第 5 章

CHAPTER 5

TPU-MLIR 开发技术分析

5.1 TPU-MLIR 简介

TPU-MLIR 现在支持 TFLite 与 ONNX 格式。这两种格式的模型可以直接转换为 TPU 可用的 bmodel。如果不是这两种格式呢？事实上，ONNX 提供了一套转换工具，可以将市场上的由主流深度学习框架编写的模型转换为 ONNX 格式，然后继续转换为 bmodel。主流深度学习框架模型转换为 ONNX 格式，然后下译到 MLIR 流程，如图 5-1 所示。

5.1.1 TPU-MLIR 的工作流程

模型转换过程中有时会出现精度损失。TPU-MLIR 支持 int8 对称与非对称量化，结合原开发企业的 Calibration 与 Tune 技术，大大地提高了性能，确保了模型的高精度。此外，TPU-MLIR 还使用了大量的图优化与算子分割优化技术，以确保模型的高效运行。

目前，TPU-MLIR 项目已应用于 SOPHGO 开发的最新一代人工智能处理器 BM1684x，该处理器的高性能 ARM 内核与相应的 SDK，可以实现深度学习算法的快速部署。实现极致性价比，打造下一代 AI 编译器。

如果神经网络模型想要支持 GPU 计算，则需要开发神经网络模型中的运算符的 GPU 版本。如果它需要适应 TPU，则需要为每个运营商开发一个 TPU 版本。在其他场景中，需要适应同一种计算处理器的不同型号的产品。如果每次都需要手动编译，则将是十分费时费力的。

AI 编译器就是为了解决上述问题而设计的。TPU-MLIR 的一系列自动优化工具，可以节省大量的手动优化时间，使在 CPU 上开发的模型能够顺利且经济高效地迁移到 TPU，以实现最终的性价比。

随着 Transformer 等神经网络结构的出现，新算子的数量不断增加。这些算子需要根据后端硬件的特点进行实现、优化与测试，以提高硬件的性能。这也导致运算符的复杂性更高，调整难度更大，并且并非所有运算符都可以由一个工具有效生成。整个人工智能编译器领域仍处于不断完善的状态。

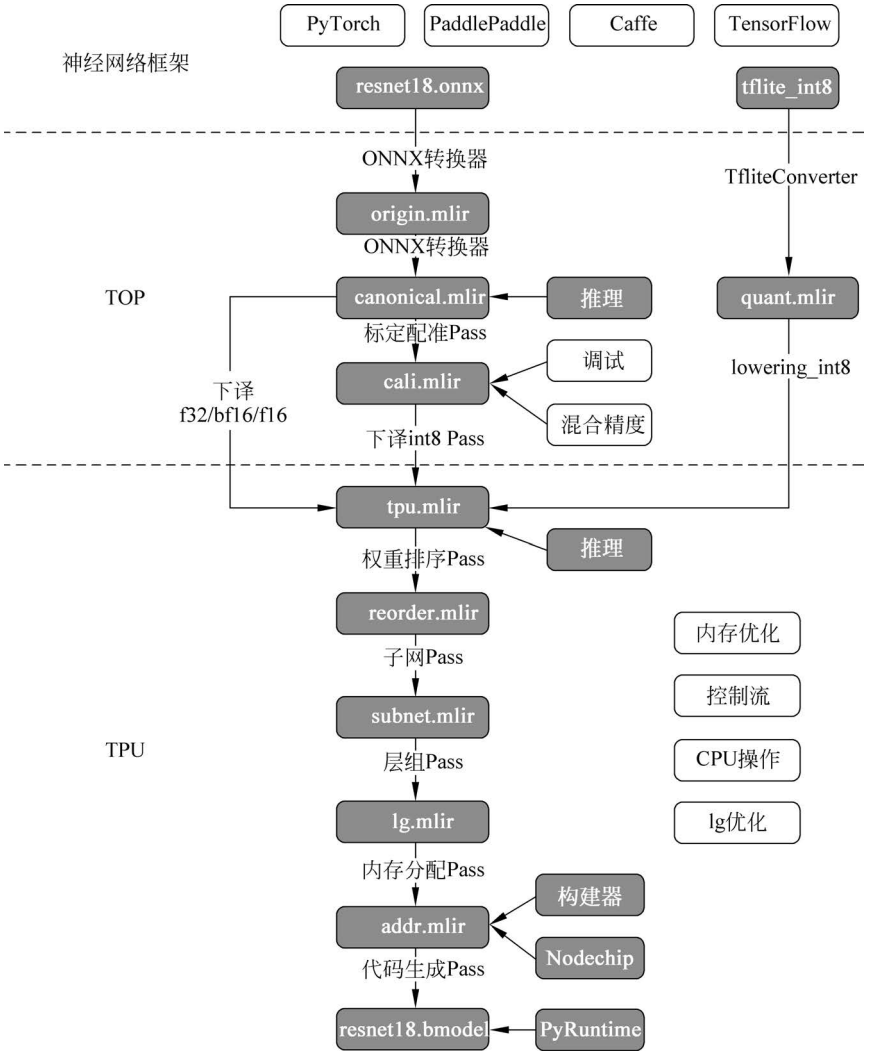


图 5-1　主流深度学习框架模型转换为 ONNX 格式，然后下译到 MLIR 流程

TPU-MLIR 还需要持续的研发投入、AI 处理器支持、代码生成性能优化、运行时调度优化等环节，这些环节还有很大的提升空间。TPU-MLIR 编译器的工作流程如图 5-2 所示。

5.1.2　TPU-MLIR 编译工程

TPU-MLIR 是 AI 芯片的 TPU 编译器工程。该工程提供了一套完整的工具链，其可以将不同框架下预训练的神经网络转换为可以在算能 TPU 上高效运算的文件 bmodel。TPU-MLIR 的整体架构如图 5-3 所示。

目前直接支持的框架有 ONNX、Caffe 与 TFLite，其他框架的模型需要转换成 ONNX 模型。

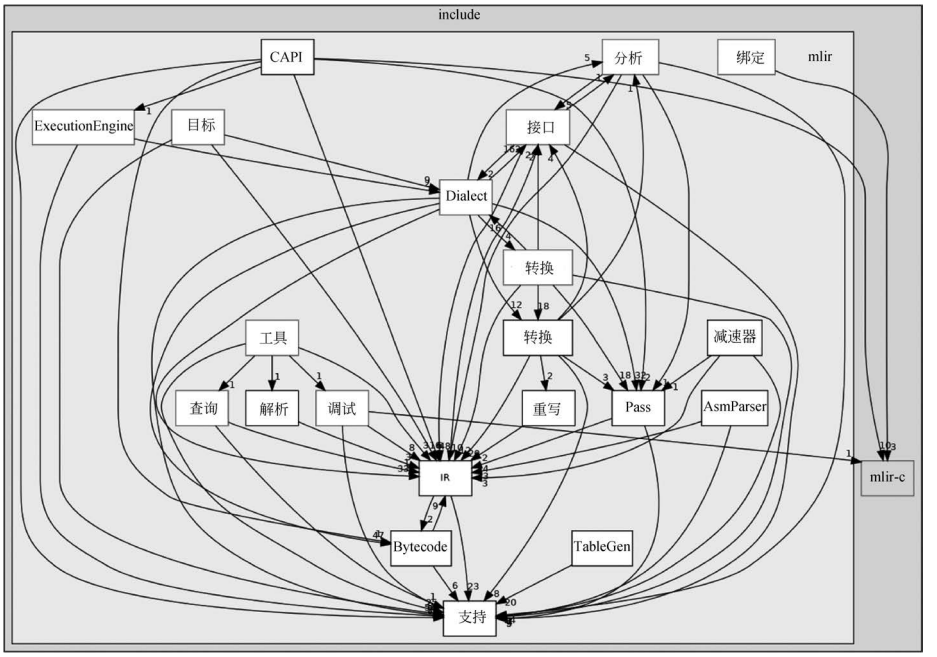


图 5-2　TPU-MLIR 编译器的工作流程

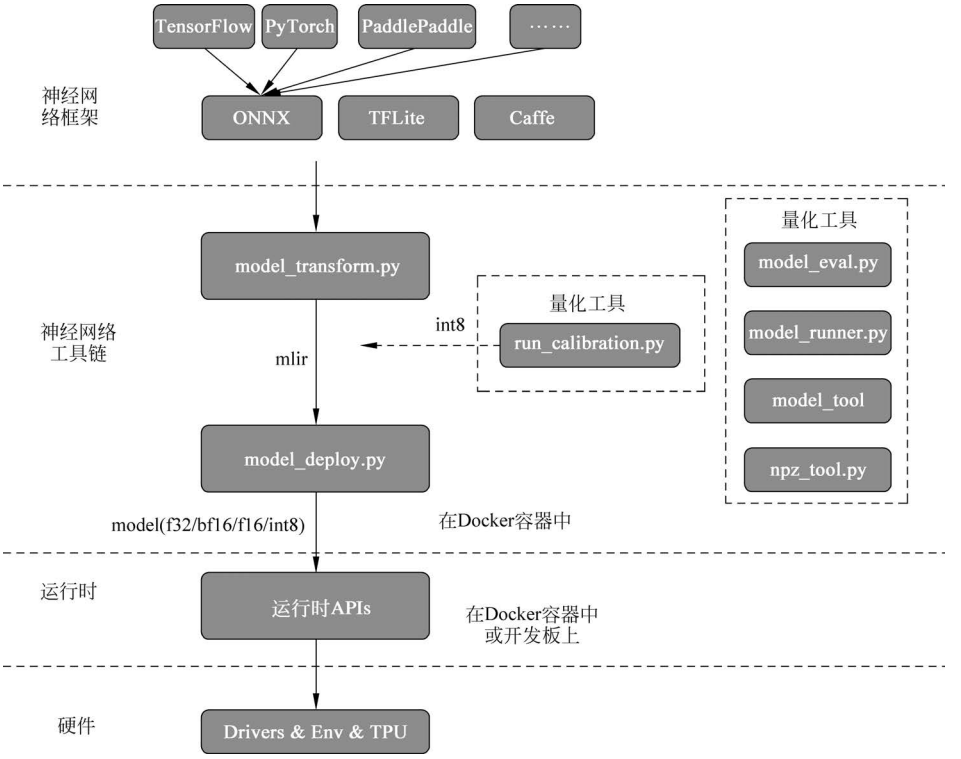


图 5-3　TPU-MLIR 的整体架构

转模型需要在指定的 Docker 执行，主要分为以下两步。

(1) 通过 model_transform.py 将原始模型转换成 MLIR 文件。

(2) 通过 model_deploy.py 将 MLIR 文件转换成 bmodel。

如果要转 int8 模型，则需要调用 run_calibration.py 生成校准表，然后传给 model_deploy.py。如果 int8 模型不满足精度需要，则可以调用 run_qtable.py 生成量化表，用来决定哪些层采用浮点计算，然后传给 model_deploy.py 生成混精度模型。

5.1.3 TPU-MLIR 开发环境配置

开发环境配置，代码在 Docker 中编译与运行。

1. 代码下载

从本书配套资源中下载 TPU-MLIR，复制该代码后，需要在 Docker 中编译。

2. Docker 配置

TPU-MLIR 在 Docker 环境开发，配置好 Docker 就可以编译与运行了。

从 DockerHub https://hub.docker.com/r/sophgo/tpuc_dev 下载所需的镜像，命令如下：

```
//第 5 章/docker_mirror.py
$ docker pull sophgo/tpuc_dev:latest
```

如果是首次使用 Docker，则可执行下述命令进行安装与配置(仅首次执行)，命令如下：

```
//第 5 章/docker_install.py
$ sudo apt install docker.io
$ sudo systemctl start docker
$ sudo systemctl enable docker
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
```

确保安装包在当前目录，然后在当前目录创建容器，命令如下：

```
//第 5 章/docker_container.py
$ docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:latest
#myname 只是举个名字的例子, 可指定成个人需要的容器的名称
```

注意，TPU-MLIR 工程在 Docker 中的路径应该是/workspace/tpu-MLIR。

3. ModelZoo(可选)

TPU-MLIR 中自带 yolov5s 模型，如果要运行其他模型，则需要下载 ModelZoo (https://github.com/sophgo/model-zoo)。下载后放在与 TPU-MLIR 同级目录，在 Docker 中的路径应该是/workspace/model-zoo。

操作工作脚本，在 Docker 的容器中，代码编译方式如下：

```
//第 5 章/mlir_model_transform.py
#代码编译
#在 Docker 的容器中代码编译方式如下
$ cd tpu - MLIR
$ source./envsetup.sh
$ ./build.sh
```

回归验证，代码如下：

```
//第 5 章/mlir_push.py
#本工程包含 yolov5s.onnx 模型，可以直接用来验证
$ pushd regression
$ ./run_model.sh yolov5s
$ popd
```

如果要验证更多网络，则需要依赖 model-zoo，回归时间比较久，代码如下（可选）：

```
//第 5 章/mlir_model_transform.py
#执行时间很长，该步骤可以跳过
$ pushd regression
$ ./run_all.sh
$ popd
```

用户的使用接口，包括转换模型的基本过程，与各类工具的使用方法。

基本操作过程是用 model_transform.py 将模型转换成 MLIR 文件，然后用 model_deploy.py 将 MLIR 转换成对应的 model，命令如下：

```
//第 5 章/mlir_deploy_transform.py
#对于 MLIR
$ model_transform.py \
    --model_name resnet \
    --model_def   resnet.onnx \
    --test_input resnet_in.npz \
    --test_result resnet_top_outputs.npz \
    --MLIR resnet.MLIR

#对于浮点模型
$ model_deploy.py \
   --MLIR resnet.MLIR \
   --quantize F32 \ #F16/BF16
   --chip bm1684x \
   --test_input resnet_in_f32.npz \
   --test_reference resnet_top_outputs.npz \
   --model resnet50_f32.bmodel
```

当用图片作为输入时，需要指定预处理信息，命令如下：

```
//第 5 章/mlir_model_name.py
$ model_transform.py \
    --model_name resnet \
    --model_def resnet.onnx \
```

```
    --input_shapes [[1,3,224,224]] \
    --mean 103.939,116.779,123.68 \
    --缩放 1.0,1.0,1.0 \
    --pixel_format bgr \
    --test_input cat.jpg \
    --test_result resnet_top_outputs.npz \
    --MLIR resnet.MLIR
```

当模型有多输入时，可以传入 1 个 NPZ 文件，或者按顺序传入多个 NPZ 文件，用逗号隔开，命令如下：

```
//第 5 章/mlir_model_somenet.py
$ model_transform.py \
    --model_name somenet \
    --model_def   somenet.onnx \
    --test_input somenet_in.npz \ #a.npy,b.npy,c.npy
    --test_result somenet_top_outputs.npz \
    --MLIR somenet.MLIR
```

如果需要转 int8 模型，则需要进行校准，命令如下：

```
//第 5 章/mlir_model_calibration.py
$ run_calibration.py somenet.MLIR \
    --dataset dataset \
    --input_num 100 \
    -o somenet_cali_table
```

传入校准表生成模型，命令如下：

```
//第 5 章/mlir_model_deploy_transform.py
$ model_deploy.py \
   --MLIR resnet.MLIR \
   --quantize INT8 \
   # --asymmetric \
   --calibration_table somenet_cali_table \
   --chip bm1684x \
   --test_input somenet_in_f32.npz \
   --test_reference somenet_top_outputs.npz \
   --tolerance 0.9,0.7 \
   --model somenet_int8.bmodel
```

当 int8 模型精度不满足业务要求时，可以尝试使用混合精度，先生成量化表，命令如下：

```
//第 5 章/mlir_model_transform.py
$ run_qtable.py somenet.MLIR \
    --dataset dataset \
    --calibration_table somenet_cali_table \
    --chip bm1684x \
    -o somenet_qtable
```

然后将量化表传入生成模型，命令如下：

```
//第 5 章/mlir_resnet_transform.py
$ model_deploy.py \
   -- MLIR resnet.MLIR \
   -- quantize INT8 \
   -- calibration_table somenet_cali_table \
   -- quantize_table somenet_qtable \
   -- chip bm1684x \
   -- model somenet_mix.bmodel
```

支持 TFLite 模型的转换，命令如下：

```
//第 5 章/mlir_model_transform.py
    #TFLite 转模型举例
 $ model_transform.py \
     -- model_name resnet50_tf \
     -- model_def ../resnet50_int8.tflite \
     -- input_shapes [[1,3,224,224]] \
     -- mean 103.939,116.779,123.68 \
     -- 缩放 1.0,1.0,1.0 \
     -- pixel_format bgr \
     -- test_input../image/dog.jpg \
     -- test_result resnet50_tf_top_outputs.npz \
     -- MLIR resnet50_tf.MLIR

$ model_deploy.py \
    -- MLIR resnet50_tf.MLIR \
    -- quantize INT8 \
    -- asymmetric \
    -- chip bm1684x \
    -- test_input resnet50_tf_in_f32.npz \
    -- test_reference resnet50_tf_top_outputs.npz \
    -- tolerance 0.95,0.85 \
    -- model resnet50_tf_1684x.bmodel
```

