

---

# TPU-MLIR 开发参考手册

发行版本 1.20-1-gd3964d047

SOPHGO

2025 年 06 月 30 日

# 目录

<b>1 TPU-MLIR 简介</b>	<b>3</b>
<b>2 开发环境配置</b>	<b>5</b>
2.1 代码下载 . . . . .	5
2.2 Docker 配置 . . . . .	5
2.3 ModelZoo(可选) . . . . .	6
2.4 代码编译 . . . . .	6
2.5 代码开发 . . . . .	7
<b>3 用户界面</b>	<b>8</b>
3.1 模型转换过程 . . . . .	8
3.1.1 支持图片输入 . . . . .	9
3.1.2 支持多输入 . . . . .	9
3.1.3 支持 INT8 对称和非对称 . . . . .	9
3.1.4 支持混精度 . . . . .	10
3.1.5 支持量化模型 TFLite . . . . .	10
3.1.6 支持 Caffe 模型 . . . . .	11
3.1.7 支持 LLM 模型 . . . . .	11
3.2 工具参数介绍 . . . . .	11
3.2.1 model_transform.py . . . . .	11
3.2.2 run_calibration.py . . . . .	13
3.2.3 model_deploy.py . . . . .	15
3.2.4 llm_convert.py . . . . .	17
3.2.5 model_runner.py . . . . .	18
3.2.6 npz_tool.py . . . . .	19
3.2.7 visual.py . . . . .	19
3.2.8 mlir2graph.py . . . . .	20
3.2.9 gen_rand_input.py . . . . .	20
3.2.10 model_tool . . . . .	22
<b>4 整体设计</b>	<b>25</b>
4.1 分层 . . . . .	25
4.2 Top Passes . . . . .	25
4.3 Tpu Passes . . . . .	27
4.4 Other Passes . . . . .	28
<b>5 前端转换</b>	<b>29</b>
5.1 主要工作 . . . . .	29

5.2	工作流程 . . . . .	29
5.3	算子转换样例 . . . . .	31
<b>6</b>	<b>量化</b>	<b>35</b>
6.1	基本概念 . . . . .	35
6.1.1	非对称量化 . . . . .	35
6.1.2	对称量化 . . . . .	36
6.2	Scale 转换 . . . . .	36
6.3	量化推导 . . . . .	37
6.3.1	Convolution . . . . .	37
6.3.2	InnerProduct . . . . .	37
6.3.3	Add . . . . .	38
6.3.4	AvgPool . . . . .	38
6.3.5	LeakyReLU . . . . .	39
6.3.6	Pad . . . . .	39
6.3.7	PReLU . . . . .	40
<b>7</b>	<b>Calibration</b>	<b>41</b>
7.1	总体介绍 . . . . .	41
7.2	默认流程介绍 . . . . .	41
7.3	校准数据筛选及预处理 . . . . .	45
7.3.1	筛选原则 . . . . .	45
7.3.2	输入格式及预处理 . . . . .	45
7.4	量化门限算法实现 . . . . .	46
7.4.1	kld 算法 . . . . .	46
7.4.2	auto-tune 算法 . . . . .	46
7.4.3	octav 算法 . . . . .	47
7.4.4	aciq 算法 . . . . .	48
7.5	优化算法实现 . . . . .	49
7.5.1	sq 算法 . . . . .	49
7.5.2	we 算法 . . . . .	49
7.5.3	bc 算法 . . . . .	51
7.5.4	search_threshold 算法 . . . . .	51
7.5.5	search_qtable 算法 . . . . .	52
7.6	示例-yolov5s 校准 . . . . .	52
7.7	可视化工具 visual 说明 . . . . .	58
<b>8</b>	<b>Lowering</b>	<b>61</b>
8.1	基本过程 . . . . .	61
8.2	混合精度 . . . . .	62
<b>9</b>	<b>SubNet</b>	<b>63</b>
<b>10</b>	<b>LayerGroup</b>	<b>64</b>
10.1	基本概念 . . . . .	64
10.2	BackwardH . . . . .	65
10.3	划分 Mem 周期 . . . . .	65
10.4	LMEM 分配 . . . . .	67

10.5 划分最优 Group . . . . .	69
<b>11 GMEM 分配</b>	<b>70</b>
11.1 目的 . . . . .	70
11.2 原理 . . . . .	70
11.2.1 weight tensor 分配 gmem . . . . .	70
11.2.2 global neuron tensors 分配 gmem . . . . .	71
<b>12 CodeGen</b>	<b>73</b>
12.1 主要工作 . . . . .	73
12.2 工作流程 . . . . .	73
12.3 TPU-MLIR 中 BM168X 及其相关类 . . . . .	74
12.4 后端函数的加载 . . . . .	75
12.5 后端 Store_cmd . . . . .	76
<b>13 MLIR 定义</b>	<b>78</b>
13.1 Top Dialect . . . . .	78
13.1.1 Operations . . . . .	78
<b>14 精度验证</b>	<b>96</b>
14.1 整体介绍 . . . . .	96
14.1.1 验证对象 . . . . .	96
14.1.2 评估指标 . . . . .	96
14.1.3 数据集 . . . . .	97
14.2 精度验证接口 . . . . .	97
14.3 精度验证样例 . . . . .	98
14.3.1 mobilenet_v2 . . . . .	98
14.3.2 yolov5s . . . . .	99
<b>15 QAT 量化感知训练</b>	<b>101</b>
15.1 基本原理 . . . . .	101
15.2 tpu-mlir QAT 实现方案及特点 . . . . .	101
15.2.1 主体流程 . . . . .	101
15.2.2 方案特点 . . . . .	102
15.3 安装方法 . . . . .	102
15.3.1 使用安装包安装 . . . . .	102
15.3.2 从源码安装 . . . . .	102
15.4 基本步骤 . . . . .	103
15.4.1 步骤 0: 接口导入及模型 prepare . . . . .	103
15.4.2 步骤 1: 用于量化参数初始化的校准及量化训练 . . . . .	104
15.4.3 步骤 2: 导出调优后的 fp32 模型及量化参数文件 . . . . .	104
15.4.4 步骤 3: 转换部署 . . . . .	104
15.5 使用样例-resnet18 . . . . .	105
15.6 QAT 测试环境 . . . . .	107
15.6.1 添加 cfg 文件 . . . . .	107
15.6.2 修改并执行 run_eval.py . . . . .	107
15.7 使用样例-yolov5s . . . . .	108
15.8 使用样例-bert . . . . .	109

<b>16 TpuLang 接口</b>	<b>110</b>
16.1 主要工作 . . . . .	110
16.2 工作流程 . . . . .	110
16.3 算子转换样例 . . . . .	111
16.4 Tpulang 接口使用方式 . . . . .	115
16.4.1 初始化 . . . . .	115
16.4.2 准备 Tensor . . . . .	116
16.4.3 构建 graph . . . . .	116
16.4.4 compile . . . . .	117
16.4.5 deinit . . . . .	117
16.4.6 deploy . . . . .	117
<b>17 用户自定义算子</b>	<b>118</b>
17.1 概述 . . . . .	118
17.2 自定义算子添加流程 . . . . .	119
17.2.1 TpuLang 自定义算子添加 . . . . .	119
17.2.2 Caffe 自定义算子添加 . . . . .	126
17.3 自定义算子示例 . . . . .	126
17.3.1 TpuLang 示例 . . . . .	126
17.3.2 Caffe 示例 . . . . .	130
17.4 自定义 AP (application processor) 算子添加流程 . . . . .	133
17.4.1 TpuLang 自定义 AP 算子添加 . . . . .	133
17.5 自定义 AP 算子示例 . . . . .	135
17.5.1 TpuLang 示例 . . . . .	135
<b>18 用 PPL 写后端算子</b>	<b>138</b>
18.1 如何编写和调用后端算子 . . . . .	138
18.2 PPL 集成到 TPU-MLIR 的流程 . . . . .	141
<b>19 final.mlir 截断方式</b>	<b>142</b>
19.1 final.mlir 结构介绍 . . . . .	142
19.2 final.mlir 截断流程 . . . . .	144
<b>20 MaskRCNN 大算子接口指南</b>	<b>146</b>
20.1 MaskRCNN 基础 . . . . .	146
20.1.1 模块快速分割方法 . . . . .	146
20.2 MaskRCNN 大算子 . . . . .	147
20.3 快速入门 . . . . .	147
20.3.1 准备您的 YAML . . . . .	147
20.3.2 模块单元测试 . . . . .	148
20.4 新前端接口 API . . . . .	148
20.4.1 [步骤 1] 运行 model_transform . . . . .	148
20.4.2 [步骤 2] 生成输入数据 . . . . .	148
20.4.3 [步骤 3] 运行 model_deploy . . . . .	149
20.5 IO_MAP 指南 . . . . .	150
20.5.1 [步骤 1] 描述模块接口 . . . . .	150
20.5.2 [步骤 2] 描述 IO_MAP . . . . .	154
20.5.3 IO_MAP 参数整理 . . . . .	157

20.6 mAP 推理 . . . . .	159
<b>21 LLMC 使用指南 . . . . .</b>	<b>160</b>
21.1 TPU-MLIR weight-only 量化 . . . . .	160
21.2 llmc_tpu . . . . .	160
21.2.1 环境准备 . . . . .	161
21.2.2 tpu 目录 . . . . .	161
21.2.3 操作步骤 . . . . .	161
<b>22 TPU Profile 工具使用及分析 . . . . .</b>	<b>166</b>
22.1 1. TPU 软件与硬件架构 . . . . .	166
22.2 2. 编译 bmodel . . . . .	167
22.3 3. 生成 Profile 原始数据 . . . . .	168
22.4 4. 可视化 Profile 数据 . . . . .	169
22.5 5. 结果分析 . . . . .	169
22.5.1 5.1 整体界面说明 . . . . .	169
22.5.2 5.2 Global Layer . . . . .	172
22.5.3 5.3 Local Layer Group . . . . .	173
22.6 6. 总结 . . . . .	175
<b>23 附录 01：从 NNTC 迁移至 TPU-MLIR . . . . .</b>	<b>176</b>
23.1 ONNX 模型导入 . . . . .	176
23.2 制作量化校准表 . . . . .	177
23.3 生成 int8 模型 . . . . .	178
<b>24 附录 02：TpuLang 的基本元素 . . . . .</b>	<b>179</b>
24.1 张量 (Tensor) . . . . .	179
24.2 张量前处理 (Tensor.preprocess) . . . . .	180
24.3 标量 (Scalar) . . . . .	181
24.4 Control Functions . . . . .	182
24.4.1 初始化函数 . . . . .	182
24.4.2 compile . . . . .	182
24.4.3 反初始化 . . . . .	183
24.4.4 重置默认图 . . . . .	184
24.4.5 获取当前默认图 . . . . .	184
24.4.6 重置图 . . . . .	184
24.4.7 舍入模式 . . . . .	184
24.5 Operator . . . . .	185
24.5.1 NN/Matrix Operator . . . . .	186
24.5.2 Base Element-wise Operator . . . . .	202
24.5.3 Element-wise Compare Operator . . . . .	214
24.5.4 Activation Operator . . . . .	226
24.5.5 Data Arrange Operator . . . . .	258
24.5.6 Sort Operator . . . . .	267
24.5.7 Shape About Operator . . . . .	271
24.5.8 Quant Operator . . . . .	275
24.5.9 Up/Down Scaling Operator . . . . .	283
24.5.10 Normalization Operator . . . . .	292

24.5.11 Vision Operator . . . . .	297
24.5.12 Select Operator . . . . .	302
24.5.13 Preprocess Operator . . . . .	310
24.5.14 Transform Operator . . . . .	311
24.5.15 Transform Operator . . . . .	315
24.5.16 Transform Operator . . . . .	316
24.5.17 Transform Operator . . . . .	319



## 法律声明

版权所有 © 算能 2025. 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

## 注意

您购买的产品、服务或特性等应受算能商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，算能对本文档内容不做任何明示或默示的声明或保证。由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

## 技术支持

### 地址

北京市海淀区丰豪东路 9 号院中关村集成电路设计园 (ICPARK)1 号楼

### 邮编

100094

### 网址

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

### 邮箱

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

### 电话

010-57590723

## 发布记录

## 目录

版本	发布日期	说明
v1.20.0	2025.06.30	支持 IO_RELOC 功能; Deconv3D INT8 精度问题修复; BatchNorm 和 Conv 反向算子支持 128 batch 训练
v1.19.0	2025.05.30	支持 AWQ 与 GPTQ 模型; 修复 Deconv3D F16, F32 精度问题
v1.18.0	2025.05.01	yolo 系列增加自动混精设置; run_calibration 增加 SmoothQuant 选择; 新增 llm 一键编译脚本
v1.17.0	2025.04.03	LLM 模型编译速度大幅提升; TPULang 支持 PPL 算子接入; 修复 Trilu bf16 在 Mars3 上随机出错问题
v1.16.0	2025.03.03	TPULang ROI_Extractor 支持; Einsum 支持 abcde,abfge->abcdgf 模式; LLMC 支持 Vila 模型
v1.15.0	2025.02.05	支持 LLMC 量化; codegen 地址越界判断; 修复若干对比问题
v1.14.0	2025.01.02	yolov8/v11 后处理融合支持; Conv3D stride 大于 15 支持; FAttention 精度提升
v1.13.0	2024.12.02	精简 Release 发布包; MaxPoolWithMask 训练算子性能优化; RoPE 大算子支持;
v1.12.0	2024.11.06	tpuv7-runtime cmodel 接入; BM1690 多核 LayerGroup 优化; 支持 PPL 编写后端算子
v1.11.0	2024.09.27	BM1688 tdb 增加 soc 模式; bmodel 支持细粒度合并; 修复若干性能下降问题
v1.10.0	2024.08.15	支持 yolov10; 增加量化调优章节; 优化 tpu-perf 日志打印
v1.9.0	2024.07.16	BM1690 新增 40 个模型回归测试; 量化算法新增 octav,aciq_guas 和 aciq_laplace
v1.8.0	2024.05.30	BM1690 支持多核 MatMul 算子; TPULang 支持输入输出顺序指定; tpuperf 移除 patchelf 依赖
v1.7.0	2024.05.15	CV186X 双核修改为单核; BM1690 测试流程与 BM1684X 一致; 支持 gemma/llama/qwen 等模型
v1.6.0	2024.02.23	添加了 Pypi 发布形式; 支持用户自定义 Global 算子; 支持了 CV186X 处理器平台
v1.5.0	2023.11.03	更多 Global Layer 支持多核并行;
v1.4.0	2023.09.27	系统依赖升级到 Ubuntu22.04; 支持了 BM1684 Winograd
v1.3.0	2023.07.27	增加手动指定浮点运算区域功能; 添加支持的前端框架算子列表; 添加 NNTC 与 TPU-MLIR 量化方式比较
v1.2.0	2023.06.14	调整了混合量化示例
v1.1.0	2023.05.26	添加使用智能深度学习处理器做后处理
v1.0.0	2023.04.10	支持 PyTorch, 增加章节介绍转 PyTorch 模型
v0.8.0	2023.02.28	添加使用智能深度学习处理器做前处理
v0.6.0	2022.11.05	增加章节介绍混精度操作过程
v0.5.0	2022.10.20	增加指定 model-zoo, 测试其中的所有模型
v0.4.0	2022.09.20	Copyright © 2022 SOFTICE CORPORATION 增加章节介绍转 Caffe 模型
v0.3.0	2022.08.24	支持 TFLite, 增加章节介绍转 TFLite 模型。
v0.2.0	2022.08.02	增加了运行 SDK 中的测试样例章节。
v0.1.0	2022.07.29	初版发布, 支持 resnet/mobilenet/vgg/ssd/yolov5s, 并用 yolov5s 作为用例。

# CHAPTER 1

---

## TPU-MLIR 简介

---

TPU-MLIR 是算能深度学习处理器的编译器工程。该工程提供了一套完整的工具链，可以将不同框架下预训练的神经网络，转化为可以在算能智能视觉深度学习处理器上高效运算的文件 bmodel。代码已经开源到 github: <https://github.com/sophgo/tpu-mlir>。

论文 <<https://arxiv.org/abs/2210.15016>> 描述了 TPU-MLIR 的整体设计思路。

TPU-MLIR 的整体架构如下：

目前直接支持的框架有 ONNX、Caffe 和 TFLite。其他框架的模型需要转换成 onnx 模型。如何将其他深度学习架构的网络模型转换成 onnx，可以参考 onnx 官网: <https://github.com/onnx/tutorials>。

模型转换需要在指定的 Docker 中执行，主要分两步，一是通过 `model_transform.py` 将原始模型转换成 mlir 文件，二是通过 `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`。

本文详细描述实现细节，用于指导开发。

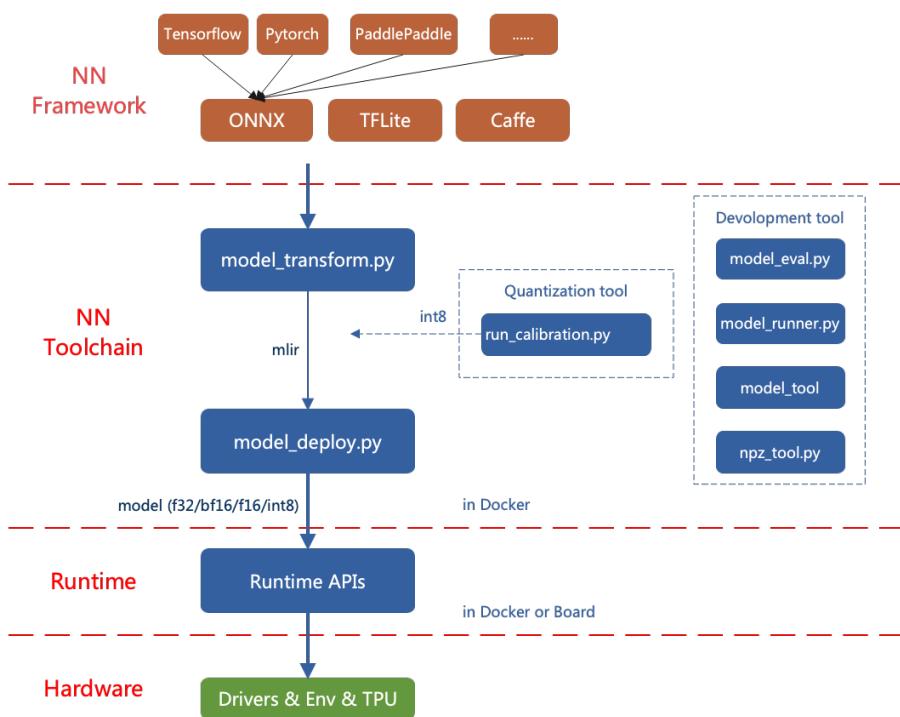


图 1.1: TPU-MLIR 整体架构

# CHAPTER 2

---

## 开发环境配置

---

本章介绍开发环境配置，代码在 Docker 中编译和运行。

### 2.1 代码下载

代码路径: <https://github.com/sophgo/tpu-mlir>

克隆该代码后，需要在 Docker 中编译。参考下文配置 Docker。

### 2.2 Docker 配置

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

从 DockerHub [https://hub.docker.com/r/sophgo/tpuc\\_dev](https://hub.docker.com/r/sophgo/tpuc_dev) 下载所需的镜像：

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

若下载失败，可从官网开发资料 <https://developer.sophgo.com/site/index/material/86/all.html> 下载所需镜像文件，或使用下方命令下载镜像：

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

如果是首次使用 Docker，可执行下述命令进行安装和配置（仅首次执行）：

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

确保安装包在当前目录, 然后在当前目录创建容器如下:

```
$ docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.4
# myname只是举个名字的例子, 请指定成自己想要的容器的名字
# 使用 --privileged 参数以获取root权限, 如果不需要root权限, 请删除该参数
```

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

## 2.3 ModelZoo(可选)

TPU-MLIR 中自带 yolov5s 模型, 如果要跑其他模型, 需要下载 ModelZoo, 路径如下:

<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

## 2.4 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, 如下:

```
# 本工程包含yolov5s.onnx模型, 可以直接用来验证
$ pushd regression
$ python run_model.py yolov5s
$ popd
```

如果要验证更多网络, 需要依赖 model-zoo, 回归时间比较久。

操作如下: (可选)

```
# 执行时间很长, 该步骤也可以跳过
$ pushd regression
$ ./run_all.sh
$ popd
```

## 2.5 代码开发

为了方便代码的阅读和开发，建议用 VSCode 编辑，在 VSCode 中需要安装这些插件：

- C/C++ Intellisense：用于 C++ 代码的智能提示和代码导航，以及代码格式化
- GitLens：用于 Git 版本控制和代码审查
- Python：用于 Python 代码的智能提示和代码导航
- yapf：用于 Python 代码格式化
- shell-format：用于 Shell 脚本格式化
- Remote-SSH：用于远程连接服务器上的代码（代码不在本地的情况下非常需要）

写完代码后右键点击格式化代码非常重要，保证代码的排版风格一致。

另外由于 TPU-MLIR 使用了 llvm-project，代码大量使用了它的头文件和库，建议安装 llvm-project，方便代码导航。操作如下：

1. 在 TPU-MLIR 的同级目录建立 third-party 目录，然后在该目录下克隆 llvm-project：

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

2. 在 TPU-MLIR 的 Docker 环境下，编译 llvm-project（编译过程中可能会提示缺失组件，根据提示安装即可）：

```
$ cd llvm-project
$ mkdir build && cd build
# 编译过程中可能会提示缺失组件，根据提示安装即可
# 比如如果提示缺失nanobind，就安装它pip3 install nanobind
$ cmake -G Ninja .. llvm \
-DLLVM_ENABLE_PROJECTS="mlir" \
-DLLVM_INSTALL_UTILS=ON \
-DLLVM_TARGETS_TO_BUILD="" \
-DLLVM_ENABLE_ASSERTIONS=ON \
-DMLIR_INCLUDE_TESTS=OFF \
-DLLVM_INSTALL_GTEST=ON \
-DMLIR_ENABLE_BINDINGS_PYTHON=ON \
-DCMAKE_BUILD_TYPE=DEBUG \
-DCMAKE_INSTALL_PREFIX=../install \
-DCMAKE_C_COMPILER=clang \
-DCMAKE_CXX_COMPILER=clang++ \
-DLLVM_ENABLE_LLD=ON
$ cmake --build . --target install
```

这样就可以在 VSCode 中关联到 llvm-project 的代码了。

# CHAPTER 3

## 用户界面

本章介绍用户的使用界面，包括转换模型的基本过程，和各类工具的使用方法。

### 3.1 模型转换过程

基本操作过程是用 `model_transform.py` 将模型转成 `mlir` 文件，然后用 `model_deploy.py` 将 `mlir` 转成对应的 `model`。以 `somenet.onnx` 模型为例，操作步骤如下：

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

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

### 3.1.1 支持图片输入

当用图片作为输入的时候, 需要指定预处理信息, 如下:

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

### 3.1.2 支持多输入

当模型有多输入的时候, 可以传入 1 个 npz 文件, 或者按顺序传入多个 npy 文件, 用逗号隔开。如下:

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

### 3.1.3 支持 INT8 对称和非对称

如果需要转 INT8 模型, 则需要进行 calibration。如下:

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

传入校准表生成模型, 如下:

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

### 3.1.4 支持混精度

当 INT8 模型精度不满足业务要求时, 可以尝试使用混精度, 先生成量化表, 如下:

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

然后将量化表传入生成模型, 如下:

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

### 3.1.5 支持量化模型 TFLite

支持 TFLite 模型的转换, 命令参考如下:

```
# 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 \
--scale 1.0,1.0,1.0 \
--pixel_format bgr \
--test_input ../image/dog.jpg \
--test_result resnet50_tf_top_outputs.npz \
--mlir resnet50_tf.mlir

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

### 3.1.6 支持 Caffe 模型

```
# Caffe转模型示例
$ 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 \
--scale 1.0,1.0,1.0 \
--pixel_format bgr \
--test_input ./image/dog.jpg \
--test_result resnet50_cf_top_outputs.npz \
--mlir resnet50_cf.mlir
```

### 3.1.7 支持 LLM 模型

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

## 3.2 工具参数介绍

### 3.2.1 model\_transform.py

用于将各种神经网络模型转换成 MLIR 文件 (.mlir``后缀) 以及配套的权重文件(``\$\{model\\_name\}\\_top\\_\\$\{quantize\}\\_all\\_weight.npz`), 支持的参数如下:

表 3.1: model\_transform 参数功能

参数名	必选 ?	说明
model_name	是	指定模型名称
model_def	是	指定模型定义文件, 比如 .onnx 或 .tflite 或 .prototxt 文件
mlir	是	指定输出的 mlir 文件名称和路径, .mlir 后缀
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
model_extern	否	其他模型定义文件, 用于与 model_def 模型合并 (目前主要用于 MaskRCNN 功能), 默认处理为 None; 多个输入模型文件时, 用 , 隔开

续下页

表 3.1 – 接上页

参数名	必选 ?	说明
model_data	否	指定模型权重文件, caffe 模型需要, 对应 .caffemodel 文件
input_types	否	当模型为 .pt 文件时指定输入的类型, 例如 int32; 多输入用 , 隔开; 不指定情况下默认处理为 float32。
keep_aspect_ratio	否	当 test_input 与 input_shapes 不同时, 在 resize 时是否保持长宽比, 默认为 false; 设置时会对不足部分补 0
mean	否	图像每个通道的均值, 默认为 0.0,0.0,0.0
scale	否	图片每个通道的比值, 默认为 1.0,1.0,1.0
pixel_format	否	图片类型, 可以是 rgb、bgr、gray、rgbd 四种情况, 默认为 bgr
channel_format	否	通道类型, 对于图片输入可以是 nhwc 或 nchw, 非图片输入则为 none, 默认是 nchw
output_names	否	指定输出的名称, 如果不指定, 则用模型的输出; 指定后按照该指定名称的顺序做输出
add_postprocess	否	将后处理融合到模型中, 指定后处理类型, 目前支持 yolov3、yolov3_tiny、yolov5、yolov8、yolov11、ssd、yolov8_seg 后处理
test_input	否	指定输入文件用于验证, 可以是 jpg 或 npy 或 npz; 可以不指定, 则不会正确性验证
test_result	否	指定验证后的输出文件, .npz 格式
excepts	否	指定需要排除验证的网络层的名称, 多个用 , 隔开
onnx_sim	否	onnx-sim 的可选项参数, 目前仅支持 skip_fuse_bn 选项, 用于关闭 batch_norm 和 Conv 层的合并
debug	否	保存可用于 debug 的模型
tolerance	否	模型转换的余弦与欧式相似度的误差容忍度, 默认为 0.99,0.99
cache_skip	否	是否在生成相同 mlir/bmodel 时跳过正确性的检查
dy-dynamic_shape_input_	否	具有动态 shape 的输入的名称列表, 例如 input1,input2. 如果设置了, model_deploy 需要设置参数' dynamic' .
shape_influencing_ir	否	在推理过程中会影响其他张量形状的输入的名称列表, 例如 input1,input2. 如果设置了, 则必须指定 test_input, 且 model_deploy 需要设置参数' dynamic' .
dynamic	否	该参数只对 onnx 模型有效. 如果设置了, 工具链会自动将模型带有 dynamic_axis 的输入加入 dynamic_shape_input_names 列表中, 将模型中 1 维的输入加入 shape_influencing_input_names 列表中, 且 model_deploy 需要设置参数' dynamic' .
resize_dims	否	预处理前的原始输入图像尺寸 h,w, 默认为模型原始输入尺寸
pad_value	否	图片缩放时边框填充大小
pad_type	否	图片缩放时的填充类型, 有 normal, center

续下页

表 3.1 – 接上页

参数名	必选 ?	说明
preprocess_list	否	输入是否需要做预处理的选项, 例如: '1,3' 表示输入 1&3 需要进行预处理, 缺省代表所有输入要做预处理
path_yaml	否	单个 yaml 文件的路径 (当前主要用于 MaskRCNN 参数配置)
enable_maskrcnn	否	是否启用 MaskRCNN 大算子.
yuv_type	否	采用'.yuv' 文件作为输入时指定其类型

转成 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。

注意:

1. model\_transform.py 阶段输入的预处理参数会用于对 test\_input 进行预处理, 并且会记录到 mlir 文件中, 后续执行 run\_calibration.py 时会读取该预处理参数对校准数据集进行预处理。若 model\_transform.py 阶段没有对应参数输入将可能影响到模型的实际量化效果。
2. 预处理参数计算方式:

$$input = scale \times (input - mean)$$

$$scale = \frac{1}{255 \times std}$$

### 3.2.2 run\_calibration.py

用少量的样本做 calibration, 得到网络的校准表, 即每一层 op 的 threshold/min/max。

支持的参数如下:

表 3.2: run\_calibration 参数功能

参数名	必选 ?	说明
无	是	指定 mlir 文件
sq	否	SmoothQuant
we	否	跨层权重均衡
bc	否	偏差校正
dataset	否	指定输入样本的目录, 该路径放对应的图片, 或 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
inference_num	否	search_threshold 和 search_qtable 过程中所需推理图片数量, 通常小于 input_num
bc_inference_num	否	偏差校正过程中所需推理图片数量
tune_num	否	指定微调样本数量, 默认为 10
tune_list	否	指定微调样本文件

续下页

表 3.2 – 接上页

参数名	必选 ?	说明
histogram_bin_num	否	直方图 bin 数量, 默认 2048
expected_cos	否	期望 search_qtable 混精模型输出与浮点模型输出的相似度, 取值范围 [0,1]
min_layer_cos	否	bias_correction 中该层量化输出与浮点输出的相似度下限, 当低于该下限时需要对该层进行补偿, 取值范围 [0,1]
max_float_layers	否	search_qtable 浮点层数量
processor	否	处理器类型
cali_method	否	选择量化门限计算方法
fp_type	否	search_qtable 浮点层数据类型
post_process	否	后处理路径
global_compare_layers	否	指定全局对比层, 例如 layer1,layer2 或 layer1:0.3,layer2:0.7
search	否	指定搜索类型, 其中包括 search_qtable,search_threshold,false。其中默认为 false, 不开启搜索
transformer	否	是否是 transformer 模型,search_qtable 中如果是 transformer 模型可分配指定加速策略
quantize_method_list	否	search_qtable 用来搜索的门限方法
benchmark_method	否	指定 search_threshold 中相似度计算方法
kurtosis_analysis	否	指定生成各层激活值的 kurtosis
part_quantize	否	指定模型部分量化, 获得 cali_table 同时会自动生成 qtable。可选择 N_mode,H_mode,custom_mode,H_mode 通常精度较好
custom_operator	否	指定需要量化的算子, 配合开启上述 custom_mode 后使用
part_asymmetric	否	指定当开启对称量化后, 模型某些子网符合特定 pattern 时, 将对应位置算子改为非对称量化
mix_mode	否	指定 search_qtable 特定的混精类型, 目前支持 8_16 和 4_8 两种
cluster	否	指定 search_qtable 寻找敏感层时采用聚类算法
quantize_table	否	search_qtable 输出的混精度量化表
o	是	输出 calibration table 文件
debug_cmd	否	debug cmd
debug_log	否	日志输出级别

校准表的样板如下:

```
# genetated time: 2022-08-11 10:00:59.743675
# histogram number: 2048
# sample number: 100
# tune number: 5
###
```

(续下页)

(接上页)

```
# op_name threshold min max
images 1.0000080 0.0000000 1.0000080
122_Conv 56.4281803 -102.5830231 97.6811752
124_Mul 38.1586478 -0.2784646 97.6811752
125_Conv 56.1447888 -143.7053833 122.0844193
127_Mul 116.7435987 -0.2784646 122.0844193
128_Conv 16.4931355 -87.9204330 7.2770605
130_Mul 7.2720342 -0.2784646 7.2720342
.....
```

它分为 4 列: 第一列是 Tensor 的名字; 第二列是阈值 (用于对称量化); 第三列第四列是 min/max, 用于非对称量化。

### 3.2.3 model\_deploy.py

将 mlir 文件转换成相应的 model, 参数说明如下:

表 3.3: model\_deploy 参数功能

参数名	必选 ?	说明
mlir	是	指定 mlir 文件
processor	是	指定模型将要用到的平台, 支持 BM1684, BM1684X, BM1688, BM1690, CV186X, CV183X, CV182X, CV181X, CV180X
quantize	是	指定默认量化类型, 支持 F32/F16/BF16/INT8 等, 不同处理器支持的量化类型如下表所示。
quant_input	否	指定输入数据类型是否与量化类型一致, 例如 int8 模型指定 quant_input, 那么输入数据类型也为 int8, 若不指定则为 F32
quant_output	否	指定输出数据类型是否与量化类型一致, 例如 int8 模型指定 quant_input, 那么输出数据类型也为 int8, 若不指定则为 F32
quant_input_list	否	选择要转换的索引, 例如 1,3 表示第一个和第三个输入的强制转换
quant_output_list	否	选择要转换的索引, 例如 1,3 表示第一个和第三个输出的强制转换
quantize_table	否	指定混精度量化表路径, 如果没有指定则按 quantize 类型量化; 否则优先按量化表量化
fuse_preprocess	否	指定是否将预处理融合到模型中, 如果指定了此参数, 则模型输入为 uint8 类型, 直接输入 resize 后的原图即可
calibration_table	否	指定校准表路径, 当存在 INT8/F8E4M3 量化的时候需要校准表
high_precision	否	打开时一部分算子会固定用 float32
tolerance	否	表示 MLIR 量化后的结果与 MLIR fp32 推理结果余弦与欧式相似度的误差容忍度, 默认为 0.8,0.5

续下页

表 3.3 – 接上页

参数名	必选 ?	说明
test_input	否	指定输入文件用于验证, 可以是 jpg 或 npy 或 npz; 可以不指定, 则不会正确性验证
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, 隔开
op_divide	否	CV183x/CV182x/CV181x/CV180x only, 尝试将较大的 op 拆分为多个小 op 以达到节省 ion 内存的目的, 适用少数特定模型
model	是	指定输出的 model 文件名称和路径
debug	否	是否保留中间文件
asymmetric	否	指定做 int8 非对称量化
dynamic	否	动态编译
includeWeight	否	tosa.mlir 的 includeWeight
customization_format	否	指定模型输入帧的像素格式
compare_all	否	指定对比模型所有的张量
num_device	否	用于并行计算的设备数量, 默认 1
num_core	否	用于并行计算的智能视觉深度学习处理器核心数量, 默认 1
skip_validation	否	跳过检查 bmodel 的正确性
merge_weight	否	将权重与之前生成的 cvimodel 合并为一个权重二进制文件, 默认否
model_version	否	如果需要旧版本的 cvimodel, 请设置版本, 例如 1.2, 默认 latest
q_group_size	否	每组定量的组大小, 仅用于 W4A16/W8A16 定量模式, 默认 0
q_symmetric	否	指定做 W4A16 对称量化, 仅用于 W4A16/W8A16 定量模式
compress_mode	否	指定模型的压缩模式: "none", "weight", "activation", "all"。支持 bm1688, 默认为"none", 不进行压缩
opt_post_processor	否	是否对 LayerGroup 的结果继续图优化, 支持 mars3, 默认为"none", 不进行
lgcache	否	指定是否暂存 LayerGroup 的切分结果: "true", "false"。默认为"true", 将每个子网的切分结果保存到工作目录 "cut_result_{subnet_name}.mlircache"
cache_skip	否	是否在生成相同 mlir/bmodel 时跳过正确性的检查
aligned_input	否	是否输入图像的宽/通道是对齐的, 仅用于 CV 系列处理器的 VPSS 输入对齐
group_by_cores	否	layer groups 是否根据 core 数目进行强制分组, 可选 auto/true/false, 默认为 auto

续下页

表 3.3 – 接上页

参数名	必选 ?	说明
opt	否	LayerGroup 优化类型, 可选 1/2/3, 默认为 2。1: 简单 LayerGroup 模式, 所有算子会尽可能做 Group, 编译速度较快; 2: 通过动态编译计算全局 cycle 最优的 Group 分组, 适用于推理图编译; 3: 线性规划 LayerGroup 模式, 适用于模型训练图编译。
addr_mode	否	设置地址分配模式 [ ‘auto’ , ‘basic’ , ‘io_alone’ , ‘io_tag’ , ‘io_tag_fuse’ , ‘io_reloc’ ], 默认为 auto
dis- able_layer_group	否	是否关闭 LayerGroup
dis- able_gdma_check	否	是否关闭 gdma 地址检查
do_winograd	否	是否使用 WinoGrad 卷积, 仅用于 BM1684 平台
mat- mul_perchannel	否	MatMul 是否使用 per-channel 量化模式, 目前支持 BM1684X 和 BM1688 处理器, 打开可能影响运行时间
enable_maskrcnn	否	是否启用 MaskRCNN 大算子.

对于不同处理器和支持的量化类型对应关系如下表所示:

表 3.4: 不同处理器支持的 quantize 量化类型

处理器	支持的 quantize
BM1684	F32, INT8
BM1684X	F32, F16, BF16, INT8, W4F16, W8F16, W4BF16, W8BF16
BM1688	F32, F16, BF16, INT8, INT4, W4F16, W8F16, W4BF16, W8BF16
BM1690	F32, F16, BF16, INT8, F8E4M3, F8E5M2, W4F16, W8F16, W4BF16, W8BF16
CV186X	F32, F16, BF16, INT8, INT4
CV183X, CV182X	CV182X, BF16, INT8
CV181X, CV180X	

其中, W4A16 与 W8A16 的 Weight-only 量化模式仅作用于 MatMul 运算, 其余算子根据实际情况仍会进行 F16 或 BF16 量化。

### 3.2.4 llm\_convert.py

用于将 HuggingFace LLM 模型转换成 bmodel, 支持的参数如下:

表 3.5: llm\_convert 参数功能

参数名	必选 ?	说明
model_path	是	指定模型路径
seq_length	是	指定序列最大长度
quantize	是	指定量化类型, 如 w4bf16/w4f16/bf16/f16
q_group_size	否	每组量化的组大小
chip	是	指定处理器类型, 支持 bm1684x/bm1688/cv186ah
max_pixels	否	多模态参数, 指定最大尺寸, 可以是 672,896, 也可以是 602112
num_device	否	指定 bmodel 部署的设备数
num_core	否	指定 bmodel 部署使用的核数, 0 表示采用最大核数
embedding_disk	否	如果设置该标志, 则将 word_embedding 导出为二进制文件, 并通过 CPU 进行推理
out_dir	是	指定输出的 bmodel 文件保存路径

### 3.2.5 model\_runner.py

对模型进行推理, 支持 mlir/pytorch/onnx/tflite/bmodel/prototxt。

执行参考如下:

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

支持的参数如下:

表 3.6: model\_runner 参数功能

参数名	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 mlir/pytorch/onnx/tflite/bmodel/prototxt
dump_all_tensors	否	开启后对导出所有的结果, 包括中间 tensor 的结果
out_fixed	否	开启后当出现 int8 类型定点数时不再自动转成 float32 类型进行打印

### 3.2.6 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等。npz\_tool.py 用于处理 npz 文件。

执行参考如下:

```
# 查看sample_out.npz中output的数据
$ npz_tool.py dump sample_out.npz output
```

支持的功能如下:

表 3.7: npz\_tool 功能

功能	描述
dump	得到 npz 的所有 tensor 信息
compare	比较 2 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

### 3.2.7 visual.py

量化网络如果遇到精度对比不过或者比较差, 可以使用此工具逐层可视化对比浮点网络和量化后网络的不同, 方便进行定位和手动调整。

执行命令可参考如下:

```
# 以使用9999端口为例
$ visual.py \
--f32_mlir netname.mlir \
--quant_mlir netname_int8_sym_tpu.mlir \
--input top_input_f32.npz --port 9999
```

支持的功能如下:

表 3.8: 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 10000, 需要在启动 docker 时映射至系统端口
host	使用的 host ip 地址, 默认 0.0.0.0
manual_run	启动后是否自动进行网络推理比较, 默认 False, 会自动推理比较

注意: 需要在 model\_deploy.py 阶段打开 --debug 选项保留中间文件供 visual.py 使用, 工具的详细使用说明见 ([可视化工具 visual 说明](#))。

### 3.2.8 mlir2graph.py

基于 dot 对 mlir 文件可视化，支持所有阶段的 mlir 文件。执行后会在 mlir 对应目录生成对应的.dot 文件和.svg 文件。其中.dot 文件可以基于 dot 渲染成其他格式的命令。.svg 是默认输出的渲染格式。可以直接在浏览器打开。

执行命令可参考如下：

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

对较大的 mlir 文件，dot 文件使用原始的渲染算法可能会消耗较长的时间，可以添加`--is_big`参数，会减少算法的迭代时间，出图更快：

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

支持的功能如下：

表 3.9: mlir2graph 功能

功能	描述
mlir	任意 mlir 文件
is_big	是否是比较大的 mlir 文件，没有明确指标，一般靠人为根据渲染用时判断
failed_keys	对比失败的节点名列表，多个用“,”隔开，在渲染后对应节点会渲染为红色
bmodel_checker_dir	使用 bmodel_checker.py 生成的 failed.npz 文件路径，当指定该路径时，会自动解析错误节点，并将对应节点渲染为红色
output	输出文件的路径，默认为 <code>-mlir</code> 的路径加格式后缀，如 <code>netname.mlir.dot</code> / <code>netname.mlir.svg</code>

### 3.2.9 gen\_rand\_input.py

在模型转换时，如果不想额外准备测试数据(`test_input`)，可以使用此工具生成随机的输入数据，方便模型验证工作。

基本操作过程是用`model_transform.py`将模型转成 mlir 文件，此步骤不进行模型验证；接下来，用`gen_rand_input.py`读取上一步生成的 mlir 文件，生成用于模型验证的随机测试数据；最后，再次使用`model_transform.py`进行完整的模型转换和验证工作。

执行的命令可参考如下：

```
# 模型初步转换为mlir文件
$ model_transform.py \
--model_name yolov5s \
--model_def ../regression/model/yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
```

(续下页)

(接上页)

```
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 350,498,646 \
--mlir yolov5s.mlir

# 生成随机测试数据，这里生成的是伪测试图片
$ gen_rand_input.py \
--mlir yolov5s.mlir \
--img --output yolov5s_fake_img.png

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

更详细的使用方法可参考如下：

```
# 可为多个输入分别指定取值范围
$ gen_rand_input.py \
--mlir ernie.mlir \
--ranges [[0,300],[0,0]] \
--output ern.npz

# 可为输入指定type类型，如不指定，默认从mlir文件中读取
$ gen_rand_input.py \
--mlir resnet.mlir \
--ranges [[0,300]] \
--input_types si32 \
--output resnet.npz

# 指定生成随机图片，不指定取值范围和数据类型
$ gen_rand_input.py \
--mlir yolov5s.mlir \
--img --output yolov5s_fake_img.png
```

支持的功能如下：

表 3.10: gen\_rand\_input 功能

参数名	必选 ?	说明
mlir	是	指定输出的 mlir 文件名称和路径, .mlir 后缀
img	否	用于 CV 任务生成随机图片, 否则生成 npz 文件。默认图片的取值范围为 [0,255], 数据类型为' uint8 ', 不通过' ranges ' 或' input_types ' 更改。
ranges	否	指定模型输入的取值范围, 以列表形式表现, 如 [[0,300],[0,0]]。如果指定生成图片, 则不需要指定取值范围, 默认 [0,255]。其他情况下, 需要指定取值范围。
input_types	否	指定模型输入的数据类型, 如 si32,f32。目前仅支持 'si32' 和 'f32' 类型。如果不填, 默认从 mlir 中读取。如果指定生成图片, 则不需要指定数据类型, 默认 ' uint8 '。
output	是	指定输出的名称

注意: CV 相关模型通常会对输入图片进行一系列预处理, 为保证模型正确性验证通过, 需要用' -img ' 生成随机图片作为输入, 不能使用随机 npz 文件作为输入。值得关注的是, 随机输入可能会引起模型正确性验证对比不通过, 特别是 NLP 相关模型, 因此建议优先使用真实的测试数据。

### 3.2.10 model\_tool

该工具用于处理最终的模型文件” bmodel ” 或者” cvimodel ”, 所有参数及对应功能描述可通过执行以下命令查看:

```
$ model_tool
```

以下均以” xxx.bmodel ” 为例, 介绍该工具的主要功能。

1) 查看 bmodel 的基本信息

执行参考如下:

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

显示模型的基本信息, 包括模型的编译版本, 编译日期, 模型中网络名称, 输入和输出参数等等。显示效果如下:

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

kernel_module name: libbm1684x_kernel_module.so
kernel_module size: 3136888
=====
net 0: [block_0] static
-----
```

(续下页)

(接上页)

```

stage 0:
input: input_states, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
input: position_ids, [1, 512], int32, scale: 1, zero_point: 0
input: attention_mask, [1, 1, 512, 512], bfloat16, scale: 1, zero_point: 0
output: /layer/Add_1_output_0_Add, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Add_1_output_0_Add, [1, 1, 512, 256], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Transpose_2_output_0_Transpose, [1, 1, 512, 256], bfloat16, scale: 1, [F]
→zero_point: 0
=====
net 1: [block_1] static
-----
stage 0:
input: input_states, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
input: position_ids, [1, 512], int32, scale: 1, zero_point: 0
input: attention_mask, [1, 1, 512, 512], bfloat16, scale: 1, zero_point: 0
output: /layer/Add_1_output_0_Add, [1, 512, 2048], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Add_1_output_0_Add, [1, 1, 512, 256], bfloat16, scale: 1, zero_point: 0
output: /layer/self_attn/Transpose_2_output_0_Transpose, [1, 1, 512, 256], bfloat16, scale: 1, [F]
→zero_point: 0

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

```

## 2) 合并多个 bmodel

执行参考如下:

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

将多个 bmodel 合并成一个 bmodel, 如果 bmodel 中存在同名的网络, 则会分不同的 stage。

## 3) 分解成多个 bmodel

执行参考如下:

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

将一个 bmodel 分解成多个 bmodel, 与 combine 命令是相反的操作。

## 4) 显示权重信息

执行参考如下:

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

显示不同网络的各个算子的权重范围信息, 显示效果如下:

```

net 0 : "block_0", stage:0
-----
tpu.Gather : [0x0, 0x40000)
tpu.Gather : [0x40000, 0x80000)
tpu.RMSNorm : [0x80000, 0x81000)

```

(续下页)

(接上页)

```

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

### 5) 更新权重

执行参考如下:

```
# 将src.bmodel中网络名为src_net的网络，在0x2000位置的权重，更新到dst.bmodel的dst_net的0x1000位置
$ model_tool --update_weight dst.bmodel dst_net 0x1000 src.bmodel src_net 0x2000
```

可以实现将模型权重进行更新。比如某个模型的某个算子权重需要更新，则将该算子单独编译成 bmodel，然后将其权重更新到原始的模型中。

### 6) 模型加密与解密

执行参考如下:

```
# -model输入combine后的模型或正常bmodel，-net输入要加密的网络，-lib实现具体的加密算法，-o输出加密后模型的名称
$ model_tool --encrypt -model combine.bmodel -net block_0 -lib libcipher.so -o encrypted.bmodel
$ model_tool --decrypt -model encrypted.bmodel -lib libcipher.so -o decrypted.bmodel
```

可以实现将模型的权重、flatbuffer 结构化数据、header 都进行加密。加解密接口必须按照 C 风格来实现，不能使用 C++，接口规定如下:

```
extern "C" uint8_t* encrypt(const uint8_t* input, uint64_t input_bytes, uint64_t* output_bytes);
extern "C" uint8_t* decrypt(const uint8_t* input, uint64_t input_bytes, uint64_t* output_bytes);
```

# CHAPTER 4

---

## 整体设计

---

### 4.1 分层

TPU-MLIR 将网络模型的编译过程分两层处理:

#### Top Dialect

与硬件无关层, 包括图优化、量化、推理等等

#### Tpu Dialect

与硬件相关层, 包括权重重排、算子切分、地址分配、推理等等

整体的流程如 ([TPU-MLIR 整体流程](#)) 图中所示, 通过 Pass 将模型逐渐转换成最终的指令, 这里具体说明 Top 层和 Tpu 层每个 Pass 的功能。后面章节会对每个 Pass 的关键点做详细说明。

### 4.2 Top Passes

#### shape-infer

做 shape 推导, 包括常量折叠。对于 shape 不确定的 op, 在这里确定 shape。

#### canonicalize

与具体 op 有关的图优化, 比如 relu 合并到 conv、shape 合并等等。

#### extra-optimize

额外的 pattern 实现, 比如求 FLOPs、去除无效输出等等。

#### processor-assign

配置处理器, 如 BM1684X 或者 CV183X 等等; 并根据处理器对 top 层进行调整, 比如 CV18XX 将输入全部调整为 F32。

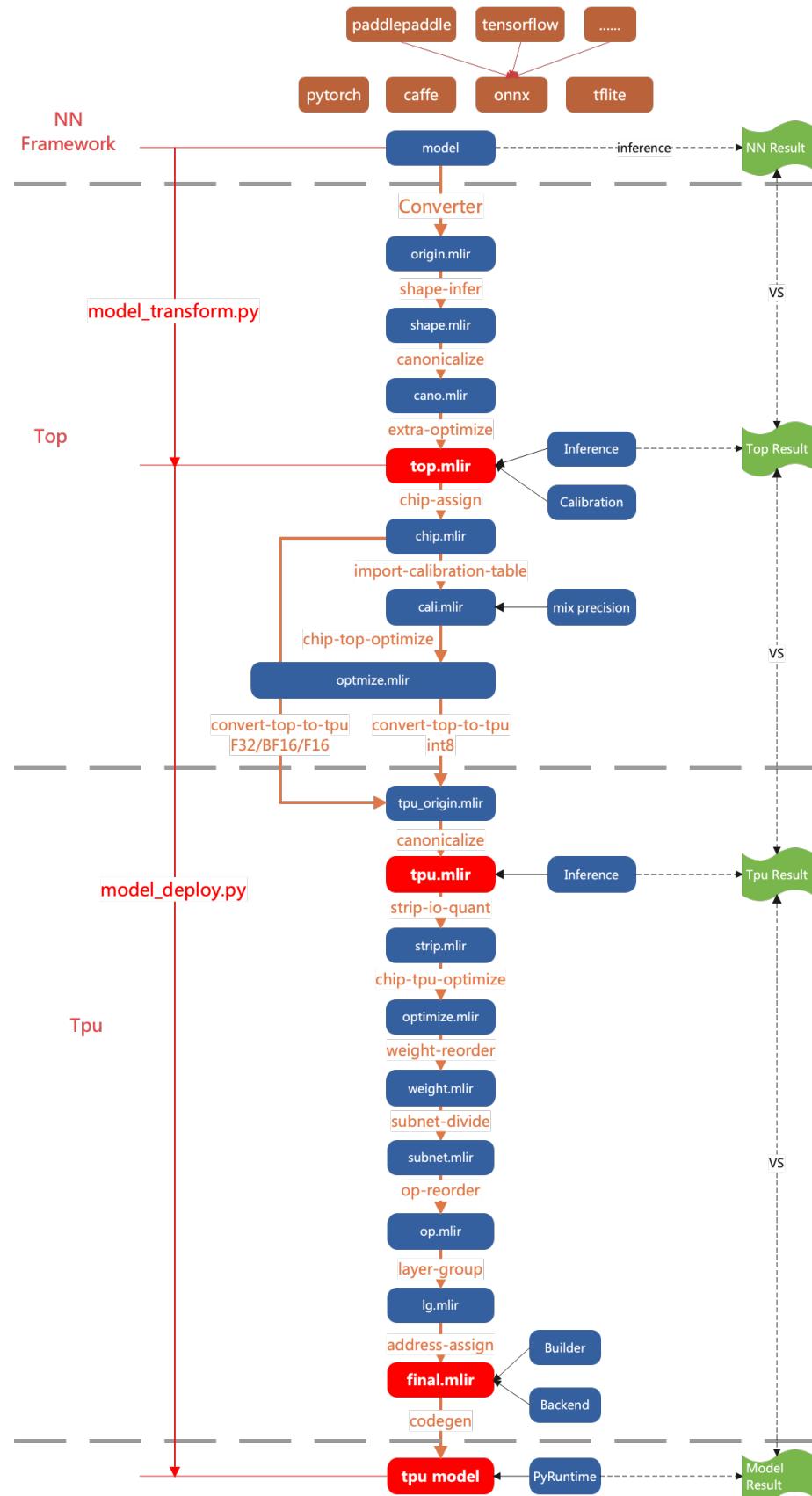


图 4.1: TPU-MLIR 整体流程  
Copyright © SOPHGO

**import-calibration-table**

按照 calibration table, 给每个 op 插入 min 和 max, 用于后续量化; 对应对称量化则插入 threshold

**processor-top-optimize**

与处理器相关的 top 层算子优化, 这是一个权衡, 有些 top 算子与处理器具有相关性

**convert-top-to-tpu**

将 top 层下沉到 tpu 层; 如果是浮点类型 (F32/F16/BF16), top 层 op 基本上直接转换成相应的 tpu 层 op 即可; 如果是 INT8 类型, 则需要量化转换

## 4.3 Tpu Passes

**canonicalize**

与 tpu 层具体 op 有关的图优化, 比如连续 Requant 的合并等等

**strip-io-quant**

决定输入或输出是否是量化类型, 否则就是默认 F32 类型

**processor-tpu-optimize**

与处理器相关的 tpu 层算子优化

**weight-reorder**

根据硬件特征对个别 op 的权重进行重新排列, 比如卷积的 filter 和 bias

**subnet-divide**

将网络按照处理器类型切分成不同的子网络, 如果所有算子都是 TPU, 则子网络只有一个

**op-reorder**

对 op 进行顺序调整, 让使用者离被使用者尽可能的靠近; 也有针对 attention 一类操作做特殊处理

**layer-group**

对网络进行切分, 使尽可能多的 op 在 local mem 中连续计算

**address-assign**

给需要 global mem 的 op 分配地址

**codegen**

执行 op 的 codegen 接口, 生成 cmdbuf。并用 Builder 模块采用 flatbuffers 格式生成最终的模型

## 4.4 Other Passes

还有一些可选的 pass 没有再图中标出来, 用于实现特定功能。

### **fuse-preprocess**

用于预处理融合, 对于图片类输入, 将图片的预处理过程合并到模型中

### **add-postprocess**

用于将 ssd 或 yolo 的后处理合并到模型中

# CHAPTER 5

---

## 前端转换

---

本章以 onnx 模型为例介绍模型/算子在本工程中的前端转换流程。

### 5.1 主要工作

前端主要负责将原始模型转换为 Top 层 (硬件无关层)mlir 模型的工作 (不包含 Canonicalize 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。

### 5.2 工作流程

1. 前提 (Prereq): Top 层算子定义, 该部分内容保存在 TopOps.td 文件
  2. 输入 (Input): 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)
    - load\_onnx\_model 部分主要是对模型进行精简化, 根据 arguments 中的 output\_names 截取模型, 并提取精简后模型的相关信息
    - init\_MLIRImporter 部分主要是生成初始的 mlir 文本
  4. generate\_mlir
    - 依次创建 input op, 模型中间 nodes op 以及 return op, 并将其补充到 mlir 文本中 (如果该 op 带有权重, 则会额外创建 weight op)
  5. 输出 (Output)
-

- 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
- 将生成的文本转为字符串并保存为 “.mlir” 文件
- 将模型权重 (tensors) 保存为 “.npz” 文件

前端转换的工作流程如图所示 (前端转换流程)。

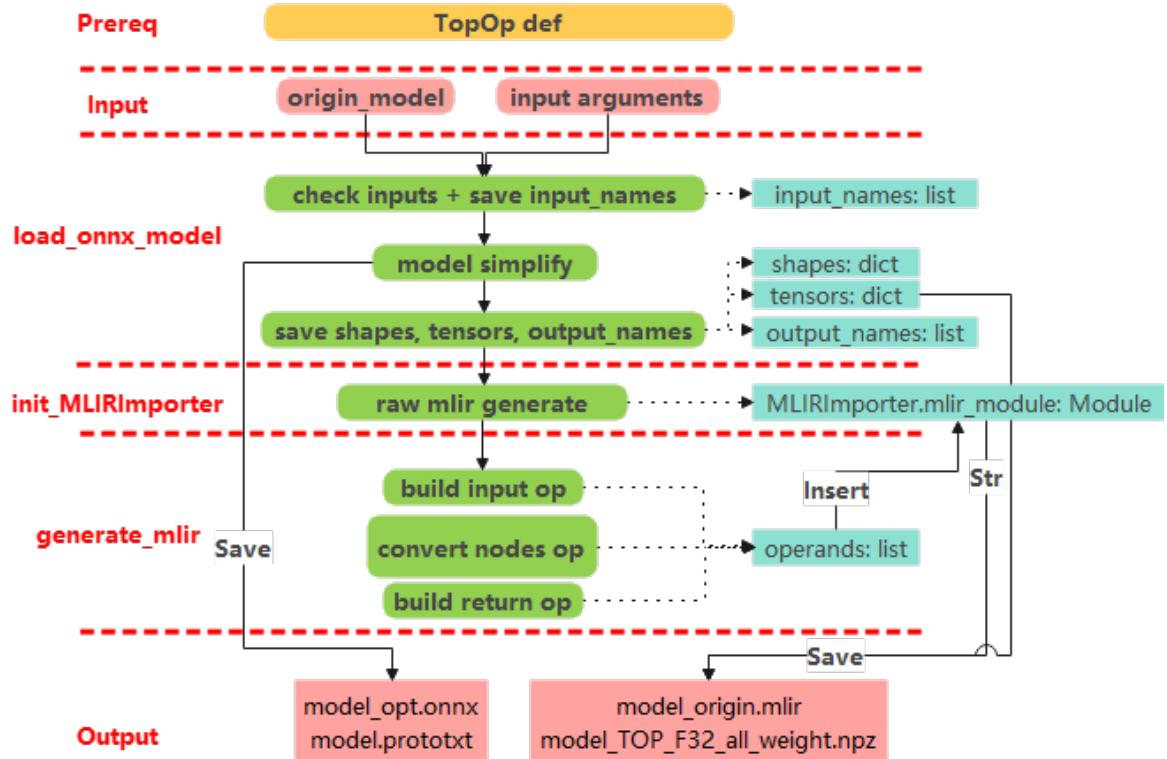


图 5.1: 前端转换流程

### 补充说明:

- Build input op 需要:
  1. `input_names`
  2. 每个 `input` 对应的 `index`
  3. 预处理参数 (若输入为图像)
- Convert nodes op 需要:
  1. 从 `operands` 获取该 node 的输入 op(即前一个已经 build 或 convert 好的算子)
  2. 从 `shapes` 中获取 `output_shape`
  3. 从 `onnx node` 中提取的 `attrs`。Attrs 会通过 `MLIRImporter` 设定为与 `TopOps.td` 定义一一对应的属性
- Build return op 需要:

依照 output\_names 从 operands 获取相应的 op

- 每创建或者转换一个算子都会执行一次插入操作, 将算子插入到 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

### 5.3 算子转换样例

本节以 Conv 算子为例, 将单 Conv 算子的 onnx 模型转换为 Top mlir, 原模型如图所示 (单 Conv 模型)

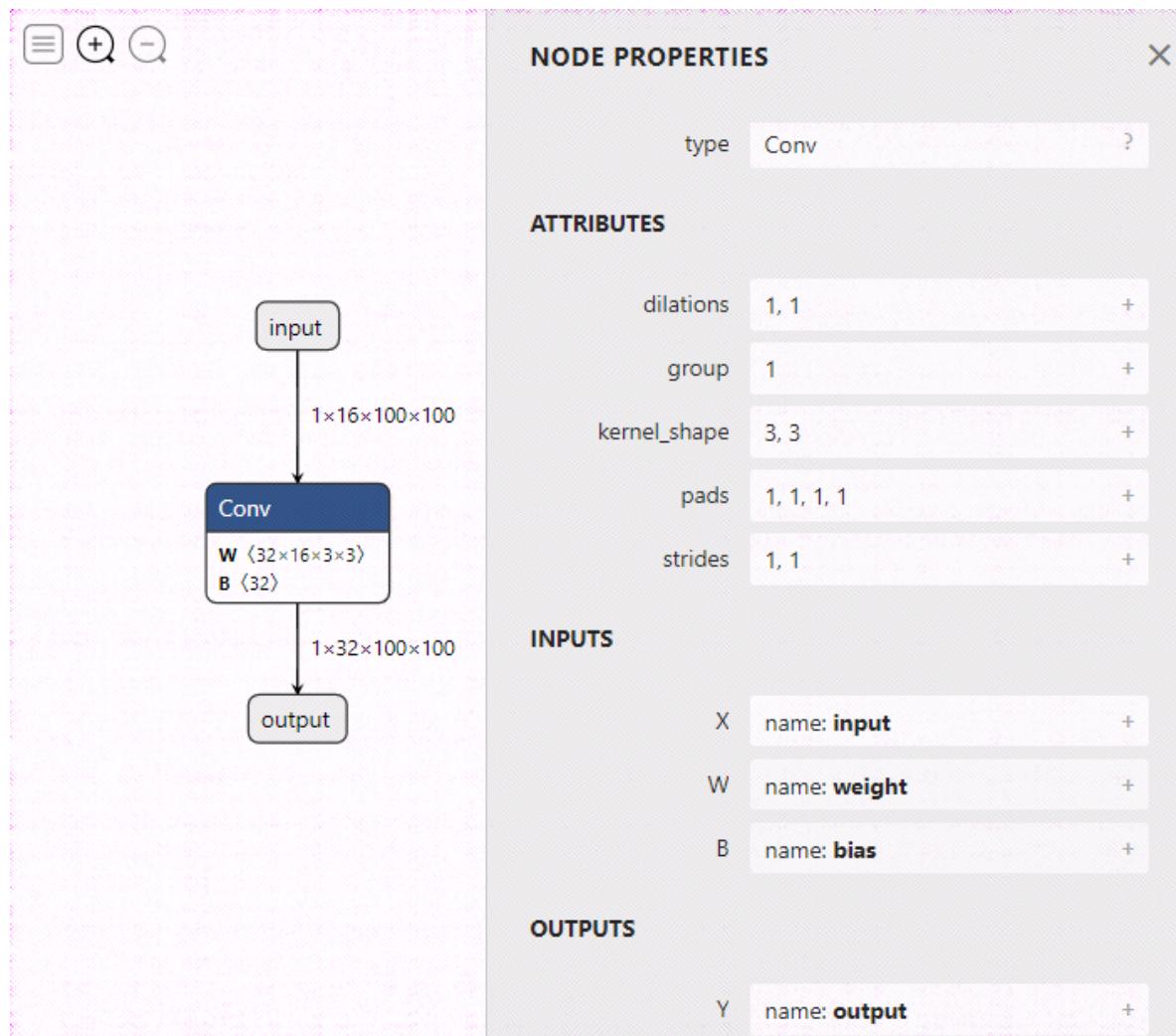


图 5.2: 单 Conv 模型

转换流程为:

- 算子定义

在 TopOps.td 中定义 Top.Conv 算子, 算子定义如图所示 ([Conv 算子定义](#))

```

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

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

```

图 5.3: Conv 算子定义

## 2. 初始化 OnnxConverter

load\_onnx\_model:

- 由于本例使用的是最简模型，所以生成的 Conv\_opt.onnx 模型与原模型相同。
- input\_names 保存了 Conv 算子的输入名 “input”
- tensors 中保存了 Conv 算子的权重 weight 与 bias
- shapes 中保存了 Conv 算子的输入和输出 shape。
- output\_names 中保存了 Conv 算子的输出名 “output”

init\_MLIRImporter:

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，如图所示 ([初始 mlir 文本](#))。

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

图 5.4: 初始 mlir 文本

## 3. generate\_mlir

- build input op，生成的 Top.inputOp 会被插入到 MLIRImporter.mlir\_module 中。
- 根据 node.op\_type (即 “Conv” ) 调用 convert\_conv\_op()，该函数中会调用 MLIRImporter.create\_conv\_op 来创建 ConvOp，而 create 函数需要的参数有：
  - 1) 输入 op: 从 ([单 Conv 模型](#)) 可知，Conv 算子的 inputs 一共包含了 input, weight 与 bias，inputOp 已被创建好，weight 与 bias 的 op 则通过 getWeightOp() 创建。
  - 2) output\_shape: 利用 onnx\_node.name 从 shapes 中获取 Conv 算子的输出 shape。
  - 3) Attributes: 从 onnx Conv 算子中获取如 ([单 Conv 模型](#)) 中的 attributes。

在 create 函数里 Top.Conv 算子的 attributes 会根据 ([Conv 算子定义](#)) 中的定义来设定。Top.ConvOp 创建后会被插入到 mlir 文本中

- 根据 output\_names 从 operands 中获取相应的 op，创建 return\_op 并插入到 mlir 文本中。到此为止，生成的 mlir 文本如图所示 ([完整的 mlir 文本](#))。

## 4. 输出

将 mlir 文本保存为 Conv\_origin.mlir，tensors 中的权重保存为 Conv\_TOP\_F32\_all\_weight.npz。

```
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 5.5: 完整的 mlir 文本

# CHAPTER 6

## 量化

量化理论源于论文: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

该论文地址: <https://arxiv.org/abs/1712.05877>

本章介绍 TPU-MLIR 的量化设计, 重点在该论文在实际量化中的应用。

## 6.1 基本概念

INT8 量化分为非对称量化和对称量化。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。

### 6.1.1 非对称量化

如上图 (非对称量化) 所示, 非对称量化其实就是在  $[min, max]$  范围内的数值定点到  $[-128, 127]$  或者  $[0, 255]$  区间。

从 int8 到 float 的量化公式表达如下:

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

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 scale, 是 float;  $Z$  是 zeropoint, 是 INT8 类型;

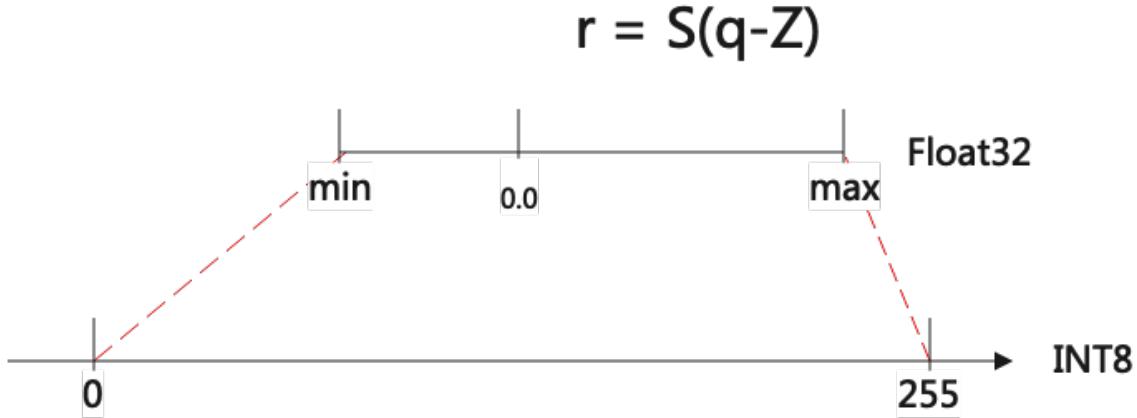


图 6.1: 非对称量化

当量化到 INT8 时, qmax=127,qmin=-128; UINT8 时, qmax=255,qmin=0

反过来从 float 到 int8 的量化公式如下:

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

### 6.1.2 对称量化

对称量化是非对称量化  $Z=0$  时的特例, 公式表达如下:

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

threshold 是阈值, 可以理解为 Tensor 的范围是  $[-\text{threshold}, \text{threshold}]$

这里  $S = \text{threshold}/128$ , 通常是 activation 情况;

对于 weight, 一般  $S = \text{threshold}/127$

对于 UINT8, Tensor 范围是  $[0, \text{threshold}]$ , 此时  $S = \text{threshold}/255.0$

## 6.2 Scale 转换

论文中的公式表达:

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

换个表述来说, 就是浮点数 Scale, 可以转换成 Multiplier 和 rshift, 如下表达:

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

举例说明:

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

Multiplier 支持的位数越高, 就越接近 Scale, 但是性能会越差。一般硬件会用 32 位或 8 位的 Multiplier。

## 6.3 量化推导

我们可以用量化公式, 对不同的 OP 进行量化推导, 得到其对应的 INT8 计算方式。

对称和非对称都用在 Activation 上, 对于权重一般只用对称量化。

### 6.3.1 Convolution

卷积的表示式简略为:  $Y = X_{(n,ic,ih,iw)} \times W_{(oc,ic,kh,kw)} + B_{(1,oc,1,1)}$

代入 int8 量化公式, 推导如下:

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

非对称量化特别注意的是, Pad 需要填入 Zx

对称量化时, Pad 填入 0, 上述推导中 Zx 和 Zy 皆为 0

在 PerAxis(或称 PerChannal) 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 Multiplier、rshift

### 6.3.2 InnerProduct

表达式和推导方式与 (Convolution) 相同

### 6.3.3 Add

加法的表达式为:  $Y = A + B$

代入 int8 量化公式, 推导如下:

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

加法最终如何用智能深度学习处理器实现, 与处理器具体的指令有关。

这里对称提供的方法是用 INT16 做中间 buffer;

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, 做加法后再重量化成 INT8

### 6.3.4 AvgPool

平均池化的表达式可以简写为:  $Y_i = \frac{\sum_{j=0}^k (X_j)}{k}$ , 其中  $k = kh \times kw$

代入 int8 量化公式, 推导如下:

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

### 6.3.5 LeakyReLU

LeakyReLU 的表达式可以简写为:  $Y = \begin{cases} X, & \text{if } X \geq 0 \\ \alpha X, & \text{if } X < 0 \end{cases}$

代入 int8 量化公式, 推导如下:

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

对称量化时,  $S_y = \frac{\text{threshold}_y}{128}, S_x = \frac{\text{threshold}_x}{128}$ , 非对称量化时,  $S_y = \frac{\max_y - \min_y}{255}, S_x = \frac{\max_x - \min_x}{255}$ 。通过 BackwardCalibration 操作后,  $\max_y = \max_x, \min_y = \min_x, \text{threshold}_y = \text{threshold}_x$ , 此时  $S_x/S_y = 1$ 。

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

当为对称量化时,  $Z_x$  和  $Z_y$  均为 0。

### 6.3.6 Pad

Pad 的表达式可以简写为:  $Y = \begin{cases} X, & \text{origin location} \\ value, & \text{padded location} \end{cases}$

代入 int8 量化公式, 推导如下:

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

通过 ForwardCalibration 操作后,  $\max_y = \max_x, \min_y = \min_x, \text{threshold}_y = \text{threshold}_x$ , 此时  $S_x/S_y = 1$ 。

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

对称量化时,  $Z_x$  和  $Z_y$  均为 0, pad 填入  $\text{round}(\text{value}/S_y)$ , 非对称量化时, pad 填入  $\text{round}(\text{value}/S_y + Z_y)$ 。

### 6.3.7 PReLU

$$\text{PReLU 的表达式可以简写为: } Y_i = \begin{cases} X_i, & \text{if } X_i \geq 0 \\ \alpha_i X_i, & \text{if } X_i < 0 \end{cases}$$

代入 int8 量化公式, 推导如下:

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

通过 BackwardCalibration 操作后,  $\max_y = \max_x, \min_y = \min_x, \text{threshold}_y = \text{threshold}_x$ , 此时  $S_x/S_y = 1$ 。

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

一共有  $oc$  个 Multiplier 和 1 个 rshift。当为对称量化时,  $Z_x$  和  $Z_y$  均为 0。

---

## Calibration

---

### 7.1 总体介绍

所谓校准，就是用真实场景数据来调校出恰当的量化参数，为何需要校准？当我们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。

tpu-mlir 的校准过程包括了门限方法自动寻优 (search\_threshold),SmoothQuant(sq), 跨层权重均衡 (we), 偏置修正 (bc) 以及自动混精功能 (search\_qtable) 等方法。总体过程如 ([量化流程图](#)) 所示。其中 sq,we,bc,search\_qtable 和 search\_threshold 都是可选择的，可以根据当前要量化的模型的实际情况进行搭配，后面章节也会具体给出各个方法的使用说明。上述过程整合在一起统一执行，最后将各个 op 的优化后的 threshold 和 min/max 值输出到一个量化校准参数文件 cali\_table 中，后续 “model\_deploy.py” 时就可使用这个参数文件来进行后续的 int8 量化。如果您使用了自动混精功能，在生成 cali\_table 的同时，还会生成混合精度表 qtable，后续 “model\_deploy.py” 时需使用这两个文件来进行后续的 int8 混合精度量化。

### 7.2 默认流程介绍

当前的校准流程囊括了多项方法，同时也提供了默认校准流程，具体过程如 ([默认流程](#)) 图如下图 ([cali\\_table 校准表](#)) 为校准最终输出的校准表  
如若使用 search\_qtable，还会生成 qtable 混精表，如下图所示。

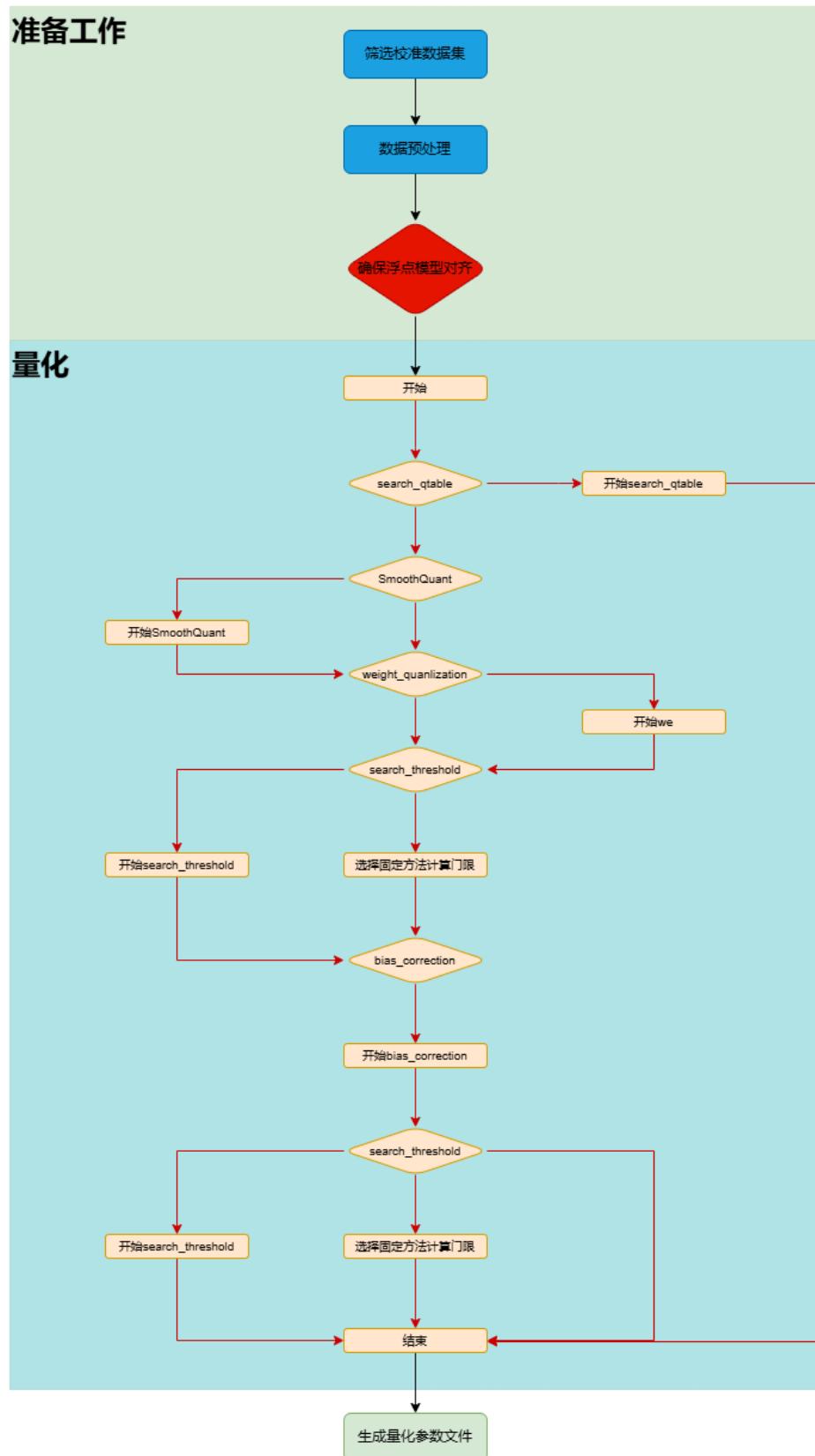


图 7.1: 量化流程图

Copyright © SOPHGO



图 7.2: 默认流程

```
# generated time: 2022-08-11 10:00:59.743675
# histogram number: 2048
# sample number: 100
# tune number: 5
###
# op_name    threshold      min      max
images 1.0000080 0.0000000 1.0000080
122_Conv 56.4281803 -102.5830231 97.6811752
124_Mul 38.1586478 -0.2784646 97.6811752
125_Conv 56.1447888 -143.7053833 122.0844193
127_Mul 116.7435987 -0.2784646 122.0844193
128_Conv 16.4931355 -87.9204330 7.2770605
130_Mul 7.2720342 -0.2784646 7.2720342
131_Conv 51.5455152 -56.4878578 26.2175255
133_Mul 22.2855371 -0.2784646 26.2175255
134_Conv 19.6111164 -28.0139256 21.4674854
136_Mul 20.8639418 -0.2784646 21.4674854
137_Add 20.5015809 -0.5569289 21.8679256
138_Conv 14.7106976 -87.1445465 32.7312393
140_Mul 27.4429126 -0.2784646 22.7210222
```

图 7.3: cali\_table 校准表

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

## 7.3 校准数据筛选及预处理

### 7.3.1 筛选原则

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, 要排除掉一些异常样例;

### 7.3.2 输入格式及预处理

表 7.1: 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 命令要指定和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 tpu-mlir 暂不支持的情形, 建议额外编写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 tensor 按字典的方式打包在一起, npy 文件是 1 个文件包含 1 个 tensor), run_calibration.py 支持直接导入 npz/npy 文件

上面 2 种格式, 在调用 run\_calibration.py 调用 mlir 文件进行推理时, 就无需再指定校准图片的预处理参数了

表 7.2: 参数指定方式

方式	描述
-dataset	对于单输入网络, 放置输入的各个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, 放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, 或者 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, 文件间通过逗号分割 (注意 npz 文件应该只有 1 个输入地址)

```

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

```

图 7.4: data\_list 要求的格式样例

## 7.4 量化门限算法实现

tpu-mlir 目前实现了七种量化门限计算方法，分别为 kld+auto-tune,octav,minmax,percentile9999, aciq\_gauss+auto-tune,aciq\_laplace+auto-tune 和基于 torch 的 histogram 算法，下面将对 kld, octav,aciq 和 auto-tune 算法进行介绍。

### 7.4.1 kld 算法

tpu-mlir 实现的 kld 算法参考 tensorRT 的实现，本质上是将  $\text{abs}(\text{fp32\_tensor})$  这个分布（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 分布若用 128 个等级的 int8 类型来表达，将相邻的多个 bin（比如 256bin 是相邻的 2 个 fp32 bin）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 个量化等级最好的模拟 fp32 的概率分布，故量化门限设在这里是最合适的。kld 算法实现伪码如下所示：

