuration is in the user’s own system, not in Docker container):

```
$ curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
$ sudo apt-get install git-lfs
```

12.2 Get the model-zoo model[1]

In the same directory of **tpu-mlir_xxxx.tar.gz** (tpu-mlir’s release package), use the following command to clone the **model-zoo** project:

```
$ git clone --depth=1 https://github.com/sophgo/model-zoo
$ cd model-zoo
$ git lfs pull --include "*.onnx,*.jpg,*.JPEG,*.npz" --exclude=""
$ cd ../
```

[1] If you get the **model-zoo** test package provided by SOPHGO, you can do the following to create and set up the **model-zoo**. After completing this step, go directly to the next section Get the tpu-perf tool.

```
$ mkdir -p model-zoo
$ tar -xvf path/to/model-zoo_<date>.tar.bz2 --strip-components=1 -C model-zoo
```

If you have cloned model-zoo, you can execute the following command to synchronize the model to the latest state:

```
$ cd model-zoo
$ git pull
$ git lfs pull --include "*.onnx,*.jpg,*.JPEG" --exclude=""
$ cd ../
```

This process downloads a large amount of data from GitHub. Due to differences in specific network environments, this process may take a long time.

12.3 Get the tpu-perf tool

Download the latest tpu-perf wheel installation package from https://github.com/sophgo/tpu-perf/releases. For example, tpu_perf-x.x.x-py3-none-manylinux2014_x86_64.whl. And put the tpu-perf package in the same directory as model-zoo. The directory structure at this point should look like this:

```
├─── tpu_perf-x.x.x-py3-none-manylinux2014_x86_64.whl
├─── tpu-mlir_xxxx.tar.gz
└─── model-zoo
```

12.4 Test process

12.4.1 Unzip the SDK and create a Docker container

Execute the following command in the tpu-mlir_xxxx.tar.gz directory (note that tpu-mlir_xxxx.tar.gz and model-zoo needs to be at the same level):

```
$ tar zxf tpu-mlir_xxxx.tar.gz
$ docker pull sophgo/tpuc_dev:v3.1
$ docker run --rm --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.1
```

After running the command, it will be in a Docker container.

12.4.2 Set environment variables and install tpu-perf

Complete setting the environment variables needed to run the tests with the following command:

```
$ cd tpu-mlir_xxxx
$ source envsetup.sh
```

There will be no prompts after the process ends. Then install tpu-perf with the following command:

```
$ pip3 install ../tpu_perf-x.x.x-py3-none-manylinux2014_x86_64.whl
```

12.4.3 Run the test

Compile the model

confg.yaml in model-zoo configures the test content of the SDK. For example, the configuration file for resnet18 is model-zoo/vision/classification/resnet18-v2/config.yaml .

Execute the following command to run all test samples:

```
$ cd ../model-zoo
$ python3 -m tpu_perf.build --mlir -l full_cases.txt
```

The following models are compiled (Due to continuous additions of models in the model-zoo, only a partial list of models is provided here; at the same time, this process also compiles models for accuracy testing, and subsequent accuracy testing sections do not require recompilation of models.):

```
* efficientnet-lite4
* mobilenet_v2
* resnet18
* resnet50_v2
* shufflenet_v2
* squeezenet1.0
* vgg16
* yolov5s
* ...
```

After the command is finished, you will see the newly generated output folder (where the test output is located). Modify the properties of the output folder to make it accessible to systems outside of Docker.

```
$ chmod -R a+rw output
```

Test model performance

12.4.4 Configure SOC device

Note: If your device is a PCIE board, you can skip this section directly.

The performance test only depends on the libsophon runtime environment, so after packaging models, compiled in the toolchain compilation environment, and model-zoo, the performance test can be carried out in the SOC environment by tpu_perf. However, the complete model-zoo as well as compiled output contents may not be fully copied to the SOC since the storage on the SOC device is limited. Here is a method to run tests on SOC devices through linux nfs remote file system mounts.