支持 Caffe 模型，命令如下：

```
//第 5 章/mlir_model_transform.py
#Caffe 转模型举例
$ model_transform.py \
    -- model_name resnet18_cf \
    -- model_def ../resnet18.prototxt \
    -- model_data../resnet18.caffemodel \
    -- input_shapes [[1,3,224,224]] \
    -- mean 104,117,123 \
    -- 缩放 1.0,1.0,1.0 \
    -- pixel_format bgr \
    -- test_input../image/dog.jpg \
    -- test_result resnet50_cf_top_outputs.npz \
-- MLIR resnet50_cf.MLIR
```

5.2 工具参数介绍

1. model_transform.py

用于将各种神经网络模型转换成 MLIR 文件，支持的参数见表 5-1。

表 5-1 model_transform 参数功能

参数名	是否必选	说明
model_name	是	指定模型名称
model_def	是	指定模型定义文件，例如`.onnx`、`.tflite`或`.prototxt`文件
model_data	否	指定模型权重文件，Caffe 模型需要，对应`.caffemodel`文件
input_shapes	否	指定输入的 shape，例如[[1,3,640,640]]；二维数组，可以支持多输入情况
resize_dims	否	原始图片需要 resize 之后的尺寸；如果不指定，则 resize 成模型的输入尺寸
keep_aspect_ratio	否	在 resize 时是否保持长宽比，默认值为 False；设置时会对不足部分补 0
mean	否	图像每个通道的均值，默认值为 0.0,0.0,0.0
scale	否	图片每个通道的比值，默认值为 1.0,1.0,1.0
pixel_format	否	图片类型，可以是 rgb、bgr、gray、rgbd 4 种类型
output_names	否	指定输出的名称，如果不指定，则用模型的输出；指定后用该指定名称作为输出
test_input	否	指定输入文件，用于验证，可以是图片、npy 或 npz；可以不指定，如果不指定，则不会进行正确性验证
test_result	否	指定验证后的输出文件
excepts	否	指定需要排除验证的网络层的名称，多个用逗号隔开
MLIR	是	指定输出的 MLIR 文件名称与路径
post_handle_type	否	将后处理融合到模型中，指定后处理类型，例如 yolo、ssd

转换成 MLIR 文件后会生成一个 ${model_name}_in_f32.npz 文件，该文件是后续模型的输入文件。

2. run_calibration.py

用少量的样本进行校准，得到网络模型的校准表，即每层 op 的 threshold/min/max。run_calibration 支持的参数，见表 5-2。

表 5-2 run_calibration 参数功能

参数名	是否必选	说明
(None)	是	指定 MLIR 文件
dataset	否	指定输入样本的目录，该路径存放对应的图片、npz 或 npy
data_list	否	指定样本列表，与 dataset 必须二选一
input_num	否	指定校准数量，如果为 0，则使用全部样本

续表

参　数　名	是否必选	说　　明
tune_num	否	指定微调样本数量,默认值为 10
histogram_bin_num	否	直方图 bin 数量,默认值为 2048
o	是	输出校准表文件

校准表的样板,操作信息如下:

```
//第 5 章/mlir_build_run.py
#生成的时间: 2024-04-8 10:00:59.743675
#直方图数目: 2048
#采样数: 100
#调试数目: 5
#
#op_name    threshold    min    max
images 1.0000080 0.0000000 1.0000080
122_conv56.4281803 -102.5830231 97.6811752
124_Mul 38.1586478 -0.2784646 97.6811752
125_conv56.1447888 -143.7053833 122.0844193
127_Mul 116.7435987 -0.2784646 122.0844193
128_conv16.4931355 -87.9204330 7.2770605
130_Mul 7.2720342 -0.2784646 7.2720342
......
```

操作信息分为 4 列:第 1 列是张量的名字;第 2 列是阈值(用于对称量化);第 3 列和第 4 列分别是 min 和 max,用于非对称量化。

3. run_qtable.py

使用 run_qtable.py 生成混合精度量化表,相关参数说明,见表 5-3。

表 5-3　run_qtable.py 参数功能

参　数　名	是否必选	说　　明
无	是	指定 MLIR 文件
dataset	否	指定输入样本的目录,该路径存放对应的图片、npz 或 npy
data_list	否	指定样本列表,与 dataset 必须二选一
calibration_table	是	输入校准表
chip	是	指定模型将要用到的平台,支持 bm1684x/bm1684/cv183x/cv182x/cv181x/cv180x
input_num	否	指定输入样本数量,默认用 10 个
loss_table	否	输出 Loss 表,默认为 full_loss_table.txt
o	是	输出混合精度量化表

混合精度量化表的样板,操作流程如下:

```
//第 5 章/mlir_mix_mode.py
#生成时间: 2024-04-07 21:35:47.981562
#采样数: 3
```

```
#全部 int8 损失: -39.03119206428528
#chip: bm1684x  mix_mode: F32
#
#op_name  quantize_mode
conv2_1/linear/bn F32
conv2_2/dwise/bn  F32
conv6_1/linear/bn F32
```

操作信息分为 2 列：第 1 列对应 layer 的名称，第 2 列对应量化模式。

同时会生成 Loss 表，默认为 full_loss_table.txt，代码如下：

```
//第 5 章/mlir_mix_mode_Layer_conv.py
#生成时间: 2024-04-07 22:30:31.912270
#采样数: 3
#全部 int8 损失: -39.03119206428528
#chip: bm1684x  mix_mode: F32
#
No.0: Layer:conv2_1/linear/bn Loss: -36.14866065979004
No.1: Layer:conv2_2/dwise/bn  Loss: -37.15774385134379
No.2: Layer:conv6_1/linear/bn Loss: -38.44639046986898
No.3: Layer:conv6_2/expand/bn Loss: -39.7430411974589
No.4: Layer:conv1/bn          Loss: -40.067259073257446
No.5: Layer:conv4_4/dwise/bn  Loss: -40.183939139048256
No.6: Layer:conv3_1/expand/bn Loss: -40.1949667930603
No.7: Layer:conv6_3/expand/bn Loss: -40.61786969502767
No.8: Layer:conv3_1/linear/bn Loss: -40.9286363919576
No.9: Layer:conv6_3/linear/bn Loss: -40.97952524820963
No.10: Layer: block_6_1         Loss: -40.987406969070435
No.11: Layer:conv4_3/dwise/bn  Loss: -41.18325670560201
No.12: Layer:conv6_3/dwise/bn  Loss: -41.193763415018715
No.13: Layer:conv4_2/dwise/bn  Loss: -41.2243926525116
......
```

它代表对应的 Layer 改成浮点计算后得到输出的 Loss。

4. model_deploy.py

将 MLIR 文件转换成相应的模型，参数说明，见表 5-4。

表 5-4　model_deploy 参数功能

参　数　名	是否必选	说　　明
MLIR	是	指定 MLIR 文件
chip	是	指定模型将要用到的平台，支持 bm1684x/bm1684/cv183x/cv182x/cv181x/cv180x
quantize	是	指定默认量化类型，支持 f32/f16/bf16/int8
quantize_table	否	指定混合精度量化表路径，如果没有指定，则按 quantize 类型量化，否则优先按量化表量化
calibration_table	否	指定校准表路径，当存在 int8 量化时需要校准表

续表

参 数 名	是否必选	说 明
tolerance	否	表示 MLIR 量化后的结果与 MLIR fp32 推理结果相似度的误差容忍度
test_input	否	指定输入文件用于验证，可以是图片、npy 或 npz；可以不指定，如果不指定，则不会进行正确性验证
test_reference	否	用于验证模型正确性的参考数据（使用 npz 格式），其为各算子的计算结果
excepts	否	指定需要排除验证的网络层的名称，多个用逗号隔开
model	是	指定输出的 model 文件名称与路径

5. model_runner.py

对模型进行推理，支持 bmodel/MLIR/onnx/tflite。

运行模型，命令如下：

```
//第 5 章/model_runner.py
$ model_runner.py \
    -- input sample_in_f32.npz \
    -- model sample.bmodel \
    -- output sample_output.npz
```

支持的参数，见表 5-5。

表 5-5 model_runner 参数功能

参 数 名	是 否 必 选	说 明
input	是	指定模型输入，npz 文件
model	是	指定模型文件，支持 bmodel/MLIR/ONNX/TFLite
dump_all_tensors	否	开启后将导出所有的结果，包括中间张量的结果

6. npz_tool.py

npz 在 TPU-MLIR 工程中会大量用到，包括输入/输出的结果等。npz_tool.py 用于处理 npz 文件。

执行 npz_tool，命令如下：

```
//第 5 章/sample_out.py
#查看 sample_out.npz 中输出的数据
$ npz_tool.py dump sample_out.npz 输出
```

支持的功能，见表 5-6。

表 5-6 npz_tool 功能

功 能	描 述
dump	得到 npz 的所有张量信息
compare	比较两个 npz 文件的差异
to_dat	将 npz 导出为 dat 文件，连续的二进制存储

7. visual.py

量化网络如果遇到精度对比不过或比较差，则可以使用此工具逐层可视化，比较浮点网络与量化后网络的不同，方便进行定位与手动调整。

运行 visual.py，执行的命令如下：

```
//第 5 章/fp32_MLIR f32.MLIR
#以使用 9999 端口为例
$ visual.py -- fp32_MLIR f32.MLIR -- quant_MLIR quant.MLIR -- input top_input_f32.npz --
port 9999
```

支持的功能，见表 5-7。

表 5-7 visual 功能

功 能	描 述
f32_MLIR	fp32 网络 MLIR 文件
quant_MLIR	量化后网络 MLIR 文件
input	测试输入数据，可以是图像文件或者 npz 文件
port	使用的 TCP 端口，默认为 10000，需要在启动 Docker 时映射至系统端口
manual_run	启动后是否自动进行网络推理比较，默认值为 False，表示会自动推理比较

5.3 整体设计

5.3.1 TPU-MLIR 分层

TPU-MLIR 将网络模型的编译过程分两层处理，包括 TOP 方言与 TPU 方言：

(1) TOP 方言：与芯片无关层，包括图优化、量化、推理等。

(2) TPU 方言：与芯片相关层，包括权重重排、算子切分、地址分配、推理等。

整体的流程(TPU-MLIR 整体流程)，如图 5-4 所示。通过 Pass 将模型逐渐转换成最终的指令，这里具体说明，TOP 层与 TPU 层的每个 Pass 有什么功能。

5.3.2 构建 Pass

构建 Pass，包括以下几个模块。

(1) 规范化：与具体 OP 有关的图优化，例如 ReLU 合并到卷积、shape 合并等。

(2) 校准：按照校准表，给每个 OP 插入 min 与 max，用于后续量化；如果对应对称量化，则插入 threshold。

(3) 下译：将 OP 根据类型下译到 TPU 层，支持的类型有 f32/f16/bf16/int8 对称/int8 非对称。

(4) 规范化：与具体 OP 有关的图优化，例如连续 Requant 的合并等。

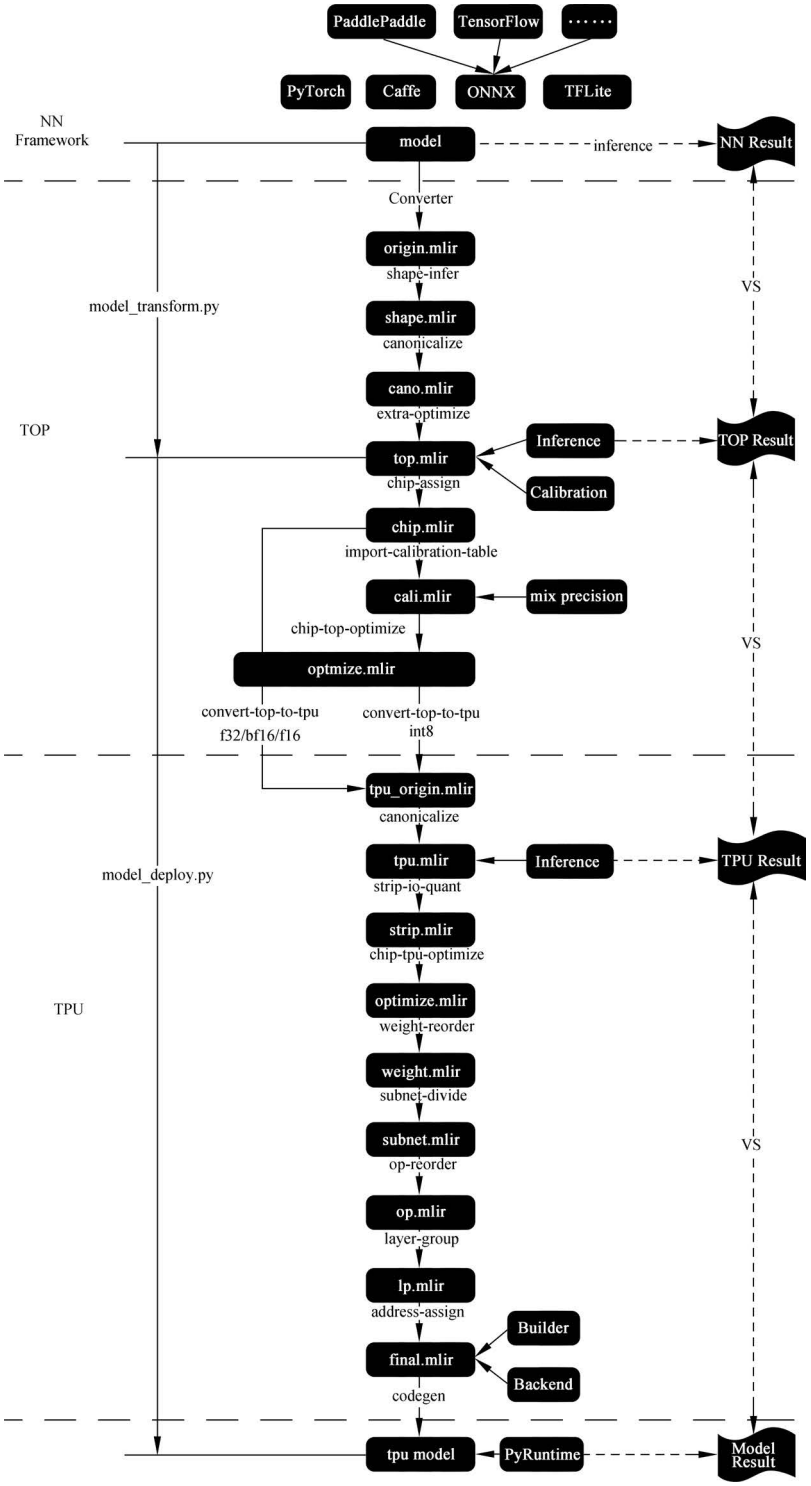


图 5-4　TPU-MLIR 整体流程：从深度学习框架到 TOP，再到 TPU

(5) 权重重新排序：根据芯片特征，对个别 OP 的权重进行重新排列，例如卷积的 filter 与 bias。

(6) 子网：将网络按照 TPU/CPU 切分成不同的子网络，如果所有的算子都是 TPU，则子网络只有一个。

(7) 图层组：对网络进行切分，使尽可能多的 OP 在片内内存中连续计算。

(8) MemAssign：给需要全局内存的 OP 分配地址。

(9) CodeGen：用 Builder 模块采用平面缓冲区格式生成最终的模型。

(10) 前端转换：以 ONNX 模型为例，介绍模型/算子在本工程中的前端转换流程。

5.3.3 TPU-MLIR 主要工作模块

前端主要负责将原始模型转换为 TOP 层(芯片无关层)、MLIR 模型的工作(不包含规范化部分，因此生成的文件名为 * _origin. MLIR)，这个过程会根据原始模型与运行 model_transform. py 时输入的参数逐一创建并添加对应的算子(OP)，最终生成 MLIR 文件与保存权重的 npz 文件。

(1) 前提(Prereq)：TOP 层算子定义，该部分内容保存在 TopOps. td 文件中。

(2) 输入：输入原始 ONNX 模型与参数(主要是预处理参数)。

(3) 初始化 Onnxconverter(load_onnx_model + initMLIRImporter)。

① load_onnx_model 部分主要用于对模型进行精简化，根据参数中的 output_names 截取模型，并提取精简后模型的相关信息。

② init_MLIRImporter 部分主要用于生成初始的 MLIR 文本。

(4) generate_MLIR：依次创建输入 OP，模型中间节点 OP 及返回 OP，并将其补充到 MLIR 文本中 (如果该 OP 带有权重，则会创建特定权重 OP)。

(5) 输出：将精简后的模型保存为 * _opt. onnx 文件，生成. prototxt 文件保存除权重外的模型信息，然后将生成的文本转换为 str 并保存为. mlir 文件。