```

1 the pseudocode of computing int8 quantize threshold by kld:
2 Prepare fp32 histogram H with 2048 bins
3 compute the absmax of fp32 value
4
5 for i in range(128,2048,128):
6     Outliers_num = sum(bin[i], bin[i+1], ..., bin[2047])
7     Fp32_distribution = [bin[0], bin[1], ..., bin[i-1]+Outliers_num]
8     Fp32_distribution /= sum(Fp32_distribution)
9
10    int8_distribution = quantize [bin[0], bin[1], ..., bin[i]] into 128 quant level
11    expand int8_distribution to i bins
12    int8_distribution /= sum(int8_distribution)
13    kld[i] = KLD(Fp32_distribution, int8_distribution)
14 end for
15
16 find i which kld[i] is minimal
17 int8 quantize threshold = (i + 0.5)*fp32 absmax/2048

```

### 7.4.2 auto-tune 算法

从 KLD 算法的实际表现来看，其候选门限相对较粗，也没有考虑到不同业务的特性，比如：对于目标检测、关键点检测等业务，tensor 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而影响到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，而评估分布相似性的方法还有其他比如欧式距离、cos 相似度等方法，这些度量方法不用考虑粗略的截取门限直接来评估 tensor 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 auto-tune 算法对这些激活的量化门限基于欧式距离度量进行微调，从而保证其 int8 量化具有更好的精度表现；

实现方案：首先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，引入量化误差；然后逐个对 op 的输入激活量化门限进行调优：在初始 KLD 量化门限和激活的最大绝对值之间，均匀选择 20 个候选值，用这些候选值对 fp32 参考激活值进行量化加扰，引入量化误差，然后输入 op 进行 fp32 计算，将输出的结果与 fp32 参考激活进行欧式距离计算，选择 20 个候选值中具有最小欧式距离的值作为调优门限；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，然后取其中较大者，比如（auto-tune 调优实现方案）图中 layer1 的输出会分别针对 layer2、layer3 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；

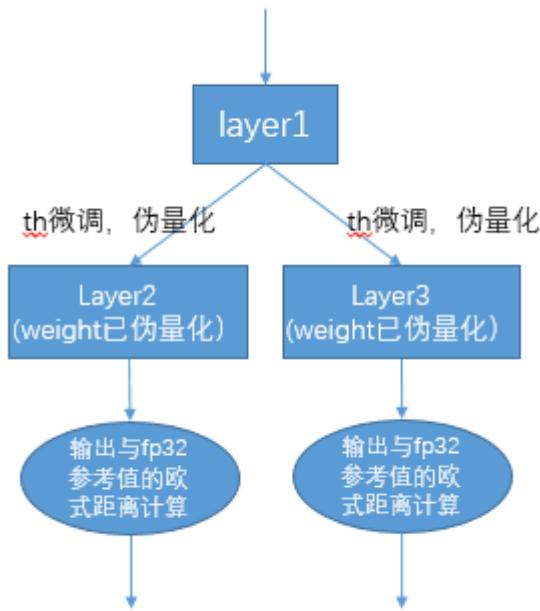


图 7.5: auto-tune 调优实现方案

### 7.4.3 octav 算法

tpu-mlir 实现的 octav 算法参考了文章《Optimal Clipping and Magnitude-aware Differentiation for Improved Quantization-aware Training》。通常人们认为量化误差来源于舍入误差和截断误差，计算每个张量的最优截断（门限）可以最小化量化误差，octav 采用了均方误差来衡量量化误差，采用递归方式并基于快速的牛顿-拉夫森（Newton-Raphson）方法用于动态确定最小化均方误差（MSE）的最优门限。下面给出了该方法最优门限迭代计算公式，如图（octav 迭代公式）所示。

其设计之初用于 QAT 量化中，但在 PTQ 量化中同样有效。下面是其实现伪码：

```

1 the pseudocode of computing int8 quantize threshold by octav:
2   Prepare T: Tensor to be quantized,
3   B: Number of quantization bits,
4   epsilon: Convergence threshold (e.g., 1e-5),
5   s_0: Initial guess for the clipping scalar (e.g., max absolute value in tensor T)
  
```

(续下页)

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

图 7.6: octav 迭代公式

(接上页)

```

6   compute s_star: Optimal clipping scalar
7
8   for n in range(20):
9       Compute the indicator functions for the current clipping scalar:
10      I_clip = 1{|T| > s_n} (applied element-wise to tensor T)
11      I_disc = 1{0 < |T| < s_n}
12
13      Update the clipping scalar s_n to the next one s_(n+1) using:
14      s_(n+1) = (\sum|x| * I_clip) / ((4^{-B}) / 3) * \sum I_disc + \sum I_clip
15      where \sum denotes the summation over the corresponding elements
16
17      If |s_(n+1) - s_n| < epsilon, the algorithm is considered to have converged
18   end for
19   s_star = s_n

```

#### 7.4.4 aciq 算法

tpu-mlir 实现的 aciq 算法参考了文章《ACIQ:ANALYTICAL CLIPPING FOR INTEGER QUANTIZATION OF NEURAL NETWORKS》。该方法假设激活值满足固定分布，然后计算该激活值对应分布的统计量，并根据理论计算获得的最优截断分位来得到最优门限。

实现方案:tpu-mlir 中提供了 aciq\_gauss 和 aciq\_laplace 两种算法，分别假设激活值满足 gauss 分布和 laplace 分布，然后根据理论上 8bit 对应的最优截断分位来计算获得最优门限。

## 7.5 优化算法实现

在校准过程中, 为了进一步提升量化模型的精度, tpu-mlir 提供了多种优化算法, 其中包括 SmoothQuant(sq), 跨层权重均衡 (we), 偏置修正 (bc), search\_qtable 和 search\_threshold, 下面是上述优化算法的介绍。

### 7.5.1 sq 算法

tpu-mlir 实现的 SmoothQuant 算法参考了文章《SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models》, 该方法通过平滑地分配模型的张量比例, 将模型的输入和权重的范围调整到一个更适合量化的范围, 从而提高量化后的模型精度, 解决大规模预训练模型 (如语言模型和视觉模型) 在量化过程中精度下降的问题。

SmoothQuant 通过调整模型的张量比例, 将激活和权重的范围进行重新分配, 使得量化过程更加稳定。具体来说, SmoothQuant 在量化前引入一个平滑因子, 将激活值的范围部分转移到权重中, 通过数学等价转换来调整模型权重, 从而降低激活值的量化误差。技术原理如图 (SmoothQuant) 所示。

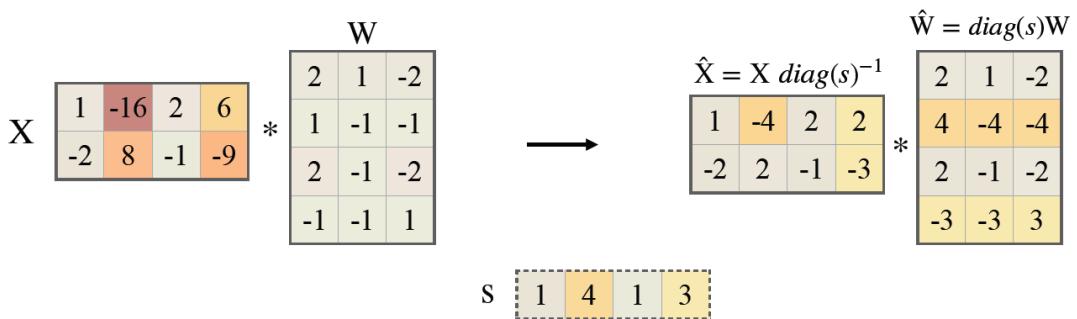


图 7.7: SmoothQuant

### 7.5.2 we 算法

tpu-mlir 实现的跨层权重均衡算法参考了文章《Data-Free Quantization Through Weight Equalization and Bias Correction》, 该方法主要针对模型权重, 通过对符合 conv-conv 和 conv-relu-conv 这两种 pattern 的权重进行均衡, 使两个相邻权重分布尽可能均匀。

之前研究发现在 mobilenet 这类可分离卷积较多的网络中, 由于可分离卷积的 channel 间数据分布差异较大, 如果采用 per-layer 的量化, 会造成较大的量化误差。we 算法很好的解决了这一问题, 其利用了 relu 函数的线性特性, 可以对相邻卷积权重进行均衡, 使得卷积 channel 间的分布差距缩小, 此时采用 per-layer 的效果可以与 per-channel 相当, 技术原理如图 (weight\_equalization) 所示。

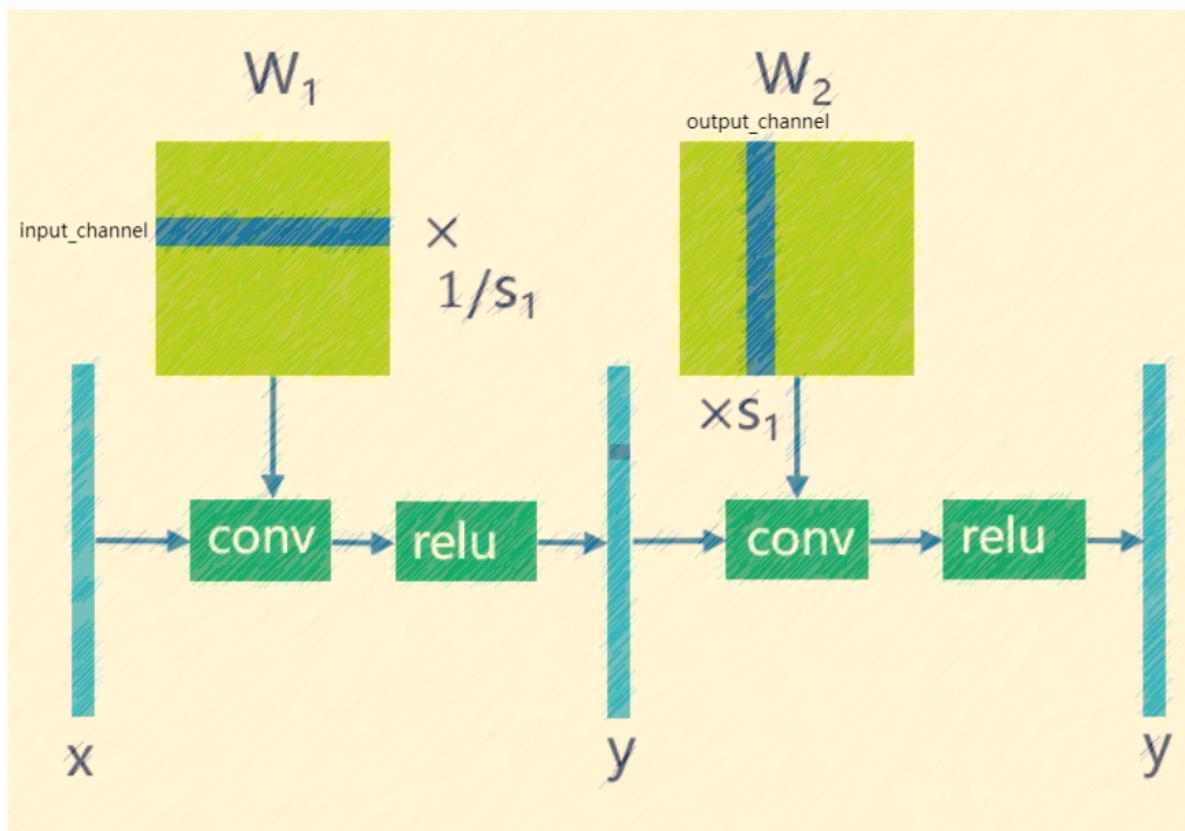


图 7.8: weight\_equalization

### 7.5.3 bc 算法

tpu-mlir 实现的偏置修正算法参考了文章《Data-Free Quantization Through Weight Equalization and Bias Correction》。通常人们认为量化模型输出误差是无偏的，也就是其满足期望值为 0，但在很多实际场景下，量化模型的输出误差是有偏的，也就是量化模型的输出与浮点模型的输出存在期望值上的偏离，这会对量化模型的精度造成影响。

偏置修正算法通过计算量化模型在校准数据上与浮点模型的统计偏差，然后对模型中 Conv/Gemm 算子的 bias 项进行补偿，从而尽可能减小二者输出的期望值偏差。效果如图 (bias\_correction) 所示。

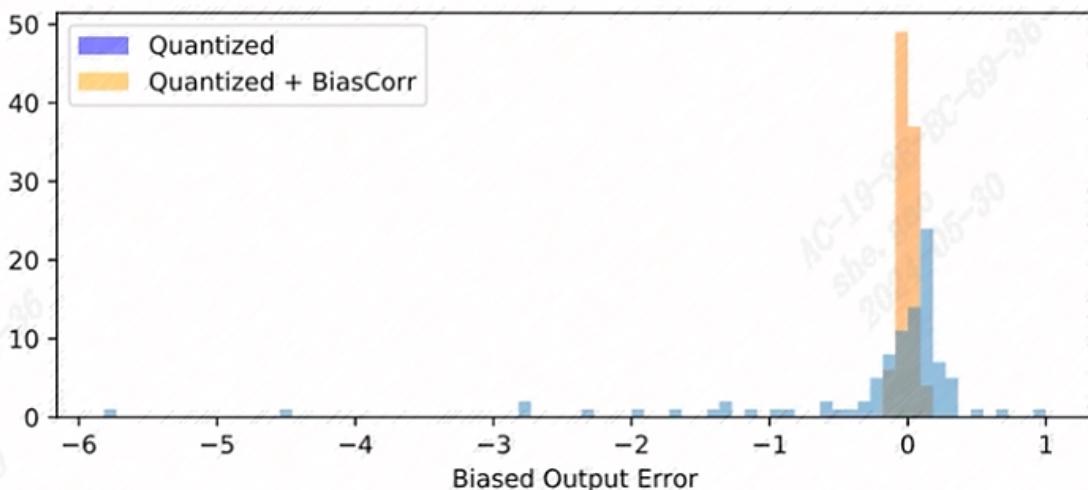


图 7.9: bias\_correction

### 7.5.4 search\_threshold 算法

tpu-mlir 提供了七种独立的门限计算方法，当我们拿到一个需要做量化的模型时，该如何择优选择门限计算方法成为一个问题。search\_threshold 针对上述问题提供了一个解决方案。

实现方案:search\_threshold 首先会同时计算 kld+tune,octav,max 和 percentile9999 四种方法的门限值，然后计算不同方法门限值生成的量化模型输出与浮点模型输出的相似度，通过比较四种门限方法的相似度，选择最大相似度对应的门限方法的门限值作为当前模型量化参数。在使用过程中，需要注意以下几点:1.search\_threshold 目前提供了 cos 和 snr 两种相似度计算方法，默认采用 cos 相似度计算方法 2. 如果量化模型与浮点模型 cos 相似度低于 0.9，该量化模型的精度下降可能比较严重，search\_threshold 搜索结果可能存在偏差，在进行实际精度验证后建议采用 search\_qtable 进行混精尝试。

### 7.5.5 search\_qtable 算法

search\_qtable 是集成于校准过程中的自动混精功能, 当全 int8 量化的模型精度无法满足需求时, 可以尝试开启 search\_qtable 算法, 该算法相比 run\_sensitive\_lyer, 速度更快, 同时提供了自定义门限算法混合以及自动生成 qtable 功能。

实现方案:search\_qtable 的输出会生成混合门限, 混合门限是指对模型每一层门限都进行择优选择, 也就是从用户所指定的多种门限计算方法结果中选择效果最好的一个, 这种选择的依据是量化模型当前层输出与原始模型当前层输出的相似度比较。除了输出混合门限,search\_qtable 还会输出模型的混精层, 当用户指定混精模型与原始模型的输出相似度后,search\_qtable 会自动输出满足该相似度所需的最少混精层。

## 7.6 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, 任意新建目录进入执行如下命令可以完成对 yolov5s 的校准过程:

```

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

```

表 7.3: model\_transform.py 参数

参数	描述
model_name	模型名
-model_def	模型类型文件 (.onnx,.pt,.tflite or .prototxt)
-model_data	指定模型权重文件, 为 caffe 模型时需要 (对应' .caffemodel' 文件)
-input_shapes	输入的形状, 例如 [[1,3,640,640]] (二维数组), 可以支持多个输入
-resize_dims	要调整到的原始图像的大小。如果未指定, 它将调整为模型的输入大小
-	调整大小时是否保持纵横比。默认为 False。设置时不足的部分会补 0
keep_aspect_ratio	
-mean	图像每个通道的平均值。默认为 0.0,0.0,0.0
-scale	图像每个通道的 scale。默认为 1.0,1.0,1.0
-pixel_format	图像类型, 可以是 rgb、bgr、gray 或 rgbd
-output_names	输出的名称。如果未指定, 则使用模型的输出, 否则使用指定的名称作为输出
-test_input	用于验证的输入文件, 可以是图像、npy 或 npz。如果不指定则不会进行验证
-test_result	输出文件保存验证结果
-excepts	验证过程中要排除的网络层名称。用逗号分隔
-debug	如果打开调试, 则立即模型文件将保留; 或将在转换完成后删除
-mlir	输出 mlir 文件名 (包括路径)

## 默认流程

```

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

```

## 使用不同量化门限计算方法

octav:

```

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

```

minmax:

```

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

```

percentile9999:

```

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

```

aciq\_gauss:

```

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

```

aciq\_laplace:

```

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

```

使用优化方法:

sq:

```

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

```

we:

```

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

```

we+bc:

```

1 $ run_calibration.py yolov5s.mlir \
2   --we \
3   --bc \
4   --dataset $REGRESSION_PATH/dataset/COCO2017 \
5   --input_num 100 \
6   --processor bm1684x \

```

(续下页)

(接上页)

```

7   --bc_inference_num 200 \
8   --cali_method use_mse \
9   -o yolov5s_cali_table

```

we+bc+search\_threshold:

```

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

```

search\_qtable:

```

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

```

表 7.4: run\_calibration.py 参数

参数	描述
mlir_file	mlir 文件
-sq	开启 SmoothQuant
-we	开启 weight_equalization
-bc	开启 bias_correction
-dataset	校准数据集
-data_list	input 列表
-input_num	校准图像数量
-inference_num	search_qtable 和 search_threshold 推理过程所需图片数量
-bc_inference_num	bias_correction 推理过程所需图片数量
-tune_list	tuning 用到的 input 列表
-tune_num	tuning 的图像数量
-histogram_bin_num	指定 kld 计算的直方图 bin 数量
-expected_cos	期望 search_qtable 混精模型输出与浮点模型输出的相似度, 取值范围 [0,1]

续下页

表 7.4 – 接上页

参数	描述
-min_layer_cos	bias_correction 中该层量化输出与浮点输出的相似度下限, 当低于该下限时需要对该层进行补偿, 取值范围 [0,1]
-max_float_layers	search_qtable 浮点层数量
-processor	处理器类型
-cali_method	量化门限计算方法选择, 可选择 use_kl,use_mse,use_percentile9999,use_max, 默认为 use_kl
-fp_type	search_qtable 浮点层数据类型
-post_process	后处理路径
-	指定全局对比层, 例如 layer1,layer2 或 layer1:0.3,layer2:0.7
global_compare_layers	
-search	指定搜索类型, 其中包括 search_qtable,search_threshold,false。其中默认为 false, 不开启搜索
-transformer	是否是 transformer 模型,search_qtable 中如果是 transformer 模型可分配指定加速策略
-quantize_method_list	search_qtable 用来搜索的门限方法, 默认仅 MSE, 支持 KL,MSE,MAX,Percentile9999 自由选择
-benchmark_method	指定 search_threshold 中相似度计算方法
-kurtosis_analysis	指定生成各层激活值的 kurtosis
-part_quantize	指定模型部分量化, 获得 cali_table 同时会自动生成 qtable。可选择 N_mode,H_mode,custom_mode,H_mode 通常精度较好
-custom_operator	指定需要量化的算子, 配合开启上述 custom_mode 后使用
-part_asymmetric	指定当开启对称量化后, 模型某些子网符合特定 pattern 时, 将对应位置算子改为非对称量化
-mix_mode	指定 search_qtable 特定的混精类型, 目前支持 8_16 和 4_8 两种
-cluster	指定 search_qtable 寻找敏感层时采用聚类算法
-quantize_table	search_qtable 输出的混精度量化表
-o	输出门限表
-debug_cmd	debug 命令, 可以选择校准模式; “percentile9999” 采用 99.99 分位作为初始门限。“use_max” 采用绝对值最大值作为门限。“use_torch_observer_for_cali” 采用 torch 的 observer 进行校准。“use_mse” 采用 octav 进行校准。
-debug_log	日志输出级别

执行结果如下图 ([yolov5s\\_cali 校准结果](#)) 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 7.10: yolov5s\_cali 校准结果

## 7.7 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候定位问题。此工具在 docker 中启动，可以在宿主机中启动浏览器打开界面。工具默认使用 tcp 端口 10000，需要在启动 docker 时候使用-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。命令启动方式如下图所示：

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

表 7.5: 可视化工具命令行参数

参数	描述
-port	服务程序的 TCP 监听端口，默认值为 10000
-f32_mlir	量化前的浮点 mlir 网络的文件名，此文件为 model_transform 生成，一般为 netname.mlir，是初始 float32 网络
-quant_mlir	量化后的定点 mlir 网络的文件名，此文件为 model_deploy 生成，一般文件名为 netname_int8_sym_tpu.mlir，生成 bmodel 用的 _final.mlir 不适用此工具。
-input	运行网络比较的输入样本数据，可以是 jpeg 图片文件或者 npy/npz 数据文件，一般可使用网络转换时的 test_input
-manual_run	浏览器客户端打开时是否自动运行网络进行数据比较，默认为 true，使用此参数则只显示网络结构

在浏览器地址栏输入 localhost:9999 可以打开程序界面，启动时候会自动进行浮点网络和量化后网络的推理，所以可能会有一定时间的等待。如下图所示：

上图中使用淡蓝色细线框出了界面的几个区域，除浏览器地址栏之外，程序界面主要显示了：

1. 当前工作目录，指定的浮点网络和量化后网络；
2. 精度数据总结区
3. layer 属性显示区域
4. 网络图形化显示区
5. tensor 数据对比区
6. tensor 数据分布和信息总结显示区（切换 tab 页面）

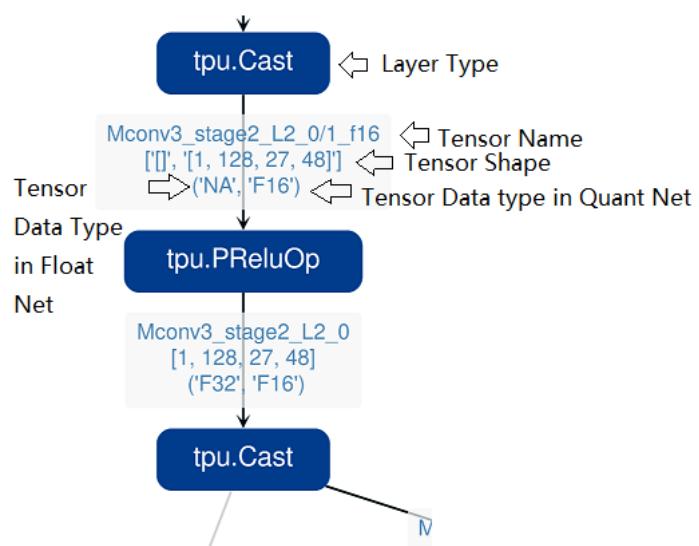
鼠标在网络显示区滚动可以放大和缩小网络显示，鼠标悬停或者点击节点可以在 layer 属性显示区中显示此 layer 的属性列表，点击 layer 之间的连线（也就是 tensor），可以在 tensor 数据对比区显示此 tensor 的量化前后数据对比。点击精度数据显示区中的点或者列表中的 tensor 或者 layer，会在网络中定位到这个选中的 layer 或者 tensor。需要注意的一点是由于网络是基于量化后网络显示，可能会相比浮点网络有变化，对于浮点网络中不存在的 tensor 会临时用量化后网络的数据替代，表现出来精度数据等都非常好，实际需要忽略而只关注浮

## CHAPTER 7. CALIBRATION



点和量化后网络都存在的 tensor，不存在的 tensor 的数据类型一般是 NA，shape 也是 [] 这样的空值。另外在 deploy 网络的时候如果没有使用--debug 参数，一些可视化工具需要的中间数据和文件会被默认清除，造成可视化工具运行不正常，需要增加--debug 选项重新生成。

tensor 上的信息解读如下：



---

Lowering

---

Lowering 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, 它涉及到量化算法; 针对不同硬件, 量化算法是不一样的, 比如有的支持 perchannel, 有的不支持; 有的支持 32 位 Multiplier, 有的只支持 8 位, 等等。

所以 Lowering 将算子从硬件无关层 (TOP), 转换到了硬件相关层 (TPU)。

## 8.1 基本过程

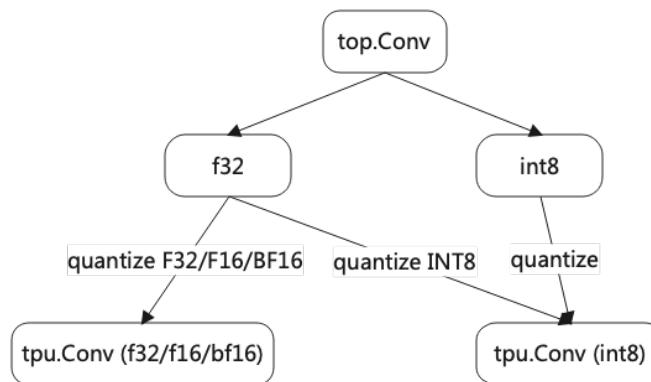


图 8.1: Lowering 过程

Lowering 的过程, 如图所示 (Lowering 过程)

- Top 算子可以分 f32 和 int8 两种, 前者是大多数网络的情况; 后者是如 tflite 等量化过的网络的情况

- f32 算子可以直接转换成 f32/f16/bf16 的 tpu 层算子, 如果要转 int8, 则需要类型是 calibrated\_type
- int8 算子只能直接转换成 tpu 层 int8 算子

## 8.2 混合精度

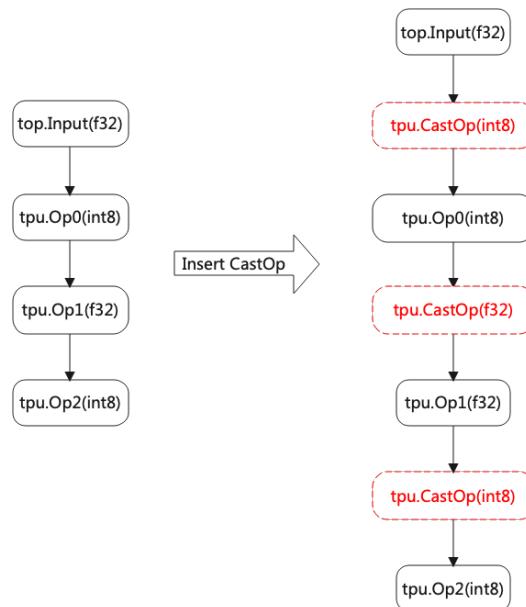


图 8.2: 混合精度

当 OP 之间的类型不一致时, 则插入 CastOp, 如图所示 (混合精度)。

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, 比如 embedding 无论输出是什么类型, 输入都是 uint 类型。

# CHAPTER 9

---

SubNet

---

待补充

# CHAPTER 10

---

LayerGroup

---

## 10.1 基本概念

智能深度学习处理器分为片外内存 (或称 Global Memory, 简称 GMEM) 和片内内存 (或称 Local Memory, 简称 LMEM)。

通常片外内存非常大 (比如 4GB), 片内内存非常小 (比如 16MB)。神经网络模型的数据量和计算量

都非常大, 通常每层的 OP 都需要切分后放到 Local Memory 进行运算, 结果再保存到 Global Memory。

LayerGroup 就是让尽可能多的 OP 经过切分后能够在 Local Memory 执行, 而避免过多的 Local 和 Global Memory 的拷贝。

### 要解决的问题:

如何使 Layer 数据保持在有限的 Local Memory 进行运算, 而不是反复进行 Local 与 Global Memory 之间的拷贝

### 基本思路:

通过切 Activation 的 N 和 H, 使每层 Layer 的运算始终在 Local Memory 中, 如图 (网  
络切分举例)

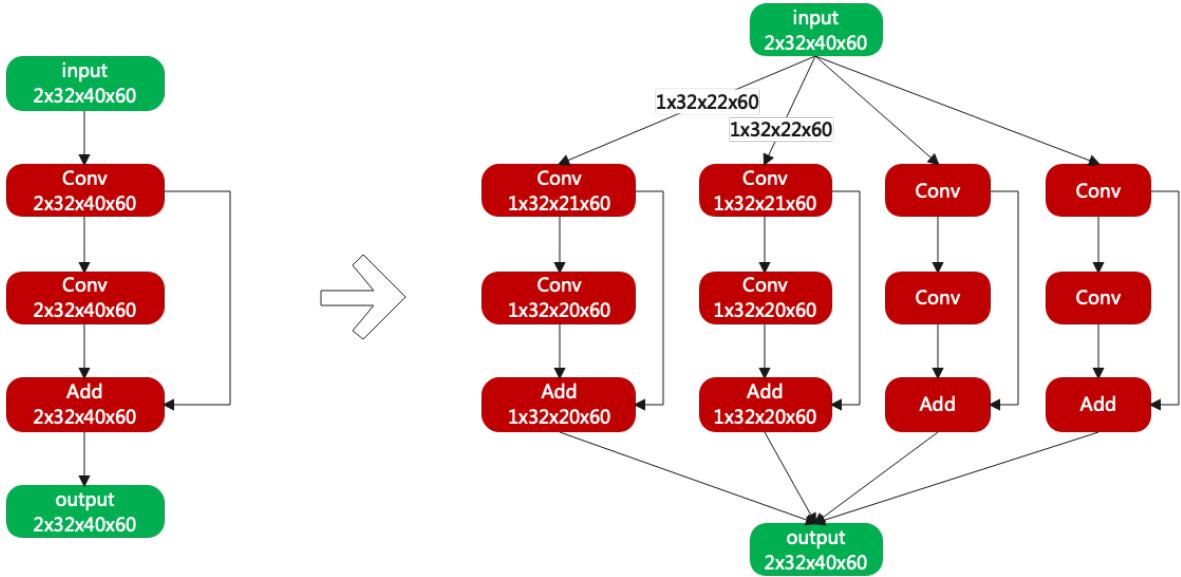


图 10.1: 网络切分举例

## 10.2 BackwardH

对网络进行 H 切分的时候, 大多数 Layer 输入和输出的 H 是一致的。但是对于 Conv、Pool 等等需要特别计算。

以 Conv 举例, 如图 (卷积 BackwardH 举例)

## 10.3 划分 Mem 周期

如何划分 group? 首先把每一层 Layer 需要的 lmem 罗列出来, 大体可以归为三类:

1. Activation Tensor, 用于保存输入输出结果, 没有使用者后直接释放
2. Weight, 用于保存权重, 不切的情况下用完就释放; 否则一直驻留在 lmem
3. Buffer, 用于 Layer 运算保存中间结果, 用完就释放

然后依次广度优先的方式配置 id, 举例如图 (LMEM 的 ID 分配)

然后再配置周期, 配置方法如图 (TimeStep 分配)

关于配置周期的细节如下:

- [T2,T7], 表示在 T2 开始的时候就要申请 lmem, 在 T7 结束的时候释放 lmem
- w4 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 w4 可以被同时加载
- 当 N 或者 H 被切分时, Weight 不需要重新被加载, 它的结束点会被修正为正无穷

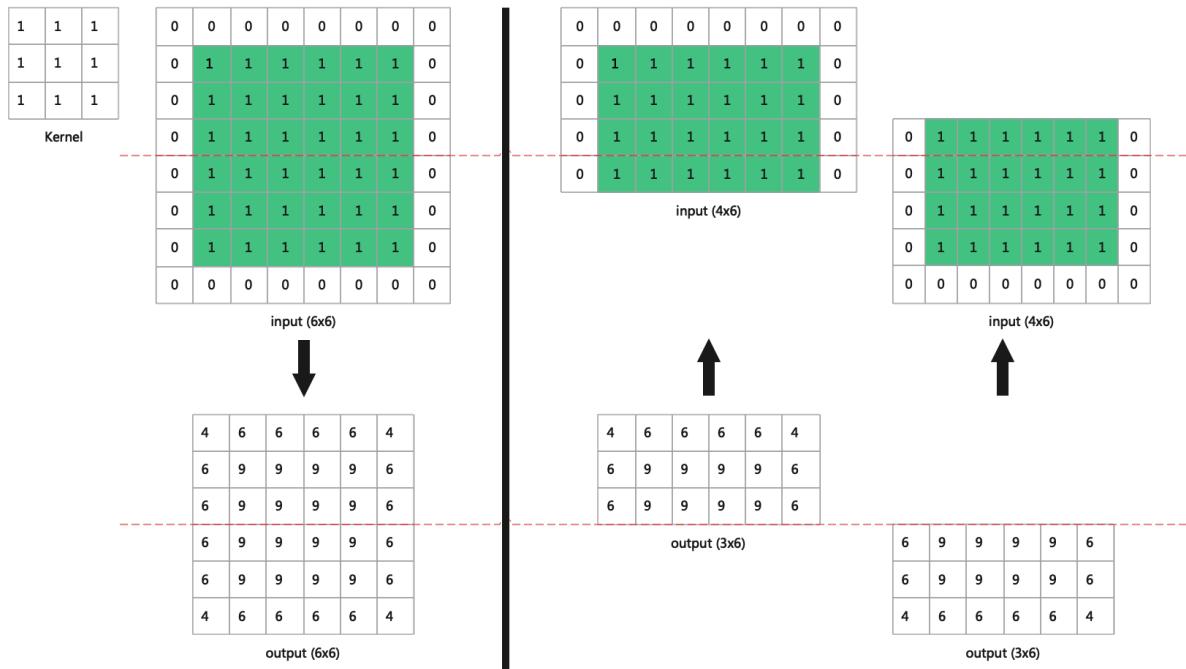


图 10.2: 卷积 BackwardH 举例

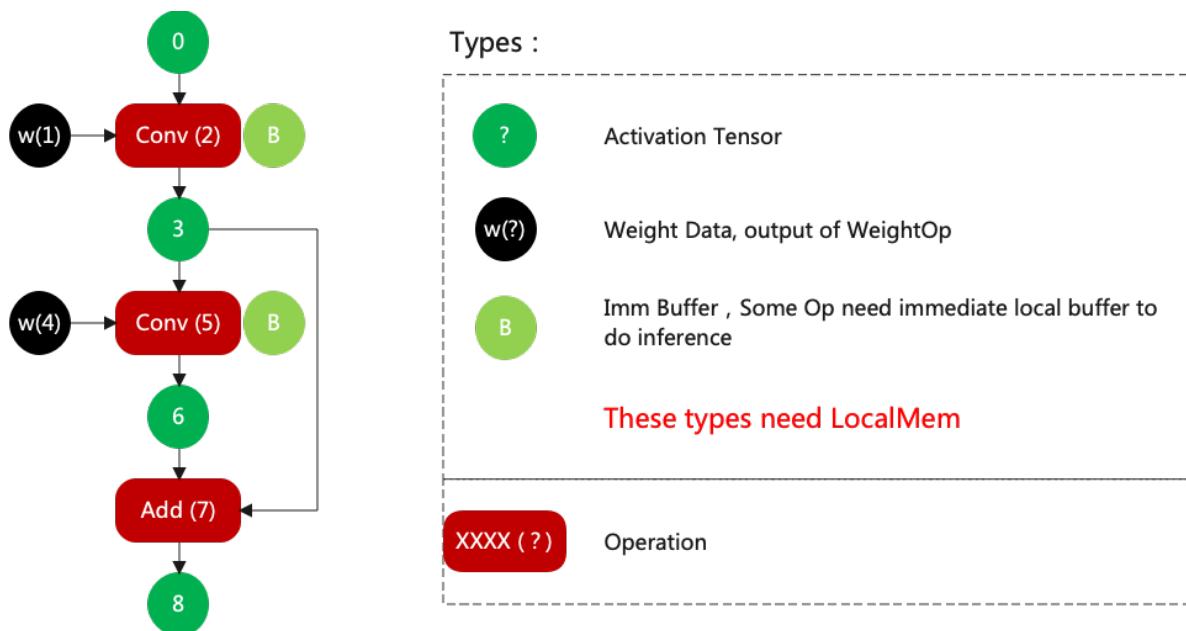


图 10.3: LMEM 的 ID 分配

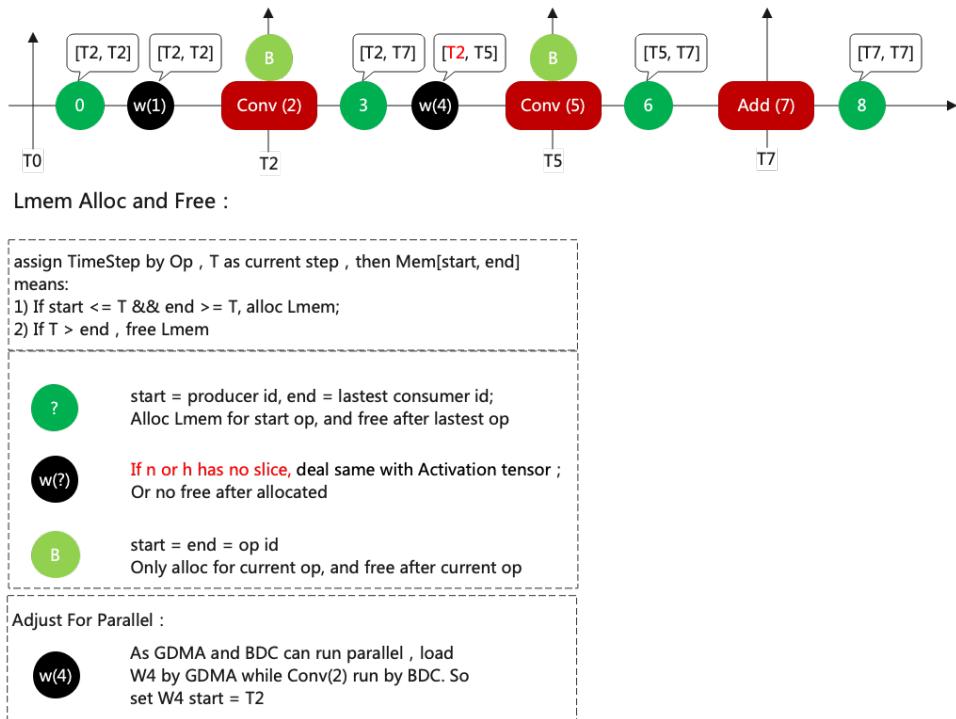


图 10.4: TimeStep 分配

## 10.4 LMEM 分配

当 n 或 h 存在切分的情况下, weight 常驻 LMEM, 每一个切分都可以继续使用 weight。

这时候会先分配 weight, 如图所示 (有切分情况的分配)

当 n 和 h 都没有切分的情况下, weight 和 activation 处理过程一样, 不使用时就释放。

这时候的分配过程, 如图所示 (无切分情况的分配)

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。

目前策略是按照 op 顺序依次分配, 优先分配 timestep 长的, 次分配 lmem 大的。

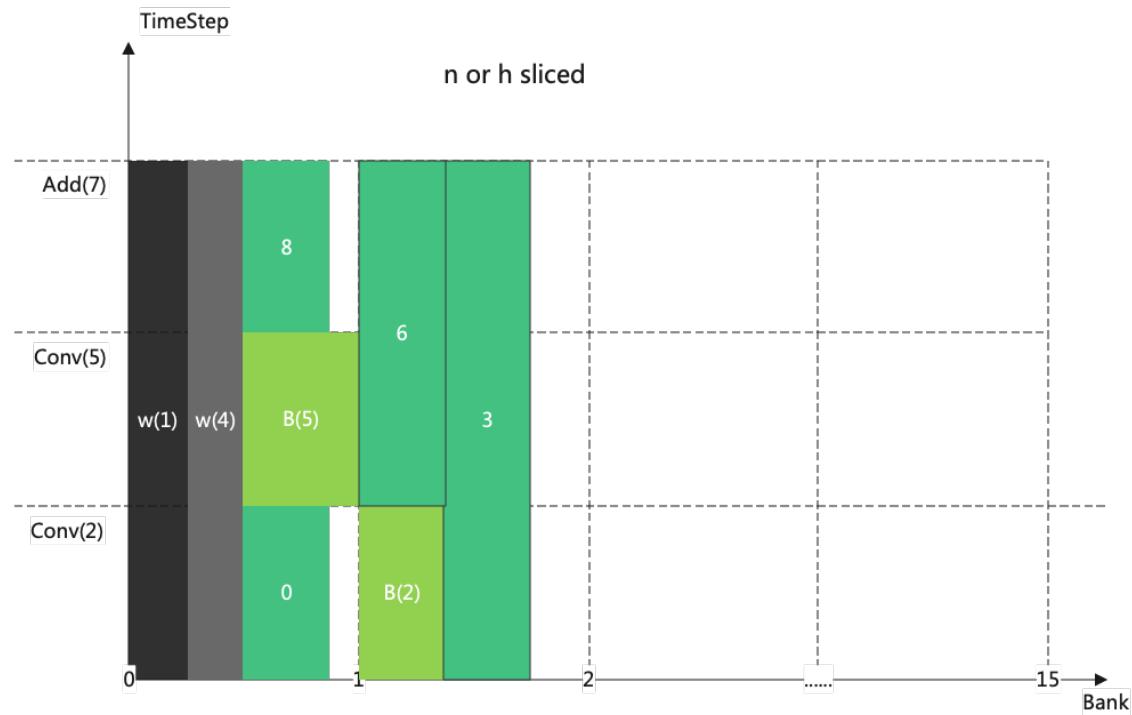


图 10.5: 有切分情况的分配

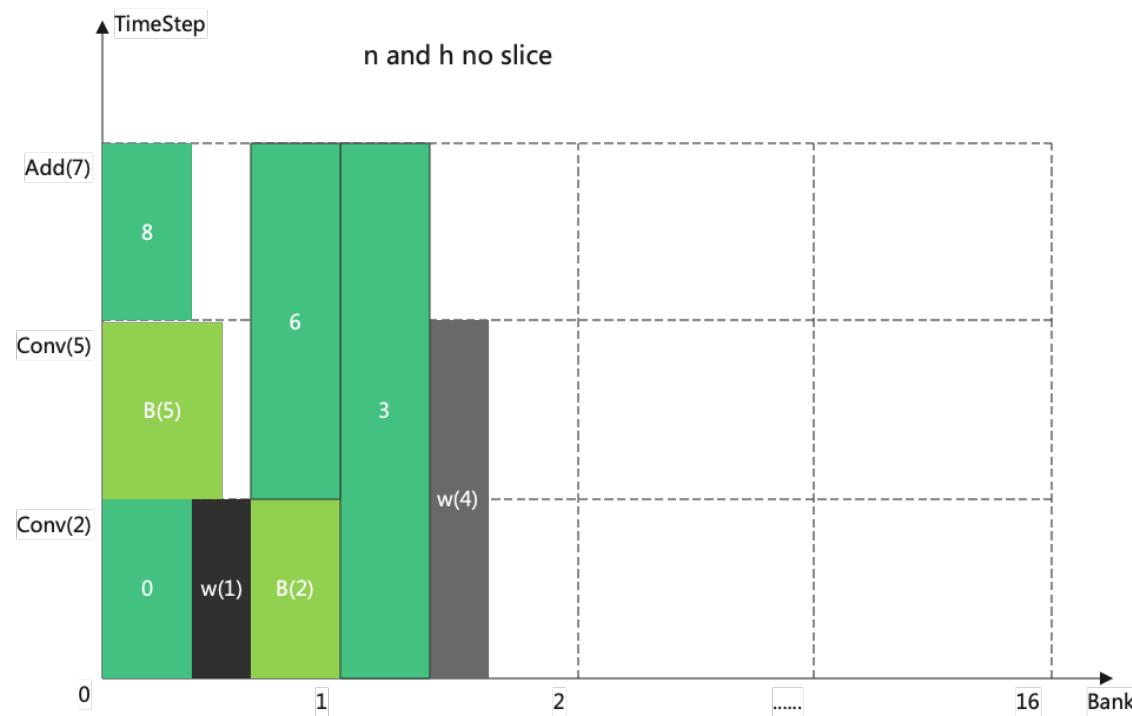


图 10.6: 无切分情况的分配

## 10.5 划分最优 Group

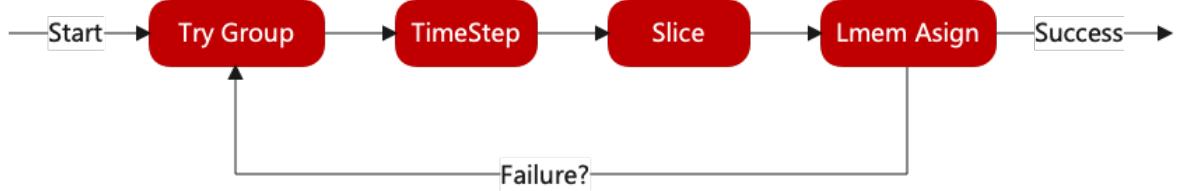


图 10.7: Group 流程

目前从尾部开始向头部方向划分 group, 优先切 N, 当 N 切到最小单位时还不能满足要求, 则切 h。

当网络很深的时候, 因为 Conv、Pool 等等算子会有重复计算部分, h 切的过多导致重复部分过多;

为了避免过多重复, 当 backward 后的 layer 的输入, 如果  $h\_slice$  重复的部分  $> h/2$ , 则认为失败。

举例: 比如 input 的  $h = 100$ , 经过切分后变成 2 个 input,  $h[0, 80)$  和  $h[20, 100)$ , 则重复部分为 60, 则认为失败; 2 个 input 对应  $h[0, 60)$  和  $h[20, 100)$ , 重复部分为 40, 认为成功。

# CHAPTER 11

---

## GMEM 分配

---

### 11.1 目的

为了节约 global memory 空间, 最大程度复用内存空间, 分配顺序: weight tensor、根据生命周期给全部 global neuron tensor 分配 gmem, 在分配过程中会复用已分配 gmem

---

**备注:** global neuron tensor 定义: 在 Op 运算结束后需要保存在 gmem 的 tensor. 如果是 layer Group, 只有 layer Group 的 input/output tensor 属于 global neuron tensor.

---

### 11.2 原理

#### 11.2.1 weight tensor 分配 gmem

遍历所有 WeightOp, 依次分配, 4K 地址对齐, 地址空间不断累加

### 11.2.2 global neuron tensors 分配 gmem

最大可能的复用内存空间，根据生命周期给全部 global neuron tensor 分配，在分配过程中会复用已分配 gmem。

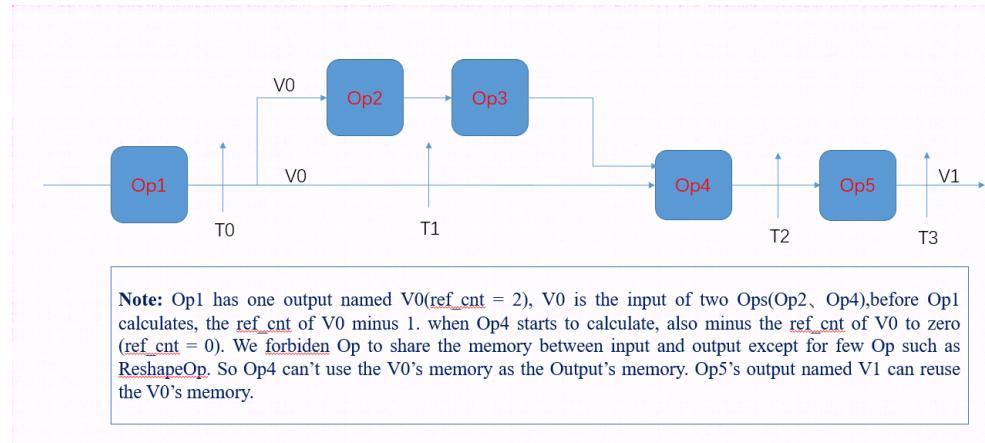
#### a. 数据结构介绍

每次分配时把对应的 tensor, address, size, ref\_cnt(这个 tensor 有几个 OP 使用)记录在 rec\_tbl。同时将 tensor, address 记录在辅助数据结构 hold\_edges,in\_using\_addr 中

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

#### b. 流程介绍

- 遍历每个 Op, 在遍历 Op 时, 判断 Op 的输入 tensor 是否位于 rec\_tbl 中, 如果 yes, 判断 ref\_cnt 是否  $\geq 1$ , 如果 yes, 则 ref\_cnt--, 表示输入 tensor 的引用数降低 1 个。  
如果 ref\_cnt 等于 0, 表示生命周期已结束, 后面的 tensor 可以复用它的地址空间。
- 在给每个 Op 的 output tensor 分配时, 先 check 是否可以复用 EOL 的 tensor 地址,check 思路, 遍历 rec\_tbl, 需要同时满足如下 5 个条件才能 reuse:
  - 对应的 tensor 不在 hold\_edges 内
  - 对应 tensor 的地址不在 in\_using\_addr 内
  - 对应 tensor 已 EOL
  - 对应 tensor 的地址空间  $\geq$  当前 tensor 所需空间
  - 当前 OP 的输入 tensor 地址不能与对应 tensor 的地址相同 (某些 Op 最终运算结果不正确, reshapeOP 例外)
- 给当前 Op 的 output tensor 分配 gmem, 如果 step2 显示可以 reuse, 就 reuse. 否则在 ddr 中新开辟 gmem.
- 调整当前 Op 的 input tensor 的生命周期, 确认它是否位于 hold\_edges 内, 如果 yes, 则在 rec\_tbl 中寻找, 检查它的 ref\_cnt 是否为 0, 如果 yes, 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 input tensor 生命周期已结束, 地址空间已释放。




---

备注: EOL 定义: end of life.

---

# CHAPTER 12

---

## CodeGen

---

TPU-MLIR 的 CodeGen 是 BModel 生成的最后一步，该过程目的是将 mlir 文件转换成最终的 bmodel。本章主要介绍模型/算子在本工程中的 CodeGen 工作流程。

### 12.1 主要工作

CodeGen 的目的在于将 mlir 文件转换成最终的 bmodel 文件输出，这个过程会执行各 op 的 CodeGen 接口，以生成 cmdbuf，并用 Builder 模版生成采用 flatbuffers 格式的最终模型。

### 12.2 工作流程

CodeGen 的大致工作流程可分为 3 个部分：指令生成、指令存储和指令取出以生成 Bmodel。具体来说：

指令生成：将不同硬件的后端函数封装到类，执行不同 op 的 CodeGen 接口，生成相应指令（二进制码）；

指令存储：通过 store\_cmd 将指令（二进制码）存储在指定数据结构中；

指令取出：所有 op 的二进制码全部生成完毕后，在编译器会调用 BM168X 系列类中封装的函数取走指令，最终生成 Bmodel。

其工作流程图如下：

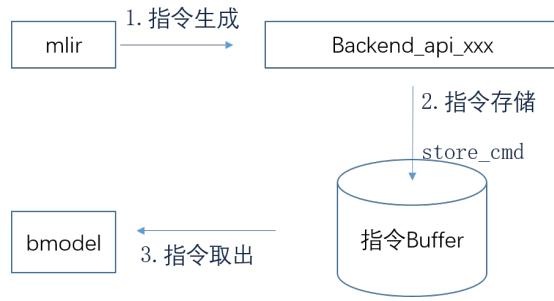


图 12.1: CodeGen 工作流程

下面对 CodeGen 过程中所需的数据结构进行介绍:

指令依据硬件的 engine 不同而有所差别, 比如 1684 有 GDMA 和 TIU, 而新架构的硬件 bm1690 会存在 sdma、cdma 等 engine。这里拿最通用的两种 engine 即 BDC(后更名为 TIU) 和 GDMA 为例:

```

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

```

bdc\_buffer: 存储 bdc 指令

gdma\_buffer: 存储 gdma 指令

gdma\_total\_id: gdma 指令存储的总数目

bdc\_total\_id: bdc 指令存储的总数目

gdma\_bytes: gdma 指令字节数

bdc\_bytes: bdc 指令字节数

### 12.3 TPU-MLIR 中 BM168X 及其相关类

TPU-MLIR 中 BM168X 及其相关类定义在 include/tpu\_mlir/Backend 文件夹下, 目的是将不同的硬件后端封装, 以实现后端与 Codegen 过程的隔离。其继承关系如下:

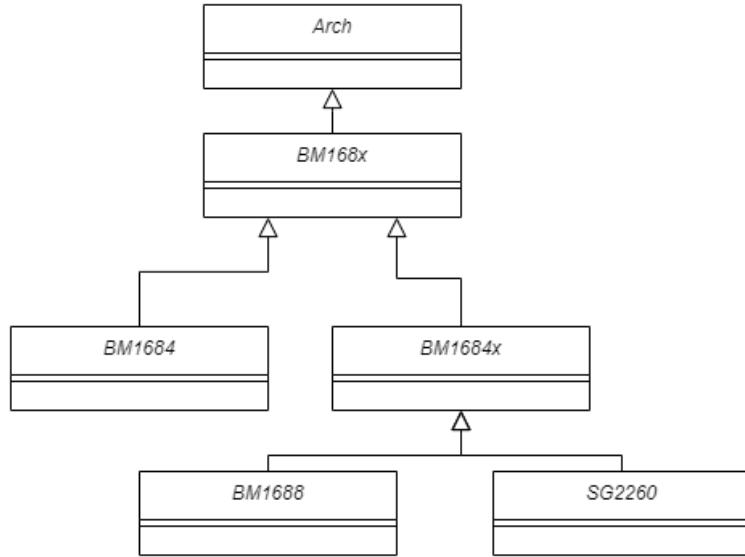


图 12.2: TPU-MLIR 中 BM168X 及其相关类继承关系

在一次运行中只存储一个类实例（设计模式中单例），该类初始化时候会经过：读取后端动态链接库、加载函数（设置后端的函数指针）、指令数据结构的初始化、设置一些硬件相关的参数例如 NPU\_NUM、L2\_SRAM 起始地址等。

## 12.4 后端函数的加载

后端作为一个动态库放入了 TPU-MLIR 工程里，具体的位置在 third\_party/nntoolchain/lib/libbackend\_xxx.so。后端函数的加载方式为：首先定义函数指针，再将动态库加载，使函数指针指向动态库中的函数。

以同步函数 tpu\_sync\_all 为例，由于之后要加上多核支持的，所以需要在相关后端 cmodel 库中定义好，

1. 注意必须和后端的函数名和参数保持一致 `typedef void (*tpu_sync_all)();`
2. 在类内部加入该函数成员 `tpu_sync_all dl_tpu_sync_all;`
3. 在该类 `load_functions` 函数的实现中加入宏，`CAST_FUNCTION(tpu_sync_all);` 该宏可以将 `dl_tpu_sync_all` 指向动态库中的函数。

获得到该类实例后即可使用动态库中的函数。

## 12.5 后端 Store\_cmd

后端 store\_cmd 的功能是在编译器调用算子的过程中，把配置的指令保存到约定空间。后端的重点函数在 store\_cmd.cpp 中，以 cmodel/src/store\_cmd.cpp; cmodel/include/store\_cmd.h 为例。

store\_cmd 分别有 EngineStorer 系列类和 CmdStorer 系列类：

1. EngineStoreInterface(接口类)、继承于 EngineStoreInterface 接口的 GDMAEngineStorer、BDEngineStorer 等具体类、EngineStorerDecorator (装饰类接口)、继承于 EngineStorerDecorator 的 VectorDumpEngineStorerDecorator 等具体装饰类。
2. CmdStorerInterface (接口)、继承于接口的 ConcretCmdStorer、StorerDecorator、VectorDumpStorerDecorator 具体装饰类。

关于类之间的关系与逻辑：

1. 使用单例设计模式，在 store\_cmd 中只存在一个 ConcretCmdStorer 类，该类中会存所有 EngineStorer 的类，当调用不同的 engine 时，会调用不同 EngineStorer，如下代码。

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

2. EngineStorer 功能为解析命令，VectorDumpEngineStorerDecorator 执行 EngineStorer 类中的 store 函数和 take\_cmds 函数，可将所有指令存储到 output\_ 中。

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

(续下页)

(接上页)

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

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

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

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

# CHAPTER 13

---

## MLIR 定义

---

本章介绍 MLIR 各个元素的定义, 包括 Dialect、Interface 等等

### 13.1 Top Dialect

#### 13.1.1 Operations

##### AddOp

###### 简述

加法操作,  $Y = coeff_0 * X_0 + coeff_1 * X_1$

###### 输入

- inputs: tensor 数组, 对应 2 个或多个输入 tensor

###### 输出

- output: tensor

###### 属性

- do\_relu: 结果是否做 Relu, 默认为 False
- relu\_limit: 如果做 Relu, 指定上限值, 如果是负数, 则认为没有上限
- coeff: 对应每个 tensor 的系数, 默认为 1.0

###### 输出

- output: 输出 tensor

**接口**

无

**范例**

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

**AvgPoolOp****简述**

将输入的 tensor 进行均值池化,  $S = \frac{1}{width * height} \sum_{i,j} a_{ij}$ 。大小给定的滑动窗口会依次将输入 tensor 进行池化

其中  $width$  和  $height$  表示 kernel\_shape 的宽度和高度。 $\sum_{i,j} a_{ij}$  则表示对 kernel\_shape 进行求和

**输入**

- input: tensor

**输出**

- output: tensor

**属性**

- kernel\_shape: 控制均值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**

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

## Depth2SpaceOp

### 简述

深度转空间操作,  $Y = Depth2Space(X)$

### 输入

- inputs: tensor

### 输出

- output: tensor

### 属性

- block\_h: tensor 高度改变的参数, i64 类型
- block\_w: tensor 宽度改变的参数, i64 类型
- is\_CRD: column-row-depth, 如果 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 true, 那么结果的形状为:  
 $[n, c * block_h * block_w, h/block_h, w/block_w]$ ,  
 否则结果的形状为:  $[n, c/(block_h * block_w), h * block_h, w * block_w]$

### 接口

无

### 范例

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

## BatchNormOp

### 简述

在一个四维输入 tensor 上执行批标准化 (Batch Normalization)。关于批标准化的更多细节可以参考论文《Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift》。

具体计算公式如下:

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

### 输入

- input: 四维输入 tensor
- mean: input 的均值 tensor
- variance: input 的方差 tensor

- gamma: 公式中的  $\gamma$  tensor, 可以为 None
- beta: 公式中的  $\beta$  tensor, 可以为 None

**输出**

- output: 结果 tensor

**属性**

- epsilon: 公式中的  $\epsilon$  常量, 默认为 1e-05
- do\_relu: 结果是否做 Relu, 默认为 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<1x3x27x27xf32>, tensor<3xf32>, tensor<3xf32>, tensor<3xf32>, F
  ↪ tensor<3xf32>) -> tensor<1x3x27x27xf32> loc("BatchNorm")
```

**CastOp**

(待补充)

**ClipOp****简述**

将给定输入限制在一定范围内

**输入**

- input: tensor

**输出**

- output: tensor

**属性**

- min: 给定的下限
- max: 给定的上限

**输出**

- output: 输出 tensor

**接口**

无

**范例**

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

## ConcatOp

### 简述

将给定的 tensor 序列在给定的维度上连接起来。所有的输入 tensor 或者都具有相同的 shape(待连接的维度除外), 或者都为空。

### 输入

- inputs: tensor 数组, 对应 2 个或多个输入 tensor

### 输出

- output: 结果 tensor

### 属性

- axis: 待连接的维度的下标
- do\_relu: 结果是否做 Relu, 默认为 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
˓→<1x3x27x27xf32>, tensor<1x3x27x27xf32>) -> tensor<1x6x27x27xf32> loc(
˓→"Concat")
```

## ConvOp

### 简述

对输入 tensor 执行二维卷积操作。

简单来说, 给定输入大小为  $(N, C_{\text{in}}, H, W)$ , 输出  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  的计算方法为:

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

其中  $\star$  是有效的 cross-correlation 操作,  $N$  是 batch 的大小,  $C$  是 channel 的数量,  $H, W$  是输入图片的高和宽。

### 输入

- input: 输入 tensor

- filter: 参数 tensor, 其形状为 ( $\text{out\_channels}$ ,  $\frac{\text{in\_channels}}{\text{groups}}$ ,  $\text{kernel\_size}[0]$ ,  $\text{kernel\_size}[1]$ ):
- bias: 可学习的偏差 tensor, 形状为 ( $\text{out\_channels}$ ).

### 输出

- output: 结果 tensor

### 属性

- kernel\_shape: 卷积核的尺寸
- strides: 卷积的步长
- pads: 输入的每一条边补充 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.Conv"(%0, %1) {kernel_shape = [3, 5], strides = [2, 1], pads = [4, 2]}  
↳ : (tensor<20x16x50x100xf32>, tensor<33x3x5xf32>) -> tensor  
↳ <20x33x28x49xf32> loc("Conv")
```

## DeconvOp

### 简述

对输入 tensor 执行反卷积操作。

### 输入

- input: 输入 tensor
- filter: 参数 tensor, 其形状为 ( $\text{out\_channels}$ ,  $\frac{\text{in\_channels}}{\text{groups}}$ ,  $\text{kernel\_size}[0]$ ,  $\text{kernel\_size}[1]$ ):
- bias: 可学习的偏差 tensor, 形状为 ( $\text{out\_channels}$ ).

### 输出

- output: 结果 tensor

### 属性

- kernel\_shape: 卷积核的尺寸
- strides: 卷积的步长

- pads: 输入的每一条边补充 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.Deconv"(%0, %1) {kernel_shape = (3, 5), strides = (2, 1), pads = (4,
˓→ 2)} : (tensor<20x16x50x100xf32>, tensor<33x3x5xf32>) -> tensor
˓→<20x33x28x49xf32> loc("Deconv")
```

**DivOp****简述**除法操作,  $Y = X_0/X_1$ **输入**

- inputs: tensor 数组, 对应 2 个或多个输入 tensor

**输出**

- output: tensor

**属性**

- do\_relu: 结果是否做 Relu, 默认为 False
- relu\_limit: 如果做 Relu, 指定上限值, 如果是负数, 则认为没有上限
- multiplier: 量化用的乘数, 默认为 1
- rshift: 量化用的右移, 默认为 0

**接口**

无

**范例**

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

## InputOp

(待补充)

## LeakyReluOp

### 简述

tensor 中每个元素执行 LeakyRelu 函数, 函数可表示为:  $f(x) = \text{alpha} * x$  for  $x < 0$ ,  $f(x) = x$  for  $x \geq 0$

### 输入

- input: tensor

### 输出

- output: tensor

### 属性

- alpha: 对应每个 tensor 的系数

### 接口

无

### 范例

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

## LSTMOp

### 简述

执行 RNN 的 LSTM 操作

### 输入

- input: tensor

### 输出

- output: tensor

### 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 是一个 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, 默认为 false
- bidirectional: 设置双向循环的 LSTM, 默认为 false
- batch\_first: 是否将 batch 放在第一维, 默认为 false

**接口**

无

**范例**

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

**LogOp****简述**

按元素计算给定输入张量的自然对数

**输入**

- input: tensor

**输出**

- output: tensor

**属性**

无

**接口**

无

**范例**

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

**MaxPoolOp****简述**

将输入的 tensor 进行最大池化

**输入**

- input: tensor

**输出**

- output: tensor

**属性**

- kernel\_shape: 控制均值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**

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

**MatMulOp****简述**二维矩阵乘法操作,  $C = A * B$ **输入**

- input: tensor: m\*k 大小的矩阵
- right: tensor: k\*n 大小的矩阵

**输出**

- output: tensor m\*n 大小的矩阵

**属性**

- bias: 偏差, 量化的时候会根据 bias 计算 bias\_scale, 可以为空
- do\_relu: 结果是否做 Relu, 默认为 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.MatMul"(%0, %1) {do_relu = false, relu_limit = -1.0} : (tensor
    ↪<3x4xf32>, tensor<4x5xf32>) -> tensor<3x5xf32> loc("matmul")
```

## MulOp

### 简述

乘法操作,  $Y = X_0 * X_1$

### 输入

- inputs: tensor 数组, 对应 2 个或多个输入 tensor

### 输出

- output: tensor

### 属性

- do\_relu: 结果是否做 Relu, 默认为 False
- relu\_limit: 如果做 Relu, 指定上限值, 如果是负数, 则认为没有上限
- multiplier: 量化用的乘数, 默认为 1
- rshift: 量化用的右移, 默认为 0

### 接口

无

### 范例

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

## MulConstOp

### 简述

和常数做乘法操作,  $Y = X * ConstVal$

### 输入

- inputs: tensor

### 输出

- output: tensor

### 属性

- const\_val: f64 类型的常量
- do\_relu: 结果是否做 Relu, 默认为 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

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

## PermuteOp

### 简述

改变 tensor 布局, 变化 tensor 数据维度的顺序, 将输入的 tensor 按照 order 给定的顺序重新布局

### 输入

- inputs: tensor 数组, 任意类型的 tensor

### 属性

- order: 指定重新布局 tensor 的顺序

### 输出

- output: 输出 tensor, 按 order 的顺序重新布局后的 tensor

### 接口

无

### 范例

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

## ReluOp

### 简述

tensor 中每个元素执行 ReLU 函数, 如果极限为零, 则不使用上限

### 输入

- input: tensor

### 输出

- output: tensor

### 属性

- 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

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

## ReshapeOp

### 简述

Reshape 算子, 返回一个给定形状的 tensor, 该 tensor 的类型和内部的值与输入 tensor 相同。reshape 可能会对 tensor 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

- input: tensor

### 输出

- output: tensor

### 属性

无

### 接口

无

### 范例

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

## ScaleOp

### 简述

Scale 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

- input: 输入 tensor
- scale: 保存 input 的放大倍数
- bias: 放大后加上的 bias

### 输出

- output: 结果 tensor

### 属性

- do\_relu: 结果是否做 Relu, 默认为 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**

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

**SigmoidOp****简述**

激活函数, 将 tensor 中元素映射到特定区间, 默认映射到 [0, 1], 计算方法为:

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

**输入**

- inputs: tensor 数组, 任意类型的 tensor

**属性**

- scale: 倍数, 默认是 1
- bias: 偏置, 默认是 0

**输出**

- output: 输出 tensor

**接口**

无

**范例**

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

**SiLUOp****简述**

激活函数,  $Y = \frac{X}{1+e^{-X}}$  或  $Y = X * Sigmoid(X)$

**输入**

- input: tensor 数组, 任意类型的 tensor

**属性**

无

**输出**

- output: 输出 tensor

**接口**

无

**范例**

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

**SliceOp****简述**

tensor 切片, 将输入的 tensor 的各个维度, 根据 offset 和 steps 数组中的偏移和步长进行切片, 生成新的 tensor

**输入**

- input: tensor 数组, 任意类型的 tensor

**属性**

- offset: 存储切片偏移的数组, offset 数组的索引和输入 tensor 的维度索引对应
- steps: 存储切片步长的数组, steps 数组的索引和输入 tensor 维度索引对应

**输出**

- output: 输出 tensor

**接口**

无

**范例**

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

**SoftmaxOp****简述**

对输入 tensor, 在指定 axis 的维度上计算归一化指数值, 计算的方法如下:

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

其中,  $\sum_{j=0}^{K-1} e^{\beta Z_j}$ , 在 axis 维度上做指数值求和, j 从 0 到 K-1, K 是输入 tensor 在 axis 维度上的尺寸。

例如: 输入 tensor 的尺寸为  $(N, C, W, H)$ , 在 axis=1 的通道上计算 Softmax, 计算方法为:

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

### 输入

- input: tensor 数组, 任意类型的 tensor

### 属性

- axis: 维度索引, 用于指定对输入 tensor 执行 Softmax 对应的维度, axis 可以取值 [-r, r-1], r 为输入 tensor 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

- output: 输出 tensor, 在指定维度做归一化指数值后的 tensor

### 接口

无

### 范例

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

## SqueezeOp

### 简述

对输入 tensor 进行指定维度的裁剪并返回裁剪后的 tensor

### 输入

- input: tensor

### 输出

- output: tensor

### 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<1x255x20x20xf32>) -> tensor
    ↳<1x255x20xf32> loc(#loc278)
```

## UpsampleOp

### 简述

上采样 op, 将输入 tensor 进行 nearest 上采样并返回 tensor

### 输入

tensor

### 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

### 输出

- output: tensor

### 接口

无

### 范例

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

## WeightOp

### 简述

权重 op, 包括权重的读取和创建, 权重会存到 npz 文件中。权重的 location 与 npz 中的 tensor 名称是对应关系。

### 输入

无

### 属性

无

### 输出

- output: 权重 Tensor

### 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<32x16x3x3xf32> loc("filter")
```

# CHAPTER 14

---

## 精度验证

---

### 14.1 整体介绍

#### 14.1.1 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。

#### 14.1.2 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。通常记录精度时

采用 IoU=0.5 时的 Average Precision (即 PASCAL VOC metric)。

**AveragePrecision(AP) :**

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

**APAcrossScales :**

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

**AverageRecall(AR) :**

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

**APAcrossScales :**

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

### 14.1.3 数据集

另外, 验证时使用的数据集需要自行下载, 分类网络使用 ILSVRC2012 的验证集 (共 50000 张图片, <https://www.image-net.org/challenges/LSVRC/2012/>)。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, 该情况下无需标签文件; 另外一种是所有图片均在同一个数据集目录下, 有一个额外的 txt 标签文件, 按照图片编号顺序每行用 1-1000 的数字表示每一张图片的类别。

目标检测网络使用 COCO2017 验证集 (共 5000 张图片, <https://cocodataset.org/#download>), 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。

## 14.2 精度验证接口

TPU-MLIR 的精度验证命令参考如下:

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

所支持的参数如下:

表 14.1: model\_eval.py 参数功能

参数名	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 和 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

## 14.3 精度验证样例

本节以 mobilenet\_v2 和 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。

### 14.3.1 mobilenet\_v2

#### 1. 数据集下载

下载 ILSVRC2012 验证集到 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, 因此不需要额外的标签文件。

#### 2. 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。具体使用方法请参照“用户界面”章节。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, 接下来我们将用该文件进行 INT8 对称量化模型的精度验证:

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

#### 3. 精度验证

使用 model\_eval.py 接口进行精度验证:

```
# 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如下:

```
# mobilenet_v2.mlir精度验证结果
2022/11/08 01:30:29 - INFO : idx:50000, top1:0.710, top5:0.899
INFO:root:idx:50000, top1:0.710, top5:0.899

# mobilenet_v2_bm1684x_int8_sym_tpu.mlir精度验证结果
2022/11/08 05:43:27 - INFO : idx:50000, top1:0.702, top5:0.895
INFO:root:idx:50000, top1:0.702, top5:0.895
```

### 14.3.2 yolov5s

#### 1. 数据集下载

下载 COCO2017 验证集到 datasets/val2017 目录下, 该目录下即包含 5000 张用于验证的图片。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 接口进行精度验证:

```
# F32 模型精度验证
$ 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
```

(续下页)

(接上页)

```
$ 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如下：

```
# 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
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
```

# CHAPTER 15

---

## QAT 量化感知训练

---

### 15.1 基本原理

相比训练后量化因为其不是全局最优而导致的精度损失，QAT 量化感知训练能做到基于 loss 优化的全局最优而尽可能的降低量化精度损失，其基本原理是：在 fp32 模型训练中就提前引入推理时量化导致的权重和激活的误差，用任务 loss 在训练集上来优化可学习的权重及量化的 scale 和 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的适应了，只要能保证推理和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。

### 15.2 tpu-mlir QAT 实现方案及特点

#### 15.2.1 主体流程

在训练过程中，用户调用模型 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（包括权重和 bias）前插入伪量化节点（可配置该节点的量化参数，比如 per-chan/layer、是否对称、量化比特数等），然后用户使用修改后模型进行正常的训练流程，完成少数几个轮次的训练后，调用转换部署 API 接口将训练过的模型转为 fp32 权重的 onnx 模型，提取伪量化节点中参数导出到量化参数文本文件中，最后将调优后的 onnx 模型和该量化参数文件输入到 tpu-mlir 工具链中，按前面讲的训练后量化方式转换部署即可。

### 15.2.2 方案特点

特点 1：基于 pytorch；QAT 是训练 pipeline 的一个附加 finetune 环节，只有与训练环境深度集成才能方便用户各种使用场景，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，方案会大不相同，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，能较方便实现模型 trace、伪量化节点插入、自定义模块替换等操作，大多数情况下，客户使用较少的用户配置即可完成量化感知训练。

特点 3：基于 SOPHGO-mq 训练框架，该框架基于商汤开源的 mqbench 修改，增加了对 SOPHGO 处理器量化特性的支持。

## 15.3 安装方法

建议在 SOPHGO 提供的 docker 镜像中使用 SOPHGO-mq，镜像可以使用 docker pull 命令获取：

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

此镜像预装了 torch2.3.0 版本和 cuda12.1，为 SOPHGO-mq 支持的最新版本，另外此镜像也支持 tpu-mlir 工具直接部署网络到处理器。

### 15.3.1 使用安装包安装

1、在 SOPHGO-mq 开源项目 <https://github.com/sophgo/sophgo-mq.git> 的 release 区获取最新的安装包，比如 sophgo\_mq-1.0.1-cp310-cp310-linux\_x86\_64.whl  
2、使用 pip 安装：pip3 install sophgo\_mq-1.0.1-cp310-cp310-linux\_x86\_64.whl

### 15.3.2 从源码安装

1、执行命令获取 github 上最新代码:git clone <https://github.com/sophgo/sophgo-mq.git>  
2、进入 SOPHGO-mq 目录后执行：

```
pip install -r requirements.txt #注:当前要求torch版本为2.3.0  
python setup.py install
```

3、执行 python -c ‘import sophgo\_mq’ 若没有返回任何错误，则说明安装正确，若安装有错，执行 pip uninstall sophgo\_mq 卸载后再尝试。

## 15.4 基本步骤

### 15.4.1 步骤 0：接口导入及模型 prepare

在训练文件中添加如下 python 模块 import 接口：

```
import torch
import torchvision.models as models
from sophgo_mq.prepare_by_platform import prepare_by_platform #初始化接口
from sophgo_mq.utils.state import enable_quantization, enable_calibration #校准和量化开关
from sophgo_mq.convert_deploy import convert_deploy #转换部署接口
    import tpu_mlir #tpu_
→mlir模块，引入之后可以实现一键式转换bmodel在处理器上部署
    from tools.model_runner import mlir_inference #tpu_
→mlir的推理模块，可以在量化感知训练阶段使用tpu_
→mlir的推理直接看到训练模型在处理器上的精度表现

#使用torchvision model zoo里的预训练resnet18模型
model = models.__dict__['resnet18'](pretrained=True)

#1.trace模型，使用字典来指定处理器类型为BM1690，量化模式为weight_
→activation，在该量化模式下，权重和激活都会被量化。指定量化策略为CNN类型
extra_prepare_dict = {
'quant_dict': {
    'chip': 'BM1690',
    'quantmode': 'weight_activation',
    'strategy': 'CNN',
    },
}
model_quantized = prepare_by_platform(model, prepare_custom_config_dict=extra_prepare_
→dict)
```

当上面接口选择处理器为 BM1690 时，此时默认的量化配置如下图所示：

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

上图量化配置中各项从上到下依次意义为：

- 1、权重量化方案为：per-channel 对称 8bit 量化，scale 系数不是 power-of-2，而是任意的
- 2、激活量化方案为：per-layer 对称 8bit 量化
- 3/4、权重和激活伪量化方案均为：LearnableFakeQuantize 即 LSQ 算法
- 5/6、权重的动态范围统计及 scale 计算方案为：MinMaxObserver，激活的为带 EMA 指数移动平均的 EMAMinMaxObserver

### 15.4.2 步骤 1：用于量化参数初始化的校准及量化训练

设置好合理的训练超参数，就可以开始量化感知训练，建议如下：

- 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；

```
#1. 打开校准开关，容许在模型上推理时用pytorch observer对象来收集激活分布并计算初始scale和zp
enable_calibration(model_quantized)
# 校准循环
for i, (images, _) in enumerate(cali_loader):
    model_quantized(images) #只需要前向推理即可
#3. 打开伪量化开关，在模型上推理时会调用QuantizeBase子对象来进行伪量化操作引入量化误差
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)
    #后向反传梯度
    loss.backward()
    #更新权重和伪量化参数
    optimizer.step()
```

### 15.4.3 步骤 2：导出调优后的 fp32 模型及量化参数文件

```
#batch-size可根据需要调整，不必与训练batch-size一致
input_shape={'input': [4, 3, 224, 224]}
# 指定导出模型类型为CNN
net_type='CNN'
#4.
→导出前先融合conv+bn层（前面train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。
convert_deploy(model_quantized, net_type, input_shape)
```

### 15.4.4 步骤 3：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。在训练阶段引入 tpu\_mlir，可以直接使用 tpu\_mlir 的推理接口直接模拟模型在处理器上的运行，从而了解训练进展，如果使用此接口，则在训练过程中就已经转化部署了模型文件，生成了 bmodel。一般可以在传统的验证流程中将模型推理替换为 mlir\_inference，输入输出为 numpy 数组，调用 tpu\_mlir 推理的示例接口如下：

```
import tpu_mlir
from tools.model_runner import mlir_inference
...
for i, (images, target) in enumerate(bmodel_test_loader):
```

(续下页)

(接上页)

```

images = images.cpu()
target = target.cpu()
inputs['data'] = images.numpy()
output = mlir_inference(inputs, mlir_model_path, dump_all = False)
output = torch.from_numpy(list(output.values())[0])
loss = criterion(output, target)

```

## 15.5 使用样例-resnet18

执行 application/imagenet\_example/main.py 对 resnet18 进行 qat 训练，命令如下：

```

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

```

在上面命令输出日志中有如下图 (原始 onnx 模型精度) 中原始模型的精度信息 (可与官方网页上精度进行比对以确认训练环境无误，比如官方标称：Acc@1 69.76 Acc@5 89.08，链接为:[https://pytorch.apache.org/#/docs/1.0/torchvision\\_models](https://pytorch.apache.org/#/docs/1.0/torchvision_models)) :

```

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

```

图 15.1: 原始 onnx 模型精度

完成 qat 训练后，跑带量化节点的 eval 精度，理论上在 tpu-mlir 的 int8 精度应该与此完全对齐，如下图 (resnet18 qat 训练精度)：

最终输出目录如下图 (resnet18 qat 训练输出模型目录)：

上图中 resnet18\_ori.onnx 为 pytorch 原始模型所转的 onnx 文件，将这个 resnet18\_ori.onnx 用 tpu-mlir 工具链进行 PTQ 量化，衡量其对称和非对称量化精度作为比较的 baseline。其中的 resnet18\_cali\_table\_from\_sophgo\_mq 为导出的量化参数文件，内容如下图 (resnet18 qat 量化参数表样例)：

```

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

```

图 15.2: resnet18 qat 训练精度

```
≡ resnet18_cali_table_from_sophgo_mq
{ } resnet18_clip_ranges.json
≡ resnet18_deploy_model.onnx
≡ resnet18_ori.onnx
≡ resnet18_q_table_from_sophgo_mq_sophgo_tpu
≡ resnet18.onnx
```