First, install the nfs service on the toolchain environment server "host system":

```
$ sudo apt install nfs-kernel-server
```

Add the following content to /etc/exports (configure the shared directory):

```
/the/absolute/path/of/model-zoo *(rw,sync,no_subtree_check,no_root_squash)
```

Where * means that everyone can access the shared directory. Moreover, it can be configured to be accessible by a specific network segment or IP, such as:

```
/the/absolute/path/of/model-zoo 192.168.43.0/24(rw,sync,no_subtree_check,no_root_squash)
```

Then execute the following command to make the configuration take effect:

```
$ sudo exportfs -a
$ sudo systemctl restart nfs-kernel-server
```

In addition, you need to add read permissions to the images in the dataset directory:

```
chmod -R +r path/to/model-zoo/dataset
```

Install the client on the SOC device and mount the shared directory:

```
$ mkdir model-zoo
$ sudo apt-get install -y nfs-common
$ sudo mount -t nfs <IP>:/path/to/model-zoo ./model-zoo
```

In this way, the test directory is accessible in the SOC environment. The rest of the SOC test operation is basically the same as that of PCIE. Please refer to the following content for operation. The difference in command execution position and operating environment has been explained in the execution place.

12.4.5 Run the test

Running the test needs to be done in an environment outside Docker (it is assumed that you have installed and configured the 1684X device and driver), so you can exit the Docker environment:

```
$ exit
```

1. Run the following commands under the PCIE board to test the performance of the generated bmodel.

```
$ pip3 install ./tpu_perf-*-py3-none-manylinux2014_x86_64.whl
$ cd model-zoo
$ python3 -m tpu_perf.run --mlir -l full_cases.txt
```

Note: If multiple SOPHGO accelerator cards are installed on the host, you can specify the running device of tpu_perf by adding --devices id when using tpu_perf. Such as:

```
$ python3 -m tpu_perf.run --devices 2 --mlir -l full_cases.txt
```

2. The SOC device uses the following steps to test the performance of the generated bmodel.

Download the latest tpu-perf, tpu_perf-x.x.x-py3-none-manylinux2014_aarch64.whl, from https://github.com/sophgo/tpu-perf/releases to the SOC device and execute the following operations:

```
$ pip3 install ./tpu_perf-x.x.x-py3-none-manylinux2014_aarch64.whl
$ cd model-zoo
$ python3 -m tpu_perf.run --mlir -l full_cases.txt
```

After that, performance data is available in output/stats.csv, in which the running time, computing resource utilization, and bandwidth utilization of the relevant models are recorded.

12.4.6 Precision test

Precision test shall be carried out in the running environment beyond docker. It is optional to exit docker environment:

```
exit
```

Run the following commands under the PCIE board to test the precision of the generated bmodel.

```
$ pip3 install ./tpu_perf-*-py3-none-manylinux2014_x86_64.whl
$ cd model-zoo
$ python3 -m tpu_perf.precision_benchmark --mlir -l full_cases.txt
```

Various types of precision data are available in individual csv files in the output directory.

Note: If multiple SOPHGO accelerator cards are installed on the host, you can specify the running device of tpu_perf by adding --devices id when using tpu_perf. Such as:

```
$ python3 -m tpu_perf.precision_benchmark --devices 2 --mlir -l full_cases.txt
```

CHAPTER 13

Appendix.04: Supported Operations

13.1 List of operators currently supported by TPU-MLIR

Table 13.1: A

Onnx	Pytorch	Caffe	TOP
Abs	aten::abs	AbsVal	top.Abs
Add	aten::adaptive_avg_pool2	ArgMax	top.AdaptiveAvgPool
And	aten::add		top.Add
ArgMax	aten::addmm		top.AddConst
ArgMin	aten::arange		top.Arange
AveragePool	aten::avg_pool1d		top.Arg
	aten::avg_pool2d		top.Attention
	aten::avg_pool3d		top.AvgPool

Table 13.2: B

Onnx	Pytorch	Caffe	TOP
BatchNormalization	aten::baddbmm	BatchNorm	top.BatchNorm
	aten::batch_norm	BN	top.BatchNormBwd
	aten::bmm		top.BatchNormTrain