(6) 将模型权重(tensors)保存为. npz 文件。

前端转换的工作流程，如图 5-5 所示。

构建输入操作需要 input_names、每个输入对应的 index 及预处理参数(若输入为图像)。

转换节点操作需要从操作数获取该节点的输入操作(前一个已经 build 或 convert 好的算子)，然后从 shapes 中获取 output_shape。

从 ONNX 节点中提取的属性会通过 MLIRImporter 设定为与 TopOps. td 定义一一对应的属性。

构建返回操作需要依照 output_names，从操作数获取相应的操作。每创建或者转换一个算子都会执行一次插入操作，将算子插入 MLIR 文本中，使最终生成的文本能从头到尾与原 ONNX 模型一一对应。

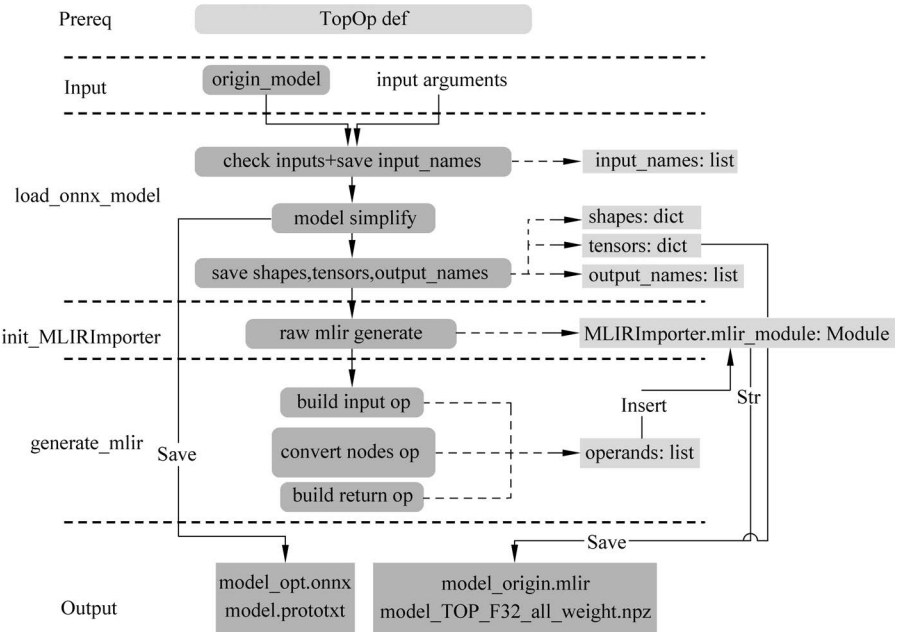


图 5-5　TPU-MLIR 前端转换流程

5.3.4　算子转换样例

以卷积算子为例，将单卷积算子的 ONNX 模型转换为 TOP MLIR，用 Netron 工具查看原模型结构，如图 5-6 所示。

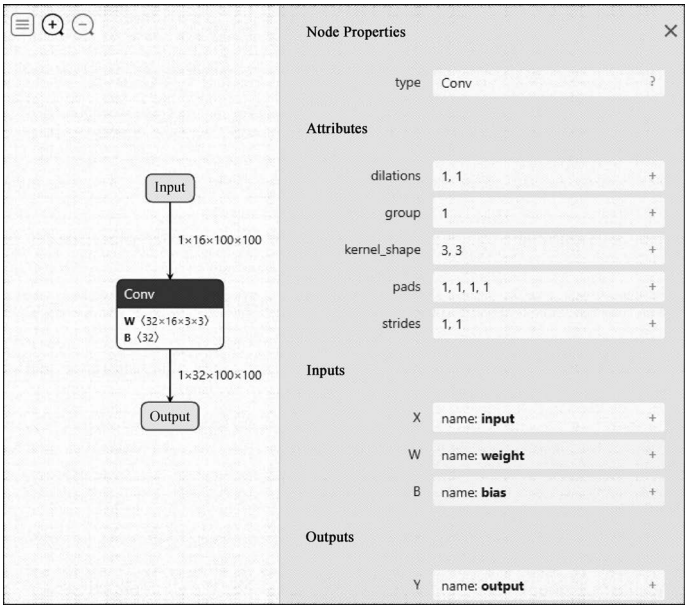


图 5-6　使用 Netron 工具查看模型文件结构

1. 算子定义

在 TopOps.td 文件中定义 Top.conv 卷积算子，如图 5-7 所示。

include > tpu_mlir > Dialect > Top > IR > TopOps.td

```
157  def Top_ConvOp: Top_Op<"Conv", [SupportFuseRelu]> {
158    let summary = " 卷积算子";
159
160    let description = [{
161      在最简单的情况下，具有输入大小的层的输出值
162      .......
163    }];
164
165    let arguments = (ins
166      AnyTensor:$input,
167      AnyTensor:$filter,
168      AnyTensorOrNone:$bias,
169      I64ArrayAttr:$kernel_shape,
170      I64ArrayAttr:$strides,
171      I64ArrayAttr:$pads, // top,left,bottom,right
172      DefaultValuedAttr<I64Attr, "1">:$group,
173      OptionalAttr<I64ArrayAttr>:$dilations,
174      OptionalAttr<I64ArrayAttr>:$inserts,
175      DefaultValuedAttr<BoolAttr, "false">:$do_relu,
176      OptionalAttr<F64Attr>:$upper_limit,
177      StrAttr:$name
178    );
179
180    let results = (outs AnyTensor:$output);
181    let extraClassDeclaration = [{
182      void parseParam(int64_t &n, int64_t &ic, int64_t &ih, int64_t &iw, int64_t &oc,
183                      int64_t &oh, int64_t &ow, int64_t &g, int64_t &kh, int64_t &kw, int64_t &
                        ins_h,
184                      int64_t &ins_w, int64_t &sh, int64_t &sw, int64_t &pt, int64_t &pb,
                        int64_t &pl,
185                      int64_t &pr, int64_t &dh, int64_t &dw, bool &is_dw, bool &with_bias, bool &
                        do_relu,
186                      float &relu_upper_limit);
187    }];
188  }
```

图 5-7　从 include/tpu_mlir/Dialect/Top/IR/TopOps.td 查看 Top_ConvOp

分析 Top_ConvOp 内部模块功能参数，如图 5-8 所示。

2. 初始化 ONNX 转换器

load_onnx_model 包括以下功能模块：

(1) 由于使用的是最简模型，所以生成的 conv _opt.onnx 模型与原模型相同。

(2) input_names 保存了卷积算子的输入名。

(3) 张量中保存了卷积算子的权重与 bias。

(4) shapes 中保存了卷积算子的输入与输出 shape。

(5) output_names 中保存了卷积算子的输出名。

init_MLIRImporter 初始化：根据 input_names 与 output_names，从 shapes 中获取了对应的 input_shape 与 output_shape，加上 model_name，生成了初始的 MLIR 文本 MLIRImporter.MLIR_module，如图 5-9 所示。

3. generate_MLIR

生成 MLIR 包括以下步骤：

(1) 构建输入操作，生成的 Top.inputOp 会被插入 MLIRImporter.MLIR_module 中。

图 5-8　分析 Top_ConvOp 内部模块功能参数

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

图 5-9　分析模块属性内部参数

（2）根据 node.op_type（卷积）调用 convert_conv_op()，该函数中会调用 MLIRImporter.create_conv_op 来创建 convOp，而 create 函数需要的参数如下。

① 输入操作：从(单卷积模型)可知，卷积算子的 inputs 一共包含输入、权重与 bias，输入 OP 已被创建好，权重与 bias 的 OP，则通过 getWeightOp()创建。

② output_shape：利用 onnx_node.name 从 shapes 中获取卷积算子的 outputshape。

③ Attributes：从 ONNX 卷积算子中获取如单卷积模型中的 attributes。

在 create 函数里 Top.conv 算子的 attributes 会根据(卷积算子定义)中的定义来设定。Top.convOp 创建后会被插入 MLIR 文本中。

(3) 根据 output_names 从 operands 中获取相应的 OP，创建 return_op 并插入 MLIR 文本中。到此为止，生成的 MLIR 文本如图 5-10 所示。

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

图 5-10　分析 Conv2d_origin.mlir 内部参数

(4) 将 MLIR 文本保存为 conv_origin.MLIR，将 tensors 中的权重保存为 conv_TOP_F32_all_weight.npz。

5.4　神经网络的量化与训练

5.4.1　量化技术概述

介绍 TPU-MLIR 的量化设计，重点介绍该论文在实际量化中的应用。

int8 量化分为非对称量化与对称量化。对称量化是非对称量化的一个特例，通常对称量化的性能会优于非对称量化，而精度上非对称量化更优。

1. 非对称量化

非对称量化其实就是把[min，max]范围内的数值，定点到[－128，127]或者[0，255]区间，如图 5-11 所示。

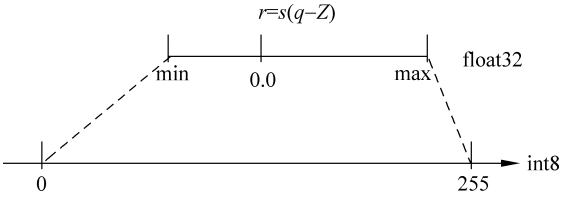


图 5-11　非对称量化算法理论

从 int8 到 float 的量化，可以用以下公式进行表示：

$$\begin{aligned} r &= S(q - Z) \\ S &= \frac{\max - \min}{q\max - q\min} \\ Z &= \mathrm{Round}\left(-\frac{\min}{S} + q\min\right) \end{aligned} \tag{5-1}$$

其中，r 是真实的值，float 类型；q 是量化后的值，int8 或者 uint8 类型；S 表示缩放，float 类型；Z 是零点，int8 类型。

当量化到 int8 时，$q\max=127$，$q\min=-128$；当量化到 uint8 时，$q\max=255$，$q\min=0$。反过来从 float 到 int8 的量化，用下式表示：

$$q = \frac{r}{S} + Z \tag{5-2}$$

2. 对称量化

对称量化是非对称量化 $Z=0$ 时的特例，公式如下。

$$\begin{aligned} \text{i8\_value} &= \text{f32\_value} \times \frac{128}{\text{threshold}} \\ \text{f32\_value} &= \text{i8\_value} \times \frac{\text{threshold}}{128} \end{aligned} \tag{5-3}$$

在式(5-3)中，解析如下：

(1) threshold 是阈值，可以理解为张量的范围是[−threshold，threshold]。

(2) 这里 $S=\text{threshold}/128$，通常是激活函数情况。

(3) 对于权重，一般 $S=\text{threshold}/127$。

(4) 对于 uint8，张量范围是[0，threshold]，此时 $S=\text{threshold}/255.0$。

3. 缩放转换

取值是一个非负数 $M=2^{-n}M_0$，其中 M_0 取值[0.5，1]，n 是一个非负数。

换个表述来讲，也就是浮点数缩放，可以转换成乘法与 rshift，用以下公式表示：

$$\mathrm{Scale} = \frac{\mathrm{Multiplier}}{2^{\mathrm{rshift}}} \tag{5-4}$$

下面举例说明，如何使用式(5-4)：

$$\begin{aligned} y &= x \times 0.1234 \\ &\Rightarrow y = x \times 0.9872 \times 2^{-3} \\ &\Rightarrow y = x \times (0.9872 \times 2^{31}) \times 2^{-34} \\ &\Rightarrow y = x \times \frac{2119995857}{1 \ll 34} \\ &\Rightarrow y = (x \times 2119995857) \gg 34 \end{aligned} \tag{5-5}$$

其中，乘法支持的位数越高，就越接近缩放，但是性能会越差。一般芯片会用 32 位或 8 位的乘法。

4. 量化推导

可以用量化公式对不同的 OP 进行量化推导，得到其对应的 int8 计算方式。对称与非对称都用在激活函数上，对于权重一般只用对称量化。

1）卷积

卷积的表达式如下：

$$Y = X_{(n,ic,ih,iw)} \times W_{(oc,ic,kh,kw)} + B_{(1,oc,1,1)} \tag{5-6}$$

在式(5-6)中，代入 int8 量化公式，用以下公式推导：

$$\begin{aligned}
&\text{float}: Y = X \times W + B \\
&\text{step0} \Rightarrow S_y(q_y - Z_y) = S_x(q_x - Z_x) \times S_w q_w + B \\
&\text{step1} \Rightarrow q_y - Z_y = S_1(q_x - Z_x) \times q_w + B_1 \\
&\text{step2} \Rightarrow q_y - Z_y = S_1 q_x \times q_w + B_2 \\
&\text{step3} \Rightarrow q_y = S_3(q_x \times q_w + B_3) + Z_y \\
&\text{step4} \Rightarrow q_y = S_3(q_x \times q_w + b_{i32}) * M_{i32} \gg \text{rshift}_{i8} + Z_y
\end{aligned} \tag{5-7}$$

非对称量化需要特别注意的是，Pad 需要填入 Z_x。

对称量化时，Pad 填入 0，上述推导中 Z_x 与 Z_y 皆为 0。

在 PerAxis(或称 PerChannel)量化时会取 Filter 的每个 OC 做量化，推导公式不变，但是会有 OC 个乘法、rshift。

2）内积

推导方式与卷积相同。

3）加法

加法的表达式如下：

$$Y = A + B \tag{5-8}$$

将式(5-8)代入 int8 量化公式，用以下公式推导：

$$\begin{aligned}
&\text{float}: Y = A + B \\
&\text{step0} \Rightarrow S_y(q_y - Z_y) = S_a(q_a - Z_a) + S_b(q_b - Z_b) \\
&\text{step1(对称)} \Rightarrow q_y = (q_a * M_a + q_b * M_b)_{i16} \gg \text{rshift}_{i8} \\
&\text{step2(非对称)} \Rightarrow q_y = \text{requant}(\text{dequant}(q_a) + \text{dequant}(q_b))
\end{aligned} \tag{5-9}$$

在式(5-9)中，加法最终如何用 TPU 实现，与 TPU 具体的指令有关。

这里对称提供的方式是用 int16 做中间缓存。

在网络中，输入 A、B 已经是量化后的结果 q_a、q_b，因此非对称是先反量化成 float，进行加法运算后再量化成 int8。

5. 平均池化

平均池化的表达式如下：

$$Y_i = \frac{\sum_{j=0}^{k}(X_j)}{k}，\text{其中 } k = kh \times kw \tag{5-10}$$

将式(5-10)代入 int8 量化公式，用以下公式推导：

$$\begin{aligned}
&\text{float}: Y_i = \frac{\sum_{j=0}^{k}(X_j)}{k} \\
&\text{step0}: => S_y(y_i - Z_y) = \frac{S_x\sum_{j=0}^{k}(x_j - Z_x) + Z_y}{k} \\
&\text{step1}: => y_i = \frac{S_x}{S_y k}\sum_{j=0}^{k}(x_j - Z_x) + Z_y \\
&\text{step2}: => y_i = \frac{S_x}{S_y k}\sum_{j=0}^{k}(x_j) - \left(Z_y - \frac{S_x}{S_y}Z_x\right) \\
&\text{step3}: => y_i = \left(\text{Scale}_{f32}\sum_{j=0}^{k}(x_j) - \text{Offset}_{f32}\right)_{i8}
\end{aligned} \tag{5-11}$$

在式(5-11)中，

$$\text{Scale}_{f32} = \frac{S_x}{S_y k}, \quad \text{Offset}_{f32} = Z_y - \frac{S_x}{S_y}Z_x \tag{5-12}$$

6. LeakyReLU

LeakyReLU 的表达式如下：

$$Y = \begin{cases} X, & \text{当 } X \geqslant 0 \text{ 时} \\ \alpha X, & \text{当 } X < 0 \text{ 时} \end{cases} \tag{5-13}$$

将式(5-13)代入 int8 量化公式，用以下公式推导：

$$\begin{cases} \text{float}: Y = \begin{cases} X, & \text{当 } X \geqslant 0 \text{ 时} \\ \alpha X, & \text{当 } X < 0 \text{ 时} \end{cases} \\ \text{step0}: =>= S_y(q_y - Z_y) = \begin{cases} S_y(q_x - Z_x), & \text{当 } q_x \geqslant 0 \text{ 时} \\ \alpha S_x(q_x - Z_x), & \text{当 } q_x < 0 \text{ 时} \end{cases} \\ \text{step1}: =>= q_y = \begin{cases} \dfrac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{当 } q_x \geqslant 0 \text{ 时} \\ \alpha\dfrac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{当 } q_x < 0 \text{ 时} \end{cases} \end{cases} \tag{5-14}$$

对称量化时，

$$S_y = \frac{\text{threshold}_y}{128}, \quad S_x = \frac{\text{threshold}_y}{128} \tag{5-15}$$

非对称量化时，

$$S_y = \frac{\max_y - \min_y}{255}, \quad S_x = \frac{\max_x - \min_x}{255} \tag{5-16}$$

在式(5-16)中,通过向后校准操作后,

$$\max_y = \max_x, \quad \min_y = \min_x, \quad \text{threshold}_y = \text{threshold}_x \tag{5-17}$$