图 15.3: resnet18 qat 训练输出模型目录

图 15.4: resnet18 qat 量化参数表样例

a、上图中第一行红色框内:work\_mode 为 QAT\_all\_int8 表示整网 int8 量化，可以在 [QAT\_all\_int8、QAT\_mix\_prec] 中选择，还会带上量化参数: 对称非对称等参数。

b、上图中 472\_Relu\_weight 表示是 conv 权重的经过 QAT 调优过的 scale 和 zp 参数，第 1 个 64 表示后面跟着 64 个 scale，第 2 个 64 表示后面跟着 64 个 zp，tpu-mlir 会导入到 top 层 weight\_scale 属性中，在 int8 lowering 时若该属性存在就直接使用该属性，不存在就按最大值重新计算。

c、上面的 min、max 是非对称量化时根据激活的 qat 调优过的 scale、zp 以及 qmin、qmax 算出来，threshold 是在对称量化时根据激活的 scale 算出来，两者不会同时有效。

## 15.6 QAT 测试环境

量化感知训练输出的网络最终要在 SOPHGO 处理器上运行，其精度可以使用端到端的推理验证程序来验证，一般在模型部署的环境中测试即可。在单机上也可以在 tpu\_mlir 阶段使用 tpu\_mlir 提供的模型验证程序在 CPU 上模拟验证，特别是简单的分类网络可以比较方便的验证其精度。一般步骤如下：

### 15.6.1 添加 cfg 文件

进入 tpu-mlir/regression/eval 目录，在 qat\_config 子目录下增加 {model\_name}\_qat.cfg，比如如下为 resnet18\_qat.cfg 文件内容：

```
dataset=${REGRESSION_PATH}/dataset/ILSVRC2012
test_input=${REGRESSION_PATH}/image/cat.jpg
input_shapes=[[1,3,224,224]] #根据实际shape修改
resize_dims=256,256 #下面为图片预处理参数，根据实际填写
mean=123.675,116.28,103.53
scale=0.0171,0.0175,0.0174
pixel_format=rgb
int8_sym_tolerance=0.97,0.80
int8_asym_tolerance=0.98,0.80
debug_cmd=use_pil_resize
```

也可增加 {model\_name}\_qat\_ori.cfg 文件：将原始 pytorch 模型量化，作为 baseline，内容可以和上面 {model\_name}\_qat.cfg 完全一样；

### 15.6.2 修改并执行 run\_eval.py

下图 (run\_eval 待测模型列表及参数) 中在 postprocess\_type\_all 中填写更多不同精度评估方式的命令字符串，比如图中已有 imagenet 分类和 coco 检测精度计算字符串；下图 (run\_eval 待测模型列表及参数) 中 model\_list\_all 填写模型名到参数的映射，比如：resnet18\_qat 的 [0,0]，其中第 1 个参数表示用 postprocess\_type\_all 中第 1 个的命令串，第 2 个参数表示用 qat\_model\_path 第 1 个目录（以逗号分隔）：

根据需要配置上图 postprocess\_type\_all 和 model\_list\_all 数组后，执行下面 run\_eval.py 命令：

```

`~ postprocess_type_all=[  

|   "--count 0 --dataset_type imagenet --postprocess_type topx --dataset /workspace/datasets/ILSVRC2012_img_val_with_subdir/",  

|   "--count 0 --dataset_type coco --postprocess_type coco_mAP --dataset /workspace/datasets/coco_for_mlir_test/val2017 --coco_annotation_file /workspace/datasets/coco/annotations/instances_val2017.json"  

]  

`~ model_list_all=[  

|   # object detection  

|   # "yolov5s_qat_ori": [1,1],  

|   # "yolov5s_qat": [1,1],  

|   # classification  

|   "resnet18_qat_ori": [0,0],  

|   "resnet18_qat": [0,0],
]

```

图 15.5: run\_eval 待测模型列表及参数

```

python3 run_eval.py  

#qat验证模式， 默认是使用tpu-mlir/regression/config中配置进行常规的模型精度测试  

--qat_eval  

--fast_test      #正式测试前的快速测试（只测试30张图的精度），确认所有case都能跑起来  

--pool_size 20  #默认起10个进程来跑，若机器闲置资源较多，可多配点  

--batch_size 10  #qat导出模型的batch-size，默认为1  

--qat_model_path '/workspace/classify_models/,/workspace/yolov5/qat_models'  

#qat模型所在目录，比如model_list_all[‘resnet18_qat’][1]的取值为0，表示其模型目标在qat_  

model_path的第一个目录地址:/workspace/classify_models/  

--debug_cmd use_pil_resize  #使用pil resize方式

```

测试后或测试过程中，查看以 {model\_name}\_qat 命名的子目录下以 log\_ 开头的 model\_eval 脚本输出日志文件，比如:log\_resnet18\_qat.mlir 表示对本目录中 resnet18\_qat.mlir 进行测试的日志；log\_resnet18\_qat\_bm1684x\_tpu\_int8\_sym.mlir 表示对本目录中 resnet18\_qat\_bm1684x\_tpu\_int8\_sym.mlir 进行测试的日志

## 15.7 使用样例-yolov5s

在 application/yolov5\_example 中执行如下命令可启动 qat 训练:

```

CUDA_VISIBLE_DEVICES=0 python train.py \  

--cfg=yolov5s.yaml \  

--weights=yolov5s.pt \  

--data=coco.yaml \  

--epochs=5 \  

--output_path=./ \  

--batch-size=8 \  

--quantize \

```

完成训练后，采取和前面 resnet18 一样的测试、转换部署流程即可。

## 15.8 使用样例-bert

在 application/nlp\_example 中执行如下命令可启动 qat 训练

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

## TpuLang 接口

---

本章节主要介绍使用 TpuLang 转换模型的流程。

### 16.1 主要工作

TpuLang 提供了 mlir 对外的接口函数。用户通过 TpuLang 可以直接组建自己的网络，将模型转换为 Top 层（硬件无关层）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。

### 16.2 工作流程

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
    - 输入参数转为 dict 格式；
    - 创建输出 tensor；
    - 设置 tensor 的量化参数（scale, zero\_point）；
    - 创建 op(op\_type, inputs, outputs, params) 并 insert 到 graph 中。
  3. 设置模型的输入输出 tensor。得到全部模型信息。
  4. 初始化 TpuLangConverter(initMLIRImporter)
  5. generate\_mlir
-

- 依次创建 input op, 模型中间 nodes op 以及 return op, 并将其补充到 mlir 文本中 (如果该 op 带有权重, 则会额外创建 weight op)
- 输出 (Output)
    - 将生成的文本转为 str 并保存为 “.mlir” 文件
    - 将模型权重 (tensors) 保存为 “.npz” 文件
  - 结束: 释放 graph。

TpuLang 转换的工作流程如图所示 (TpuLang 转换流程)。

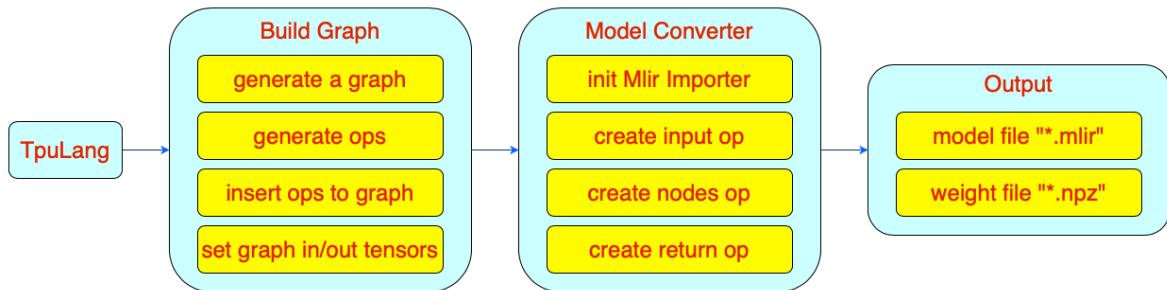


图 16.1: TpuLang 转换流程

### 补充说明:

- op 接口需要:
  - op 的输入 tensor(即前一个算子的输出 tensor 或 graph 输入 tensor, coeff);
  - 从接口中提取的 attrs。Attrs 会通过 MLIRImporter 设定为与 TopOps.td 定义一一对应的属性
  - 如果接口中包括量化参数 (scale, zero\_point), 则该参数对应的 tensor 需要设置(或检查)量化参数.
  - 返回该 op 的输出 tensor(tensors)
- 在所有算子都插入 graph, 并设置 graph 的 input/output tensors 之后, 才会启动转换到 mlir 文本的工作。该部分由 TpuLangConverter 来实现。
- TpuLang Converter 转换流程与 onnx 前端转换流程相同, 具体参考 (前端转换).

## 16.3 算子转换样例

本节以 Conv 算子为例, 将单 Conv 算子模型转换为 Top mlir

```

import numpy as np

def model_def(in_shape):
    tpul.init("BM1684X")

```

(续下页)

(接上页)

```

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

```

### 单 Conv 模型

转换流程为：

1. 接口定义

conv 接口定义如下：

```

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

```

### 参数说明

- input: Tensor 类型，表示输入 Tensor，4 维 NCHW 格式。
- weight: Tensor 类型，表示卷积核 Tensor，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。
- bias: Tensor 类型，表示偏置 Tensor。为 None 时表示无偏置，反之则要求 shape 为 [1, oc, 1, 1]。
- dilation: List[int]，表示空洞大小，取 None 则表示 [1,1]，不为 None 时要求长度为 2。List 中顺序为 [长, 宽]
- pad: List[int]，表示填充大小，取 None 则表示 [0,0,0,0]，不为 None 时要求长度为 4。List 中顺序为 [上, 下, 左, 右]
- stride: List[int]，表示步长大小，取 None 则表示 [1,1]，不为 None 时要求长度为 2。List 中顺序为 [长, 宽]

- groups: int 型, 表示卷积层的组数。若 ic=oc=groups 时, 则卷积为 depthwise conv
- out\_dtype: string 类型或 None, 表示输出 Tensor 的类型。输入 tensor 类型为 float16/float32 时, 取 None 表示输出 tensor 类型与输入一致, 否则取 None 表示为 int32。取值范围: /int32/uint32/float32/float16
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动产生名称。

在 TopOps.td 中定义 Top.Conv 算子, 算子定义如图所示 (Conv 算子定义)

```

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

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

```

图 16.2: Conv 算子定义

## 2. 构建 Graph

- 初始化模型: 创建空 Graph。
- 模型输入: 给定 shape 与 data type 创建输入 tensor x。此处也可以指定 tensor name。
- conv 接口:

- 调用 conv 接口，指定输入 tensor 以及输入参数。

- 生成输出 tensor

```
output = Tensor(dtype=out_dtype, name=out_name)
```

- attributes，将输入参数打包成 ([Conv 算子定义](#)) 定义的 attributes

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

- 定义输出 tensor

- 插入 conv op，将 Top.ConvOp 插入到 Graph 中。

- 返回输出 tensor

- 设置 Graph 的输入，输出 tensors。

### 3. init\_MLIRImporter:

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，如图所示 ([初始 mlir 文本](#))。

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

图 16.3: 初始 mlir 文本

### 4. generate\_mlir

- build input op，生成的 Top.inputOp 会被插入到 MLIRImporter.mlir\_module 中。
- 调用 Operation.create 来创建 Top.ConvOp，而 create 函数需要的参数有：
  - 输入 op：从接口定义可知，Conv 算子的 inputs 一共包含了 input, weight 与 bias，inputOp 已被创建好，weight 与 bias 的 op 则通过 getWeightOp() 创建。
  - output\_shape：利用 Operator 中存储的输出 tensor 中获取其 shape。
  - Attributes：从 Operator 中获取 attributes，并将 attributes 转换为 MLIRImporter 识别的 Attributes

Top.ConvOp 创建后会被插入到 mlir 文本中

- 根据 output\_names 从 operands 中获取相应的 op, 创建 return\_op 并插入到 mlir 文本中。到此为止, 生成的 mlir 文本如图所示 (完整的 mlir 文本)。

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

图 16.4: 完整的 mlir 文本

## 5. 输出

将 mlir 文本保存为 Conv\_origin.mlir, tensors 中的权重保存为 Conv\_TOP\_F32\_all\_weight.npz。

## 16.4 Tpulang 接口使用方式

目前 TpuLang 只适用于推理框架的推理部分。类 tensorflow 等框架的静态图, 使用 TpuLang 进行网络集成时, 用户需要首先使用 tpul.init('processor') 初始化 (processor 可以是 BM1684X 或者 BM1688), 然后准备 tensor, 接着使用 operator 构建网络, 最后调用 tpul.compile 接口编译生成 bmodel。下面详细介绍一下每一步怎么做, 以下使用到的各种接口 (tpul.init, deinit, Tensor 以及算子接口等) 都可以在 appx02(附录 02: TpuLang 的基本元素) 中查看到详细介绍。

以下步骤假定当前已经完成 tpu-mlir 发布包的加载。

### 16.4.1 初始话

具体的定义参见 (初始化函数)

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

tpul.init('BM1684X')
```

### 16.4.2 准备 Tensor

具体的定义参见 ([tensor](#))

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

### 16.4.3 构建 graph

接着利用现有的 OP(Operator) 和刚刚准备好的 Tensor 构建 graph，下面是一个简单的模型构建示例：

```
def conv_op(x,
            kshape,
            stride,
            pad=None,
            group=1,
            dilation=[1, 1],
            bias=False,
            dtype="float32"):
    oc = kshape[0]
    weight_data = np.random.randn(*kshape).astype(np.float32)
    weight = tpul.Tensor(dtype=dtype, shape=kshape, data=weight_data, ttype="coeff")
    bias_data = np.random.randn(oc).astype(np.float32)
    bias = tpul.Tensor(dtype=dtype, shape=[oc], data=bias_data, ttype="coeff")
    conv = tpul.conv(x,
                     weight,
                     bias=bias,
                     stride=stride,
                     pad=pad,
                     dilation=dilation,
                     group=group)
    return conv

def model_def(x):
    conv0 = conv_op(x, kshape=[32, 1, 5, 5], stride=[1, 1], pad=[2, 2, 2, 2], dtype='float32')
    relu1 = tpul.relu(conv0)
    maxpool2 = tpul.maxpool(relu1, kernel=[2, 2], stride=[2, 2], pad=[0, 0, 0, 0])
    conv3 = conv_op(maxpool2, kshape=[64, 32, 5, 5], stride=[1, 1], pad=[2, 2, 2, 2], F
                    dtype='float32')
    relu4 = tpul.relu(conv3)
    maxpool5 = tpul.maxpool(relu4, kernel=[2, 2], stride=[2, 2], pad=[0, 0, 0, 0])
    conv6 = conv_op(maxpool5, kshape=[1024, 64, 7, 7], stride=[1, 1], dtype='float32')
    relu7 = tpul.relu(conv6)
    softmax8 = tpul.softmax(relu7, axis=1)
    return softmax8

y = model_def(x)
```

#### 16.4.4 compile

调用 tpul.compile 函数 (compile), 编译完成后会得到 example\_f32.bmodel :

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

#### 16.4.5\_deinit

具体的定义参见 ([反初始化函数](#))

```
tpul._deinit()
```

#### 16.4.6 deploy

最后使用 model\_deploy.py 完成模型部署，具体使用方法参考定义 (model\_deploy)。

## 用户自定义算子

---

### 17.1 概述

tpu-mlir 当前已经包含了丰富的算子库，可以满足大部分神经网络模型的编译需求。但在某些场景下，可能需要用户自定义算子来实现对张量的计算。如：

1. tpu-mlir 还未支持的算子，且无法通过其它算子组合实现
2. 算子为用户私有，未对公众开源
3. 使用多个算子 API 组合无法取得最佳计算性能，直接从 tpu-kernel 层自定义运算可以提高运行效率

自定义算子功能允许用户自由使用 tpu-kernel 中的接口，自定义 tensor 在 tpu 上的计算，并将这一计算过程封装为后端算子（后端算子开发请参考 TPU-KERNEL 开发参考手册）。其中，后端算子计算涉及到 global layer 与 local layer 相关操作：

- a. 算子必须实现 global layer，global layer 的输入和输出数据都放在 ddr 中，数据需要从 global mem 搬运到 local mem 中执行运算，再将结果搬运至 global mem。其优点是 local mem 可以任意使用，比较灵活；缺点是会产生较多的 gdma 搬运，tpu 利用率较低。
- b. 算子根据需要实现 local layer，local layer 的输入和输出的数据都放在 local mem 中，可以与其他 layer 组合进行 LayerGroup 优化，避免该 layer 计算时数据要搬入搬出到 global mem 中。其优点是可以节省 gdma 搬运，运算效率高；缺点是比较复杂，local mem 在模型部署时会提前分配好，不可任意使用，在部分算子中无法实现，关于 local layer 的更多细节请参考 LayerGroup 章节。
- c. 用户还需要实现自定义算子的一些补丁函数，用于在编译阶段进行正确性对比，形状推断以及实现更复杂的 local layer 等。

完成后端算子的封装后，前端可以通过 tpulang 或 caffe 构建包含自定义算子的模型，并最终通过 tpu-mlir 的模型转换接口完成模型部署。本章主要介绍在 tpu-mlir 发布包中使用自定义算子的流程。

## 17.2 自定义算子添加流程

注意：在下文中，{op\_name} 表示算子的名字，且字符串长度应不超过 20。{processor\_arch} 表示架构名称，当前可选 BM1684X 和 BM1688。

### 17.2.1 TpuLang 自定义算子添加

#### 1. 加载 tpu-mlir

以下操作需要在 Docker 容器中。关于 Docker 的使用，请参考 Docker 配置。

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

envsetup.sh 会添加以下环境变量：

表 17.1: 环境变量

变量名	值	说明
TPUC_ROOT	tpu-mlir_xxx	解压后 SDK 包的位置
MODEL_ZOO_PATH	\${TPUC_ROOT}/../model-zoo	model-zoo 文件夹位置，与 SDK 在同一级目录
REGRESSION_PATH	\${TPUC_ROOT}/regression	regression 文件夹的位置

envsetup.sh 对环境变量的修改内容为：

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

#### 2. 定义参数结构体与解析函数

- a. 在 \$TPUC\_ROOT/customlayer/include/backend\_custom\_param.h 中定义算子参数的结构体，该结构体会被应用在后续步骤中的各个函数里。结构体示例如下：

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

- b. 用户需要根据算子所需的参数在 \$TPUC\_ROOT/customlayer/include/param\_parser.h 中实现相应的函数用于解析由工具链前端传递过来的参数。工具链前端的参数是通过 custom\_param\_t 数组的指针进行传递，其中，数组的第一个元素是保留的，从第二个元素开始，每个元素对应前端的一个属性，每个 custom\_param\_t 结构体中包含了一个参数的信息，参数值会被存放在相应数据类型（其中包含了整数，浮点数，整数数组与浮点数数组）的 custom\_param\_t 成员变量中。参数的顺序与用户后续调用 tpulang 接口时提供的参数顺序相同。custom\_param\_t 结构体的定义如下：

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

解析函数的示例如下：

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

### 3. 编译器补丁

有时候，需要对编译器进行修改，以对不同的自定义算子在不同参数下的编译行为进行控制，这时候就需要添加一些补丁。当前已支持以下补丁函数（在文件夹./plugin 中定义）：

- a. （必选）推理函数。此补丁函数用于 TOP 层和 TPU 层的数据比对。补丁函数形式如下：

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

其中，input\_shapes 为输入张量形状的数组，input\_dims 为输入张量维度的数组。inputs 表示输入张量的指针数组，outputs 表示输出张量的指针数组。

- b. （可选）形状推断函数。此补丁函数用于 TOP 层形状推断，若不实现，默认只有一个输入一个输出，且输出形状跟输入形状相同。补丁函数形式如下：

```
void shape_inference_absadd(void* param, int param_size, const int (*input_
    ↵shapes)[MAX_SHAPE_DIMS],
    const int* input_dims, int (*output_shapes)[MAX_SHAPE_DIMS], int* output_
    ↵dims);
```

其中，`input_shapes/output_shapes` 为输入/出张量形状的数组，`input_dims/output_dims` 为输入/出张量维度的数组。

- c. (可选) 强制动态运行。某些算子的形状会动态变化 (如 NonZero 算子)，即使在静态编译下也需要强制动态运行。补丁函数形式如下：

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

若不提供该函数，则默认 `{op_name}` 对应的算子必须静态运行。

- d. (可选) 是否支持与其他算子组合 (用于 local layer)。补丁函数形式如下：

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

若不提供该函数，则默认 `{op_name}` 对应的算子不支持与其他算子组合，即强制走 global layer。否则，需要实现对应的 local layer 调用接口 `api_xxx_local` 和 `api_xxx_local_bfsz` (可选)。

- e. (可选) 在支持与其他算子组合时，是否允许对轴 `axis` 进行切割。补丁函数形式如下：

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

若不提供该函数，则默认与其他算子组合时，`{op_name}` 对应的算子允许对所有的轴进行切割。

- f. (可选) 切片反向推导函数，同样应用于 local layer (详情请参考 LayerGroup 章节)。补丁函数形式如下：

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

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

其中，`in_idx` 和 `in_slice` 分别表示指向该层输入张量切片的索引和大小的指针，`out_idx` 和 `out_slice` 表示该层输出张量切片的索引索引和大小。若不提供该函数，则 `in_idx` 指向的数值与 `out_idx` 相同，`in_slice` 指向的数值与 `out_slice` 相同。

#### 4. 编写后端算子

后端算子可基于 tpu-kernel 编写 (4.1)，也可基于 ppl 编写 (4.2)

##### 4.1 基于 tpu-kernel 编写后端算子

假定当前处于 `$TPUC_ROOT/customlayer` 路径下：

- a. 在./include/tpu\_impl\_custom\_ops.h 头文件中，声明 global layer 与 local layer 的自定义算子函数

```
void tpu_impl_{op_name}_global // 必选
void tpu_impl_{op_name}_local // 可选
```

- b. 在./src 下添加 tpu\_impl\_{op\_name}.c 文件，在其中调用 tpu-kernel 接口实现自定义算子 kenel 函数。
- c. 在./src 下添加 interface\_{op\_name}.c 文件，在其中实现自定义算子调用接口：

```
void api_{op_name}_global // 必选，用于调用 void tpu_impl_{op_name}_global
void api_{op_name}_local // 可选，用于调用 void tpu_impl_{op_name}_local
```

## 4.2 基于 ppl 编写后端算子

假定当前处于 \$TPUC\_ROOT/customlayer 路径下：

- a. 在./PplBackend/src 下导入 {op\_name}.pl (ppl kernel 定义与实现)
- b. 在./PplBackend/src 下导入 {op\_name}\_tile.cpp (切分函数，指定 dtype 对应的后端实现)

```
// kernelFunc 定义和函数名{op_name}.pl 中保持一致
using KernelFunc = int (*)(global_addr_t, global_addr_t,
                           float, int, int, int, int, bool);

int {op_name}_tiling/{op_name}(...) { // 必选
    KernelFunc func;
    if (dtype == SG_DTYPE_FP32) {
        func = {op_name}_f32;
    } else if (dtype == SG_DTYPE_FP16) {
        func = {op_name}_f16;
    } else if (dtype == SG_DTYPE_BFP16) {
        func = {op_name}_bf16;
    }
    ...
} else {
    assert(0 && "unsupported dtype");
}
// 切分函数 (可选)
...
}
```

- c. 在./PplBackend/src 下导入 {op\_name}\_api.c (接口函数)

```
extern int {op_name}_tiling/{op_name}(...); // 必选
void api_addconst_global/local(..., const void *param) { // 必选
    PARSE_PARAM({op_name}, {op_name}_param, param);
```

(续下页)

(接上页)

```
{op_name}_tiling/{op_name}(...);  
}
```

### 1. 编写算子通用接口

在./src 目录下添加自定义算子函数的调用接口:

```
int64_t api_{op_name}_global_bfsz (可选, 计算 global layer 需要的缓存大小)
```

```
int api_{op_name}_local_bfsz (可选, 计算 local layer 需要的缓存大小, 缓存用于存储计算的中间结果, 提前计算用于 LayerGroup 搜索 layer 间的最佳组合)
```

```
void type_infer_{op_name} (可选, 动态时使用, 从输入的形状和数据类型推理出输出的形状和数据类型, 若不实现, 则默认只有一个输入和一个输出, 且输出的形状和数据类型与输入的形状和数据类型相同)
```

```
void slice_infer_{op_name} (可选, 动态时使用, 从输入的切片推理出输出的切片, 若不实现, 则默认只有一个输入和一个输出, 且输出的切片与输入的切片相同)
```

### 6. 注册后端算子调用接口

在 register\_ops.cmake 中添加算子的名字以注册自定义算子:

```
register_custom_op({op_name}) // 4.1 基于tpu-kernel编写后端算子  
// OR  
register_custom_ppl_op({op_name}) // 4.2 基于ppl编写后端算子
```

假如自定义算子存在 local layer, 则需要注册一下:

```
register_custom_local_op({op_name}) // 4.1 基于tpu-kernel编写后端算子  
// OR  
register_custom_ppl_local_op({op_name}) // 4.2 基于ppl编写后端算子
```

假如自定义算子 global layer 需要缓存, 则需要注册一下:

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

假如自定义算子 local layer 需要缓存, 则需要注册一下:

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

### 7. 编译并安装动态库

先初始化环境:

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

然后需要完成补丁的编译（得到 libplugin\_custom.so）：

```
rebuild_custom_plugin
```

自定义算子后端接口的编译（得到 libbackend\_custom.so）：

```
rebuild_custom_backend
```

之后根据实际使用场景编译对应的固件（用于动态算子）：

- a. CMODEL 模式（得到 libcmodel\_custom\_xxx.so）

```
rebuild_custom_firmware_cmodel {processor_arch}
```

- b. SoC 模式（得到 libxxx\_kernel\_module\_custom\_soc.so）

```
rebuild_custom_firmware_soc {processor_arch}
```

- c. PCIe 模式（得到 libxxx\_kernel\_module\_custom\_pcie.so）

```
rebuild_custom_firmware_pcie {processor_arch}
```

至此我们就完成了自定义算子后端部分的工作。

## 8. 调用 TpuLang 构建模型

有关如何使用 TpuLang 的说明，请参阅“TPULang 接口”部分。

TpuLang 提供了 TpuLang.custom 接口来在工具链前端构建自定义算子（请确保 op\_name 与后端算子的名称匹配）：注意，params 应该是 python 中的字典，其 key 应该是一个表示参数名称的字符串，值应该是整数或浮点数，或者是整数或浮点数的列表（列表的长度不应大于 16）。在构建神经网络时，对于相同的自定义运算符和相同的键，键的数量和顺序应保持相同，如果其值为列表，则长度应保持相同。

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

(续下页)

(接上页)

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

### a. 定义自定义算子的 tpulang 接口

为了方便起见,可以在文件 \$TPUC\_ROOT/customlayer/python/my\_tpulang\_layer.py 中标准化自定义运算符:

```
import transform.TpuLang as tpul  
  
class xxx:  
    @staticmethod  
    def native(...):  
        ...  
        return ...  
    @staticmethod  
    def tpulang(inputs, ...):  
        params = dict(...)  
        outputs = tpul.custom(  
            tensors_in=inputs,  
            op_name={op_name},  
            params=params,  
            out_dtypes=...)  
        return outputs
```

其中 native 函数用于计算自定义层的参考输出数据。tpulang 函数使用 TpuLang.custom 函数构造自定义层。

### b. 单元测试

定义完自定义算子后, 需要测试一下这个接口是否可靠。在目录 \$TPUC\_ROOT/customlayer/test\_if/unittest 中, 创建一个名为 test\_{op\_name}.py 的 python 文件。在此文件中, 创建一个派生自 TestTPULangCustom 的类并创建测试函数。

下面的 shell 命令将用于执行单元测试:

```
run_custom_unittest {processor_arch}
```

## 9. 上卡测试

当网络中存在动态自定义算子时, bmodel 中包含的固件可能无法使 bmrt\_test 正常工作, 这时就需要替换固件了, 使用 shell 命令可以达到这一目标:

```
tpu_model --kernel_update xxx.bmodel libxxx_kernel_module_custom_soc.so  
↳#SoC模式下  
  
tpu_model --kernel_update xxx.bmodel libxxx_kernel_module_custom_pcie.so  
↳#PCIe模式下
```

### 17.2.2 Caffe 自定义算子添加

#### 1. 定义 Caffe 的自定义算子

要 定 义 Caffe 的 自 定 义 算 子，你 需 要 在 \$TPUC\_ROOT/customlayer/python/my\_caffe\_layer.py 文件中 定 义 一 个 类，该类继承自 caffe.Layer，并根据需要重写 setup, reshape, forward 和 backward 函数。

#### 2. 实现自定义算子前端转换函数

通过 Python 实现的自定义算子的 caffe 层类型为 “Python”，需要在 \$TPUC\_ROOT/customlayer/python/my\_converter.py 中的 MyCaffeConverter 类里根据之前的自定义算子定义一个针对 caffe 层类型为 “Python”的前端算子转换函数。完成转换函数后便可通过 MyCaffeConverter 对包含自定义算子的 Caffe 模型进行前端转换。

定义完成后，可以调用 my\_converter.py 接口进行 Top MLIR 转换：

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

后端部分与 “TpuLang 自定义算子添加” 中的步骤相同，此处不再赘述。

## 17.3 自定义算子示例

本节内容假定已经完成了 tpu-mlir 发布包加载。

### 17.3.1 TpuLang 示例

本小节提供了一个 swapchanel 算子实现与通过 TpuLang 接口应用的样例。

#### 1. 算子参数解析

在文件 \${TPUC\_ROOT}/customlayer/include/backend\_custom\_param.h 中 定 义 参 数 结 构 体 swapchannel\_param\_t：

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

其中，这里的字段 order 对应前端的属性 order。

值得注意的是，从编译器传递到后端的是一个 custom\_param\_t 的数组 A，它的第一个元素是保留的，从第二个元素开始，每个元素对应前端的一个属性。为方便起见，在头文件 \${TPUC\_ROOT}/customlayer/include/api\_common.h 中，

提供了一个宏来完成了一个对应: PARSE\_PARAM(swapchannel, sc\_param, param) , 其中, param 表示数组 A, sc\_param 表示后端参数结构体。用户需要在文件 \${TPUC\_ROOT}/customlayer/include/param\_parser.h 中定义一个 swapchannel\_parse\_param 解析函数来完成这种转换, 其输入实际上是数组 A 的剔除第一个元素后的子数组的指针。在文件 \${TPUC\_ROOT}/customlayer/include/param\_parser.h 中, 实现参数解析代码:

```
static swapchannel_param_t swapchannel_parse_param(const void* param) {
    swapchannel_param_t sc_param = {0};
    for (int i = 0; i < 3; i++) {
        sc_param.order[i] = ((custom_param_t *)param)[0].int_arr_t[i];
    }
    return sc_param;
}
```

参数解析在补丁函数和后端实现中都会被用到。

## 2. 补丁函数

在文件 \${TPUC\_ROOT}/customlayer/plugin/plugin\_swapchannel.c 中:

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

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

## 3. 后端算子实现

在 \${TPUC\_ROOT}/customlayer/include/tpu\_impl\_custom\_ops.h 头文件中添加如下声明:

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

`${TPUC_ROOT}/customlayer/src/tpu_impl_swapchannel.c` 代码如下：

```
#include "tpu_impl_custom_ops.h"

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

#### 4. 后端接口

在文件  `${TPUC_ROOT}/customlayer/src/interface_swapchannel.c` 中定义函数 `void type_infer_swapchannel` 和 `void api_swapchannel_global`：

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

// type infer function
void type_infer_swapchannel(
    const global_tensor_spec_t *input,
    global_tensor_spec_t *output,
```

(续下页)

(接上页)

```

const void *param) {
    output->dtype = input->dtype;
    output->dims = input->dims;
    memcpy(output->shape, input->shape, output->dims);
    output->elem_num = input->elem_num;
}

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

```

## 5. 后端算子注册

在文件 \${TPUC\_ROOT}/customlayer/register\_ops.cmake 添加如下代码，可用于注册后端算子：

```
register_custom_op(swapchannel)
```

完成后，可参考《TpuLang 自定义算子添加》小节进行动态库编译与安装。

## 6. 前端准备

在文件 \${TPUC\_ROOT}/customlayer/python/my\_tpuLang\_layer.py 中调用 TpuLang 接口构建自定义算子 swapChannel，它只有一个输入和一个输出，且有一个属性 order，是一个长度为 3 的整数列表：

```

import transform.TpuLang as tpul

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

```

(续下页)

(接上页)

```
op_name="swapchannel",
params=params,
out_dtypes=[dtype])
return outs
```

在文件 \${TPUC\_ROOT}/customlayer/test\_if/unittest/test\_swapchannel.py 中, 对自定义的 swapChannel 算子进行单元测试:

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

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

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

### 17.3.2 Caffe 示例

本小节提供了 Caffe 中 absadd 和 ceiladd 自定义算子的应用示例。

#### 1. 定义 Caffe 自定义算子

absadd 和 ceiladd 在 \${TPUC\_ROOT}/customlayer/python/my\_caffe\_layer.py 中的定义如下:

```
import caffe
import numpy as np

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

    def setup(self, bottom, top):
        params = eval(self.param_str)
        # define attributes here
```

(续下页)

(接上页)

```

self.b_val = params['b_val']

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

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

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

class CeilAdd(caffe.Layer):

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

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

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

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

```

Caffe prototxt 中相应算子的表达如下：

```

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

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

```

(续下页)

(接上页)

## 2. 实现算子前端转换函数

在 `$TPUC_ROOT/customlayer/python/my_converter.py` 中的 `MyCaffeConverter` 类里定义一个 `convert_python_op` 函数，代码如下：

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

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

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

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

## 3. Caffe 前端转换

通过调用 `my_converter.py` 接口完成对 `$TPUC_ROOT/customlayer/test` 目录下的 `my_model.prototxt`, `my_model.caffemodel` Caffe 模型进行转换（该 Caffe 模型中包含了 `absadd` 与 `ceiladd` 算子），命令如下：

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

通过以上步骤可获得 `caffe_test_net.mlir` 的 Top 层 mlir 文件，后续的模型部署过程请参考“用户接口”章节。

## 4. 后端算子与接口实现

`absadd` 与 `ceiladd` 的实现部分和 `swapchannel` 算子相似，可在 `$TPUC_ROOT/customlayer/include` 和 `$TPUC_ROOT/customlayer/src` 目录下找到相应代码。

## 17.4 自定义 AP (application processor) 算子添加流程

### 17.4.1 TpuLang 自定义 AP 算子添加

#### 1. 加载 tpu-mlir

与 TPU 自定义算子时加载 tpu-mlir 一致。

#### 2. 编写 AP 算子实现

假定当前处于 \$TPUC\_ROOT/customlayer 路径下，在./include/custom\_ap/ap\_impl\_{op\_name}.h 头文件中，声明一个继承 ap\_layer 类的自定义派生类 layer (其中“forward()”声明具体实现方法，“shape\_infer()”声明推理前后张量形状变化方法，“dtype\_infer()”声明推理前后数据类型变化方法，“get\_param()”声明参数解析方法)。并且在./ap\_src 目录下添加 ap\_impl\_{op\_name}.cpp，在其中实现相应的函数，定义新的成员变量，重写其中的成员函数。

#### 3. 注册自定义算子

##### a. 在 ap\_impl\_{op\_name}.cpp 中添加算子的名字以注册自定义算子：

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

##### b. 并在./customlayer/include/customap\_common.h 中的枚举类型 ‘AP\_CUSTOM\_LAYER\_TYPE\_T’ 中定义成员 AP\_CUSTOM\_{OP\_NAME}，其中 OP\_NAME 为大写。

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

##### c. 在 customlayer/ap\_src/ap\_layer.cpp 中定义实例化方法

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

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

#### 4. 编译器补丁

有时候，需要对编译器进行修改，以对不同的自定义算子在不同参数下的编译行为进行控制，这时候就需要添加一些补丁。当前已支持以下补丁函数（在文件夹./plugin 中定义）：

- （必选）需要自行实现算子参数解析函数，用于获取算子所需的关键参数，重写自定义 layer 的 get\_param() 方法：

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

- （必选）推理函数，即算子的 C++ 实现。重写自定义 layer 的 forward() 方法：

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

- （可选）形状推断函数。此补丁函数用于编译器形状推断，若不实现，默认只有一个输入一个输出，且输出形状跟输入形状

相同。补丁函数形式如下：

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

其中，input\_shapes/output\_shapes 为输入/出张量形状的数组，input\_dims/output\_dims 为输入/出张量维度的数组。

## 5. 编译并安装动态库

先初始化环境：

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

然后需要完成补丁的编译（得到 libplugin\_custom.so）：

```
rebuild_custom_plugin
```

根据处理器架构编译自定义算子库文件（在目录 build\_ap 下得到 libcustomapop.so），需要特别注意的是，编译自定义 AP 算子的环境要与 bmodel 运行环境中的 glic 版本兼容，命令如下：

- x86 架构

```
rebuild_custom_apop_x86
```

- arm 架构

```
rebuild_custom_apop_aarch64
```

至此我们完成了自定义 AP 算子后端部分的工作。

## 6. 利用 TpuLang 构建自定义 AP 算子

关于 TpuLang 的使用方式请参考 TpuLang 接口章节。

TpuLang 中提供了 TpuLang.custom 接口可以同样用于自定义 AP 算子, 使用方法与自定义 Tpu 算子基本一致, 区别在定义 “TpuLang.custom” 对象时, “op\_name” 参数要以 “ap.” 开头字段作为区分, 例如 “ap.topk”:

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

## 7. 上卡测试

当网络中存在自定义 AP 算子时, bmodel 需要包含算子信息, 使用命令将 libcus-tomapop.so 写入 bmodel 文件, 所有主机处理器架构均使用:

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

注: 需要特别注意的是, 编译自定义 AP 算子的环境要与 bmodel 运行环境中的 glibc 版本兼容。

## 17.5 自定义 AP 算子示例

本节内容假定已经完成了 tpu-mlir 发布包加载。

### 17.5.1 TpuLang 示例

本小节提供了一个 swapchanel 算子实现与通过 TpuLang 接口应用的样例。

#### 1. 自定义算子派生类

其中, 这里的字段 order 对应前端的属性 order。

在 {TPUC\_ROOT}/customlayer/ap\_src/ap\_impl\_{op\_name}.cpp 的自定义类中定义成员变量:

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

在 {TPUC\_ROOT}/customlayer/ap\_src/ap\_impl\_{op\_name}.cpp 的自定义类中重写接口 get\_param()。值得注意的是，从编译器传递到后端的是一个 custom\_param\_t 的数组 A，它的第一个元素是保留的，从第二个元素开始，每个元素对应前端的一个属性：

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

在 {TPUC\_ROOT}/customlayer/ap\_src/ap\_impl\_{op\_name}.cpp 的自定义类中重写接口 shape\_infer()：

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

## 2. AP 算子实现

在 {TPUC\_ROOT}/customlayer/ap\_src/ap\_impl\_{op\_name}.cpp 的自定义类中重写接口 forward()：

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

## 3. AP 算子注册

- 在 ap\_impl\_{op\_name}.cpp 中添加算子的名字以注册自定义算子：

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

- 并 在 ./customlayer/include/customap\_common.h 中的 枚举 类型 ‘AP\_CUSTOM\_LAYER\_TYPE\_T’ 中定义成员 AP\_CUSTOM\_TOPK。

```
typedef enum {
    AP_CUSTOM
```

(续下页)

(接上页)