Table 13.3: C

Onnx	Pytorch	Caffe	TOP
Cast	aten::cat	Concat	top.Cast
Ceil	aten::ceil	ContinuationIndicator	top.Ceil
Clip	aten::channel_shuffle	Convolution	top.Clip
Concat	aten::chunk	ConvolutionDepthwise	top.Compare
Constant	aten::clamp	Crop	top.CompareConst
ConstantOfShape	aten::constant_pad_nd		top.Concat
Conv	aten::contiguous		top.ConstantFill
ConvTranspose	aten::_convolution		top.Conv
Cos	aten::_convolution_mode		top.ConvBwd_Weight
CumSum	aten::copy		top.Copy
	aten::cos		top.Cos
	aten::cosh		top.Cosh
			top.Csc
			top.CumSum
			top.Custom

Table 13.4: D

Onnx	Pytorch	Caffe	TOP
DepthToSpace	aten::detach	Deconvolution	top.Deconv
DequantizeLinear	aten::div	DetectionOutput	top.DeformConv2D
Div	aten::dropout	Dropout	top.Depth2Space
Dropout		DummyData	top.DequantizeLinear
			top.DetectionOutput
			top.Div

Table 13.5: E

Onnx	Pytorch	Caffe	TOP
Einsum	aten::elu	Eltwise	top.Einsum
Elu	aten::embedding	Embed	top.Elu
Equal	aten::empty		top.EmbDenseBwd
Erf	aten::eq		top.Erf
Exp	aten::erf		top.Exp
Expand	aten::exp		top.Expand
	aten::expand		
	aten::expand_as		

Table 13.6: F

Onnx	Pytorch	Caffe	TOP
Flatten	aten::flatten	Flatten	top.Flatten
Floor	aten::floor	FrcnDetection	top.Floor
	aten::floor_divide		top.FrcnDetection

Table 13.7: G

Onnx	Pytorch	Caffe	TOP
Gather	aten::gather		top.GELU
GatherElements	aten::ge		top.GRU
GatherND	aten::gelu		top.Gather
GELU	aten::grid_sampler		top.GatherElements
Gemm	aten::group_norm		top.GatherND
GlobalAveragePool	aten::gru		top.GridSampler
GlobalMaxPool	aten::gt		top.GroupNorm
Greater			
GreaterOrEqual			
GridSample			
GroupNormalization			
GRU			

Table 13.8: H

Onnx	Pytorch	Caffe	TOP
HardSigmoid	aten::hardsigmoid		top.HardSigmoid
HardSwish	aten::hardswish		top.HardSwish
	aten::hardtanh		

Table 13.9: I

Onnx	Pytorch	Caffe	TOP
Identity	aten::index_select	ImageData	top.If
If	aten::instance_norm	InnerProduct	top.Input
InstanceNormalization		Input	top.InstanceNorm
		Interp	top.Interp

Table 13.10: L

Onnx	Pytorch	Caffe	TOP
LayerNormalization	aten::layer_norm	LRN	top.LRN
LeakyRelu	aten::le	LSTM	top.LSTM
Less	aten::leaky_relu	Lstm	top.LayerNorm
LessOrEqual	aten::linear		top.LayerNormBwd
Log	aten::log_sigmoid		top.LayerNormTrain
LogSoftmax	aten::log_softmax		top.LeakyRelu
Loop	aten::lstm		top.List
LRN	aten::lt		top.Log
LSTM			top.Loop

Table 13.11: M

Onnx	Pytorch	Caffe	TOP
MatMul	aten::masked_fill	MatMul	top.MaskedFill
Max	aten::matmul	Mish	top.MatMul
MaxPool	aten::max		top.MatchTemplate
Min	aten::max_pool1d		top.Max
Mul	aten::max_pool2d		top.MaxConst
	aten::max_pool3d		top.MaxPool
	aten::mean		top.MaxPoolWithMask
	aten::meshgrid		top.MaxUnpool
	aten::min		top.MeshGrid
	aten::mish		top.Min
	aten::mm		top.MinConst
	aten::mul		top.Mish
			top.Mul
			top.MulConst