在式(5-17)中,

$$S_x / S_y = 1 \tag{5-18}$$

继续用以下公式推导:

$$\begin{cases} \text{step2}: => = q_y = \begin{cases} (q_x - Z_x) + Z_y, & \text{当 } q_x \geqslant 0 \text{ 时} \\ \alpha(q_x - Z_x) + Z_y, & \text{当 } q_x < 0 \text{ 时} \end{cases} \\ \text{step3}: => = q_y = \begin{cases} q_x - Z_x + Z_y, & \text{当 } q_x \geqslant 0 \text{ 时} \\ M_{i8} \gg \text{rshift}_{i8}(q_x - Z_x) + Z_y, & \text{当 } q_x < 0 \text{ 时} \end{cases} \end{cases} \tag{5-19}$$

在式(5-19)中,当为对称量化时,Z_x 与 Z_y 均为 0。

7. Pad 填充

Pad 的表达式如下:

$$Y = \begin{cases} \text{X}, & \text{原始位置} \\ \text{value}, & \text{填充位置} \end{cases} \tag{5-20}$$

将式(5-20)代入 int8 量化公式,用以下公式推导:

$$\begin{cases} \text{float}: Y = \begin{cases} \text{X}, & \text{原始位置} \\ \text{value}, & \text{填充位置} \end{cases} \\ \text{step0}: => S_y(q_y - Z_y) = \begin{cases} S_x(q_x - Z_x), & \text{原始位置} \\ \text{value}, & \text{填充位置} \end{cases} \\ \text{step1}: => q_y = \begin{cases} \dfrac{S_x}{S_y}(q_x - Z_x) + Z_y, & \text{原始位置} \\ \dfrac{\text{value}}{S_y}, & \text{填充位置} \end{cases} \end{cases} \tag{5-21}$$

在式(5-21)中,通过前向校准操作后,用以下公式表示,

$$\max_y = \max_x, \quad \min_y = \min_x, \quad \text{threshold}_y = \text{threshold}_x \tag{5-22}$$

在式(5-22)中,$S_x / S_y = 1$。

继续用以下公式推导:

$$\text{step2}: => q_y = \begin{cases} (q_x - Z_x) + Z_y, & \text{原始位置} \\ \dfrac{\text{value}}{S_y}, & \text{填充位置} \end{cases} \tag{5-23}$$

在式(5-23)中,当进行对称量化时,Z_x 与 Z_y 均为 0,pad 填入 round(value/S_y),当进

行非对称量化时，pad 填入 round(value/S_y+Z_y)。

8. PReLU

PReLU 的表达式，用以下公式表示：

$$Y_i=\begin{cases}X_i, & \text{当 } X_i\geqslant 0 \text{ 时}\\ \alpha_i X_i, & \text{当 } X_i<0 \text{ 时}\end{cases} \tag{5-24}$$

将式(5-24)代入 int8 量化公式，用以下公式表示：

$$\begin{cases}\text{float}: Y_i=\begin{cases}X_i, & \text{当 } X_i\geqslant 0 \text{ 时}\\ \alpha_i X_i, & \text{当 } X_i<0 \text{ 时}\end{cases}\\ \text{step0}: => S_y(y_i-Z_y)=\begin{cases}S_x(x_i-Z_x), & \text{当 } x_i\geqslant 0 \text{ 时}\\ S_\alpha q_{\alpha_i} S_x(x_i-Z_x), & \text{当 } x_i<0 \text{ 时}\end{cases}\\ \text{step1}: => y_i=\begin{cases}\dfrac{S_x}{S_y}(x_i-Z_x)+Z_y, & \text{当 } x_i\geqslant 0 \text{ 时}\\ S_\alpha q_{\alpha_i}\dfrac{S_i}{S_y}(x_i-Z_x)+Z_y, & \text{当 } x_i<0 \text{ 时}\end{cases}\end{cases} \tag{5-25}$$

在式(5-25)中，通过向后校准操作后，用以下公式表示：

$$\max_y=\max_x,\quad \min_y=\min_x,\quad \text{threshold}_y=\text{threshold}_x \tag{5-26}$$

在式(5-26)中，$S_x/S_y=1$。

继续用以下公式推导：

$$\begin{cases}\text{step2}: =>=y_i=\begin{cases}(x_i-Z_x)+Z_y, & \text{当 } x_i\geqslant 0 \text{ 时}\\ S_\alpha q_i(x_i-Z_x)+Z_y, & \text{当 } x_i<0 \text{ 时}\end{cases}\\ \text{step3}: =>=y_i=\begin{cases}(x_i-Z_x)+Z_y, & \text{当 } x_i\geqslant 0 \text{ 时}\\ q_\alpha * M_{i8}(x_i-Z_x)+Z_y\gg \text{rshift}_{i8}, & \text{当 } x_i<0 \text{ 时}\end{cases}\end{cases} \tag{5-27}$$

在式(5-27)中，一共有多个乘法与 1 个 rshift。当进行对称量化时，Z_x 与 Z_y 均为 0。

5.4.2　校准技术

1. 总体介绍

所谓校准，也就是用真实场景数据来校准出恰当的量化参数，为何需要校准？当对激活进行非对称量化时，需要预先知道其总体的动态范围，即 min-max 值，对激活进行对称量化时，需要预先使用合适的量化门限算法，在激活总体数据分布的基础上经计算得到其量化门限，而一般训练输出的模型是不带有激活这些数据统计信息的，因此这两者都要依赖于在一个微型的训练集子集上进行推理，收集各个输入的各层输出激活，汇总得到总体 min-max 及数据点分布直方图，并根据 KLD 等算法得到合适的对称量化门限，最后会启用自动调谐

算法，使用各 int8 层输出激活与 fp32 激活的欧氏距离，来对这些 int8 层的输入激活量化门限进行调优；上述过程整合在一起，统一执行，最后将各个 OP 的优化后的门限与 min-max 值，输出到一个量化参数文本文件中，后续运行 model_deploy.py 文件时，就可使用这个参数文件来进行后续的 int8 量化，总体量化过程如图 5-12 所示。

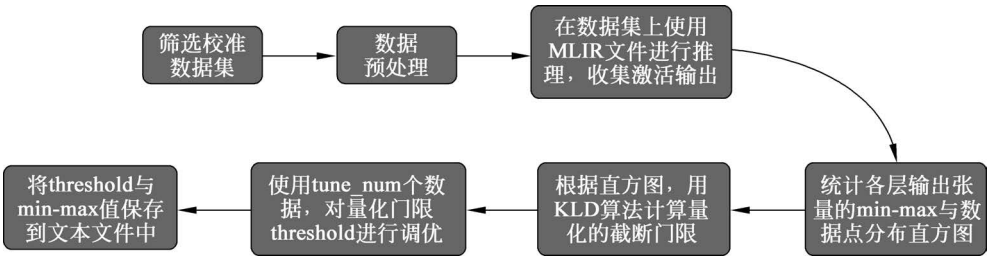


图 5-12　int8 总体量化工程

校准最终输出的量化参数文件样例，如图 5-13 所示。

```
#步骤5：调整后转储阈值表
tuned_threshold_list = []
with open(self.args.calibration_table, 'w') as f:
    f.write("# genetated time: {}\n".format(datetime.datetime.now()))
    f.write("# histogram number: {}\n".format(self.histogram_bin_num))
    f.write("# sample number: {}\n".format(self.num_samples))
    f.write("# tune number: {}\n###\n".format(self.args.tune_num))
    f.write("# op_name    threshold    min    max\n")
```

图 5-13　调整后转储阈值表

2. 校准数据筛选及预处理

在训练集中挑选约 100～200 张覆盖各个典型场景风格的图片来进行校准，采用类似训练数据清洗的方式，要排除一些异常样例。

3. 输入格式及预处理

输入格式，见表 5-8。

表 5-8　输入格式

格　　式	描　　述
原始图片	对于 CNN 类图片输入网络，支持直接输入图片，要求在前面生成 MLIR 文件时，model_transform.py 命令要指定与训练时完全一致的图片预处理参数
npz 或 npy 文件	对于非图片输入或图片预处理类型较复杂的 TPU-MLIR 暂不支持的情形，建议编写特定脚本，将完成预处理后的输入数据保存到 npz、npy 文件中（npz 文件是多个输入张量，按字典的方式打包在一起，npy 文件是 1 个文件包含 1 个张量），run_calibration.py 支持直接导入 npz、npy 文件

在 run_calibration.py 文件中调用 MLIR 文件进行推理时，无须再指定校准图片的预处理参数。

参数描述方式见表 5-9。

表 5-9　参数描述方式

方　式	描　述
dataset	对于单输入网络，配置输入的各张图片或已预处理的输入 npy、npz 文件（无顺序要求）；对于多输入网络，配置各个样本的已预处理的 npz 文件
data_list	将各个样本的图片文件地址、npz 文件地址或者 npy 文件地址，一行放一个样本，放置在文本文件中，若网络有多个输入文件，文件间通过逗号分隔（注意 npz 文件应该只有一个输入地址）

图片库数据列表示例，包括存储路径与图片命名，如图 5-14 所示。

```
/data/cali_100pics/n00.jpeg
/data/cali_100pics/n01.jpeg
/data/cali_100pics/n02.jpeg
/data/cali_100pics/n03.jpeg
/data/cali_100pics/n04.jpeg
/data/cali_100pics/n05.jpeg
```

图 5-14　图片库存储路径与图片命名

5.4.3　算法实现

1. KLD 算法

TPU-MLIR 实现的 KLD 算法参考了 tensorRT 的实现，本质上是将 abs(fp32_tensor) 这个波形（用 2048 个 fp32 bin 的直方图表示）截掉一些高位的离群点后（截取的位置固定在 128bin、256bin...一直到 2048bin）得到 fp32 参考概率分布 P，这个 fp32 波形若用 128 个等级的 int8 类型来表达，则将相邻的多个 bin（例如 256bin 是相邻的两个 fp32bin）合并成 1 个 int8 值等级计算分布概率后，再扩展到相同的 bin 数，以保证和 P 具有相同的长度，最终得到量化后 int8 值的概率分布 Q，计算 P 和 Q 的 KL 散度，在一个循环中，分别对 128bin、256bin、……、2048bin 这些截取位置计算 KL 散度，找出具有最小散度的截取位置，这说明在这里截取，能用 int8 这 128 个量化等级最好地模拟 fp32 的概率分布，故量化门限设在这里是最合适的。

利用 KLD 计算 int8 量化阈值，代码如下：

```
//第 5 章/mlir_distribution.py
for i in range(128,2048,128):
          Outliers_num = sum(bin[i], bin[i + 1], … , bin[2047])
          Fp32_distribution = [bin[0], bin[1], … , bin[i - 1] + Outliers_num]
          Fp32_distribution/ = sum(Fp32_distribution)
          int8_distribution = quantize [bin[0], bin[1], … , bin[i]]
into 128 quant level
          expand int8_distribution to i bins
          int8_distribution / = sum(int8_distribution)
          kld[i] = KLD(Fp32_distribution, int8_distribution)
        end for

        find i which kld[i] is minimal
        int8 quantize threshold = (i + 0.5) * fp32 absmax/2048
```

2. 自动调谐算法

从 KLD 算法的实际表现来看，其候选门限相对较粗，没有考虑到不同业务的特性，例如，对于目标检测、关键点检测等业务，张量的离群点，可能对最终的结果的表现更加重要，此时要求量化门限更大，以避免对这些离群点进行饱和，进而影响这些分布特征的表达；另外，KLD 算法是基于量化后 int8 概率分布与 fp32 概率分布的相似性来计算量化门限的，而评估波形相似性的方法还有欧氏距离、cos 相似度等其他方法，这些度量方法不用考虑粗略的截取门限，而是直接来评估张量数值分布相似性，很多时候能有更好的表现，因此，在高效的 KLD 量化门限的基础上，TPU-MLIR 提出了自动调谐算法，对这些激活的量化门限基于欧氏距离度量进行微调，从而保证其 int8 量化具有更好的精度表现。

3. 实现方案

算法实现步骤如下：

(1) 统一对网络中带权重 layer 的权重进行伪量化，即从 fp32 量化为 int8，再反量化为 fp32，此操作会引入量化误差。

(2) 逐个对 OP 的输入激活量化门限进行调优：在初始 KLD 量化门限与激活的最大绝对值之间均匀选择 10 个候选值，用这些候选者对 fp32 参考激活值进行量化调优，引入量化误差，然后输入 OP 进行 fp32 计算，将输出的结果与 fp32 参考激活进行欧氏距离计算，选择 10 个候选值中具有最小欧氏距离的值作为调优门限。

(3) 对于 1 个 OP 输出连接到后面多个分支的情形，多个分支分别按上述方法计算量化门限，然后取其中的较大者，例如，自动调谐调优实现方案中层 1 的输出会分别针对层 2、层 3 调节一次，两次调节独立进行，根据实验证明，取最大值能兼顾两者，如图 5-15 所示。

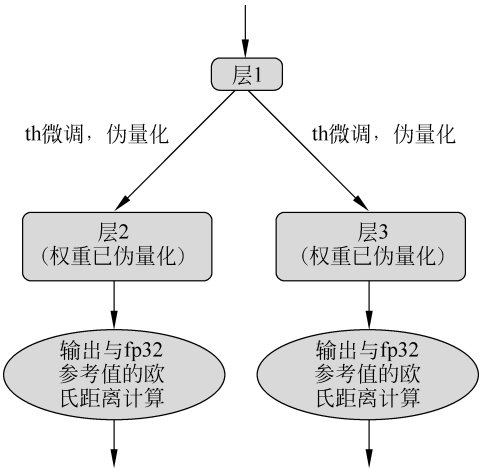


图 5-15　分层量化算法

4. 示例-yolov5s 校准

在 TPU-MLIR 的 Docker 环境中，在 TPU-MLIR 目录执行 source envsetup.sh 初始化环境后，任意新建目录，进入新建目录后执行如下命令，可以完成对 yolov5s 的校准过程，代码如下：

```
//第 5 章/mlir_distribution.py
$ model_transform.py \
    -- model_name yolov5s \
    -- model_def   ${REGRESSION_PATH}/model/yolov5s.onnx \
    -- input_shapes [[1,3,640,640]] \
    -- keep_aspect_ratio \   #keep_aspect_ratio、均值、缩放、pixel_format 均为预处理参数
    -- mean 0.0,0.0,0.0 \
    -- 缩放 0.0039216,0.0039216,0.0039216 \
    -- pixel_format rgb \
    -- output_names 350,498,646 \
    -- test_input ${REGRESSION_PATH}/image/dog.jpg \
    -- test_result yolov5s_top_outputs.npz \
    -- MLIR yolov5s.MLIR

$ run_calibration.py yolov5s.MLIR \
    -- dataset $REGRESSION_PATH/dataset/COCO2017 \
    -- input_num 100 \
    -- tune_num 10 \
    - o yolov5s_cali_table
```

执行 yolov5s_cali 校准结果，如图 5-16 所示。

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

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

图 5-16　yolov5s_cali 校准操作流程

5.4.4 可视化工具 visual 说明

可视化工具 visual.py 可以用来比较量化网络与原始网络的数据相似性，有助于在量化后精度不够满意时定位问题。此工具在 Docker 中启动，可以在宿主机中启动浏览器打开接口。工具默认使用 TCP 端口 10000，需要在启动 Docker 时使用-p 命令映射到宿主机，而工具的启动目录必须在网络所在目录。

由于网络是基于量化后的网络显示，所以可能会相比浮点网络有变化，对于浮点网络中不存在的张量会临时用量化后网络的数据替代，表现出来精度数据等都非常好，实际需要忽略，而只关注浮点与量化后网络都存在的张量，不存在的张量的数据类型一般是 NA，shape 也是[]这样的空值。

张量上的信息解读，如图 5-17 所示。

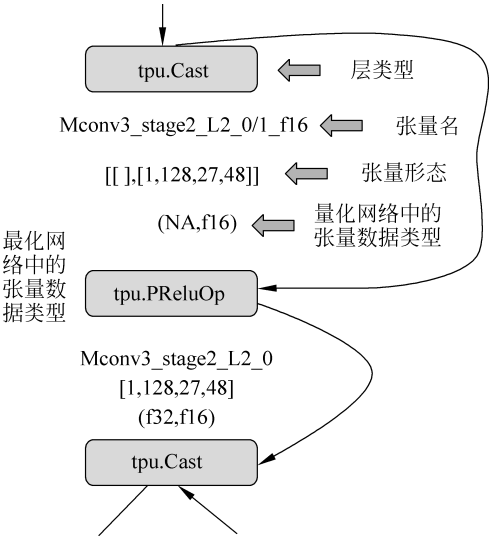


图 5-17 张量参数信息解析示例

1. 下译

下译将 TOP 层 OP 下沉到 TPU 层 OP，它支持的类型有 f32/f16/bf16/int8 对称/int8 非对称。

当转换 int8 时，它涉及量化算法；针对不同的芯片，量化算法是不一样的，例如有的支持每个通道，有的不支持；有的支持 32 位乘法，有的只支持 8 位乘法等，所以下译将算子从芯片无关层(TOP)，转换到了芯片相关层(TPU)。