```
AP_CUSTOM_TOPK          = 10002,  
AP_CUSTOM_LAYER_NUM     ,  
AP_CUSTOM_LAYER_UNKNOW = AP_CUSTOM_LAYER_NUM,  
} AP_CUSTOM_LAYER_TYPE_T;
```

c. 在 customlayer/ap\_src/ap\_layer.cpp 中定义实例化方法

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

#### 4. 前端准备

调用 TpuLang 接口构建自定义 AP 算子的流程与 TPU 自定义算子基本一致，区别在定义 “TpuLang.custom” 对象时，“op\_name” 参数要以 “ap.” 开头字段作为区分，例如 “ap.topk”

## 用 PPL 写后端算子

PPL 是基于 C/C++ 语法扩展的、针对 TPU 编程的专用编程语言 (DSL)。开发者可以通过 PPL 在 TPU-MLIR 中编写后端算子。本章节以 `add_const_fp` 算子为例，介绍如何编写后端算子，以及 PPL 代码是如何被编译和使用的。

PPL 后端算子的实现位于 `tpu-mlir/lib/PplBackend/src` 目录；如果是发布包，则在 TPU-MLIR 发布包的 `PplBackend/src` 目录。有关如何编写 PPL 源码的详细信息，请参考 `tpu-mlir/third_party/ppl/doc` 中的文档。

### 18.1 如何编写和调用后端算子

第一步：实现三个源码文件

需要创建三个源码文件，一个是设备端的 `pl` 源码，一个是主机端的接口 `cpp` 源码，另一个是主机端的 `tiling` 函数 `cpp` 源码。以 `add_const_fp` 为例，文件名分别为：

- `add_const_fp.pl`: 实现 `add_const_f32`, `add_const_f16` 及 `add_const_bf16` 等 kernel 接口。
- `add_const_fp_tile.cpp`: 实现 `add_tiling` 函数以调用这些 kernel 接口。
- `add_const_fp_api.cpp`: 实现 `api_add_const_fp_global` 函数以调用 `add_tiling` 接口。

`tiling.cpp` 文件示例

```
// 添加pl文件自动生成的头文件
#include "add_const_fp.h"
// 添加tpu-mlir数据类型及结构体头文件
#include "tpu_mlir/Backend/BM168x/Param.h"
```

(续下页)

(接上页)

```

// 需要用extern C来定义入口函数
extern "C" {
    // 如果pl文件提供了多个算子，可以提前定义函数指针，这样可以减少一些重复代码
    // 注意pl文件中的指针类型需要用gaddr_t定义
    using KernelFunc = int (*)(gaddr_t, gaddr_t, float, int, int, int, int, int,
                                bool);

    // 添加入口函数，输入参数由用户自定义
    int add_tiling(gaddr_t ptr_dst, gaddr_t ptr_src, float rhs, int N, int C, int H,
                    int W, bool relu, int dtype) {
        KernelFunc func;
        // 根据输入数据类型，选择合适的算子
        if (dtype == DTYPE_FP32) {
            func = add_const_f32;
        } else if (dtype == DTYPE_FP16) {
            func = add_const_f16;
        } else if (dtype == DTYPE_BFP16) {
            func = add_const_bf16;
        } else {
            assert(0 && "unsupported dtype");
        }

        // 计算block size，可以将block size对齐到EU_NUM，
        // 这样可以减少内存分配失败的次数，并且因为TPU上的内存大部分是按照EU_
        // →NUM对齐的，
        // 所以不会影响到内存分配
        int block_w = align_up(N * C * H * W, EU_NUM);
        int ret = -1;
        while (block_w > 1) {
            ret = func(ptr_dst, ptr_src, rhs, N, C, H, W, block_w, relu);
            if (ret == 0) {
                return 0;
            } else if (ret == PplLocalAddrAssignErr) {
                // 当错误类型为PplLocalAddrAssignErr时，说明block size太大，
                // local 内存放不下，需要减小block size
                block_w = align_up(block_w / 2, EU_NUM);
                continue;
            } else if (ret == PplL2AddrAssignErr) {
                // 当错误类型为PplL2AddrAssignErr时，说明block size太大，
                // L2 内存放不下，需要减小block size，本示例没有分配L2内存，
                // 因此不会出现这个错误
                assert(0);
            } else {
                // 其他错误，需要debug
                assert(0);
                return ret;
            }
        }
        return ret;
    }
}

```

## 注意事项

- add\_const\_fp.h 头文件中包含了一些错误码和芯片相关的参数定义：
- pl 文件中的指针需要使用 gaddr\_t 类型定义

表 18.1: 内置错误码

参数名	说明
PplLocalAddrAssignErr	Local 内存分配失败
FileErr	
LlvmFeErr	
PplFeErr	AST 转 IR 失败
PplOpt1Err	优化 pass opt1 失败
PplOpt2Err	优化 pass opt2 失败
PplFinalErr	优化 pass final 失败
PplTransErr	代码生成失败
EnvErr	环境变量异常
PplL2AddrAssignErr	L2 内存分配失败
PplShapeInferErr	shape 推导失败
PplSetMemRefShapeErr	
ToPplErr	
PplTensorConvErr	
PplDynBlockErr	

表 18.2: 内置芯片参数

参数名	说明
EU_NUM	EU 数量
LANE_NUM	LANE 数量

第二步：调用 Kernel 接口

在 lib/Dialect/Tpu/Interfaces/BM1684X/AddConst.cpp 的 void tpu::AddConstOp::codegen\_global\_bm1684x() 函数中，调用 api\_add\_const\_fp\_global，代码如下：

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

如果该算子支持局部执行，则实现 api\_xxxxOp\_local，并使用 BM168x::call\_ppl\_local\_func 进行调用。

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

以上便完成了后端算子的实现。

## 18.2 PPL 集成到 TPU-MLIR 的流程

1. 将 PPL 编译器精简后放入 `third_party/ppl` 目录，并更新 PPL 编译器，参考该目录下的 `README.md` 文件。
2. 在 `model_deploy.py` 中集成 PPL 源码编译，流程如图所示：

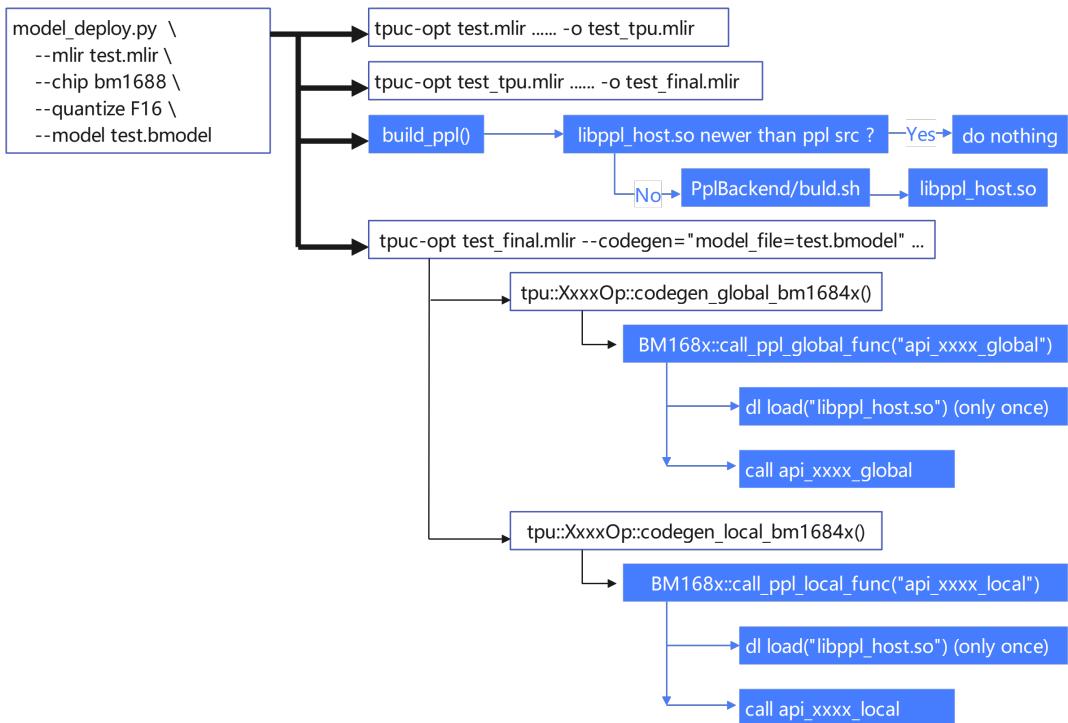


图 18.1: PPL Workflow

# CHAPTER 19

## final.mlir 截断方式

final.mlir 作为 codegen 的输入文件，是模型在经过了所有硬件无关与硬件相关的优化后生成的最终中间表达 (IR)。因为包含了硬件相关信息，结构相对于之前的 IR 要复杂的多。

而在进行模型适配时有时会出现 Tpu 层 MLIR 文件与 bmodel 的 cmodel 推理结果不一致的情况，为了快速定位到出问题的位置，除了使用 bmodel\_checker.py 工具对每一层输出进行对比外，还可以手动对 final.mlir 文件进行截断，生成一个截断后的模型。

因此，本章主要会对 final.mlir 的结构进行剖析，并讲解如何基于 final.mlir 对模型进行截断以便于后续的问题定位。

- 建议使用 IDE：VSCode。
- 建议使用插件：MLIR。

### 19.1 final.mlir 结构介绍

final.mlir 中的单个算子组成部分如下：

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

图 19.1: final.mlir 单算子示例

注意：

- value 表示算子的输入/输出，为 SSA 形式

- out\_num: 表示输出的数量。如果是单输出算子，则不会显示 :out\_num。
- 对于多输出算子的值，用户将按 %value#in\_index 方式引用（index 从 0 开始）
- 每个输入/输出值都有对应的 Tensor type。
- 完整的 Tensor type 包含形状、数据类型和全局内存地址（Gmem addr）。

除了单算子外 final.mlir 中还存在着 LayerGroup 后生成的 tpu.Group 算子，其中包含了多个中间算子，这些算子均在 Lmem 上完成计算，由 tpu.Group 统一通过 tpu.Load 和 tpu.Store 控制输入数据加载和输出数据存储，所以中间算子的 Tensor type 并没有 Gmem addr：

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

图 19.2: tpu.Group 示例

- local\_type 指代不带有 Gmem addr 的 Tensor type。
- 算子尾部的 loc(#loc32) 指代模型某层输出的 location，即该输出的编号，可根据该编号在 final.mlir 文件尾部找到对应的输出名。
- Yield 表示 tpu.Group 的输出集合。

完整的 final.mlir 文件中存在的结构大致如下：

```
module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    func.func @main(%arg0: host_in_type) -> (out1_type, out2_type, out3_type) {
      %0 = "top.Input"(%arg0) {...} : (host_in_type) -> in_type loc(#loc1)
      %1:3 = call @subfunc_0(%0) : (in_type) -> (out1_type, out2_type, out3_type) loc(#loc)
      return %1#0, %1#1, %1#2 : out1_type, out2_type, out3_type loc(#loc)
    } loc(#loc)

    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out1_type, out2_type, out3_type)
    attributes {...} {
      ...
      return %1, %2, %3 : out1_type, out2_type, out3_type loc(#loc)
    } loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("layer_name_1")
#loc2 = loc("layer_name_2")
#loc3 = loc("layer_name_3")
...
#loc950 = loc(fused[#loc2, #loc3])
...
```

图 19.3: final.mlir 结构示例

- 双层 module 中包含了 mainfunc 和 subfunc， mainfunc 和 subfunc 存在调用关系。
- mainfunc 中的 arg0 指代 host 端的输入，因此 host\_in\_type 不带有 Gmem addr。
- 多输出的 location 会被添加在 final.mlir 文件的最尾端，并表述出与每个具体输出 location 间的包含关系，例如 #loc950 = loc(fused[#loc2, #loc3])。

## 19.2 final.mlir 截断流程

1. 修改 subfunc。删减 subfunc 内部结构，并将返回值的 value 与对应 type:

```

module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    func.func @main(%arg0: host_in_type) -> (out1_type, out2_type, out3_type) {
      ...
    } loc(#loc)
    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out4_type) attributes {...} {
      ...
      return %1, %2, %3 : out1_type, out2_type, out3_type loc(#loc)
      return %4 : out4_type loc(#loc)
    } loc(#loc)           Sync the out type
  } loc(#loc)           Remove the unneeded parts and change return values
} loc(#loc)
#loc1 = loc("layer_name_1")
...

```

图 19.4: 截断流程 Step1

2. 同步 mainfunc 中 subfunc 的调用方式 (value 与 type ):

```

module @"ModelA" attributes { model_info0 ...} {
  module @"ModelA" attributes { model_info1 ...} {
    func.func @main(%arg0: host_in_type) -> (out4_type) attributes {...} {
      ...
      %%0 = "top.Input"(%arg0) {...} : (host_in_type) -> in_type loc(#loc1)
      %%1:3 = call @subfunc_0(%0) : (in_type) -> (out4_type, out2_type, out3_type) loc(#loc)
      %%1 = call @subfunc_0(%0) : (in_type) -> (out4_type) loc(#loc)
      return %%1:0, %%1:1, %%1:2 : out1_type, out2_type, out3_type loc(#loc)
      return %%1: out4_type loc(#loc)           Update mainfunc according to the modified subfunc
    } loc(#loc)
    func.func @subfunc_0(%arg0: in_type loc("input0")) -> (out4_type) attributes {...} {
      ...
    } loc(#loc)
  } loc(#loc)
} loc(#loc)
#loc1 = loc("layer_name_1")
...

```

图 19.5: 截断流程 Step2

3. 检查 bmodel 是否修改成功。可首先通过执行 codegen 步骤看是否可以正常生成 bmodel (<…> 请替换为实际的文件或参数):

```
$ tpuc-opt <final.mlir> --codegen="model_file=<bmodel_file> embed_debug_info=<true/false>
→ model_version=latest" -o /dev/null
```

当需要使用 profile 进行性能分析时，embed\_debug\_info 设置为 true。

4. 使用 model\_tool 检查该 bmodel 的输入输出信息是否符合预期：

```
$ model_tool --info <bmodel_file>
```

注意：

1. 截断时以算子为单位进行模型结构的删除，每个 tpu.Group 应当被看作是一个算子。
2. 仅修改函数返回值不对冗余的模型结构进行删除可能会造成输出结果错误的情况，该情况是由于每个激活的 Gmem addr 分配会根据激活的生命周期进行复用，一旦生命周期结束，将会被分配给下一个合适的激活，导致该地址上的数据被后续操作覆盖。
3. 需要确保 tpu.Group 的每个输出都有 user，否则可能会出现 codegen 步骤报错的情况，如果不想输出 tpu.Group 的某个结果又不便将其完整删除，可以为没有 user 的输出添加一个无意义的 tpu.Reshape 算子，并配上相同的 Gmem addr 和 location，例如：

```
...
%2:2 = "tpu.Group"(%1) ({
  ...
  %10 = "tpu.Store"(%8, %0) {...} : (%8_local_type, none) -> out0_type loc(#loc2)
  %11 = "tpu.Store"(%9, %0) {...} : (%9_local_type, none) -> out1_type loc(#loc3)
  "tpu.Yield"(%10, %11) : (out0_type, out1_type) -> () loc(#loc950)
}) {...} : (%1_type) -> (out0_type, out1_type) loc(#loc950)

# only returns %2#0, so add a meaningless reshape as the user of %2#1
%3 = "tpu.Reshape"(%2#1) {shape = []} : (out1_type) -> out1_type loc(#loc3)
return %2#0 : out0_type loc(#loc)
...}
```

图 19.6: reshape 添加示例

4. 对模型进行删减后可以更新 module 模块中的 module.coeff\_size 信息以减少裁剪后生成的 bmodel 大小，公式如下：

$$\text{CoeffSize} = \text{NumElement}_{\text{weight}} * \text{DtypeBytes}_{\text{weight}} + \text{Addr}_{\text{weight}} - \text{CoeffAddr}$$

上述公式中的 weight 指代截断后 final.mlir 中最后一个 top.Weight.neuron（即激活）因为会对地址进行复用，因此不建议进行修改。

# CHAPTER 20

---

## MaskRCNN 大算子接口指南

---

### 20.1 MaskRCNN 基础

两阶的 MaskRCNN 由两类组成:

- **3 个有权值模块:** backbone.pt 和 2 个 bbox/mask 中间有权值层 (按顺序命名为 torch\_bbox/mask.pt ).
- **5 个动态无权值模块:** 包括 RPN head, bbox pooler, bbox head, mask pooler, mask head.

因此, 完整的 MaskRCNN 可以通过以下过程表示:

- **bbox 检测头:** backbone.pt => RPN head => bbox pooler => torch\_bbox.pt => bbox head.
- **mask 检测头:** backbone.pt => RPN head => mask pooler => torch\_mask.pt => mask head.

#### 20.1.1 模块快速分割方法

由于 MaskRCNN 拆分依赖原框架工程的兼容情况, 用户可能无法 trace 每部分, 本章节中以 mask head 为无法 trace 的例子.

MaskRCNN 的两类模块分割点, 即再次进入有权值模块首层的接入点.

## 20.2 MaskRCNN 大算子

由于基于细粒度操作的 MaskRCNN 部署, 操作动态 IR 的难度较高, 因此提出了以下 MaskRCNN 大算子解决方案:

**粗粒度:**

1. **内置 MaskRCNN 专属后端:** 现在 mlir-backend 直接支持动态无权值模块, 目前包括 RPN head, bbox head, bbox pooler 和 mask pooler. 因此, 大多数与前端推理图解析和优化相关的繁重工作得以节省, 如避免了大量动态形参推理或变种细粒度算子的支持.
2. **模型重建:** 用户只需 4 个结构信息即可重建完整的 MaskRCNN:
  - **io\_map:** 描述模块接口, 与 MaskRCNN 拓扑同构. 定义为 (目标模块索引, 操作数索引):(源模块索引: 操作数索引).
  - **config.yaml:** 用于存储 MaskRCNN 超参数的 YAML 文件, 事先提供.
  - **BackBone:** 通常从顶部到 RPN, 事先从原始 MaskRCNN 中拆分.
  - **有权值模块:** bbox/mask 中间有权值层, 事先从原始 MaskRCNN 中拆分.

## 20.3 快速入门

在深入了解新的 MaskRCNN 特性之前, 请先了解新的 MaskRCNN yaml 文件格式和单元测试.

### 20.3.1 准备您的 YAML

在 regression/dataset/MaskRCNN/CONFIG\_MaskRCNN.yaml 中准备了一个默认的 YAML, 其结构如下:

- **model\_transform 的编译参数:** 重建 MaskRCNN 的结构信息.
  - **io\_map:** 即定义 (目标模块索引, 操作数索引):(源模块索引: 操作数索引), 其中 -1 表示整体模型的顶层输入,-2 表示整体模型的顶层输出,0、1、2…表示 MaskRCNN 模块的 ID.  
例如,{(0,0):(-1,0),(1,0):(0,0),(-2,0):(1,0)}, 表示模块 [0] 的 input[0] 来自整体模型的 input[0], 模块 [1] 的 input[0] 来自模块 [0] 的 output[0], 整体模型的 output[0] 来自模块 [1] 的 output[0].
  - **maskrcnn\_output\_num:** 整体 MaskRCNN 的最终输出操作数的数量.
  - **maskrcnn\_structure:** 描述 MaskRCNN 模块顺序.1 表示 torch.pt 模型,0 表示 PPLOp. 例如,[1,0,1] 表示第一个模块是 torch 模型, 第二个模块是 PPLOp, 第三个模块是 torch 模型.
  - **maskrcnn\_ppl\_op:** 后端已经用 PPL 实现的 MaskRCNN 算子名称.
  - **numPPLOp\_InWithoutWeight\_MaskRCNN:** 每个 PPLOp 的输入操作数; 请不要计入权重.

- **MaskRCNN 的超参数:** 必要 MaskRCNN 配置参数, 来自源码 MaskRCNN 框架.

### 20.3.2 模块单元测试

--case 提供单试模块的选择, 当前支持 4 个动态无权值模块测试: RPN head, bbox pooler, bbox head, mask pooler.

更多指导请参见 test\_MaskRCNN.py.

```
$ test_MaskRCNN.py --case MaskRCNN_Utest_RPNGetBboxes --debug
```

## 20.4 新前端接口 API

### 20.4.1 [步骤 1] 运行 model\_transform

用于将 MaskRCNN 转换为 MLIR 文件.

- **跳过推理:** 请注意, 在此步骤中不需要形参推理或计算推理, 无需输入/比较参考的数据 .npz 文件, 但需要事先提供 config.yaml.
- **跳过预处理:** 请注意, 在此步骤中, 默认无预处理.
- **新的启动符:** 注意 enable\_maskrcnn.

```
$ model_transform.py \
--model_def backbone.pt \
--model_extern torch_bbox.pt,torch_mask.pt \
--model_name MaskRCNN \
--input_shapes [[1,3,800,1216],[1,1,4741,4],[1,1,20000,4],[1,1,20000,4],[1,1,100,4]] \
--mlir MaskRCNN.mlir \
--enable_maskrcnn \
--path_yaml regression/dataset/MaskRCNN/CONFIG_MaskRCNN.yaml
```

### 20.4.2 [步骤 2] 生成输入数据

#### MaskRCNN 输入格式

MaskRCNN 大算子框架需要 5 个输入:

- **预处理图片:** 经过预处理的图片.
- **max\_shape\_RPN/max\_shape\_GetBboxB:** 如果输入图片的形参为 S1, 原始形参为 S0, 则最大形参为 int(S0 \* S1 / S0), 并扩展为常量权重张量.
- **scale\_factor\_GetBboxB/scale\_factor\_MaskPoolerB:** 如果输入图片的形参为 S1, 原始形参为 S0, 则缩放因子为 float(S1 / S0), 并扩展为常量权重张量.

## 输入格式化工具

在 tpu-mlir/python/tools/tool\_maskrcnn.py 提供了一个格式化输入数据工具, 以帮助您生成满足上述要求的数据.

- **跳过预处理:** 输入图像应为经过预处理的图像, 因为 MaskRCNN 的预处理过程通常很复杂, 并依赖于原框架中的特定函数.

除了 path\_yaml, 还需要指定三个参数:

- **path\_input\_image:** 经过预处理的图像, 保存为 npz 格式.
- **basic\_max\_shape\_inverse:** 预处理后的高度和宽度.
- **basic\_scalar\_factor:** 正是上述的  $\text{float}(S_1 / S_0)$ , basic\_max\_shape\_inverse 除以原始形状重新排序后的 height, width.

结果数据将存储在与 path\_input\_image 相同的路径中, 但后缀为 SuperiorMaskRCNNInputPreprocessed.

请查看 tool\_maskrcnn.py 以获取更多指导.

```
$ tool_maskrcnn.py \
--path_yaml      ./regression/dataset/MaskRCNN/CONFIG_MaskRCNN.yaml \
--path_input_image  Superior_IMG_BackBone.npz \
--basic_max_shape_inverse 1216,800 \
--basic_scalar_factor   1.8734375,1.8735363 \
--debug
```

### 20.4.3 [步骤 3] 运行 model\_deploy

- **跳过推理:** 此处跳过量化比较和仿真比较。
- **强制参数:** --quantize 模式被强制为 F32, --processor 被强制为 BM1684X
- **新的启动符:** 注意 enable\_maskrcnn.

```
$ model_deploy.py \
--mlir MaskRCNN.mlir \
--quantize F32 \
--processor BM1684X \
--model MaskRCNN.bmodel \
--debug \
--enable_maskrcnn
```

## 20.5 IO\_MAP 指南

手动生成 io\_map 分为两个步骤:

- **模块接口的完备定义:** 准确收集输入和输出的操作数和形参, 以及模块连接.
- **创建相应的 io\_map:** 应准确且唯一地重建完整的 MaskRCNN.

### 20.5.1 [步骤 1] 描述模块接口

如开始所述, 完整的 MaskRCNN 被截断为多个模块.

请为每个模块描述以下信息:

- **输入:** 输入操作数或常量权重
- **形参:** 以 4 维 shape 表示.
- **数据类型:** 仅支持 fp32 或 int32.
- **连接:** 每个输入找出其来源的上层模块 (可能不是上一个相邻模块), 和上层模块相应输出的操作序数.

请注意, -1 表示完整 MaskRCNN 的输入, 而 -2 表示完整模型的输出

#### **[-1] Top\_In**

输入编号	名称	形参	数据类型
输入 0)	'img.1'	[1,3,800,1216]	
输入 1)	'max_shape_RPN'	[bs,1,max_filter_num,4]	int32
输入 2)	'max_shape_GetBboxB'	[1,bs*20000,1,4]	int32
输入 3)	'scale_factor_GetBboxB'	[1,bs,20000,4]	FP32
输入 4)	'scale_factor_MaskPooler'	[bs,1,roi_slice,4]	FP32

#### **[Torch] SubBlock-0: BackBone.pt**

IO 类型	名称	形参	数据类型	连接信息 [源模块-操作数序号]
输入 0)	'img.1'	[1,3,800,1216]	FP32	from_[TOP_IN]Input-0
输出 0)	'11'	[1,256,200,304]	FP32	
输出 1)	'12'	[1,256,100,152]	FP32	
输出 2)	'13'	[1,256,50,76]	FP32	
输出 3)	'16'	[1,256,25,38]	FP32	
输出 4)	'15'	[1,256,13,19]	FP32	
输出 5)	'18'	[1,3,200,304]	FP32	
输出 6)	'19'	[1,3,100,152]	FP32	
输出 7)	'20'	[1,3,50,76]	FP32	
输出 8)	'21'	[1,3,25,38]	FP32	
输出 9)	'22'	[1,3,3,19]	FP32	
输出 10)	'23'	[1, 2,200,304]	FP32	
输出 11)	'24'	[1,12,100,152]	FP32	
输出 12)	'25'	[1,12,50,76]	FP32	
输出 13)	'26'	[1,12,25,38]	FP32	
输出 14)	'27'	[1,12,13,19]	FP32	

[PPL] SubBlock-1: ppl::RPN\_get\_bboxes

IO 类型	名称	形参	连接信息 [源模块-操作数序号]
输出	0 result_list	[bs,1,max_per_img,num_levels]	
输入	1 cls_scores_0	[bs,3,200,304]	[Torch][SubBlock-0]Output 5)
输入	2 cls_scores_1	[bs,3,100,152]	[Torch][SubBlock-0]Output 6)
输入	3 cls_scores_2	[bs,3,50,76]	[Torch][SubBlock-0]Output 7)
输入	4 cls_scores_3	[bs,3,25,38]	[Torch][SubBlock-0]Output 8)
输入	5 cls_scores_4	[bs,3,13,19]	[Torch][SubBlock-0]Output 9)
输入	6 bbox_preds_0	[bs,12,200,304]	[Torch][SubBlock-0]Output 10)
输入	7 bbox_preds_1	[bs,12,100,152]	[Torch][SubBlock-0]Output 11)
输入	8 bbox_preds_2	[bs,12,50,76]	[Torch][SubBlock-0]Output 12)
输入	9 bbox_preds_3	[bs,12,25,38]	[Torch][SubBlock-0]Output 13)
输入	10 bbox_preds_4	[bs,12,13,19]	[Torch][SubBlock-0]Output 14)
输入	11 max_shape	[bs,1,max_filter_num,4]	[TOP_IN]Input-1
输入	12 mlvl_anchors_0	[bs,1,3*200*304,4]	[mlir][Weight]
输入	13 mlvl_anchors_1	[bs,1,3*100*152,4]	[mlir][Weight]
输入	14 mlvl_anchors_2	[bs,1,3*50*76,4]	[mlir][Weight]
输入	15 mlvl_anchors_3	[bs,1,3*25*38,4]	[mlir][Weight]
输入	16 mlvl_anchors_4	[bs,1,3*13*19,4]	[mlir][Weight]

[PPL] SubBlock-2: ppl::Bbox\_Pooler

IO 类型	名称	形参	连接信息 [源模块-操作数序号]
输出	0 result_res	[bs*250,256,PH,PW]	
输出	1 result_rois	[bs,max_per_img,1,roi_len]	
输入	2 feat0	[bs,256,H,W]	[Torch][SubBlock-0]Output 0)
输入	3 feat1	[bs,256,H/2,W/2]	[Torch][SubBlock-0]Output 1)
输入	4 feat2	[bs,256,H/4,W/4]	[Torch][SubBlock-0]Output 2)
输入	5 feat3	[bs,256,H/8,W/8]	[Torch][SubBlock-0]Output 3)
输入	6 rois_multi_batch	[bs,roi_slice,1,roi_len]	[PPL][SubBlock-1]result_list

### [Torch] SubBlock-3: torch\_bbox.pt

Batch	IO 类型	名称	形参	数据类型	连接信息 [源模块-操作数序号]
Batch-1	输入	0	[250,256,7,7]	FP32	[PPL][SubBlock-2]result_res
	输出	0	[250,81]	FP32	
	输出	1	[250,320]	FP32	

### [PPL] SubBlock-4: ppl::get\_bboxes\_B

Batch	IO 类型	名称	形参	连接信息 [源模块-操作数序号]
Batch 1	输出	re_sult_det_bboxes	[bs,1,100,5]	
	输出	result_det_labels	[bs,1,100,1]	
	输入	rois	[1,bs*250,1,5]	[PPL][SubBlock-2]1-result_rois
	输入	bbox_pred	[1,bs*250,1,320]	[Torch][SubBlock-3]Output 1
	输入	cls_score	[1,bs*250,1,81]	[Torch][SubBlock-3]Output 0
	输入	max_val	[1,bs*20000,1,4]	[TOP_IN]Input-2
	输入	scale_factor	[1,bs,20000,4]	[TOP_IN]Input-3

### [PPL] SubBlock-5: ppl::Mask\_Pooler

IO 类型	序号	名称	形参	连接信息 [源模块-操作数序号]
输出	0	result_res	[roi_num,C,PH,PW]	
输入	1	x0	[bs,256,H,W]	[Torch][SubBlock-0]Output 0
输入	2	x1	[bs,C,H/2,W/2]	[Torch][SubBlock-0]Output 1
输入	3	x2	[bs,C,H/4,W/4]	[Torch][SubBlock-0]Output 2
输入	4	x3	[bs,C,H/8,W/8]	[Torch][SubBlock-0]Output 3
输入	5	det_bboxes_multi_b	[bs,1,roi_slice,roi_]	[PPL][SubBlock-4]0- result_det_bboxes
输入	6	det_labels_multi_ba	[bs,1,roi_slice,1]	[PPL][SubBlock-4]1- result_det_labels
输入	7	scale_factor	[bs,1,roi_slice,4]	[TOP_IN]Input-4

### [Torch] SubBlock-6: torch\_mask.pt

Batch	IO 类型	序号	名称	形参	数据类型	连接信息 [源模块-操作数序号]
Batch 1	输入	0	in-put.2	[100,256,14,14]	FP32	[PPL][SubBlock-5]0-result_res
	输出	0	75	[100,80,28,28]	FP32	
Batch 4	输入	0	in-put.2	[400,256,14,14]	FP32	
	输出	0	75	[400,80,28,28]	FP32	

### [2] TOP\_OUT

IO 类型	序号	形参	数据类型	连接信息 [源模块-操作数序号]
输出	0	[bs,1,100,5]	FP32	[PPL][SubBlock-5]0-result_det_bboxes
输出	1	[bs,1,100,1]	FP32	[PPL][SubBlock-5]1-result_det_labels
输出	2	[100,80,28,28]	FP32	[Torch][SubBlock-6]

## 20.5.2 [步骤 2] 描述 IO\_MAP

以以下格式重新组织上述模块接口:

- **模块名称:** 一个模块的名称和序号.
- **上层输入:** 每个输入找出其来源的上层模块 (可能不是上一个相邻模块), 和上层模块相应输出的操作序数.
- **连接数:** 记录输入操作数的总数.
- **映射:** (目标模块索引, 操作数索引):(源模块索引: 操作数索引)

请注意, -1 表示完整 MaskRCNN 的输入, 而 -2 表示完整模型的输出.

### [0]TORCH\_0-rpn

- 上层输入:
  - $\leftarrow [-1]\text{TOP\_IN}[0]$
- 连接数: 1
- 映射:
  - $(0,0):(-1,0)$

### [1]PPL-RPNGetBboxes

- 上层输入:
  - $\leftarrow [0]\text{TORCH\_0-rpn}[5:15]$
  - $\leftarrow [-1]\text{TOP\_IN}[1]$
- 连接数: 10
- 映射:
  - $(1,0):(0,5)$
  - $(1,1):(0,6)$
  - $(1,2):(0,7)$
  - $(1,3):(0,8)$
  - $(1,4):(0,9)$
  - $(1,5):(0,10)$
  - $(1,6):(0,11)$
  - $(1,7):(0,12)$
  - $(1,8):(0,13)$
  - $(1,9):(0,14)$
  - $(1,10):(-1,1)$

### [2]PPL-Bbox\_Pooler

- 上层输入:
  - $\leftarrow [0]\text{TORCH\_0-rpn}[0:4]$
  - $\leftarrow [1]\text{PPL-RPNGetBboxes}[0]$
- 连接数:  $4 + 1$
- 映射:
  - $(2,0):(0,0)$

- (2,1):(0,1)
- (2,2):(0,2)
- (2,3):(0,3)
- (2,4):(1,0)

### [3]Torch-2

- 上层输入:
  - $\leftarrow [2]\text{PPL-Bbox\_Pooler}$
- 连接数: 1
- 映射:
  - (3,0):(2,0)

### [4]PPL-GetBboxB

- 上层输入:
  - $\leftarrow [2]\text{PPL-Bbox\_Pooler}[1]$
  - $\leftarrow [3]\text{Torch-2}[0:2]_{\text{inverse}}$
  - $\leftarrow [-1]\text{TOP\_IN}[2:4]$
- 连接数:  $1 + 2$  (逆向) + 2
- 映射:
  - (4,0):(2,1)
  - (4,1):(3,1)
  - (4,2):(3,0)
  - (4,3):(-1,2)
  - (4,4):(-1,3)

### [5]ppl-MaskPooler

- 上层输入:
  - $\leftarrow [0]\text{Torch-RPN}[0:4]$
  - $\leftarrow [4]\text{PPL-GetBboxB}[0:2]$
  - $\leftarrow [-1]\text{TOP\_IN}[4]$
- 连接数:  $4 + 2$
- 映射:
  - (5,0):(0,0)
  - (5,1):(0,1)

- (5,2):(0,2)
- (5,3):(0,3)
- (5,4):(4,0)
- (5,5):(4,1)
- (5,6):(-1,4)

### [6]Torch-3

- 上层输入:
  - $\leftarrow [5]\text{ppl-MaskPooler}$
- 连接数: 1
- 映射:
  - (6,0):(5,0)

### [2]TOP\_OUT

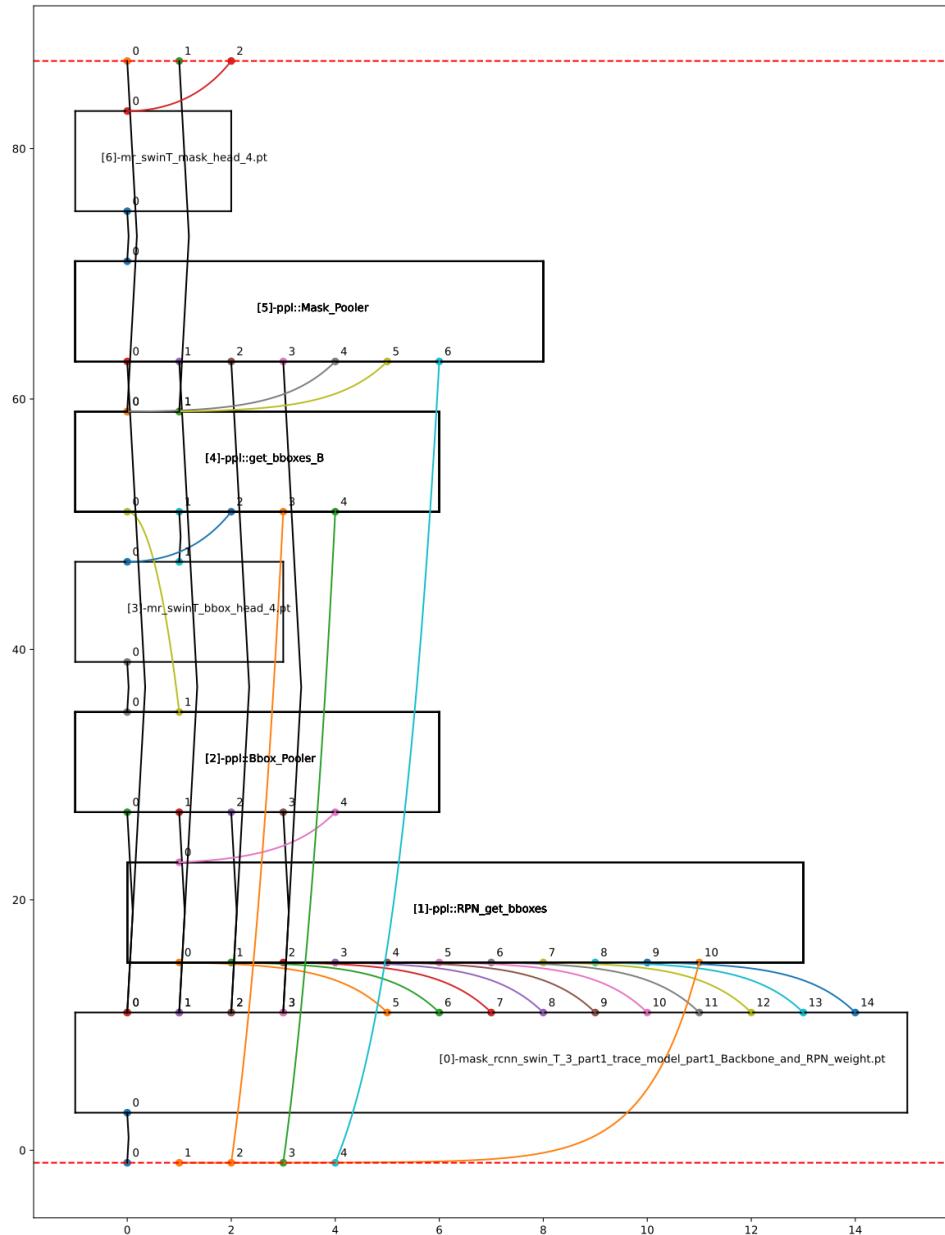
- 上层输入:
  - $\leftarrow [4]\text{PPL-GetBboxB}[0:2]$
  - $\leftarrow [6]\text{Torch-3}$
- 连接数: 2 + 1
- 映射:
  - (-2,0):(4,0)
  - (-2,1):(4,1)
  - (-2,2):(6,0)

#### 20.5.3 IO\_MAP 参数整理

收集上述所有映射信息后, 生成 io\_map 字典:

- **io\_map:**  $\{(0,0):(-1,0), (1,0):(0,5), (1,1):(0,6), (1,2):(0,7), (1,3):(0,8), (1,4):(0,9), (1,5):(0,10), (1,6):(0,11), (1,7):(0,12), (1,8):(0,13), (1,9):(0,14), (1,10):(-1,1), (2,0):(0,0), (2,1):(0,1), (2,2):(0,2), (2,3):(0,3), (2,4):(1,0), (3,0):(2,0), (4,0):(2,1), (4,1):(3,1), (4,2):(3,0), (4,3):(-1,2), (4,4):(-1,3), (5,0):(0,0), (5,1):(0,1), (5,2):(0,2), (5,3):(0,3), (5,4):(4,0), (5,5):(4,1), (5,6):(-1,4), (6,0):(5,0), (-2,0):(4,0), (-2,1):(4,1), (-2,2):(6,0)\}$

现直接在 model\_transform 中使用它, 编译过程中将生成一个 revised\_io\_map\_\${model\_name}.svg 图片, 以帮助您检查和可视化 io\_map, 如下图.



## 20.6 mAP 推理

转换和部署这样的粗粒度 MaskRCNN 进行到这里还不够，要在 COCO2017 数据集上 mAP 推理，需要仔细地接入原始推理框架。

有关更多推断细节，请参阅我们的 model-zoo 项目。

# CHAPTER 21

---

## LLMC 使用指南

---

### 21.1 TPU-MLIR weight-only 量化

TPU-MLIR 中支持对大模型进行 weight-only 仅权重量化, 采用的量化算法是 RTN(round to nearest) 算法, 量化粒度为 per-channel 或者 per-group。具体的量化配置如下:

表 21.1: weight-only 量化参数

bit	symmetric	granularity	group_size
4	False	per-channel or per-group	-1 or 64(default)
8	True	per-channel	-1

RTN 量化算法简洁高效, 但也面临一些不足, 对于一些要求模型精度比较高的场景, RTN 算法量化的模型可能无法满足精度需求, 此时需要借助大模型量化工具 llmc\_tpu 来进一步提升精度。

### 21.2 llmc\_tpu

本项目源自 ModelTC/llmc。ModelTC/llmc 是非常优秀的项目, 专为压缩 LLM 设计, 利用最先进的压缩算法提高效率并减少模型体积, 同时不影响预测精度。如果要深入了解 llmc 项目, 请转到 <https://github.com/ModelTC/llmc>。

本项目是基于 ModelTC/llmc 进行一些定制化修改, 用于支持 Sophgo 处理器。

### 21.2.1 环境准备

#### 1. 下载本项目