Table 13.12: N

Onnx	Pytorch	Caffe	TOP
Neg	aten::ne	Normalize	top.Nms
NonMaxSuppression	aten::neg		top.NonZero
NonZero	aten::new_ones		top.None
Not	aten::new_zeros		top.Normalize

Table 13.13: O

Onnx	Pytorch	Caffe	TOP
	aten::ones		
	aten::ones_like		

Table 13.14: P

Onnx	Pytorch	Caffe	TOP
Pad	aten::pad	Padding	top.PRelu
PixelNormalization	aten::permute	Permute	top.Pack
Pow	aten::pixel_shuffle	Pooling	top.Pad
PRelu	aten::pixel_unshuffle	Power	top.Permute
	aten::pow	PReLU	top.PixelNorm
	aten::prelu	PriorBox	top.PoolMask
		Proposal	top.Pow
			top.Pow2
			top.Preprocess
			top.PriorBox
			top.Proposal

Table 13.15: Q

Onnx	Pytorch	Caffe	TOP
QuantizeLinear			top.QuantizeLinear

Table 13.16: R

Onnx	Pytorch	Caffe	TOP
Range	aten::reflection_pad1d	ReLU	top.RMSNorm
Reciprocal	aten::reflection_pad2d	ReLU6	top.ROIPooling
ReduceL1	aten::relu	Reorg	top.Range
ReduceL2	aten::remainder	Reshape	top.Reciprocal
ReduceMax	aten::repeat	RetinaFaceDetection	top.Reduce
ReduceMean	aten::replication_pad1d	Reverse	top.Relu
ReduceMin	aten::replication_pad2d	ROIPooling	top.Remainder
ReduceProd	aten::reshape		top.Repeate
ReduceSum	aten::rsqrt		top.Reshape
Relu	aten::rsub		top.RetinaFaceDetection
Reshape			top.Reverse
Resize			top.RoiAlign
RoiAlign			top.Round
Round			

Table 13.17: S

Onnx	Pytorch	Caffe	TOP
ScatterElements	aten::scatter	Scale	top.Scale
ScatterND	aten::select	ShuffleChannel	top.ScaleLut
Shape	aten::sigmoid	Sigmoid	top.ScatterElements
Sigmoid	aten::silu	Silence	top.ScatterND
Sin	aten::sin	Slice	top.Shape
Slice	aten::sinh	Softmax	top.ShuffleChannel
Softmax	aten::size	Split	top.SiLU
Softplus	aten::slice		top.Sigmoid
Split	aten::softmax		top.Sin
Sqrt	aten::softplus		top.Sinh
Squeeze	aten::split		top.Size
Sub	aten::split_with_sizes		top.Slice
Sum	aten::sqrt		top.SliceAxis
	aten::squeeze		top.Softmax
	aten::stack		top.SoftmaxBwd
	aten::sub		top.Softplus
	aten::sum		top.Softsign
			top.Split
			top.Sqrt
			top.Squeeze
			top.StridedSlice
			top.Sub
			top.SubConst
			top.SwapChannel
			top.SwapDimInner

Table 13.18: T

Onnx	Pytorch	Caffe	TOP
Tanh	aten::t	TanH	top.Tan
Tile	aten::tan	Tile	top.Tanh
TopK	aten::tanh		top.Tile
Transpose	aten::tile		top.TopK
	aten::to		top.Transpose
	aten::transpose		top.Tuple
	aten::type_as		

Table 13.19: U

Onnx	Pytorch	Caffe	TOP
Unsqueeze	aten::unsqueeze	Upsample	top.UnTuple
Upsample	aten::upsample_bilinear2d		top.Unpack
	aten::upsample_nearest2d		top.Unsqueeze
			top.Upsample

Table 13.20: V

Onnx	Pytorch	Caffe	TOP
	aten::view		top.View

Table 13.21: W

Onnx	Pytorch	Caffe	TOP
Where	aten::where		top.Weight
			top.WeightReorder
			top.Where

Table 13.22: Y

Onnx	Pytorch	Caffe	TOP
		YoloDetection	top.Yield
			top.YoloDetection

Table 13.23: Z

Onnx	Pytorch	Caffe	TOP
	aten::zeros		
	aten::zeros_like		