2. 基本过程

如图 5-18 所示，下译过程包括以下流程：

(1) TOP 算子可以分为 f32 与 int8 两种，前者是大多数网络的情况。

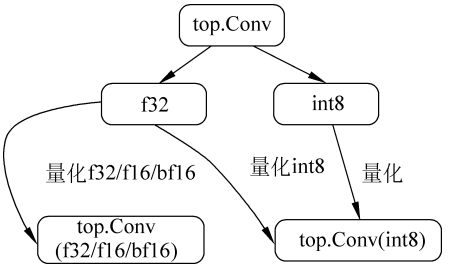


图 5-18　TOP 卷积算子量化流程

(2) 后者是如 TFLite 等量化过的网络的情况。

(3) f32 算子可以直接转换成 f32/f16/bf16 的 TPU 层算子，如果要转换成 int8，则需要类型是 calibrated_type。

(4) int8 算子只能直接转换成 TPU 层 int8 算子。

3. 混合精度

当 OP 之间的类型不一致时，可以插入 CastOp，进行混合精度量化，如图 5-19 所示。

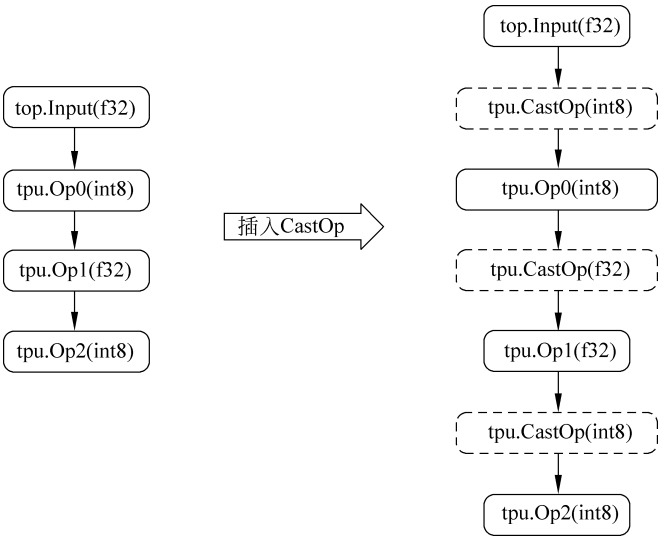


图 5-19　TOP 算子混合精度量化流程

这里假定输出的类型与输入的类型相同，如果不同，则需要特殊处理，例如 embedding 无论输出是什么类型，输入都是 uint 类型。

5.4.5　图层组

1. 基本概念

TPU 芯片分为片外内存(或称 Global Memory，GMEM)与片内内存(或称 Local Memory，LMEM)。

通常片外内存非常大(例如 4GB),片内内存非常小(例如 16MB)。神经网络模型的数据量与计算量都非常大,通常每层的 OP 都需要切分后放到片内内存进行运算,将结果再保存到片外内存。

2. 要解决的问题

图层组就是让尽可能多的 OP 经过切分后能够在片内内存执行,而避免过多的片内与片外内存的复制。

3. 基本思路

通过切分激活函数的 N 与 H,使每层的运算始终在片内内存中进行网络切分,如图 5-20 所示。

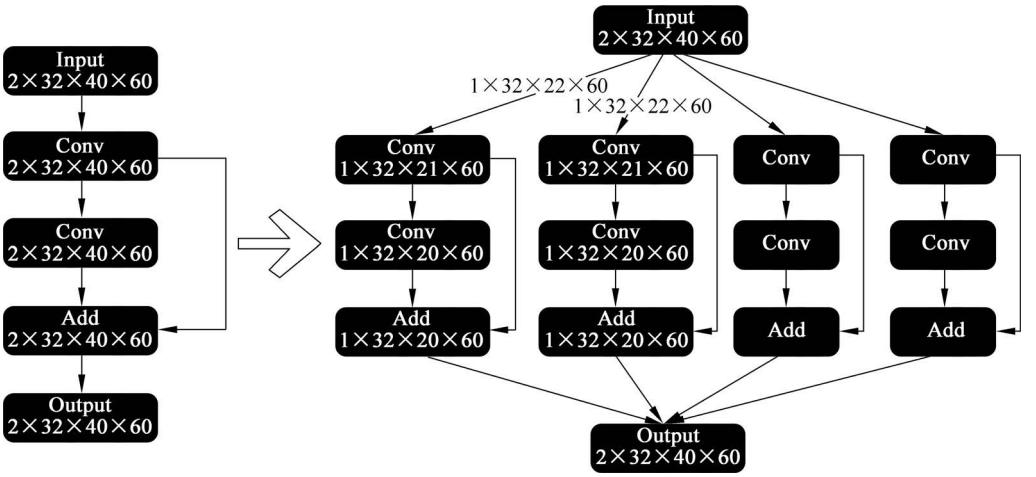


图 5-20 TOP 卷积算子网络切分举例

4. BackwardH

对网络进行 H 切分时,大多数 Layer 输入与输出的 H 是一致的,但是对于卷积、池化等需要特别计算。

以卷积 BackwardH 举例,如图 5-21 所示。

5.4.6 划分存储周期

如何划分 group? 首先把每层 Layer 需要的 LMEM 罗列出来,大体可以归为三类:

(1) 激活张量,用于保存输入/输出结果,若没有使用,就直接释放。

(2) 权重用于保存权重,在不切的情况下用完就释放,否则一直驻留在 LMEM 中。

(3) 缓存,用于 Layer 运算保存中间结果,用完就释放。

依次使用广度优先的方式,配置 LMEM 的 ID 分配,如图 5-22 所示。

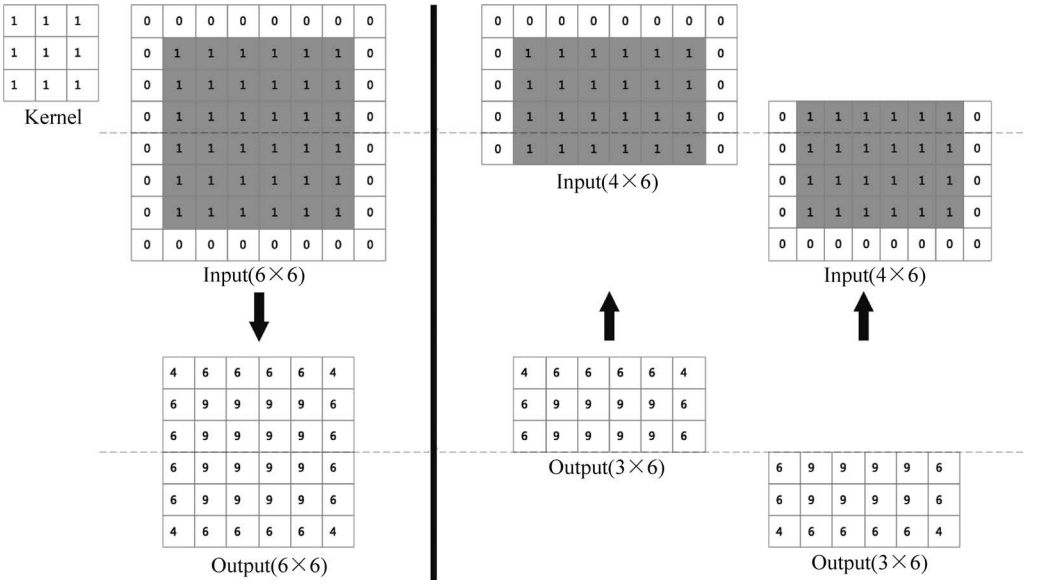


图 5-21　TOP 卷积算子后端 *H* 切分举例

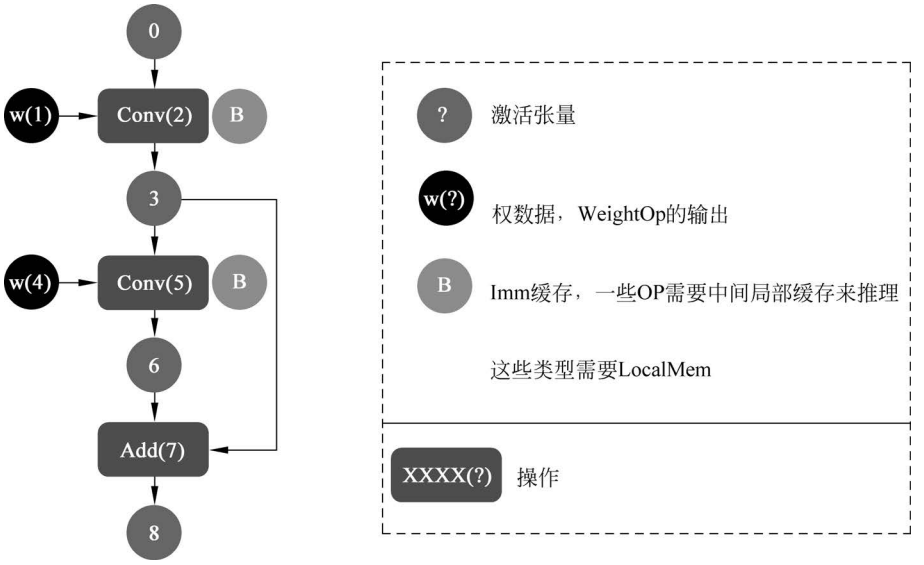


图 5-22　TOP 卷积算子后端 *H* 切分举例

配置时间步长分配，如图 5-23 所示。

关于配置周期的细节，方法如下：

(1) [T2，T7]，表示在 T2 开始时就要申请 LMEM，在 T7 结束时释放 LMEM。

(2) w(4)的原始周期应该是[T5，T5]，但是被修正成[T2，T5]，因为在 T2 做卷积运算时，w(4)可以被同时加载。

(3) 当 *N* 或者 *H* 被切分时，权重不需要重新被加载，它的结束点会被修正为正无穷。

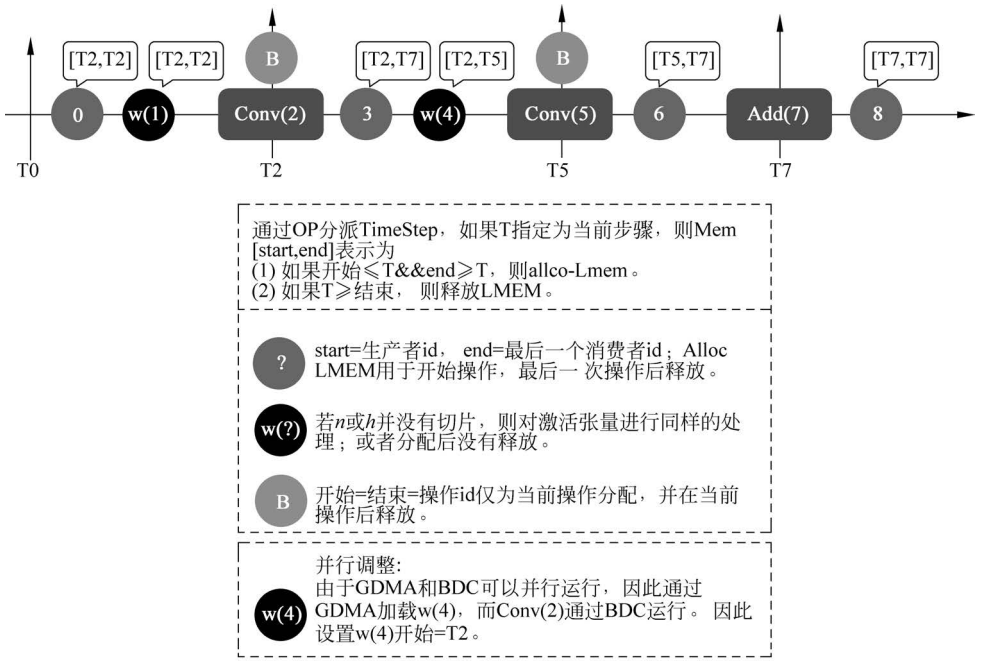


图 5-23　配置时间步长分配

1. LMEM 分配

当 n 或 h 存在切分时，权重常驻 LMEM，每个切分都可以继续使用权重。这时会先分配权重切分，如图 5-24 所示。

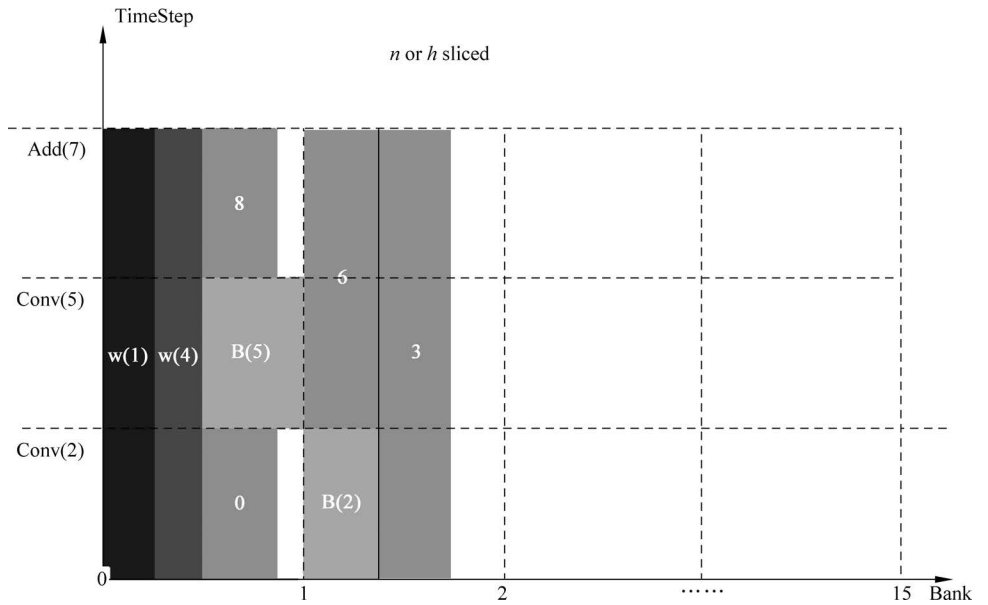


图 5-24　分配权重切分方法示例

当 n 与 h 都没有切分的情况下，权重与 activation 处理过程一样，不使用时就释放。这时的无切分情况的分配过程如图 5-25 所示。

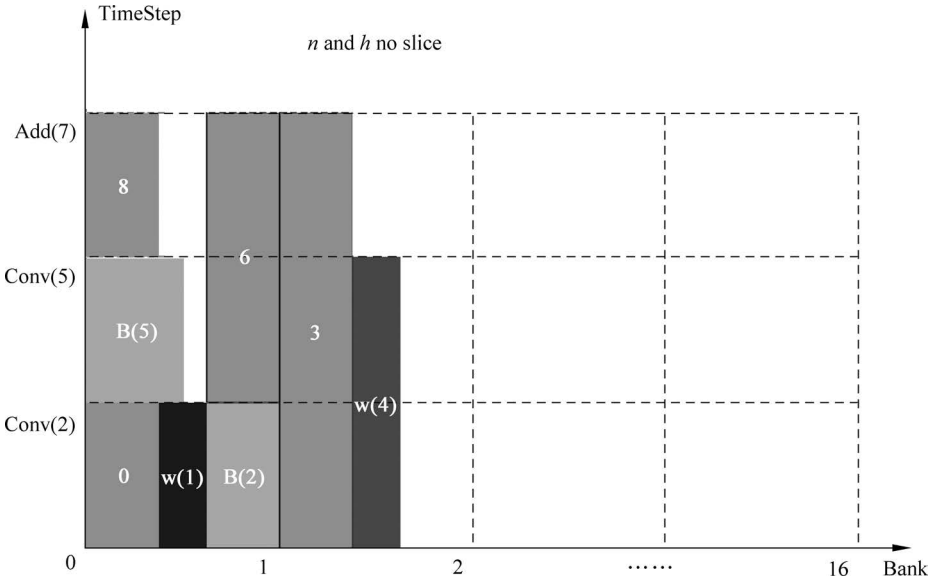


图 5-25　无切分分配方法示例

那么 LMEM 分配问题，就可以转换成这些方块如何摆放问题（注意方块只能左右移动，不能上下移动）。另外，LMEM 分配时优先不要跨 bank。目前的策略是按照 OP 的顺序依次分配，优先分配 timestep 长的，其次分配 LMEM 大的。

2. 划分最优组

目前从尾部开始向头部方向划分组，优先切 N，当 N 切到最小单位时还不能满足要求，则切 h。当网络很深时，因为卷积、池化等算子会有重复计算部分，如果 h 切得过多，则会导致重复部分过多。

为了避免过多重复，如果输入后向层的 h_slice 重复的部分$>h/2$，则认为失败，例如，输入的 $h = 100$，经过切分后变成两个输入，$h[0,80)$与 $h[20,100)$，重复部分为 60，则认为失败；两个输入对应 $h[0,60)$与 $h[20,100)$，重复部分为 40，则认为成功。