```
1 git clone git@github.com:sophgo/llmc-tpu.git
```

#### 2. 准备您需要量化的 LLM 或者 VLM 模型, 放到 ‘llmc-tpu ‘的同级目录

比如 huggingface 上下载 Qwen2-VL-2B-Instruct, 如下:

```
1 git lfs install
2 git clone git@hf.co:Qwen/Qwen2-VL-2B-Instruct
```

#### 3. 下载 Docker 并建立 Docker 容器

pull docker images

```
1 docker pull registry.cn-hangzhou.aliyuncs.com/yongyang/llmcompression:pure-latest
```

create container. llmc\_test is just a name, and you can set your own name

```
1 docker run --privileged --name llmc_test -it --shm-size 64G --gpus all -v $PWD:/workspace F
→registry.cn-hangzhou.aliyuncs.com/yongyang/llmcompression:pure-latest
```

#### 4. 进入 ‘llmc-tpu ‘, 安装依赖包

注意现在已经在 docker 容器中

```
1 cd /workspace/llmc-tpu
2 pip3 install -r requirements.txt
```

### 21.2.2 tpu 目录

```
|--- README.md |--- data |--- LLM |--- cali # 校准数据集 |--- eval
# 推理数据集 |--- VLM |--- cali |--- eval |--- config |--- LLM
# LLM 量化 config |--- Awq.yml #Awq config |--- GPTQ.yml #GPTQ config
|--- VLM #VLM 量化 config |--- Awq.yml #Awq config |--- example.yml #
量化参数参考例子 |--- lm_quant.py # 量化主程序 |--- run_llmc.sh # 量化运行脚本
```

### 21.2.3 操作步骤

#### 【阶段一】准备校准数据集和测试数据集

- 注意点 1: **校准数据集**可以是开源数据集或者业务数据集, 如果模型经过下游业务数据集微调, 则需要选用业务数据集做校准
- 注意点 2: **测试数据集**主要用来评估当前模型的精度表现, 包括预训练 (pretrain) 模型或者量化 (fake\_quant) 模型的精度

可以选择用开源数据集, 也可以选择用业务数据集。

## 开源数据集

如果有业务数据集最好，没有的话可以用开源数据集，如下：

表 21.2: 数据集选取

模型类型	量化算法	校准数据集 (开源)	测试数据集 (开源)
LLM	Awq	pileval	wikitext2
LLM	GPTQ	wikitext2	wikitext2
VLM	Awq	MME	MME

校准数据集的选取与模型类型和量化算法相关，例如如果量化的是 LLM 模型，使用的是 Awq 算法，通常推荐使用 pileval 数据集作为校准集。针对这些开源数据集本文档提供了对应的下载命令，可以运行下载相应的数据集。具体操作如下：可打开 llmc-tpu/tools 文件，里面对应有 download\_calib\_dataset.py 和 download\_eval\_dataset.py 两个 python 脚本，分别用于下载校准集和测试集。

如果是 VLM 模型，建议使用 Awq 算法，下载数据集命令如下：

```
1 cd /workspace/llmc-tpu
    · 校准数据集
1 python3 tools/download_calib_dataset.py --dataset_name MME --save_path tpu/data/VLM/calib
    · 测试数据集
1 python3 tools/download_eval_dataset.py --dataset_name MME --save_path tpu/data/VLM/eval
```

如果是 LLM 模型，建议用 Awq 算法，下载数据集命令如下：

```
1 cd /workspace/llmc-tpu
    · 校准数据集
1 python3 tools/download_calib_dataset.py --dataset_name pileval --save_path tpu/data/LLM/calib
    · 测试数据集
1 python3 tools/download_eval_dataset.py --dataset_name wikitext2 --save_path tpu/data/LLM/eval
```

## 业务数据集

### 1. 业务校准数据集

如果模型经过下游业务数据集微调，在选择校准集时，通常应该选择业务数据集。<sup>\*</sup> 如果是 LLM，将业务数据集放置于上述 LLM/cali 目录下即可。至于数据集具体的格式，用户可以将一条一条数据文本，写到 txt 文件里面，每一行代表一条文本数据，使用上述的配置，可以实现自定义数据集的校准。<sup>\*</sup> 如果是 VLM，将业务数据集放置于上述 VLM/cali 目录下即可。至于数据集具体的格式，可以参考 VLM/cali/general\_custom\_data 中的格式，选择符合需求的格式即可。这里一定需要注意，最后的 json 文件应该命名为 samples.json。

### 2. 业务测试数据集

如果模型经过下游业务数据集校准，在选择测试集时，通常应该选择业务数据集测试。<sup>\*</sup> 如果是 LLM，将业务数据集放置于上述 LLM/eval 目录下即可。至于数据集具体的格式，用户可以将一条一条数据文本，写到 txt 文件里面，每一行代表一条文本数据，使用上述的配置，可以实现自定义数据集的测试。<sup>\*</sup> 如果是 VLM，将业务数据集放置于上述 VLM/eval 目录下即可。至于数据集具体的格式，可以参考 VLM/cali/general\_custom\_data 中的格式，选择符合需求的格式即可。这里一定需要注意，最后的 json 文件应该命名为 samples.json。

## 【阶段二】配置量化 config 文件

- 注意点：量化 config 文件包括了量化过程中所需的量化配置，用户可按照需求进行选择，同时为了对齐 TPU 硬件的配置也会对某些参数做出限制，具体可看下文详细介绍。

### config 文件参数说明

```

1 base:
2   seed: &seed 42
3 model:
4   type: Qwen2VL # 设置模型名，具体支持的模型参见llmc/models目录
5   path: /workspace/Qwen2-VL-2B-Instruct # 设置模型权重路径，请改成您需要的模型
6   torch_dtype: auto
7 calib:
8   name: mme # 设置成实际的校准数据集名称，mme, pileval等等
9   download: False
10  path: /workspace/llmc-tpu/tpu/data/VLM/cali/MME # 设置校准数据集路径
11  n_samples: 128
12  bs: 1
13  seq_len: 512
14  preproc: pileval_awq
15  seed: *seed
16 eval:
17  eval_pos: [pretrain, fake_quant]
18  name: mme # 设置成实际的测试数据集名称，mme, wikitext2等等
19  download: False
20  path: /workspace/llmc-tpu/tpu/data/VLM/eval/MME # 设置测试数据集路径
21  bs: 1

```

(续下页)

(接上页)

```

22     seq_len: 2048
23
24     quant:
25         method: Awq
26         quant_objects: [language] # 默认只量化LLM部分，如要量化VIT部分，则设置成[vision,F
27             ↪language]
28         weight:
29             bit: 4 # 设置成想要的量化bit，可以支持4或8
30             symmetric: False # 4bit填False；8bit填True
31             granularity: per_group # 4bit填per_group；8bit，填per_channel
32             group_size: 64 # 4bit填64(与TPU-MLIR对应)；8bit，填-1
33
34     special:
35         trans: True
36         trans_version: v2
37         weight_clip: True
38         clip_sym: True
39
40     save:
41         save_trans: True      # 当设置为True，可以保存下调整之后的浮点权重
42         save_path: ./save_path # 设置保存权重的路径
43
44     run:
45         task_name: awq_w_only
46         task_type: VLM # 设置成VLM或者LLM

```

上面是以 Awq 算法为例构建的一个完整的 config 文件。为了简便用户操作，用户可以将上面直接拷贝到自己的 config 中，然后对有注解的部分参数进行修改。下面对重要的一些参数做详细的说明：

表 21.3: 相关参数介绍

参数	描述
model	模型名称，支持的模型在 llmc/models 目录，可以自行支持新模型 llmc/models/xxxx.py
calib	calib 类参数主要指定校准集相关的参数
eval	eval 类参数主要指定了和测试集相关的参数
quant	指定量化参数，一般建议用 Awq 算法,quant_objects 一般选 language，关于 weight 量化参数参考下表

为了与 ‘TPU-MLIR ‘对齐,weight 量化相关参数配置如下:

表 21.4: weight-only 量化参数

bit	symmetric	granularity	group_size
4	False	per-channel or per-group	-1 or 64(default)
8	True	per-channel	-1

### 【阶段三】执行量化算法

```
1 cd /workspace/llmc-tpu  
2 python3 tpu/llm_quant.py --llmc_tpu_path . --config_path ./tpu/example.yml
```

- config\_path 则表示量化 config 文件对应的路径, llmc\_tpu\_path 表示当前 llmc\_tpu 路径

# CHAPTER 22

---

## TPU Profile 工具使用及分析

---

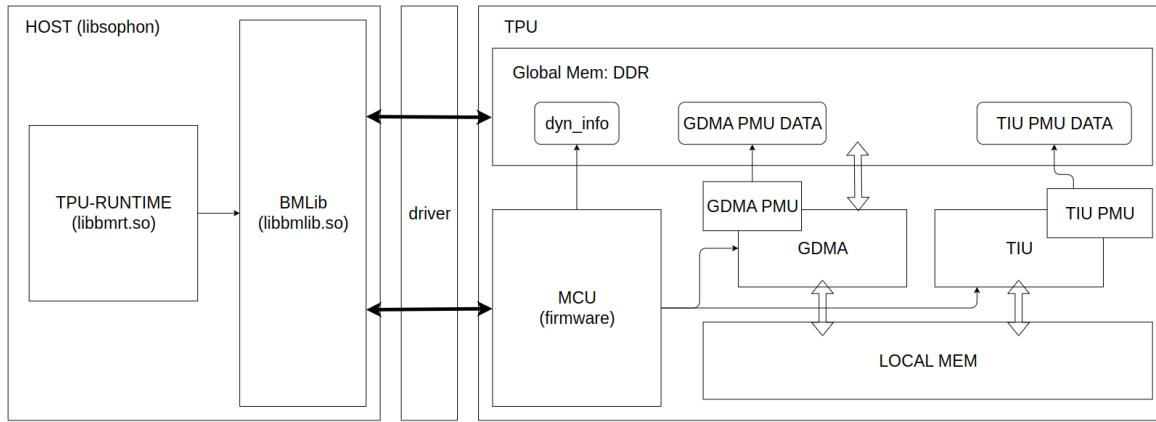
### 22.1 1. TPU 软件与硬件架构

完整的 TPU 推理应用是通过软硬件相互配合完成的，如下图所示：

软件方面，Host 端实现了 libsophon、驱动两个软件包。驱动负责对实际的 Host 与设备基础通信和资源管理机制的抽象，提供了基础的功能接口。对于 TPU 推理来说，libsophon 提供了具体的功能，其中 BMLib(libbmlib.so) 实现对驱动接口封装，提供兼容保证，简化调用流程，提升编程的效率和可移植性，TPU-RUNTIME(libbmrt.so) 提供了模型 (bmodel) 的加载、管理与执行等功能。

硬件方面，TPU 内部主要由 MCU、GDMA、TIU 三个 engine 来完成工作的。

- MCU 在 BM1684X 上是一个单核的 A53 处理器，通过 firmware 固件程序完成向 GDMA、TIU 两个 engine 下发命令、驱动通信、简单计算等具体功能，实现了算子的具体逻辑。
- GDMA 和 TIU 是实际的执行引擎，GDMA 用于 Global mem 与 Local mem 之间传输数据，实现了 1D、矩阵、4D 等数据搬运功能；TIU 对 local mem 中的数据执行密集计算命令，包括卷积、矩阵乘法、算术等原子操作。



TPU Profile 是将 Profile 数据转换为可视化网页的工具。Profile 数据的来源包括内部 GDMA PMU 和 TIU PMU 两个硬件模块记录的运行计时数据、各个软件模块关键函数信息、bmodel 里的元数据等。这些数据是在编译模型、以及应用运行时收集的。在实际部署中，默认是关闭的，可以通过环境变量来开启。

本文主要是利用 Profile 数据及 TPU Profile 工具，可视化模型的完整运行流程，来让读者对 TPU 内部有一个直观的认识。

## 22.2 2. 编译 bmodel

(本操作及下面操作会用到 TPU-MLIR)

由于 Profile 数据会编译中的一些 layer 信息保存到 bmodel 中，导致 bmodel 体积变大，所以默认是关闭的。打开方式是在调用 model\_deploy.py 加上 --debug 选项。如果在编译时未开启该选项，运行时开启 Profile 得到的数据在可视化时，会有部分数据缺失。

下面以 tpu-mlir 工程中的 yolov5s 模型来演示。

```
# 生成 top mlir
model_transform.py \
--model_name yolov5s \
--model_def ./yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 350,498,646 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s.mlir
```

```
# 将top mlir转换成fp16精度的bmodel
model_deploy.py \
--mlir yolov5s.mlir \
```

(续下页)

(接上页)

```
--quantize F16 \
--chip bm1684x \
--test_input yolov5s_in_f32.npz \
--test_reference yolov5s_top_outputs.npz \
--model yolov5s_1684x_f16.bmodel \
--debug # 记录profile数据
```

通过以上命令，将 yolov5s.onnx 编译成了 yolov5s\_bm1684x\_f16.bmodel。更多用法可以参见 [TPU-MLIR](#)。

### 22.3 3. 生成 Profile 原始数据

同编译过程，运行时的 Profile 功能默认是关闭的，防止在做 profile 保存与传输时产生额外时间消耗。需要开启 profile 功能时，在运行编译好的应用前设置环境变量 BMRUNTIME\_ENABLE\_PROFILE=1 即可。下面用 libspthon 中提供的模型测试工具 bmrt\_test 来作为应用，生成 profile 数据。

```
# 通过环境变量(BMRUNTIME_ENABLE_PROFILE)使能profile, 生成二进制数据
BMRUNTIME_ENABLE_PROFILE=1 bmrt_test --bmodel resnet50_fix8b.bmodel
```

下面是开启 Profile 后运行输出的日志：

```
[bmrt][load_bmodel:1084] INFO:Loading bmodel from [yolov5s_1684x_f16.bmodel]. Thanks for your patience...
[bmrt][load_bmodel:1023] INFO:pre net num: 0, load net num: 0
[bmrt][show_net_info:1520] INFO: #####
[bmrt][show_net_info:1521] INFO: NetName: yolov5s, Index=0
[bmrt][show_net_info:1523] INFO: ---- stage 0 ----
[bmrt][show_net_info:1532] INFO: Input 0) 'Images' shape=[ 1 3 640 640 ] dtype=FLOAT32 scale=1 zero_point=0
[bmrt][show_net_info:1542] INFO: Output 0) '350_Transpose_f32' shape=[ 1 3 80 80 85 ] dtype=FLOAT32 scale=1 zero_point=0
[bmrt][show_net_info:1542] INFO: Output 1) '498_Transpose_f32' shape=[ 1 3 40 40 85 ] dtype=FLOAT32 scale=1 zero_point=0
[bmrt][show_net_info:1542] INFO: Output 2) '646_Transpose_f32' shape=[ 1 3 20 20 85 ] dtype=FLOAT32 scale=1 zero_point=0
[bmrt][show_net_info:1545] INFO: #####
[bmrt][bmrt_test:782] INFO:>= running network #0, name: yolov5s, loop: 0
[bmrt][bmrt_test:868] INFO:reading input #0, bytesize=4915200
[bmrt][print_array:706] INFO:-> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1228800
[bmrt][write_block:295] INFO:write_block: type=1, len=36
[bmrt][write_block:295] INFO:write_block: type=8, len=44
[bmrt][end:76] INFO:bdc record_num=1838, max_record_num=1048576
[bmrt][write_block:295] INFO:write_block: type=3, len=58816
[bmrt][end:89] INFO:gdma record_num=196, max_record_num=1048576
[bmrt][write_block:295] INFO:write_block: type=4, len=37632
[bmrt][write_block:295] INFO:write_block: type=5, len=256
[bmrt][write_block:295] INFO:write_block: type=6, len=1072
[bmrt][print_note:94] INFO:*****
[bmrt][print_note:95] INFO:***** PROFILE MODE due to BMRUNTIME_ENABLE_PROFILE=1
[bmrt][print_note:96] INFO: Note: BMRuntime will collect time data during running * Profile开启提示, 防止在部署时忘记关闭
[bmrt][print_note:97] INFO: that will cost extra time.
[bmrt][print_note:98] INFO: Close PROFILE Mode by "unset BMRUNTIME_ENABLE_PROFILE"
[bmrt][print_note:99] INFO:*****
[bmrt][bmrt_test:1085] INFO:reading output #0, bytesize=6528000
[bmrt][print_array:706] INFO:-> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1632000
[bmrt][bmrt_test:1085] INFO:reading output #1, bytesize=1632000
[bmrt][print_array:706] INFO:-> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=408000
[bmrt][bmrt_test:1085] INFO:reading output #2, bytesize=408000
[bmrt][print_array:706] INFO:-> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=102000
[bmrt][bmrt_test:1089] INFO:net[yolov5s] stage[0], launch total time is 375609 us (npu 5650 us, cpu 369959 us) 由于开启了Profile, CPU时间不准确, 可忽略
[bmrt][bmrt_test:1042] INFO:+++ The network[yolov5s] stage[0] output_data ++
[bmrt][print_array:706] INFO:output data #0 shape: [1 3 80 80 85 ] < 0.30957 -0.289551 0.0744629 -0.203003 -11.9375 -1.54297 -5.25391 -3.05859 -5.39453 -5.64844 -95 -6.26172 ... > len=1632000
[bmrt][print_array:706] INFO:output data #1 shape: [1 3 40 40 85 ] < -0.0398254 0.253174 -0.383057 -0.520996 -11.0391 -1.3125 -5.11328 -3.32031 -5.46484 -5.97656 812 -6.29688 ... > len=408000
[bmrt][print_array:706] INFO:output data #2 shape: [1 3 20 20 85 ] < 0.713379 0.654297 -0.534668 -0.241577 -9.82031 -1.23047 -5.77344 -3.35547 -5.92578 -5.12109 -44 -6.63672 ... > len=102000
[bmrt][bmrt_test:1083] INFO:load input time(s): 0.003650
[bmrt][bmrt_test:1084] INFO:calculate time(s): 0.375613
[bmrt][bmrt_test:1085] INFO:get output time(s): 0.003838
[bmrt][bmrt_test:1086] INFO:compare time(s): 0.000827
```

同时在当前目录生成 bmprofile\_data-1 文件夹，为全部的 Profile 数据。

## 22.4 4. 可视化 Profile 数据

tpu-mlir 提供了 tpu\_profile.py 脚本，来把生成的二进制 profile 数据转换成网页文件，来进行可视化。命令如下：

```
# 将bmprofile_data_0目录的profile原始数据转换成网页放置到bmprofile_out目录
# 如果有图形界面，会直接打开浏览器，直接看到结果
tpu_profile.py bmprofile_data-1 bmprofile_out

ls bmprofile_out
# echarts.min.js profile_data.js result.html
```

用浏览器打开 bmprofile\_out/result.html 可以看到 profile 的图表。

此外，该工具还有其他用法，可以通过 tpu\_profile.py --help 来查看。

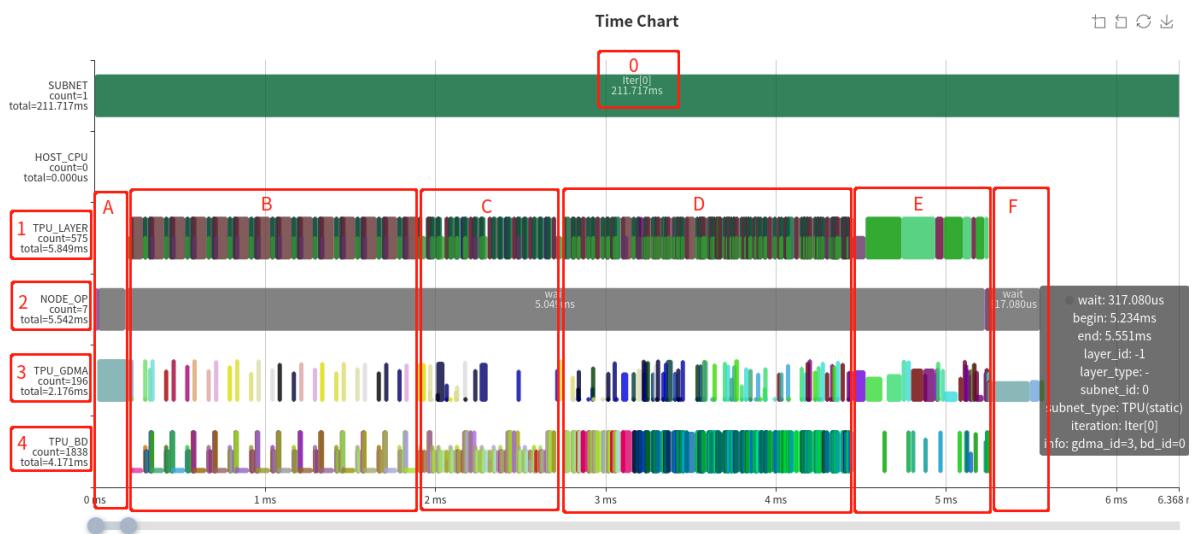
## 22.5 5. 结果分析

### 22.5.1 5.1 整体界面说明

完整界面大致可分为运行时序图和内存时空图。默认情况下内存时空图是折叠的，需要通过界面的“显示 LOCALMEM”和“显示 GLOBAL MEM”来展开。

下面对这两部分分别说明如何来分析 TPU 运行状态：

**Profile: yolov5s on BM1684X**



上图是运行时序图，根据图中标号说明如下：

- 在做 Profile 时，在 Host 的时间可能不准确，该部分仅用于表示子网分隔标记。

1. 该行表示的是整个网络中各个 Layer 的时序，是由下面的 TPU\_GDMA, TPU\_BD(TIU) 实际运行衍生计算得来。一个 Layer Group 会将一段算子分成数据搬运和计算两部分，并且是并行运行的，所以用半高的色块表示数据搬运，全高表示计算，避免重叠。
2. 该行表示 MCU 上的操作，记录的关键函数包括设置 GDMA、TIU 指令及等待完成等。加和后通常可以表示完整的实际运行时间。
3. 该行表示 TPU 中 GDMA 操作的时序。其中色块的高度表示实际使用的数据传输带宽大小。
4. 该行表示 TPU 中 TIU 操作的时序。其中色块高度表示该计算的有效利用率。

从 NODE\_OP 的下方的统计 total=5.542ms，说明整个网络运行时间是 5.542ms，也可以看出在实际网络运行时，配置指令只占非常短的时间，大部时间在等待。

整体运行过程可以分为三个部分 A 段, B-E 段, F 段。其中，A 段是利用 MCU 将用户空间的输入数据搬运到计算指令空间；F 段是利用 MCU 将计算指令空间的输出数据搬回到用户空间。下面主要对 B-E 段的模型计算过程进行说明。

熟悉 [TPU-MLIR](#) 的同学应该清楚，完整的网络并不是 Layer By Layer 来运行的，中间会经过将多个 Layer 根据硬件资源和调度关系进行融合，将加载、计算、保存分离出来，去掉中间不必要的数据搬进与搬出，形成一个 Layer Group，并划分成多个 Slice 来周期运行。整个网络根据结构可能会分成多个 Layer Group。可以观察 B、C、D 段的 Layer Pattern，中间有半高的加载保存操作，而且呈现了一定周期的循环，根据这些，我们可以判断出 B、C、D 是三个被融合后的 Layer Group。而且后面 E 段并没有明显的周期，这几个 Layer 是没有被融合的 Global Layer。整体上看，网络中只有 20% 的部分没有被融合，在这个层面上看，网络结构对于编译器相对比较友好。



上图是整体的内存时空图，包括了 LOCAL MEM 和 GLOBAL MEM 上下两部分。横轴表示时间，可以结合上面的运行时序图来看。纵轴表示内存空间范围。图中绿色块高度表示占用空间大小，宽度表示占用时间长短，此外，红色表示 GDMA 写入或 TIU 输出，绿色表示 GDMA 读取或 TIU 输入。

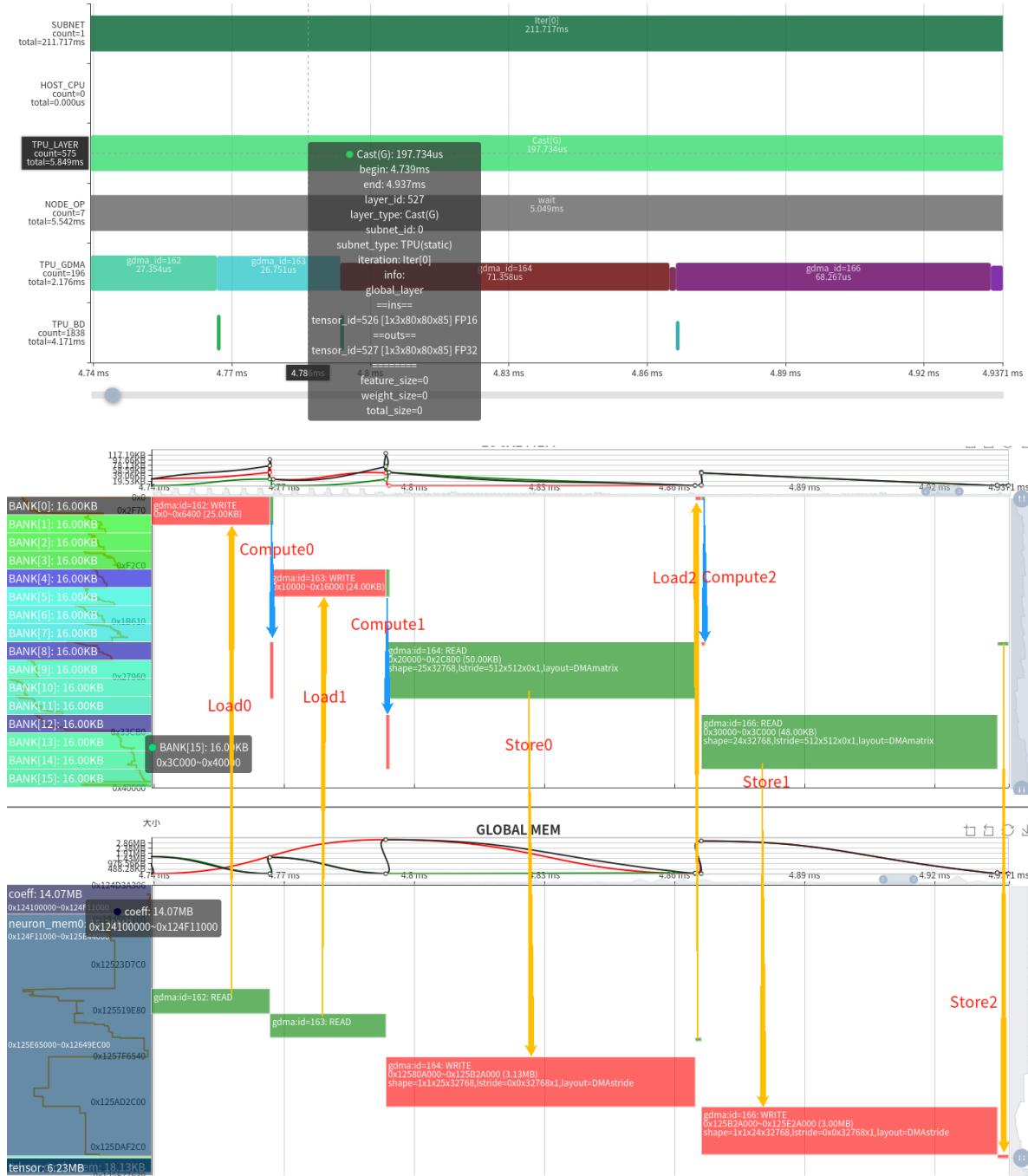
- LOCAL MEM 是 TPU 内部计算空间，对于 BM1684X 来说，TIU 一共有 64 个 Lane，每个 Lane 可使用 128KB 的内存，并分为了 16 个 bank。由于各个 Lane 的操作与内存是一致的，故图中只放了 Lane0 的内存占用情况。在计算过程中，还有一个需要注意的地方，计算的输入和输出最好不要在同一个 Bank 上，由于数据读写冲突，会影响计算效率。
- GLOBAL MEM 空间相对比较庞大，通常在 4GB–12GB 范围，为方便显示，只针对运行时使用的空间块进行显示。由于只有 GDMA 能与 GLOBAL MEM 通信，故绿色表示 GDMA 的读取操作，红色表示 GDMA 的写入操作。

从内存时空图中可以看出，对于 Layer Group 来说，Local Mem 的使用也呈周期性；TIU 的输入和输出通常是在每个 Bank 边界上，并且没有冲突。仅就这个网络来说，Local Mem 占用空间相对均匀，整个范围都有分布。从 GLOBAL MEM 的时空图上可以看到，以 Layer Group 运行时，写数据操作相对较少，读数据偏多。而在 Global Layer 运行时，会经过写回-> 读出-> 写回->…等操作。

此外，还可以看到运行时的 GLOBAL MEM 空间占用细节，可以分解为 Coeff 占用 14.07MB、Runtime 占用 15.20MB、Tensor 占用 6.23MB。

### 22.5.2 5.2 Global Layer

下面以比较简单的 Global Layer 来分析，根据 Layer 信息，Cast 的前一层由于是 Permute (图中未显示) 导致无法与其他算子融合。



从 Layer 上可以看到参数信息，当前层是将  $1 \times 3 \times 80 \times 80 \times 85$  的 fp16 tensor 数据转换为 fp32。计算过程为：

time ——————>

**Load0 | Compute0 | Store0 | |**

Load1 | Compute1 | Store1 |  
| Load2 | Compute2 | Store2

由于只有一个 GDMA 器件，Load 和 Store 只能串行执行，所以流水变成了：

time ——————>

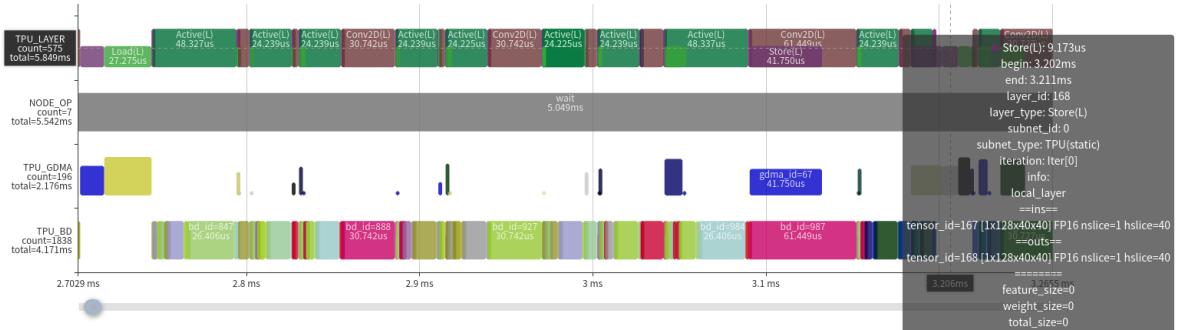
GDMA: Load0 | Load1 | Store0, Load2 | Store1 | Store2 TIU: | Compute0 | Compute1 | Compute2 |

从对应的内存时空图也可以看出完整的数据流动关系，输入数据是 fp16 转换到输出 fp32 后，内存翻倍了，因而传输时间大概为原来的两倍。

在计算过程中虽然已经做到流水并行，但由于受带宽限制，无法满足算力的需要，所以整个运行时间取决于数据搬进与搬出的时间。从另一方面也说明了 Layer 融合的必要性。

### 22.5.3 5.3 Local Layer Group

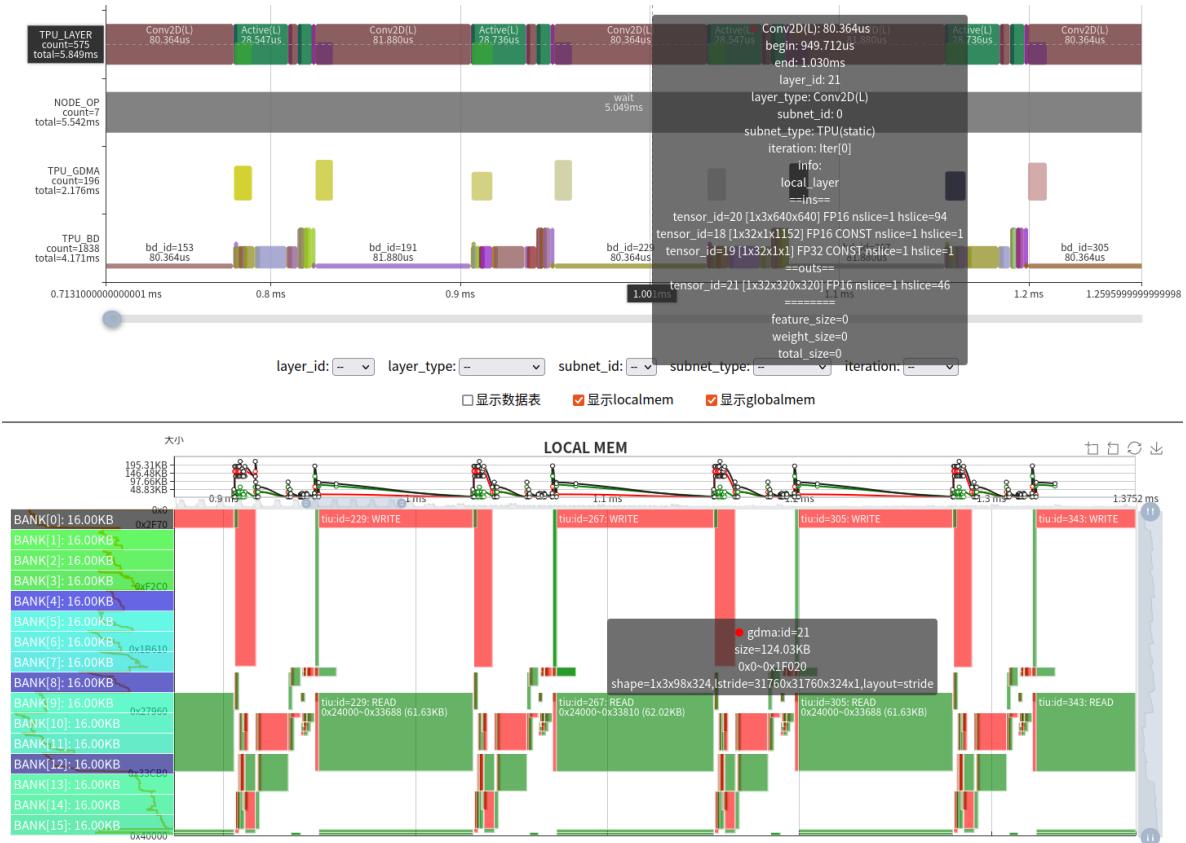
根据上面的 Layer Group 的情况，分为两种 case 来分析：



1. 效率较高的情况。主要特征是：

- 除了前面和后面，中间的只有很少的 GDMA 操作，显著地减少了数据的搬进搬出。
- TIU 操作效率都比较高，几乎算力全部是有效的。
- TIU 操作之间没有空隙（也是因为 GDMA 传输时间比较短）。

在这种情况下，可提升的空间非常有限了，只能从网络结构或其他方面来优化。



- 算力利用率比较低的情况。这种情况主要是网络算子参数与 TPU 架构不友好造成的。我们的 BM1684X 上有 64 个 Lane, 对应于输入的 IC, 也就是说输入 IC 是 64 的倍数才能充分利用 TIU 的 Conv 原子操作。但从图中参数可以看到网络 Conv 的输入 Channel 为 3, 导致有效计算只有 3/64。

遇到参数不友好的情况，有以下几种解决办法：

- 充分利用 LOCAL MEM 增大 Slice 以减少循环次数；
- 利用一些变换，如数据排列，来充分利用 TPU。其实对于首层为输入 Channel 为 3 的情况，我们引入了一种 3IC 的技术，已经解决了这种计算效率低的问题；
- 修改原始代码，调整相关计算。

在实际中，也遇到过很多无法避免的效率低下的情况，只能随着我们对 TPU 计算理解逐步加深，通过改进 TPU 架构或指令来解决。

## 22.6 总结

本文演示了对 TPU 做 Profile 的完整流程，并介绍了如何利用 Profile 的可视化图表来分析 TPU 中的运行过程与问题。

Profile 工具对我们开发 AI 编译器来说，是一个必要的工具。我们不仅需要在理论上分析和思考优化手段和方法，还需要从芯片内部实际运行角度来观察计算过程中的瓶颈，为软件和硬件设计和演进提供深层次的信息。另外，Profile 工具也为我们 Debug 提供了一种手段，可以直观地发现错误，比如内存踩踏、同步出错等问题。

此外，TPU Profile 的显示功能在不断完善中。

# CHAPTER 23

## 附录 01：从 NNTC 迁移至 TPU-MLIR

NNTC 所使用 Docker 版本为 sophgo/tpuc\_dev:v2.1, MLIR 使用的版本及环境初始化请参考[开发环境配置](#)。

下面将以 yolov5s 为例, 讲解 NNTC 和 TPU-MLIR 在量化方面的异同, 浮点模型编译方面可以直接参考《TPU-MLIR 快速入门指南》的“编译 ONNX 模型”章节内容, 以下内容假设已经按照《TPU-MLIR 快速入门指南》中描述准备好了 yolov5s 模型。

### 23.1 ONNX 模型导入

在 TPU-MLIR 中要对模型进行量化首先要把原始模型转为 top 层的 mlir 文件, 这一步可以类比为 NNTC 中分步量化生成 fp32umodel 的过程。

#### 1. TPU-MLIR 的模型转换命令

```
$ model_transform.py \
  --model_name yolov5s \
  --model_def ./yolov5s.onnx \
  --input_shapes [[1,3,640,640]] \
  --mean 0,0,0,0 \
  --scale 0.0039216,0.0039216,0.0039216 \
  --keep_aspect_ratio \
  --pixel_format rgb \
  --output_names 350,498,646 \
  --test_input ./image/dog.jpg \
  --test_result yolov5s_top_outputs.npz \
  --mlir yolov5s.mlir
```

TPU-MLIR 可以直接把图片预处理编码到转换出的 mlir 文件中。

## 2. NNTC 的模型转换命令

```
$ python3 -m ufw.tools.on_to_umodel \
-m ./yolov5s.onnx \
-s '(1,3,640,640)' \
-d 'compilation' \
--cmp
```

NNTC 导入模型的时候不能指定预处理方式。

## 23.2 制作量化校准表

想要生成定点模型都需要经过量化工具对模型进行量化, nntc 中分步量化这里使用的是 calibration\_use\_pb, mlir 使用的是 run\_calibration.py

输入数据的数量根据情况准备 100~1000 张左右, 用现有的 100 张来自 COCO2017 的图片举例, 执行 calibration:

在 nntc 中使用分步量化还需要自行使用图片量化数据集制作 lmdb 量化数据集, 并且修改 fp32\_prototxt, 将数据输入指向 lmdb 文件

---

**备注:** 关于 NNTC 量化数据集制作方式可以参考《TPU-NNTC 开发参考手册》的“模型量化”章节内容, 且注意该 lmdb 数据集与 TPU-MLIR 并不兼容。TPU-MLIR 可以直接使用原始图片作为量化工具输入。如果是语音、文字等非图片数据, 需要将其转化为 npz 文件。

---

### 1. MLIR 量化模型

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

经过量化之后会得到量化表 yolov5s\_cali\_table。

### 2. NNTC 量化模型