划分最优组方法的示例，如图 5-26 所示。

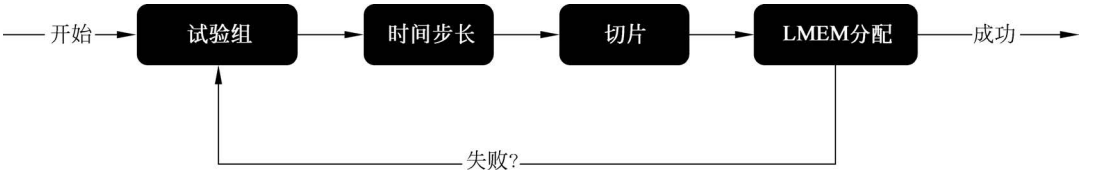


图 5-26　划分最优组方法示例

5.4.7 GMEM 分配

1. 目的

为了节约片外内存空间，最大程度地复用内存空间，分配顺序为权重张量，根据生命周期给全部全局神经元张量分配 GMEM，在分配过程中会复用已分配的 GMEM。

全局神经元张量的定义为在 OP 运算结束后需要保存在 GMEM 的 tensor。如果是图层组，则只有图层组的输入/输出张量属于全局神经张量。

2. 原理

权重张量分配 GMEM 遍历所有的 WeightOp，依次分配，4K 地址对齐，地址空间不断累加。

全局神经网络分配 GMEM 最大可能的复用内存空间，根据生命周期分配给全部全局神经网络，在分配过程中会复用已分配的 GMEM。

3. 数据结构介绍

每次分配时会把对应的张量、address、size、ref_cnt(这个张量有几个 OP 使用)记录在 rec_tbl。同时将张量、address 记录在辅助数据结构 hold_edges、in_using_addr 中，代码如下：

```
//第 5 章/mlir_value_offset.c
//Value, offset, size, ref_cnt
using gmem_entry = std::tuple;
std::vector rec_tbl;
std::vector hold_edges;
std::set in_using_addr;
```

4. 流程介绍

流水线计算流程包括以下步骤：

(1) 遍历每个 Op，在遍历 Op 时，判断 Op 的输入张量是否位于 rec_tbl 中，如果是，则判断 ref_cnt 是否≥1。如果是，则 ref_cnt-，表示输入张量的引用数降低 1 个；如果 ref_cnt 等于 0，则表示生命周期已结束，后面的张量可以复用它的地址空间。

(2) 在给每个 Op 的输出张量分配 GMEM 时，先判断是否可以复用 EOL 的张量地址、判断思路、遍历 rec_tbl，需要同时满足以下 7 个条件才能重用：

① 对应的张量不在 hold_edges 内。

② 对应张量的地址不在 in_using_addr 内。

③ 对应张量已 EOL。

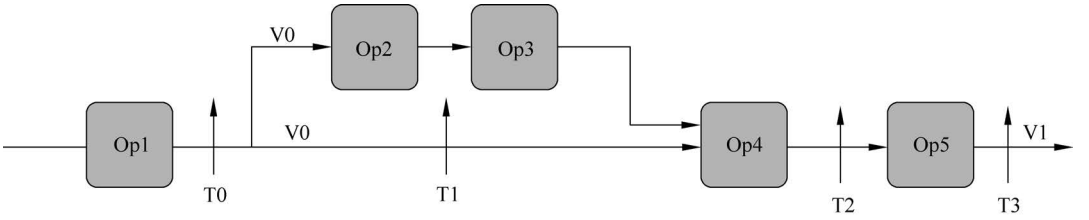
④ 对应张量的地址空间≥当前张量所需的空间。

⑤ 当前 Op 的输入张量地址不能与对应张量的地址相同(某些 Op 最终运算结果不正确，ReshapeOP 例外)。

⑥ 给当前 Op 的输出张量分配 GMEM，如果 step2 显示可以重用，就重用，否则在 DDR 中新开辟的 GMEM。

⑦ 调整当前 Op 的输入张量的生命周期，确认它是否位于 hold_edges 内，如果是，则在 rec_tbl 中寻找，检查它的 ref_cnt 是否为 0。如果是，则把它从 hold_edges 中删除，并且把它的 addr 从 in_using_addr 中删除，意味着这个输入张量生命周期已结束，地址空间已释放。

多算子流水线计算流程示例，如图 5-27 所示。



注意格式: Op1有一个名为V0的输出(ref_cnt=2)，V0是两个Ops(Op2，Op4)的输入，在Op1计算之前，V0的ref_cnt减去1。当Op4开始计算时也减去V0的ref_cnt(ref_ cnt=0)。禁止Op在输入和输出之间共享内存，但少数Op除外，如ReshapeOp。因此Op4不能使用V0的内存作为Output的内存。Op5的输 出名为V1，可以重用V0的内存。

图 5-27　多算子流水线计算流程示例

5.4.8　TOP 方言操作

1. AddOp

AddOp 及性能描述见表 5-10。

表 5-10　AddOp 及性能描述

功 能 模 块	描　　述
简述	加法操作，$Y=\text{coeff}_0 * X_0+\text{coeff}_1 * X_1$
输入	inputs：张量数组，对应两个或多个输入张量
输出	output：张量
属性	do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则指定上限值；如果是负数，则认为没有上限 coeff：对应每个张量的系数，默认值为 1.0
输出	output：output 张量
接口	无
范例	%2 = top.Add(%0,%1) {do_relu = false}：(tensor<1×3×27×27×f32>,tensor<1×3×27×27×f32>) -> tensor<1×3×27×27×f32> loc(add)

2. AddPoolOp

AddPoolOp 及性能描述见表 5-11。

表 5-11 AddPoolOp 及性能描述

功能模块	描　　述
简述	将输入的张量进行平均池化，$S=\frac{1}{\text{width}*\text{height}}\sum_{i,j}a_{i,j}$。大小给定的滑动窗口会依次对输入张量进行池化 其中 width 与 height 表示 kernel_shape 的宽度与高度。$\sum_{i,j}a_{i,j}$ 则表示对 kernel_shape 进行求与
输入	输入：张量
输出	输出：张量
属性	kernel_shape：控制平均池化滑动窗口的大小 strides：步长，控制滑动窗口每次滑动的距离 pads：控制填充形状，方便池化 pad_value：填充内容，常数，默认值为 0 count_include_pad：结果是否需要对填充的 pad 进行计数 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则指定上限值；如果是负数，则认为没有上限
接口	无
范例	%90 = top.AvgPool(%89) {do_relu = false, kernel_shape = [5, 5], pads = [2, 2, 2, 2], strides = [1, 1]}: (tensor<1×256×20×20×f32>) -> tensor<1×256×20×20×f32> loc(resnetv22_pool1_fwd_GlobalAveragePool)

3. Depth2SpaceOp

Depth2SpaceOp 及性能描述见表 5-12。

表 5-12 Depth2SpaceOp 及性能描述

功能模块	描　　述
简述	深度转空间操作，$Y=\text{Depth2Space}(X)$
输入	inputs：张量
输出	输出：张量
属性	block_h：张量，高度改变的参数，i64 类型 block_w：张量，宽度改变的参数，i64 类型 is_CRD：column-row-depth，如果值为 True，则数据沿深度方向的排布按照 HWC，否则为 CHW，bool 类型 is_inversed：如果值为 True，则结果的形状为$[n,c*\text{block}_h*\text{block}_w,h/\text{block}_h,w/\text{block}_w]$，否则结果的形状为：$[n,c/(\text{block}_h*\text{block}_w),h*\text{block}_h,w*\text{block}_w]$
输出	输出：输出张量
接口	无
范例	%2 = top.Depth2Space(%0) {block_h = 2, block_w = 2, is_CRD = true, is_inversed = false}: (tensor<1×8×2×3×f32>) -> tensor<1×2×4×6×f32> loc(add)

4. BatchNormOp

BatchNormOp 及性能描述，见表 5-13。

表 5-13　BatchNormOp 及性能描述

功能模块	描　　述
简述	在一个四维输入张量上执行批标准化(Batch Normalization)。 具体计算公式如下： $$y=\frac{x-E[x]}{\sqrt{\mathrm{Var}[x]+\varepsilon}}*\gamma+\beta \tag{5-28}$$
输入	输入：四维输入张量 Mean：输入的均值张量 Variance：输入的方差张量 Gamma：公式中的 γ 张量，可以为 None beta：公式中的 β 张量，可以为 None
输出	输出：结果张量
属性	epsilon：公式中的 ε 常量，默认值为 1e-05 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，指定上限值；如果是负数，则认为没有上限
接口	无
范例	%5 = top.BatchNorm(%0, %1, %2, %3, %4) {epsilon = 1e-05, do_relu = false}: (tensor<1×3×27×27×f32>, tensor<3×f32>, tensor<3×f32>, tensor<3×f32>, tensor<3×f32>) -> tensor<1×3×27×27×f32> loc(BatchNorm)

5. ClipOp

ClipOp 及性能描述见表 5-14。

表 5-14　ClipOp 及性能描述

功能模块	描　　述
简述	将给定输入限制在一定范围内
输入	输入：张量
输出	输出：张量
属性	min：给定的下限 max：给定的上限
输出	输出：输出张量
接口	无
范例	%3 = top.Clip(%0) {max = 1%: f64, min = 2%: f64}: (tensor<1×3×32×32×f32>) -> tensor<1×3×32×32×f32> loc(Clip)

6. ConcatOp

ConcatOp 及性能描述见表 5-15。

表 5-15　ConcatOp 及性能描述

功能模块	描　　述
简述	将给定的张量序列在给定的维度上连接起来。所有的输入张量或者都具有相同的 shape(待连接的维度除外)或者都为空

续表

功能模块	描　　述
输入	inputs：张量数组，对应两个或多个输入张量
输出	输出：结果张量
属性	axis：待连接的维度的下标 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，指定上限值；如果是负数，则认为没有上限
接口	无
范例	%2 = top.Concat(%0, %1) {axis = 1, do_relu = false}：(tensor<1×3×27×27×f32>, tensor<1×3×27×27×f32>)-> tensor<1×6×27×27×f32> loc(Concat)

7. ConvlutionOp

ConvlutionOp 及性能描述见表 5-16。

表 5-16　ConvlutionOp 及性能描述

功能模块	描　　述
简述	对输入张量执行二维卷积操作。 简单来讲，给定输入大小为(N,C_{in},H,W)，output(N,C_{out},H_{out},W_{out})的计算方法为 $$\text{out}(N_{in},C_{out}) = \text{bias}(C_{out}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_j,k) * \text{input}(N_i,k) \tag{5-29}$$ 在式(5-29)中，* 是有效的 cross-correlation 操作，N 是 batch 的大小，C 是 channel 的数量，而 H 和 W 是输入图片的高与宽
输入	输入：输入张量 filter：参数张量，其形状为 $$\left(\text{out\_channels},\frac{\text{in\_channels}}{\text{groups}},\text{kernel\_size}[0],\text{kernel\_size}[1]\right) \tag{5-30}$$ 在式(5-30)中，bias 为可学习的偏差张量，形状为(out_channels)。
输出	输出：结果张量
属性	kernel_shape：卷积核的尺寸 strides：卷积的步长 pads：输入的每条边补充 0 的层数 group：从输入通道到输出通道的阻塞连接数，默认值为 1 dilations：卷积核元素之间的间距，可选 inserts：可选 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
接口	无
范例	%2 = top.conv(%0, %1) {kernel_shape = [3, 5], strides = [2, 1], pads = [4, 2]}：(tensor<20×16×50×100×f32>, tensor<33×3×5×f32>) -> tensor<20×33×28×49×f32> loc(卷积)

8. DeconvOp

DeconvOp 及性能描述见表 5-17。

表 5-17　DeconvOp 及性能描述

功能模块	描　　述
简述	对输入张量执行反卷积操作
输入	输入：输入张量 filter：参数张量，其形状为 $$\left(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1]\right) \tag{5-31}$$ 在式(5-31)中，bias 为可学习的偏差张量，形状为(out_channels)
输出	输出：结果张量
属性	kernel_shape：卷积核的尺寸 strides：卷积的步长 pads：输入的每条边补充 0 的层数 group：从输入通道到输出通道的阻塞连接数，默认值为 1 dilations：卷积核元素之间的间距，可选 inserts：可选 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
接口	无
范例	%2 = top.Deconv(%0, %1) {kernel_shape = (3, 5), strides = (2, 1), pads = (4, 2)}: (tensor<20×16×50×100×f32>, tensor<33×3×5×f32>)-> tensor<20×33×28×49×f32> loc(Deconv)

9. DivOp

DivOp 及性能描述见表 5-18。

表 5-18　DivOp 及性能描述

功能模块	描　　述
简述	除法操作，$Y = X_0 / X_1$
输入	inputs：张量数组，对应两个或多个输入张量
输出	输出：张量
属性	do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值，如果是负数，则认为没有上限 乘法：量化用的乘数，默认值为 1 rshift：量化用的右移，默认值为 0
输出	输出：输出张量
接口	无
范例	%2 = top.Div(%0, %1) {do_relu = false, relu_limit = -1.0, 乘法 = 1, rshift = 0}: (tensor<1×3×27×27×f32>, tensor<1×3×27×27×f32>) -> tensor<1×3×27×27×f32> loc(div)

10. LeakyReluOp

LeakyReluOp 及性能描述见表 5-19。

表 5-19 LeakyReluOp 及性能描述

功能模块	描　　述
简述	张量中的每个元素执行 LeakyReLU 函数，函数可表示为 f(x) = alpha * x for x < 0，f(x) = x for x >= 0
输入	输入：张量
输出	输出：张量
属性	alpha：对应每个张量的系数
输出	输出：输出张量
接口	无
范例	%4 = top.LeakyRelu(%3) {alpha = 0.67000001668930054：f64}：(tensor<1×32×100×100×f32>) -> tensor<1×32×100×100×f32> loc(LeakyRelu)

11. LSTMOp

LSTMOp 及性能描述见表 5-20。

表 5-20 LSTMOp 及性能描述

功能模块	描　　述
简述	执行 RNN 的 LSTM 操作
输入	输入：张量
输出	输出：张量
属性	filter：卷积核 recurrence：循环单元 bias：LSTM 的参数，偏置 initial_h：LSTM 中的每句话经过当前 cell 后会得到一个 state，state 是个 tuple(c，h)，其中 h=[batch_size，hidden_size] initial_c：c=[batch_size，hidden_size] have_bias：是否设置偏置 bias，默认值为 False bidirectional：设置双向循环的 LSTM，默认值为 False batch_first：是否将 batch 放在第一维，默认值为 False
输出	输出：输出张量
接口	无
范例	%6 =top.LSTM(%0，%1，%2，%3，%4，%5) {batch_first = false，bidirectional = true，have_bias = true}：(tensor<75×2×128×f32>，tensor<2×256×128×f32>，tensor<2×256×64×f32>，tensor<2×512×f32>，tensor<2×2×64×f32>，tensor<2×2×64×f32>) -> tensor<75×2×2×64×f32> loc(LSTM)

12. LogOp

LogOp 及性能描述见表 5-21。

表 5-21　LogOp 及性能描述

功能模块	描　　述
简述	按元素计算给定输入张量的自然对数
输入	输入：张量
输出	输出：张量
属性	无
输出	输出：输出张量
接口	无
范例	%1 = top.Log(%0)：(tensor<1×3×32×32×f32>) -> tensor<1×3×32×32×f32> loc(Log)

13. MaxPoolOp

MaxPoolOp 及性能描述见表 5-22。

表 5-22　MaxPoolOp 及性能描述

功能模块	描　　述
简述	对输入的张量进行最大池化
输入	输入：张量
输出	输出：张量
属性	kernel_shape：控制平均池化滑动窗口的大小 strides：步长，控制滑动窗口每次滑动的距离 pads：控制填充形状，方便池化 pad_value：填充内容，常数，默认值为 0 count_include_pad：结果是否需要对填充的 pad 进行计数 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
接口	无
范例	%8 = top.MaxPool(%7) {do_relu = false, kernel_shape = [5, 5], pads = [2, 2, 2, 2], strides = [1, 1]}：(tensor<1×256×20×20×f32>) -> tensor<1×256×20×20×f32> loc(resnetv22_pool0_fwd_MaxPool)

14. MatMulOp

MatMulOp 及性能描述见表 5-23。

表 5-23　MatMulOp 及性能描述

功能模块	描　　述
简述	二维矩阵乘法操作，$C=A*B$
输入	输入：tensor：$m*k$ 大小的矩阵 right：tensor：$k*n$ 大小的矩阵
输出	输出：张量 $m*n$ 大小的矩阵

续表

功能模块	描　　述
属性	bias：偏差，量化时会根据 bias 计算 bias_scale，可以为空 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
输出	输出：输出张量
接口	无
范例	%2 = top.MatMul(%0, %1) {do_relu = false, relu_limit = -1.0}：(tensor<3×4×f32>, tensor<4×5×f32>) -> tensor<3×5×f32> loc(matmul)

15. MulOp

MulOp 及性能描述见表 5-24。

表 5-24　MulOp 及性能描述

功能模块	描　　述
简述	乘法操作，$Y=X_0 * X_1$
输入	inputs：张量数组，对应两个或多个输入张量
输出	输出：张量
属性	do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限 乘法：量化用的乘数，默认值为 1 rshift：量化用的右移，默认值为 0
输出	输出：输出张量
接口	无
范例	%2 = top.Mul(%0, %1) {do_relu = false, relu_limit = -1.0, 乘法 = 1, rshift = 0}：(tensor<1×3×27×27×f32>, tensor<1×3×27×27×f32>) -> tensor<1×3×27×27×f32> loc(mul)

16. MulConstOp

MulConstOp 及性能描述见表 5-25。

表 5-25　MulConstOp 及性能描述

功能模块	描　　述
简述	与常数做乘法操作，$Y=X * \text{Constval}$
输入	inputs：张量
输出	输出：张量
属性	const_val：f64 类型的常量 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
输出	输出：输出张量
接口	无

续表

功能模块	描　　述
范例	%1 = arith.constant4.7：f64 %2 =top.MulConst(%0) {do_relu = false，relu_limit = −1.0}：(tensor<1×3×27×27×f64>，%1) -> tensor<1×3×27×27×f64> loc(mulconst)

17．PermuteOp

PermuteOp 及性能描述见表 5-26。

表 5-26　PermuteOp 及性能描述

功能模块	描　　述
简述	改变张量布局，变化张量数据维度的顺序，将输入的张量按照 order 给定的顺序重新布局
输入	inputs：张量数组，任意类型的张量
属性	order：指定重新布局张量的顺序
输出	输出：输出张量，按 order 的顺序重新布局后的张量
接口	无
范例	%2 =top.Permute(%1) {order = [0，1，3，4，2]}：(tensor<4×3×85×20×20×f32>) -> tensor<4×3×20×20×85×f32> loc(output_Transpose)

18．ReluOp

ReluOp 及性能描述见表 5-27。

表 5-27　ReluOp 及性能描述

功能模块	描　　述
简述	张量中的每个元素执行 ReLU 函数，如果极限为 0，则不使用上限
输入	输入：张量
输出	输出：张量
属性	relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
输出	输出：输出张量
接口	无
范例	%1 = top.Relu(%0) {relu_limit = 6.000000e+00：f64}：(tensor<1×3×32×32×f32>) -> tensor<1×3×32×32×f32> loc(Clip)

19．ReshapeOp

ReshapeOp 及性能描述见表 5-28。

表 5-28　ReshapeOp 及性能描述

功能模块	描　　述
简述	Reshape 算子，返回一个给定形状的张量，该张量的类型与内部的值与输入张量相同。Reshape 可能会对张量的任何一行进行操作。在 Reshape 过程中不会有任何数据的值被修改
输入	输入：张量

续表

功能模块	描　　述
输出	输出：张量
属性	无
接口	无
范例	%133 = top.Reshape(%132)：(tensor<1×255×20×20×f32>) -> tensor<1×3×85×20×20×f32> loc(resnetv22_flatten0_reshape0_Reshape)

20. ScaleOp

ScaleOp 及性能描述见表 5-29。

表 5-29　ScaleOp 及性能描述

功能模块	描　　述
简述	缩放操作 $Y=X*S+B$，其中 X/Y 的 shape 为[N，C，H，W]，S/B 的 shape 为[1，C，1，1]。
输入	输入：输入张量 缩放：保存输入的放大倍数 bias：放大后加上的 bias
输出	输出：结果张量
属性	do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
接口	无
范例	%3 = top.scale(%0，%1，%2) {do_relu = false}：(tensor<1×3×27×27×f32>，tensor<1×3×1×1×f32>，tensor<1×3×1×1×f32>) -> tensor<1×3×27×27×f32> loc(缩放)

21. SigmoidOp

SigmoidOp 及性能描述见表 5-30。

表 5-30　SigmoidOp 及性能描述

功能模块	描　　述
简述	激活函数，将张量中的元素映射到特定区间，默认映射到[0，1]，计算方法为 $$Y=\frac{\text{scale}}{1+e^{-X}}+\text{bias} \tag{5-32}$$
输入	inputs：张量数组，任意类型的张量
属性	缩放：倍数，默认值为 1 bias：偏置，默认值为 0
输出	输出：输出张量
接口	无
范例	%2 = top.Sigmoid(%1) {bias = 0.000000e+00：f64，scale = 1.000000e+00：f64}：(tensor<1×16×64×64×f32>) -> tensor<1×16×64×64×f32> loc(output_Sigmoid)

22. SiLUOp

SiLUOp 及性能描述见表 5-31。

表 5-31　SiLUOp 及性能描述

功能模块	描　　述
简述	激活函数，$Y=\frac{X}{1+e^{-X}}$或 $Y=X*\mathrm{Sigmoid}(X)$
输入	输入：张量数组，任意类型的张量
属性	无
输出	输出：输出张量
接口	无
范例	%1 =top. SiLU(%0)：(tensor<1×16×64×64×f32>) -> tensor<1×16×64×64×f32> loc(output_Mul)

23. SliceOp

SliceOp 及性能描述见表 5-32。

表 5-32　SliceOp 及性能描述

功能模块	描　　述
简述	张量切片，将输入的张量的各个维度，根据 offset 与 steps 数组中的偏移与步长进行切片，生成新的张量
输入	输入：张量数组，任意类型的张量
属性	offset：存储切片偏移的数组，offset 数组的索引与输入张量的维度索引对应 steps：存储切片步长的数组，steps 数组的索引与输入张量维度索引对应
输出	输出：输出张量
接口	无
范例	%1 = top. Slice(%0) {offset = [2, 10, 10, 12], steps = [1, 2, 2, 3]}：(tensor<5×116×64×64×f32>) -> tensor<3×16×16×8×f32> loc(output_Slice)

24. SoftwareOp

SoftwareOp 及性能描述见表 5-33。

表 5-33　SoftwareOp 及性能描述

功能模块	描　　述
简述	对输入张量，在指定 axis 的维度上计算归一化指数值，计算的方法如下： $$\sigma(Z)_i=\frac{e^{\beta Z_i}}{\sum_{j=0}^{K-1}e^{\beta Z_j}} \tag{5-33}$$ 在式(5-33)中，$\sum_{j=0}^{K-1}e^{\beta Z_j}$，在 axis 维度上做指数值求与，$j$ 从 0 到 $K-1$，K 是输入张量在 axis 维度上的尺寸。 例如，输入张量的尺寸为(N,C,W,H)，在 axis=1 的通道上计算 Softmax，计算方法为 $Y_{n,i,w,h}=\frac{e^{\beta X_{n,i,w,h}}}{\sum_{j=0}^{C-1}e^{\beta X_{n,j,w,h}}}$

续表

功能模块	描　　述
输入	输入：张量数组，任意类型的张量
属性	axis：维度索引，用于指定对输入张量执行 Softmax 对应的维度，axis 可以取值[$-r$，$r-1$]，r 为输入张量维度的数量，当 axis 为负数时，表示倒序维度 beta：TFLite 模型中对输入的缩放系数，非 TFLite 模型无效，默认值为 1.0
输出	输出：输出张量，在指定维度做归一化指数值后的张量
接口	无
范例	%1 = top.Softmax(%0) {axis = 1：i64}：(tensor<1×1000×1×1×f32>) -> tensor<1×1000×1×1×f32> loc(output_Softmax)

25. SqueezeOp

SqueezeOp 及性能描述见表 5-34。

表 5-34　SqueezeOp 及性能描述

功能模块	描　　述
简述	对输入张量进行指定维度的裁剪并返回裁剪后的张量
输入	输入：张量
输出	输出：张量
属性	axes：指定需要裁剪的维度，0 代表第一个维度，-1 代表最后一个维度
接口	无
范例	%133 =top.Squeeze(%132) {axes = [-1]}：(tensor<1×255×20×20×f32) -> tensor<1×255×20×f32> loc(#loc278)

26. UpsampleOp

UpsampleOp 及性能描述见表 5-35。

表 5-35　UpsampleOp 及性能描述

功能模块	描　　述
简述	上采样 OP，将输入张量进行最近邻上采样并返回张量
输入	张量
属性	scale_h：目标图像与原图像的高度之比 scale_w：目标图像与原图像的宽度之比 do_relu：结果是否做 ReLU，默认值为 False relu_limit：如果做 ReLU，则需指定上限值；如果是负数，则认为没有上限
输出	输出：张量
接口	无
范例	%179 = top.Upsample(%178) {scale_h = 2：i64，scale_w = 2：i64}：(tensor<1×128×40×40×f32>) -> tensor<1×128×80×80×f32> loc(268_Resize)

27. WeightOp

WeightOp 及性能描述见表 5-36。

表 5-36 WeightOp 及性能描述

功能模块	描 述
简述	权重 OP,包括权重的读取与创建,权重会被保存到 NPZ 文件中。权重的 location 与 NPZ 文件中的张量名称是对应关系
输入	无
属性	无
输出	输出：权重张量
接口	read：读取权重数据,类型由模型指定 read_as_float：将权重数据转换成 float 类型读取 read_as_byte：将权重数据按字节类型读取 create：创建权重 OP clone_bf16：将当前权重转换成 bf16,并创建权重 OP clone_f16：将当前权重转换成 f16,并创建权重 OP
范例	%1 = top.Weight()：() -> tensor<32×16×3×3×f32> loc(filter)

5.4.9 评估验证

1. 验证对象

TPU-MLIR 中的精度验证主要针对 MLIR 模型,fp32 采用 TOP 层的 MLIR 模型进行精度验证,而 int8 对称与非对称量化模式则采用 TPU 层的 MLIR 模型。

2. 评估指标

当前主要用于测试的网络有分类网络与目标检测网络,分类网络的精度指标采用 Top-1 与 Top-5 准确率,而目标检测网络采用 COCO 的 12 个评估指标,如下所示。通常在记录精度时,采用 IoU=0.5 时的平均精度 PASCAL VOC 度量。

AP:\% AP at IoU=.50:.05:.95（主要挑战度量）

AP^{IoU}=.50\% AP at IoU=.50（PASCAL VOC 度量）

AP^{IoU}=.75\% AP at IoU=.75（严格度量）

AP^{small}：\% 小目标 AP：area<32^2

AP^{medium}\% 中目标 AP：32^2<area<96^2

AP^{large}\% 大目标 AP：area>96^2

$AR^{max=1}$\% AR 给出每幅图像 1 次检测

$AR^{max=10}$\% AR 给出每幅图像 10 次检测

$AR^{max=100}$\% AR 给出每幅图像 100 次检测

AP^{small}\% 小目标 AP:area<32^2

AP^{medium}\% 中目标 AP：32^2<area<96^2

AP^{large}\% 大目标 AP：area>96^2

3. 数据集

验证时使用的数据集需要自行下载，分类网络使用 ILSVRC2012 验证集（共 50000 幅图像）。数据集中的图片有两种摆放方式，一种是数据集目录下有 1000 个子目录，对应 1000 个类别，每个子目录下有 50 张该类别的图片，该情况下无须标签文件；另外一种是所有图片均在同一个数据集目录下，有一个特定的 TXT 标签文件，按照图片编号顺序每行用数字 1～1000 表示每张图片的类别。

目标检测网络使用 COCO2017 验证集（共 5000 张图片），所有图片均在同一数据集目录下，另外还需要下载与该数据集对应的标签文件.json。

1）精度验证接口

TPU-MLIR 的精度验证命令参考，代码如下：

```
//第 5 章/eval_datasets.py
$ model_eval.py \
    -- model_file mobiLeNet_v2.MLIR \
    -- count 50 \
    -- dataset_type ImageNet \
    -- postprocess_type topx \
-- dataset datasets/ILSVRC2012_img_val_with_subdir
```

所支持的参数见表 5-37。

表 5-37 model_eval.py 参数功能

参 数 名	是否必选	说 明
model_file	是	指定模型文件
dataset	否	数据集目录
dataset_type	否	数据集类型，当前主要支持 ImageNet、COCO，默认为 ImageNet
postprocess_type	是	精度评估方式，当前支持 topx 与 coco_mAP
label_file	否	TXT 标签文件，在验证分类网络精度时可能需要
coco_annotation	否	JSON 标签文件，在验证目标检测网络时需要
count	否	用来验证精度的图片数量，默认使用整个数据集

2）精度验证样例

以 mobiLeNet_v2 与 yolov5s 分别作为分类网络与目标检测网络的代表进行精度验证。

3）数据集下载

将 ILSVRC2012 验证集下载到 datasets/ILSVRC2012_img_val_with_subdir 目录下，数据集的图片采用带有子目录的摆放方式，因此不需要特定的标签文件。

4. 模型转换

使用 model_transform.py 接口，将原模型转换为 mobiLeNet_v2.mlir 模型，并通过 run_calibration.py 接口获得 mobiLeNet_v2_cali_table。TPU 层的 int8 模型则通过下方的命令获得，运行完命令后会获得一个名为 mobiLeNet_v2_bm1684x_int8_sym_tpu.mlir 的中间文件，接下来将用该文件进行 int8 对称量化模型的精度验证，代码如下：

```
//第 5 章/model_deploy.py
#int8 对称量化模型
$ model_deploy.py \
    --MLIR mobiLeNet_v2.MLIR \
    --quantize INT8 \
    --calibration_table mobiLeNet_v2_cali_table \
    --chip bm1684x \
    --test_input mobiLeNet_v2_in_f32.npz \
    --test_reference mobiLeNet_v2_top_outputs.npz \
    --tolerance 0.95,0.69 \
    --model mobiLeNet_v2_int8.bmodel
```

5. 精度验证

使用 model_eval.py 接口进行精度验证，代码如下：

```
//第 5 章/model_eval.py
#f32 模型精度验证
$ model_eval.py \
     --model_file mobiLeNet_v2.MLIR \
     --count 50000 \
     --dataset_type ImageNet \
     --postprocess_type topx \
     --dataset datasets/ILSVRC2012_img_val_with_subdir

#int8 对称量化模型精度验证
$ model_eval.py \
     --model_file mobiLeNet_v2_bm1684x_int8_sym_tpu.MLIR \
     --count 50000 \
     --dataset_type ImageNet \
     --postprocess_type topx \
     --dataset datasets/ILSVRC2012_img_val_with_subdir
```

f32 模型与 int8 对称量化模型的精度验证结果，代码如下：

```
//第 5 章/model_eval.py
#mobiLeNet_v2.MLIR 精度验证结果
2023/11/08 01:30:29 - INFO: idx:50000, top1:0.710, top5:0.899
INFO:root:idx:50000, top1:0.710, top5:0.899

#mobiLeNet_v2_bm1684x_int8_sym_tpu.MLIR 精度验证结果
2023/11/08 05:43:27 - INFO: idx:50000, top1:0.702, top5:0.895
INFO:root:idx:50000, top1:0.702, top5:0.895
```

6. yolov5s

1）数据集下载

将 COCO2017 验证集下载到 datasets/val2017 目录下，该目录下包含 5000 张用于验证的图片。将对应的标签文件 instances_val2017.json 下载到 datasets 目录下。

2）模型转换

转换流程与 mobiLeNet_v2 相似。

3）精度验证

使用 model_eval.py 接口进行 f32 精度验证，代码如下：

```
//第 5 章/mlir_model_eval_build.py
$ model_eval.py \
    -- model_file yolov5s.MLIR \
    -- count 5000 \
    -- dataset_type coco \
    -- postprocess_type coco_mAP \
    -- coco_annotation datasets/instances_val2017.json \
    -- dataset datasets/val2017
```

int8 对称量化模型精度验证，代码如下：

```
//第 5 章/mlir_model_eval_build_1.py
$ model_eval.py \
    -- model_file yolov5s_bm1684x_int8_sym_tpu.MLIR \
    -- count 5000 \
    -- dataset_type coco \
    -- postprocess_type coco_mAP \
    -- coco_annotation datasets/instances_val2017.json \
    -- dataset datasets/val2017
```

f32 模型与 int8 对称量化模型的精度验证结果，代码如下：

```
//第 5 章/mlir_model_eval_build_2.py
#yolov5s.MLIR 精度验证结果
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.369
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.561
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.393
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.217
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.422
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.470
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.300
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.502
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.542
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.359
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.602
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.670
```

yolov5s_bm1684x_int8_sym_tpu.MLIR 精度验证结果，代码如下：

```
//第 5 章/mlir_model_eval_build_3.py
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.337
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.544
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.365
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.196
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.382
```

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

5.5 QAT量化感知训练

5.5.1 QAT量化技术基本原理

相比训练后量化，因为其不是全局最优，所以导致精度损失，QAT量化感知训练能做到基于Loss优化的全局最优，而尽可能地降低量化精度损失，其基本原理是：在fp32模型训练中就提前引入了推理时量化导致的权重与激活的误差，用任务Loss在训练集上来优化可学习的权重及量化的缩放与zp值，当任务Loss即使面临这个量化误差的影响也能经学习达到比较低的Loss值时，在后面真正推理部署量化时，因为量化引入的误差早已在训练时被很好地适应了，所以只要能保证推理与训练时的计算完全对齐，理论上就保证了推理时量化不会有精度损失。

5.5.2 TPU-MLIR QAT实现方案及特点

1. 主体流程

当用户进行训练时，调用模型QAT量化API对训练模型进行修改：推理时OP融合后需要量化的OP的输入(包括权重与bias)前插入伪量化节点(可配置该节点的量化参数，例如per-chan/layer、是否对称、量化比特数等)，然后用户使用修改后的模型进行正常训练流程，完成少数几个轮次的训练后，调用转换部署API，将训练过的模型转换为fp32权重的ONNX模型，提取伪量化节点中的参数并导出到量化参数文本文件中，最后将调优后的ONNX模型与该量化参数文件输入TPU-MLIR工具链中，按前面讲述的训练后量化方式转换部署即可。

2. 方案特点

方案特点主要包括以下3点。

特点1：基于PyTorch。QAT是训练pipeline的一个附加微调环节，只有与训练环境深度集成才能方便用户在各种场景使用，考虑PyTorch具有最广泛的使用率，故目前方案仅基于PyTorch。若QAT后续要支持其他框架，则方案会大不相同，因为其trace、module替换等机制深度依赖原生训练平台的支持。

特点2：客户基本无感。区别于早期需人工深度介入模型转换的方案，本方案基于

PyTorch fx，能实现自动的完成模型 trace、伪量化节点插入、自定义模块替换等操作，在大多数情况下，客户使用默认配置即可一键式完成模型转换。

特点 3：基于商汤科技开源的 MQBench QAT 训练框架，已有一定的社区基础，方便工业界与学术界在 TPU 上进行推理性能与精度评估。

5.5.3 TPU-MLIR 环境配置方法

1. 从源码安装

TPU-MLIR 下载方法如下：

(1) 执行命令获取 GitHub 上的最新代码：git clone https://github.com/sophgo/MQBench。

(2) 进入 MQBench 目录后执行的命令如下：

```
pip install -r requirements.txt        # 当前要求的 torch 版本为 1.10.0
python setup.py install
```

(3) 执行命令 python -c 'import mqbench'，若没有返回任何错误，则说明安装正确；若安装有错，则可执行命令 pip uninstall mqbench 卸载后再尝试。

2. 安装 wheel 文件

从 https://MQBench-1.0.0-py3-none-any.whl 链接下载 Python whl 包，执行 pip3 install MQBench-1.0.0-py3-none-any.whl 直接安装即可。

5.5.4 QAT 示例化基本步骤

1. 接口导入及模型准备

在训练文件中添加如下 Python 模块 import 接口，代码如下：

```
//第 5 章/mlir_mqbench.py
from mqbench.prepare_by_platform import prepare_by_platform, BackendType
#初始化接口
from mqbench.utils.state import enable_calibration, enable_quantization
#校准与量化开关
from mqbench.convert_deploy importconvert_deploy
#转换部署接口
#使用 torchvision model zoo 里的预训练 ResNet18 模型
model = torchvision.models.__dict__[resnet18](pretrained = True)
Backend = BackendType.sophgo_tpu
#1.trace 模型，然后基于 sophgo_tpu 硬件的要求添加特定方式的量化节点
model_quantized = prepare_by_platform(model, Backend)
```

当上面接口选择 sophgo_tpu 后端时，该接口的第 3 个参数 prepare_custom_config_dict 默认不用配置，此时默认的量化配置如图 5-28 所示。

在图 5-28 中，sophgo_tpu 后端的 dict 中的各项从上到下依次如下：

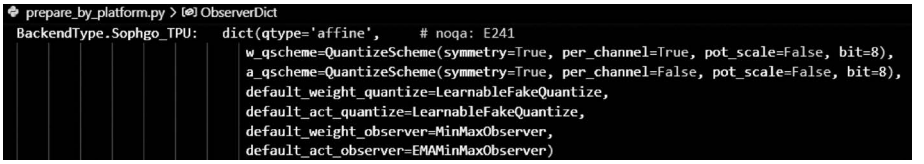


图 5-28　默认量化配置示例

(1) 权质量化方案为 per-chan 对称 8 位量化,缩放系数不是 power-of-2,而是任意的。

(2) 激活量化方案为 per-layer 非对称 8 位量化。

(3) 权重与激活伪量化方案均为 LearnableFakeQuantize,即 LQ 算法。

(4) 权重的动态范围统计及缩放计算方案为 MinMaxObserver,激活的为带 EMA 指数移动平均的 EMAMinMaxObserver。

步骤 1：用于量化参数初始化的校准及量化训练。

打开校准开关,允许在模型上推理时,用 PyTorch 观察对象来收集激活分布,并计算初始缩放,代码如下：

```
//第 5 章/mlir_enable_calibration.py
enable_calibration(model_quantized)
#校准循环
for i, (images, _) in enumerate(cali_loader):
    model_quantized(images) #只需前向推理
```

打开伪量化开关,在模型上推理时,调用 QuantizeBase 子对象来进行伪量化操作引入量化误差,代码如下：

```
//第 5 章/mlir_enable_calibration_1.py
enable_quantization(model_quantized)
#训练循环
for i, (images, target) in enumerate(train_loader):
    #前向推理并计算 loss
    output = model_quantized(images)
    loss = criterion(output, target)
    #后向反向梯度
    loss.backward()
    #更新权重与伪量化参数
    optimizer.step()
```

步骤 2：导出调优后的 fp32 模型及量化参数文件。

在调优量化过程中,batch-size 可根据需要调整,不必与训练 batch-size 一致,代码如下：

```
//第 5 章/mlir_convert.py
input_shape = {'data': [4, 3, 224, 224]}
```

导出前先融合 conv+bn 层(前面训练时未真正融合),将伪量化节点参数保存到参数文件,然后移除,代码如下：

```
//第 5 章/mlir_convert_1.py
convert_deploy(model_quantized, backend, input_shape)
```

步骤 3：启动训练。

设置好合理的训练超参数，建议如下：

```
//第 5 章/mlir_epochs.py
- epochs = 1:约在 1～3 即可
- lr = 1e - 4:学习率应该是 fp32 收敛时的学习率，甚至更低些
- optim = sgd:默认使用 sgd
```

步骤 4：转换部署。

使用 TPU-MLIR 的 model_transform.py 及 model_deploy.py 脚本完成到 sophg-tpu 硬件的转换部署。运行 example/ImageNet_example/main.py 文件对 ResNet18 进行 QAT 训练，命令如下：

```
//第 5 章/mlir_ImageNet_example.py
python3 ImageNet_example/main.py
     -- arch = resnet18
     -- batch - size = 192
     -- epochs = 1
     -- lr = 1e - 4
     -- gpu = 0
     -- pretrained
     -- backend = sophgo_tpu
     -- optim = sgd
     -- deploy_batch_size = 10
     -- train_data = /data/imagenet/for_train_val/
     -- val_data = /data/imagenet/for_train_val/
 -- output_path = /workspace/classify_models
```

2. TpuLang 接口

TpuLang 提供了 MLIR 对外的接口函数。用户通过 TpuLang 可以直接组建用户自己的网络，将模型转换为 TOP 层（芯片无关层）MLIR 模型（不包含 Canonicalize 部分，因此生成的文件名为 * _origin.MLIR）。这个过程会根据输入的接口函数，逐一创建并添加算子（OP），最终生成 MLIR 文件与保存权重的 npz 文件。

TPU-MLIR 工作流程如下。

（1）初始化：设置运行平台，创建模型 Graph。

（2）添加 OPS：循环添加模型的 OP。

（3）将输入参数转换为 dict 格式。

（4）推理 outputshape，并创建输出张量。

（5）设置张量的量化参数（scale，zero_point）。

（6）创建 op（op_type，inputs，outputs，params）并插入 Graph 中。

（7）设置模型的输入/输出张量，得到全部模型信息。

① 初始化 TpuLangconverter(initMLIRImporter)。

② generate_MLIR。

③ 依次创建输入算子、模型中间节点算子及返回算子,并将其补充到 MLIR 文本中(如果该算子带有权重,则会特定创建权重算子)。

(8) 输出。

① 将生成的文本转换为 str 并保存为.MLIR 文件。

② 将模型权重(tensors)保存为.npz 文件。

(9) 结束:释放 Graph。

TpuLang 转换的工作流程如图 5-29 所示(TpuLang 转换流程)。

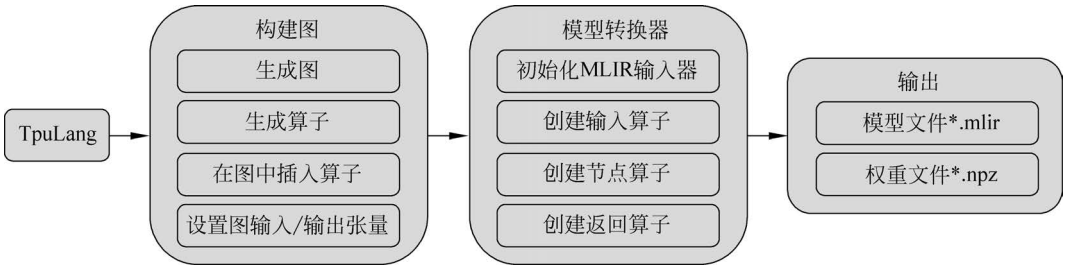


图 5-29　从 TpuLang 到构件图、模型转换器、输出的流程示例

3. 补充说明

操作接口需要操作的输入张量(前一个算子的输出张量或 Graph 的输入张量,coeff),包括以下几个模块:

(1) 根据接口提取的参数,推理获取 output_shape,即需要进行 shape_inference。

(2) 从接口中提取的属性。属性会通过 MLIRImporter 设定为与 TopOps.td 定义一一对应的属性。

(3) 如果接口中包括量化参数(scale,zero_point),则该参数对应的张量需要设置(或检查)量化参数。

(4) 返回该操作的输出张量(tensors)。

(5) 在所有算子都插入 Graph,并设置 Graph 的输入/输出 tensors 之后,才会启动转换到 MLIR 文本的工作。该部分由 TpuLang 转换器来实现。

(6) TpuLang 转换器转换流程与 ONNX 前端转换流程相同,具体参考(前端转换)。

从 TpuLang 到构件图、模型转换器、输出的流程示例,如图 5-29 所示。

4. 算子转换样例

以卷积算子为例,将单卷积算子模型转换为 TOP MLIR,单卷积模型原模型的定义如图 5-30 所示。

conv_v2 接口定义,代码如下:

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

图 5-30　将单卷积算子模型转换为 TOP MLIR 示例

```
//第 5 章/conv_v2.c
Def conv_v2(tensor_i,
            weight,
            bias = None,
            stride = None,
            dilation = None,
            pad = None,
            group = 1,
            input_zp = None,
            weight_zp = None,
            out_dtype = None,
            out_name = None):
   #pass
```

参数说明如下。

(1) tensor_i：张量类型，表示输入张量，4 维[N，C，H，W]格式。

(2) weight：张量类型，表示卷积核张量，4 维[oc，ic，kh，kw]格式，其中 oc 表示输出 channel 数，ic 表示输入 channel 数，kh 是 kernel_h，kw 是 kernel_w。

(3) bias：张量类型，表示偏置张量。当为 None 时表示无偏置，反之则要求 shape 为[1，oc，1，1]。

(4) dilation：List[int]，表示空洞大小，如果取 None，则表示[1，1]，当不为 None 时要求长度为 2。List 中的顺序为[长，宽]。

(5) pad：List[int]，表示填充大小，如果取 None，则表示[0，0，0，0]，当不为 None 时要求长度为 4。List 中的顺序为[上，下，左，右]。

(6) stride：List[int]，表示步长大小，如果取 None，则表示[1，1]，当不为 None 时要求长度为 2。List 中的顺序为[长，宽]。

(7) group：int 型，表示卷积层的组数。若 ic＝oc＝groups 时，则卷积为 depthwise 卷积。

(8) input_zp：List[int]型或 int 型，表示输入偏移。如果取 None，则表示 0，当取 List 时要求长度为 ic。

(9) weight_zp：List[int]型或 int 型，表示卷积核偏移。如果取 None，则表示 0，当取

List 时要求长度为 ic，其中 ic 表示输入的 channel 数。

（10）out_dtype：string 类型或 None，表示输出张量的类型。当输入张量类型为 float16/float32 时，取 None 表示输出张量类型与输入一致，否则取 None，表示为 int32。取值范围为/int32/uint32/float32/float16。

（11）out_name：string 类型或 None，表示输出张量的名称，当为 None 时内部会自动产生名称。

在 TopOps.td 文件中定义 Top.conv 算子。

1）构建 Graph

构建 Graph 图，包括以下几个模块。

（1）初始化模型：创建空 Graph。

（2）模型输入：给定 shape 与 data type 创建输入张量 x。此处也可以指定张量 name。

（3）conv_v2 接口：调用 conv_v2 接口，指定输入张量及输入参数。推理 outputshape，并生成输出张量，代码如下：

```
//第 5 章/shape_inference.c
def _shape_inference():
   kh_ext = dilation[0] * (weight.shape[2] - 1) + 1
   kw_ext = dilation[1] * (weight.shape[3] - 1) + 1
   oh = (input.shape[2] + pad[0] + pad[1] - kh_ext) //stride[0] + 1
   ow = (input.shape[3] + pad[2] + pad[3] - kw_ext) //stride[1] + 1
   return [input.shape[0], weight.shape[0], oh, ow]
output = tensor(_shape_inference(), dtype = out_dtype, name = out_name)
```

将输入参数打包成（卷积算子定义）定义的 attributes，代码如下：

```
//第 5 章/shape_inference_1.c
attr = {
   kernel_shape: ArrayAttr(weight.shape[2:]),
   strides: ArrayAttr(stride),
   dilations: ArrayAttr(dilation),
   pads: ArrayAttr(pad),
   do_relu: Attr(False, bool),
   group: Attr(group)
}
```

（4）插入卷积 OP，将 Top.convOp 插入 Graph 中。

（5）返回输出张量。

（6）设置 Graph 的输入，输出张量。

2）init_MLIRImporter

根据 input_names 与 output_names 从 shapes 中获取对应的 input_shape 与 output_shape，加上 model_name，生成初始的 MLIR 文本 MLIRImporter.MLIR_module，以及初始 MLIR 文本，如图 5-31 所示。

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

图 5-31　模块属性初始化示例

3）生成 MLIR 的示例

生成 MLIR 的示例包括以下几个步骤：

（1）build 输入 OP，生成的 Top. inputOp 会被插入 MLIRImporter. MLIR_module 中。

（2）调用 Operation. create 来创建 Top. convOp，而 create 函数需要的参数有以下几个。

① 输入 OP：从接口定义可知，卷积算子的 inputs 一共包含了 input、权重与 bias，输入 OP 已被创建好，权重与 bias 的 OP 则通过 getWeightOp()创建。

② output_shape：利用 Operator 中存储的输出张量获取其 shape。

③ Attributes：从 Operator 中获取 attributes，并将 attributes 转换为 MLIRImporter 识别的 Attributes。

Top. convOp 创建后会被插入 MLIR 文本中。

（3）根据 output_names 从 operands 中获取相应的 OP，创建 return_op 并插入 MLIR 文本中。到此为止，生成的完整的 MLIR 文本如图 5-32 所示。

```
#loc = loc(unknown)
module attributes {module.FLOPs = 109428480 : i64, module.chip = "ALL", module.name = "model_def", module.state = "TOP_F32", module.weigh
t_file = "model_def_top_f32_all_weight.npz"} {
  func.func @main(%arg0: tensor<1x3x173x141xf32> loc(unknown)) -> tensor<1x64x84x69xf32> {
    %0 = "top.Input"(%arg0) : (tensor<1x3x173x141xf32>) -> tensor<1x3x173x141xf32> loc(#loc1)
    %1 = "top.Weight"() : () -> tensor<64x1x7x7xf32> loc(#loc2)
    %2 = "top.Weight"() : () -> tensor<64xf32> loc(#loc3)
    %3 = "top.Conv"(%0, %1, %2) {dilations = [1, 1], do_relu = false, group = 1 : i64, kernel_shape = [7, 7], pads = [0, 0, 1, 1], relu_l
imit = -1.000000e+00 : f64, strides = [2, 2]} : (tensor<1x3x173x141xf32>, tensor<64x1x7x7xf32>, tensor<64xf32>) -> tensor<1x64x84x69xf32>
 loc(#loc4)
    return %3 : tensor<1x64x84x69xf32> loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("BMTensor0")
#loc2 = loc("BMTensor1")
#loc3 = loc("BMTensor2")
#loc4 = loc("BMTensor3")
```

图 5-32　模块属性初始化示例

（4）将 MLIR 文本保存为 conv_origin. MLIR，将 tensors 中的权重保存为 conv_TOP_F32_all_weight. npz。