```
$ calibration_use_pb quantize \
--model=./compilation/yolov5s_bmnemo_test_fp32.prototxt \
--weights=./compilation/yolov5s_bmnemo.fp32umodel \
-save_test_proto=True --bitwidth=TO_INT8
```

在 nntc 中, 量化之后得到的是 int8umodel 以及 prototxt。

### 23.3 生成 int8 模型

转成 INT8 对称量化模型, 执行如下命令:

1. MLIR:

```
$ model_deploy.py \
--mlir yolov5s.mlir \
--quantize INT8 \
--calibration_table yolov5s_cali_table \
--processor bm1684 \
--test_input yolov5s_in_f32.npz \
--test_reference yolov5s_top_outputs.npz \
--tolerance 0.85,0.45 \
--model yolov5s_1684_int8_sym.bmodel
```

运行结束之后得到 yolov5s\_1684\_int8\_sym.bmodel。

2. NNTC:

在 NNTC 中, 则是使用 int8umodel 以及 prototxt 使用 bmnetu 工具生成 int8 的 bmodel。

```
$ bmnetu --model=./compilation/yolov5s_bmneto_deploy_int8_unique_top.prototxt \
--weight=./compilation/yolov5s_bmneto.int8umodel
```

运行结束之后得到 compilation.bmodel。

# CHAPTER 24

## 附录 02：TpuLang 的基本元素

本章将介绍 TpuLang 程序的基本元素：Tensor、Scalar、Control Functions 和 Operator。

### 24.1 张量 (Tensor)

TpuLang 中 Tensor 的 name, data, data type, tensor type 均最多只能声明或者设置 1 次。

一般情况下推荐创建 Tensor 不指定 Name，以免因为 Name 相同导致问题。只有在必须指定 Name 时，才需要在创建 Tensor 时指定 Name。

对于作为 Operator 输出的 Tensor，可以不指定 shape，因为 Operator 会自行推导。即使指定了 shape，若 Tensor 是 Operator 的输出，则同样由 Operator 自行推导并修改。

TpuLang 中 Tensor 的定义如下：

```
class Tensor:

    def __init__(self,
                 shape: list = [],
                 name: str = None,
                 ttype="neuron",
                 data=None,
                 dtype: str = "float32",
                 scale: Union[float, List[float]] = None,
                 zero_point: Union[int, List[int]] = None)
        #pass
```

如上所示，TpuLang 中 Tensor 有 5 个参数。

- shape: Tensor 的形状, List[int], 对于 Operator 输出的 Tensor, 可以不指定 shape, 默认值为 []。
- Name: Tensor 的名称, string 或 None, 该值推荐使用默认值 None 以免因为 Name 相同导致问题;
- ttype: Tensor 的类型, 可以是”neuron”或”coeff”, 初始值为”neuron”;
- data: Tensor 的数据, ndarray 或 None, 默认值为 None, 此时 Tensor 将根据指定的形状初始化为全零。当 ttype 为 coeff 时, 不可以为 None, data 为 ndarray, 此时 data 的 shape, dtype 必须与输入 shape, dtype 一致。
- dtype: Tensor 的数据类型, 默认值为”float32”, 否则取值范围为”float32”, “float16”, “int32”, “uint32”, “int16”, “uint16”, “int8”, “uint8”;
- scale: Tensor 的量化参数, float 或 List[float], 默认值为 None;
- zero\_point: Tensor 的偏移参数, int 或 List[int], 默认值为 None;

声明 Tensor 的示例:

```
#activation
input = tpul.Tensor(name='x', shape=[2,3], dtype='int8')
#weight
weight = tpul.Tensor(dtype='float32', shape=[3,4], data=np.random.uniform(0,1,
˓→shape).astype('float32'), ttype="coeff")
```

## 24.2 张量前处理 (Tensor.preprocess)

TpuLang 中 Tensor 如果是输入, 且需要对输入进行前处理, 可以调用该函数

TpuLang 中 Tensor.preprocess 的定义如下:

```
class Tensor:

    def preprocess(self,
                  mean : List[float] = [0, 0, 0],
                  scale : List[float] = [1.0, 1.0, 1.0],
                  pixel_format : str = 'bgr',
                  channel_format : str = 'nchw',
                  resize_dims : List[int] = None,
                  keep_aspect_ratio : bool = False,
                  keep_ratio_mode : str = 'letterbox',
                  pad_value : int = 0,
                  pad_type : str = 'center',
                  white_level : float = 4095,
                  black_level : float = 112):
        #pass
```

如上所示, TpuLang 中 Tensor 的 preprocess 有以下几个参数。

- mean: Tensor 的每个 channel 的平均值, 默认值为 [0, 0, 0];

- scale: Tensor 的每个 channel 的 scale 值, 默认值为 [1, 1, 1];
- pixel\_format: Tensor 的 pixel 的方式, 默认值为 'bgr', 取值范围为: 'rgb', 'bgr', 'gray', 'rgba', 'gbrg', 'grbg', 'bggr', 'rggb';
- channel\_format: Tensor 的格式, channel 维在前还是在最后。默认值为 'nchw', 取值范围为 "nchw", "nhwc"。
- resize\_dims: Tensor 的 resize 后的 [h, w], 默认值为 None, 表示取 Tensor 的 h 和 w;
- keep\_aspect\_ratio: resize 参数, 是否保持相同的 scale。bool 量, 默认值为 False;
- keep\_ratio\_mode: resize 参数, 如果使能 keep\_aspect\_ratio 的两种模式, 默认值 'letterbox', 取值范围为 'letterbox', 'short\_side\_scale';
- pad\_value: resize 参数, 当 resize 时 pad 的值。int 类型, 默认值为 0;
- pad\_type: resize 参数, 当 resize 时 pad 的方式。str 类型, 默认值为 'center', 取值范围为 'normal', 'center';
- white\_level: raw 参数。str 类型, 默认值为 4095;
- black\_level: raw 参数。str 类型, 默认值为 112;

声明 Tensor.preprocess 的示例:

```
#activation
input = tpul.Tensor(name='x', shape=[2,3], dtype='int8')
input.preprocess(mean=[123.675,116.28,103.53], scale=[0.017,0.017,0.017])
# pass
```

## 24.3 标量 (Scalar)

定义一个标量 Scalar。Scalar 是一个常量, 在声明时指定, 且不能修改。

```
class Scalar:
    def __init__(self, value, dtype=None):
        #pass
```

Scalar 构造函数有两个参数,

- value: Variable 型, 即 int/float 型, 无默认值, 必须指定;
- dtype: Scalar 的数据类型, 为默认值 None 等同于 "float32", 否则取值范围为 "float32", "float16", "int32", "uint32", "int16", "uint16", "int8", "uint8";

使用实例:

```
pad_val = tpul.Scalar(1.0)
pad = tpul.pad(input, value=pad_val)
```

## 24.4 Control Functions

控制函数 (control functions) 主要包括控制 TpuLang 使用时的初始化、启动编译生成目标文件等。

控制函数常用于 TpuLang 程序的 Tensor 和 Operator 之前和之后。比如在写 Tensor 和 Operator 之前，可能需要做初始化。在完成 Tensor 和 Operator 编写之后，可能需要启动编译和反初始化。

### 24.4.1 初始化函数

初始化 Function，在一个程序中构建网络之前使用。

初始化函数接口如下所示，选择处理器型号。

```
def init(device):
    #pass
```

- device: string 类型。取值范围” BM1684X” |” BM1688” |” CV183X”。

### 24.4.2 compile

#### 接口定义

```
def compile(name: str,
           inputs: List[Tensor],
           outputs: List[Tensor],
           cmp=True,
           refs=None,
           mode='f32',      # unused
           dynamic=False,
           asymmetric=False,
           no_save=False,
           opt=2,
           mlir_inference=True,
           bmodel_inference=True,
           log_level="normal",
           embed_debug_info=False):
```

## 功能描述

用于将 TpuLang 模型编译为 bmodel。

## 参数说明

- name: string 类型。模型名称。
- inputs: List[Tensor]，表示编译网络的所有输入 Tensor；
- outputs: List[Tensor]，表示编译网络的所有输出 Tensor；
- cmp: bool 类型，True 表示需要结果比对，False 表示仅编译；如果 mlir\_inference 为 False，cmp 参数无效。
- refs: List[Tensor]，表示编译网络的所有需要比对验证的 Tensor；
- mode: string 类型，废弃。
- dynamic: bool 类型，是否进行动态编译。
- no\_save: bool 类型，是否将中间文件暂存到共享内存并随进程释放，启用该项时 Compile 会返回生成的 bmodel 文件的 bytes-like object，用户需要自行接收和处理，如使用 f.write(bmodel\_bin) 保存。
- asymmetric: bool 类型，是否为非对称量化。
- opt: int 类型，表示编译器 group 优化级别。0，表示不需要进行 group；1，表示尽可能进行 group；2，表示根据动态规划进行 group。默认值为 2。
- mlir\_inference: bool 类型，是否执行 mlir 的推理，如果为 False，cmp 参数无效。
- bmodel\_inference: bool 类型，是否执行 bmodel 的推理。
- log\_level 用来控制日志等级，目前支持 only-pass、only-layer-group、normal、quiet：
  - only-pass: 主要打印图优化 pattern 匹配情况。
  - only-layer-group: 主要打印 layer group 信息。
  - normal: 编译生成 bmodel 的日志都会打印出来
  - quiet: 什么都不打印
- embed\_debug\_info: bool 类型，是否开启 profile 模式。

### 24.4.3 反初始化

在网络构建之后，需要进行反初始化结束。只有在反初始化后，之前 Tpulang 的数据才会得到释放

```
def deinit():
    #pass
```

#### 24.4.4 重置默认图

在网络构建之前，需要进行重置默认图操作。如果输入 graph 为 None，重置默认图后，当前 graph 为空图。如果设置输入 graph，会设置 graph 为默认图。如果只有一个子图，可以不需要显示调用 reset\_default\_graph。因为 init 函数会调用该函数。

```
def reset_default_graph(graph = None):
    #pass
```

#### 24.4.5 获取当前默认图

在网络构建之后，如果需要得到默认的子图，调用该函数可以得到默认的 graph。

```
def get_default_graph():
    #pass
```

#### 24.4.6 重置图

如果需要清除 graph 以及其保存的 Tensor 信息，可以调用该函数。graph 为 None 时，清除当前默认图的信息。

```
def reset_graph(graph = None):
    #pass
```

注意：如果 graph 中的 Tensor 还被其他 graph 使用，不要调用该函数清除 graph 信息

#### 24.4.7 舍入模式

舍入是指按照一定的规则舍去某些数字后面多余的尾数的过程，以得到更简短、明确的数字表示。给定  $x$ ，舍入结果是  $y$ ，有下面的舍入模式供选择。

四舍五入，当小数值为 0.5 时舍入到邻近的偶数，对应的值是 half\_to\_even。

四舍五入，正数接近于正无穷，负数接近于负无穷，对应的值是 half\_away\_from\_zero，公式如下

$$y = \text{sign}(x) \lfloor |x| + 0.5 \rfloor = -\text{sign}(x) \lceil -|x| - 0.5 \rceil$$

无条件舍去，接近于原点，对应的值是 towards\_zero，公式如下

$$y = \text{sign}(x) \lfloor |x| \rfloor = -\text{sign}(x) \lceil -|x| \rceil = \begin{cases} \lfloor x \rfloor & \text{if } x > 0, \\ \lceil x \rceil & \text{otherwise.} \end{cases}$$

接近于负无穷，对应的值是 down，公式如下

$$y = \lfloor x \rfloor = -\lceil -x \rceil$$

接近于正无穷，对应的值是 up，公式如下

$$y = \lceil x \rceil = -\lfloor -x \rfloor$$

四舍五入，接近于正无穷，对应的值是 half\_up，公式如下

$$y = \lceil x + 0.5 \rceil = -\lfloor -x - 0.5 \rfloor = \left\lceil \frac{\lfloor 2x \rfloor}{2} \right\rceil$$

四舍五入，接近于正无穷，对应的值是 half\_down，公式如下

$$y = \lfloor x - 0.5 \rfloor = -\lceil -x + 0.5 \rceil = \left\lfloor \frac{\lceil 2x \rceil}{2} \right\rfloor$$

下表列出不同舍入模式下 x 与 y 的对应关系。

	Half to Even	Half Away From Zero	Towards Zero	Down	Up	Half Up	Half Down
+1.8	+2	+2	+1	+1	+2	+2	+2
+1.5	+2	+2	+1	+1	+2	+2	+1
+1.2	+1	+1	+1	+1	+2	+1	+1
+0.8	+1	+1	0	0	+1	+1	+1
+0.5	0	+1	0	0	+1	+1	0
+0.2	0	0	0	0	+1	0	0
-0.2	0	0	0	-1	0	0	0
-0.5	0	-1	0	-1	0	0	-1
-0.8	-1	-1	0	-1	0	-1	-1
-1.2	-1	-1	-1	-2	-1	-1	-1
-1.5	-2	-2	-1	-2	-1	-1	-2
-1.8	-2	-2	-1	-2	-1	-2	-2

## 24.5 Operator

为了 TpuLang 编程时可以考虑到获取好的性能，下面会将 Operator 分成本地操作（Local Operator）、受限本地操作（Limited Local Operator）和全局操作（Global Operator）。

- 本地操作：在启动编译时，可以与其它的本地操作进行合并优化，使得操作之间的数据只存在于 TPU 的本地存储中。
- 受限本地操作：在一定条件下才能作为本地操作与其它本地操作进行合并优化。
- 全局操作：不能与其它操作进行合并优化，操作的输入输出数据都需要放到 TPU 的全局存储中。

以下操作中，很多属于按元素计算 (Element-wise) 的操作，要求输入输出 Tensor 的 shape 具备相同数量的维度。

当操作的输入 Tensor 是 2 个时，分为支持 shape 广播和不支持 shape 广播两种。支持 shape 广播表示 tensor\_i0 (输入 0) 和 tensor\_i1 (输入 1) 的同一维度的 shape 值可以不同，此时其中一个 tensor 的 shape 值必须是 1，数据将被广播扩展到另一个 tensor 对应的 shape 值。不支持 shape 广播则要求 tensor\_i0 (输入 0) 和 tensor\_i1 (输入 1) 的 shape 值一致。

### 24.5.1 NN/Matrix Operator

`conv`

#### 接口定义

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

#### 功能描述

二维卷积运算。可参考各框架下的二维卷积定义。该操作属于 **本地操作**。

#### 参数说明

- `input`: Tensor 类型, 表示输入 Tensor, 4 维 NCHW 格式。
- `weight`: Tensor 类型, 表示卷积核 Tensor, 4 维 NCHW 格式。
- `bias`: Tensor 类型, 表示偏置 Tensor。为 `None` 时表示无偏置, 反之则要求 `shape` 为 `[1, oc, 1, 1]`, `oc` 表示输出 Channel 数。
- `stride`: `List[int]`, 表示每个空间维度的步长大小, 取 `None` 则表示 `[1,1]`, 不为 `None` 时要求长度 2。
- `dilation`: `List[int]`, 表示每个空间维度的空洞大小, 取 `None` 则表示 `[1,1]`, 不为 `None` 时要求长度为 2。
- `pad`: `List[int]`, 表示每个空间维度的填充大小, 遵循 `[x1_begin, x2_begin, ..., x1_end, x2_end, ...]` 顺序。取 `None` 则表示 `[0,0,0,0]`, 不为 `None` 时要求长度为 4。
- `groups`: `int` 型, 表示卷积层的组数。
- `out_dtype`: `string` 类型或 `None`, 为 `None` 时与 `input` 数据类型一致。取值为范围为 “`float32`”, “`float16`”。表示输出 Tensor 的数据类型。
- `out_name`: `string` 类型或 `None`, 表示输出 Tensor 的名称, 为 `None` 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。

## conv\_int

### 接口定义

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

### 功能描述

二维卷积定点运算。可参考各框架下的二维卷积定义。

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    output = (input - izp) Conv (weight - wzp) + bias[c];
```

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型, 表示输入 Tensor, 4 维 NCHW 格式。
- weight: Tensor 类型, 表示卷积核 Tensor, 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。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [1, oc, 1, 1]。bias 的数据类型为 int32。
- stride: List[int], 表示每个空间维度的步长大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- dilation: List[int], 表示每个空间维度的空洞大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- pad: List[int], 表示每个空间维度的填充大小, 遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end, …] 顺序。取 None 则表示 [0,0,0,0], 不为 None 时要求长度为 4。
- groups: int 型, 表示卷积层的组数。若 ic=oc=groups 时, 则卷积为 depthwise conv。
- input\_zp: List[int] 型或 int 型, 表示输入偏移。取 None 则表示 0, 取 List 时要求长度为 ic。当前不支持 List[int] 型。
- weight\_zp: List[int] 型或 int 型, 表示卷积核偏移。取 None 则表示 0, 取 List 时要求长度为 ic, 其中 ic 表示输入的 Channel 数。
- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int32。取值范围: int32/uint32。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

**conv\_quant****接口定义**

```
def conv_quant(input: Tensor,
              weight: Tensor,
              bias: Tensor = None,
              stride: List[int] = None,
              dilation: List[int] = None,
              pad: List[int] = None,
              group: int = 1,
              input_scale: Union[float, List[float]] = None,
              weight_scale: Union[float, List[float]] = None,
              output_scale: Union[float, List[float]] = None,
              input_zp: Union[int, List[int]] = None,
              weight_zp: Union[int, List[int]] = None,
              output_zp: Union[int, List[int]] = None,
              out_dtype: str = None,
              out_name: str = None):
    # pass
```

**功能描述**

二维卷积定点运算。可参考各框架下的二维卷积定义。

```
for c in channel
    izp = is_ipz_const ? ipz_val : ipz_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    conv_i32 = (input - ipz) Conv (weight - wzp) + bias[c];
    output = requant_int(conv_i32, mul, shift) + ozp
    其中mul, shift由iscale, wscale, oscale得到
```

该操作属于 **本地操作**。

**参数说明**

- tensor\_i: Tensor 类型，表示输入 Tensor，4 维 NCHW 格式。
- weight: Tensor 类型，表示卷积核 Tensor，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。
- bias: Tensor 类型，表示偏置 Tensor。为 None 时表示无偏置，反之则要求 shape 为 [1, oc, 1, 1]。bias 的数据类型为 int32。
- stride: List[int]，表示每个空间维度的步长大小，取 None 则表示 [1,1]，不为 None 时要求长度为 2。
- dilation: List[int]，表示每个空间维度的空洞大小，取 None 则表示 [1,1]，不为 None 时要求长度为 2。

- pad: List[int]，表示每个空间维度的填充大小，遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end,…] 顺序。取 None 则表示 [0,0,0,0]，不为 None 时要求长度为 4。
- groups: int 型，表示卷积层的组数。若 ic=oc=groups 时，则卷积为 depthwise conv。
- input\_scale: List[float] 型或 float 型，表示输入量化参数。取 None 则使用 input Tensor 中的量化参数，取 List 时要求长度为 ic。当前不支持 List[float] 型。
- weight\_scale: List[float] 型或 float 型，表示卷积核量化参数。取 None 则使用 weight Tensor 中的量化参数，取 List 时要求长度为 oc。
- output\_scale: List[float] 型或 float 型，表示卷积核量化参数。不可以取 None，取 List 时要求长度为 oc。当前不支持 List[float] 型。
- input\_zp: List[int] 型或 int 型，表示输入偏移。取 None 则表示 0，取 List 时要求长度为 ic。当前不支持 List[int] 型。
- weight\_zp: List[int] 型或 int 型，表示卷积核偏移。取 None 则表示 0，取 List 时要求长度为 oc。
- output\_zp: List[int] 型或 int 型，表示卷积核偏移。取 None 则表示 0，取 List 时要求长度为 oc。当前不支持 List[int] 型。
- out\_dtype: string 类型或 None，表示输入 Tensor 的类型，取 None 表示为 int8。取值范围: int8/uint8。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688：输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X：输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

## deconv

### 接口定义

```
def deconv(input: Tensor,
           weight: Tensor,
           bias: Tensor = None,
           stride: List[int] = None,
           dilation: List[int] = None,
           pad: List[int] = None,
```

(续下页)

(接上页)

```

output_padding: List[int] = None,
group: int = 1,
out_dtype: str = None,
out_name: str = None):
#pass

```

## 功能描述

二维反卷积运算。可参考各框架下的二维反卷积定义。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入 Tensor, 4 维 NCHW 格式。
- weight: Tensor 类型, 表示卷积核 Tensor, 4 维 NCHW 格式。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [1, oc, 1, 1], oc 表示输出 Channel 数。
- stride: List[int], 表示每个空间维度的步长大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- dilation: List[int], 表示每个空间维度的空洞大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- pad: List[int], 表示每个空间维度的填充大小, 遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end, …] 顺序。取 None 则表示 [0,0,0,0], 不为 None 时要求长度为 4。
- output\_padding: List[int], 表示输出每个空间维度的填充大小, 取 None 则表示 [0,0], 不为 None 时要求长度为 2。
- group: int 类型, 表示表示反卷积层的组数。
- out\_dtype: string 类型或 None, 为 None 时与 input 数据类型一致。取值为范围为 “float32”, “float16”。表示输出 Tensor 的数据类型。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。

### deconv\_int

#### 接口定义

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

#### 功能描述

二维反卷积定点运算。可参考各框架下的二维卷积定义。

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    output = (input - izp) Deconv (weight - wzp) + bias[c];
```

该操作属于 **本地操作**。

#### 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor，4 维 NCHW 格式。
- weight: Tensor 类型，表示卷积核 Tensor，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。
- bias: Tensor 类型，表示偏置 Tensor。为 None 时表示无偏置，反之则要求 shape 为 [1, oc, 1, 1]。bias 的数据类型为 int32

- stride: List[int], 表示每个空间维度的步长大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- dilation: List[int], 表示每个空间维度的空洞大小, 取 None 则表示 [1,1], 不为 None 时要求长度为 2。
- pad: List[int], 表示每个空间维度的填充大小, 遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end, …] 顺序。取 None 则表示 [0,0,0,0], 不为 None 时要求长度为 4。
- output\_padding: List[int], 表示输出的填充大小, 取 None 则表示 [0,0], 不为 None 时要求长度为 1 或 2。
- groups: int 型, 表示反卷积层的组数。
- input\_zp: List[int] 型或 int 型, 表示输入偏移。取 None 则表示 0, 取 List 时要求长度为 ic。当前不支持 List[int] 型。
- weight\_zp: List[int] 型或 int 型, 表示卷积核偏移。取 None 则表示 0, 取 List 时要求长度为 ic, 其中 ic 表示输入的 Channel 数。
- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int32。取值范围: int32/uint32。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

## conv3d

### 接口定义

```
def conv3d(input: Tensor,
           weight: Tensor,
           bias: Tensor = None,
           stride: List[int] = None,
           dilation: List[int] = None,
           pad: List[int] = None,
           group: int = 1,
           out_dtype: str = None,
```

(续下页)

(接上页)

```
out_name: str = None):
#pass
```

## 功能描述

三维卷积运算。可参考各框架下的三维卷积定义。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入 Tensor, 5 维 NCDHW 格式。
- weight: Tensor 类型, 表示卷积核 Tensor, 4 维 NCDHW 格式。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [1, oc, 1, 1, 1] 或 [oc], oc 表示输出 Channel 数。
- stride: List[int], 表示每个空间维度的步长大小, 取 None 则表示 [1,1,1], 不为 None 时要求长度为 3。
- dilation: List[int], 表示每个空间维度的空洞大小, 取 None 则表示 [1,1,1], 不为 None 时要求长度为 3。
- pad: List[int], 表示每个空间维度的填充大小, 遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end, …] 顺序。取 None 则表示 [0,0,0,0,0,0], 不为 None 时要求长度为 6。
- groups: int 型, 表示卷积层的组数。
- out\_dtype: string 类型或 None, 为 None 时与 input 数据类型一致。取值为范围为 “float32”, “float16”。表示输出 Tensor 的数据类型。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 weight 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。

## conv3d\_int

### 接口定义

```
def conv3d_int(input: Tensor,
               weight: Tensor,
               bias: Tensor = None,
               stride: List[int] = None,
               dilation: List[int] = None,
               pad: List[int] = None,
               group: int = 1,
               input_zp: Union[int, List[int]] = None,
               weight_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
```

### 功能描述

三维卷积定点运算。可参考各框架下的三维卷积定义。

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    kzp = is_kzp_const ? kzp_val : kzp_vec[c];
    output = (input - izp) Conv3d (weight - kzp) + bias[c];
```

其中 Conv3d 表示 3D 卷积计算。

该操作属于 **本地操作**。

### 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor，5 维 NCTHW 格式。
- weight: Tensor 类型，表示卷积核 Tensor，5 维 [oc, ic, kt, kh, kw] 格式。其中 oc 表示输出 Channel 数，ic 表示输入 channel 数，kt 是 kernel\_t，kh 是 kernel\_h，kw 是 kernel\_w。
- bias: Tensor 类型，表示偏置 Tensor。为 None 时表示无偏置，反之则要求 shape 为 [1, oc, 1, 1, 1]。
- stride: List[int]，表示每个空间维度的步长大小，取 None 则表示 [1,1,1]，不为 None 时要求长度为 3。
- dilation: List[int]，表示每个空间维度的空洞大小，取 None 则表示 [1,1,1]，不为 None 时要求长度为 3。
- pad: List[int]，表示每个空间维度的填充大小，遵循 [x1\_begin, x2\_begin, ..., x1\_end, x2\_end, ...] 顺序。取 None 则表示 [0,0,0,0,0,0]。不为 None 时要求长度 6。

- groups: int 型, 表示卷积层的组数。若  $ic=oc=groups$  时, 则卷积为 depthwise conv3d。
- input\_zp: List[int] 型或 int 型, 表示输入偏移。取 None 则表示 0, 取 List 时要求长度为  $ic$ 。当前不支持 List[int] 型。
- weight\_zp: List[int] 型或 int 型, 表示卷积核偏移。取 None 则表示 0, 取 List 时要求长度为  $ic$ , 其中  $ic$  表示输入的 Channel 数。
- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int32。取值范围: int32/uint32。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

## conv3d\_quant

### 接口定义

```
def conv3d_quant(input: Tensor,
                 weight: Tensor,
                 bias: Tensor = None,
                 stride: List[int] = None,
                 dilation: List[int] = None,
                 pad: List[int] = None,
                 group: int = 1,
                 input_scale: Union[float, List[float]] = None,
                 weight_scale: Union[float, List[float]] = None,
                 output_scale: Union[float, List[float]] = None,
                 input_zp: Union[int, List[int]] = None,
                 weight_zp: Union[int, List[int]] = None,
                 output_zp: Union[int, List[int]] = None,
                 out_dtype: str = None,
                 out_name: str = None):
    # pass
```

## 功能描述

二维卷积定点运算。可参考各框架下的二维卷积定义。

```
for c in channel
    izp = is_izp_const ? izp_val : izp_vec[c];
    wzp = is_wzp_const ? wzp_val : wzp_vec[c];
    conv_i32 = (input - izp) Conv (weight - wzp) + bias[c];
    output = requant_int(conv_i32, mul, shift) + ozp
    其中mul, shift由iscale, wscale, oscale得到
```

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor，5 维 NCTHW 格式。
- weight: Tensor 类型，表示卷积核 Tensor，5 维 [oc, ic, kt, kh, kw] 格式。其中 oc 表示输出 Channel 数，ic 表示输入 channel 数，kt 是 kernel\_t，kh 是 kernel\_h，kw 是 kernel\_w。
- bias: Tensor 类型，表示偏置 Tensor。为 None 时表示无偏置，反之则要求 shape 为 [1, oc, 1, 1, 1]。bias 的数据类型为 int32。
- stride: List[int]，表示每个空间维度的步长大小，取 None 则表示 [1,1,1]，不为 None 时要求长度为 3。
- dilation: List[int]，表示每个空间维度的空洞大小，取 None 则表示 [1,1,1]，不为 None 时要求长度为 3。
- pad: List[int]，表示每个空间维度的填充大小，遵循 [x1\_begin, x2\_begin…x1\_end, x2\_end, …] 顺序。取 None 则表示 [0,0,0,0,0,0]。不为 None 时要求长度 6。
- groups: int 型，表示卷积层的组数。若 ic=oc=groups 时，则卷积为 depthwise conv3d
- input\_scale: List[float] 型或 float 型，表示输入量化参数。取 None 则使用 input Tensor 中的量化参数，取 List 时要求长度为 ic。当前不支持 List[float] 型。
- weight\_scale: List[float] 型或 float 型，表示卷积核量化参数。取 None 则使用 weight Tensor 中的量化参数，取 List 时要求长度为 oc。
- output\_scale: List[float] 型或 float 型，表示卷积核量化参数。不可以取 None，取 List 时要求长度为 oc。当前不支持 List[float] 型。
- input\_zp: List[int] 型或 int 型，表示输入偏移。取 None 则表示 0，取 List 时要求长度为 ic。当前不支持 List[int] 型。
- weight\_zp: List[int] 型或 int 型，表示卷积核偏移。取 None 则表示 0，取 List 时要求长度为 oc。
- output\_zp: List[int] 型或 int 型，表示卷积核偏移。取 None 则表示 0，取 List 时要求长度为 oc。当前不支持 List[int] 型。

- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int8。取值范围: int8/uint8。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入和权重的数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

## matmul

### 接口定义

```
def matmul(input: Tensor,
           right: Tensor,
           bias: Tensor = None,
           right_transpose: bool = False,
           left_transpose: bool = False,
           output_transpose: bool = False,
           keep_dims: bool = True,
           out_dtype: str = None,
           out_name: str = None):
    #pass
```

### 功能描述

矩阵乘运算。可参考各框架下的矩阵乘定义。该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示输入左操作数, 大于或等于 2 维, 设最后两维 shape=[m,k]。
- right: Tensor 类型, 表示输入右操作数, 大于或等于 2 维, 设最后两维 shape=[k,n]。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [n]。
- left\_transpose: bool 型, 默认为 False。表示计算时是否对左矩阵进行转置。

- right\_transpose: bool 型, 默认为 False。表示计算时是否对右矩阵进行转置。
- output\_transpose: bool 型, 默认为 False。表示计算时是否对输出矩阵进行转置。
- keep\_dims: bool 型, 默认为 True。表示结果是否保持原来的 dim, False 则 shape 为 2 维。
- out\_dtype: string 类型或 None, 为 None 时与 input 数据类型一致。取值为范围为 “float32”, “float16”。表示输出 Tensor 的数据类型。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

要求左右 Tensor 的维度长度一致。当 Tensor 的维度长度为 2 时, 表示矩阵和矩阵乘运算。当 Tensor 的维度长度大于 2 时, 表示批矩阵乘运算。要求 `input.shape[-1] == right.shape[-2]`, `input.shape[:-2]` 和 `right.shape[:-2]` 需要满足广播规则。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 right 的数据类型必须一致。bias 的数据类型必须是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。input 与 right,bias 的数据类型必须一致。

## matmul\_int

### 接口定义

```
def matmul_int(input: Tensor,
               right: Tensor,
               bias: Tensor = None,
               right_transpose: bool = False,
               left_transpose: bool = False,
               output_transpose: bool = False,
               keep_dims: bool = True,
               input_zp: Union[int, List[int]] = None,
               right_zp: Union[int, List[int]] = None,
               out_dtype: str = None,
               out_name: str = None):
    #pass
```

## 功能描述

矩阵乘运算。可参考各框架下的矩阵乘定义。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入左操作数, 大于或等于 2 维, 设最后两维 shape=[m,k]。
- right: Tensor 类型, 表示输入右操作数, 大于或等于 2 维, 设最后两维 shape=[k,n]。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [n]。
- left\_transpose: bool 型, 默认为 False。表示计算时是否对左矩阵进行转置。
- right\_transpose: bool 型, 默认为 False。表示计算时是否对右矩阵进行转置。
- output\_transpose: bool 型, 默认为 False。表示计算时是否对输出矩阵进行转置。
- keep\_dims: bool 型, 默认为 True。表示结果是否保持原来的 dim, False 则 shape 为 2 维。
- input\_zp: List[int] 型或 int 型, 表示 input 的偏移。取 None 则表示 0。当前不支持 List[int] 型。
- right\_zp: List[int] 型或 int 型, 表示 right 的偏移。取 None 则表示 0。当前不支持 List[int] 型。
- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int32。取值范围: int32/uint32
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

要求左右 Tensor 的维度长度一致。当 Tensor 的维度长度为 2 时, 表示矩阵和矩阵乘运算。当 Tensor 的维度长度大于 2 时, 表示批矩阵乘运算。要求  $\text{input.shape}[-1] == \text{right.shape}[-2]$ ,  $\text{input.shape}[:-2]$  和  $\text{right.shape}[:-2]$  需要满足广播规则。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 指定。

## 处理器支持

- BM1688: 输入数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

### matmul\_quant

#### 接口定义

```
def matmul_quant(input: Tensor,
                 right: Tensor,
                 bias: Tensor = None,
                 right_transpose: bool = False,
                 keep_dims: bool = True,
                 input_scale: Union[float, List[float]] = None,
                 right_scale: Union[float, List[float]] = None,
                 output_scale: Union[float, List[float]] = None,
                 input_zp: Union[int, List[int]] = None,
                 right_zp: Union[int, List[int]] = None,
                 output_zp: Union[int, List[int]] = None,
                 out_dtype: str = None,
                 out_name: str = None):
    #pass
```

#### 功能描述

量化的矩阵乘运算。可参考各框架下的矩阵乘定义。该操作属于 **本地操作**。

#### 参数说明

- input: Tensor 类型, 表示输入左操作数, 大于或等于 2 维, 设最后两维 shape=[m,k]。
- right: Tensor 类型, 表示输入右操作数, 大于或等于 2 维, 设最后两维 shape=[k,n]。
- bias: Tensor 类型, 表示偏置 Tensor。为 None 时表示无偏置, 反之则要求 shape 为 [n]。
- right\_transpose: bool 型, 默认为 False。表示计算时是否对右矩阵进行转置。
- keep\_dims: bool 型, 默认为 True。表示结果是否保持原来的 dim, False 则 shape 为 2 维。
- input\_scale: List[float] 型或 float 型, 表示 input 的量化参数。取 None 则使用 input Tensor 中的量化参数。当前不支持 List[float] 型。
- right\_scale: List[float] 型 float 型, 表示 right 的量化参数。取 None 则使用 right Tensor 中的量化参数。当前不支持 List[float] 型。

- output\_scale: List[float] 型 float 型, 表示 output 的量化参数。不可以取 None。当前不支持 List[float] 型。
- input\_zp: List[int] 型或 int 型, 表示 input 的偏移。取 None 则表示 0。当前不支持 List[int] 型。
- right\_zp: List[int] 型或 int 型, 表示 right 的偏移。取 None 则表示 0。当前不支持 List[int] 型。
- output\_zp: List[int] 型或 int 型, 表示 output 的偏移。取 None 则表示 0。当前不支持 List[int] 型。
- out\_dtype: string 类型或 None, 表示输入 Tensor 的类型, 取 None 表示为 int8。取值范围: int8/uint8
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

要求左右 Tensor 的维度长度一致。当 Tensor 的维度长度为 2 时, 表示矩阵和矩阵乘运算。当 Tensor 的维度长度大于 2 时, 表示批矩阵乘运算。要求 input.shape[-1] == right.shape[-2], input.shape[:-2] 和 right.shape[:-2] 需要满足广播规则。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 指定。

## 处理器支持

- BM1688: 输入数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。
- BM1684X: 输入数据类型可以是 INT8/UINT8。偏置的数据类型为 INT32。

### 24.5.2 Base Element-wise Operator

#### add

#### 接口定义

```
def add(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素加法运算。 $tensor\_o = tensor\_i0 + tensor\_i1$ 。该操作支持广播。该操作属于 **本地操作**。

## 参数说明

- `tensor_i0`: Tensor 类型或 Scalar、int、float，表示输入左操作 Tensor 或 Scalar。
- `tensor_i1`: Tensor 类型或 Scalar、int、float，表示输入右操作 Tensor 或 Scalar。`tensor_i0` 和 `tensor_i1` 至少有一个是 Tensor。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 `tensor_i0`, `tensor_i1`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 `tensor_i0`, `tensor_i1`, `output` 的 zero\_point。
- `out_dtype`: string 类型或 None，表示输出 Tensor 的数据类型，为 None 时会与输入数据类型一致。可选参数为'float32' /' float16' /' int8' /' uint8'。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型由 `out_dtype` 指定，或与输入数据类型一致（当其中一个输入为' int8' 则输出默认为' int8' 类型）。当输入为' float32' /' float16' 时，输出数据类型必须与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时，`tensor_i0` 与 `tensor_i1` 的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时，`tensor_i0` 与 `tensor_i1` 的数据类型必须一致。

**sub****接口定义**

```
def sub(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

**功能描述**

张量和张量的按元素减法运算。 $tensor_o = tensor_{i0} - tensor_{i1}$ 。该操作支持广播。该操作属于 **本地操作**。

**参数说明**

- tensor\_i0: Tensor 类型或 Scalar、int、float，表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int、float，表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 tensor\_i0, tensor\_i1, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 tensor\_i0, tensor\_i1, output 的 zero\_point。
- out\_dtype: string 类型或 None，表示输出 Tensor 的数据类型，为 None 时会与输入数据类型一致。可选参数为'float32' / 'float16' / 'int8'。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

**返回值**

返回一个 Tensor，该 Tensor 的数据类型由 out\_dtype 指定，或与输入数据类型一致。当输入为'float32' / 'float16' 时，输出数据类型必须与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时, tensor\_i0 与 tensor\_i1 的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时, tensor\_i0 与 tensor\_i1 的数据类型必须一致。

## mul

### 接口定义

```
def mul(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

### 功能描述

张量和张量的按元素乘法运算。 $tensor_o = tensor_i0 * tensor_i1$ 。该操作支持广播。该操作属于 **本地操作**。

### 参数说明

- tensor\_i0: Tensor 类型或 Scalar、int、float, 表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int、float, 表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- scale: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为 tensor\_i0, tensor\_i1, output 的 scale。
- zero\_point: List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为 tensor\_i0, tensor\_i1, output 的 zero\_point。
- out\_dtype: string 类型或 None, 表示输出 Tensor 的数据类型, 为 None 时会与输入数据类型一致。可选参数为' float32' /' float16' /' int8' /' uint8'。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型由 `out_dtype` 指定，或与输入数据类型一致（当其中一个输入为' int8' 则输出默认为' int8' 类型）。当输入为' float32' /' float16' 时，输出数据类型必须与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时，`tensor_i0` 与 `tensor_i1` 的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。当数据类型为 FLOAT16/FLOAT32 时，`tensor_i0` 与 `tensor_i1` 的数据类型必须一致。

`div`

## 接口定义

```
def div(tensor_i0: Union[Tensor, Scalar],  
        tensor_i1: Union[Tensor, Scalar],  
        out_name: str = None):  
    #pass
```

## 功能描述

张量和张量的按元素除法运算。 $tensor_o = tensor_{i0}/tensor_{i1}$ 。该操作支持广播。该操作属于 **本地操作**。

## 参数说明

- `tensor_i0`: Tensor 类型或 Scalar、int、float，表示输入左操作 Tensor 或 Scalar。
- `tensor_i1`: Tensor 类型或 Scalar、int、float，表示输入右操作 Tensor 或 Scalar。`tensor_i0` 和 `tensor_i1` 至少有一个是 Tensor。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32。

### max

#### 接口定义

```
def max(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

#### 功能描述

张量和张量的按元素取最大值。 $tensor_o = \max(tensor_{i0}, tensor_{i1})$ 。该操作支持广播。该操作属于 **本地操作**。

#### 参数说明

- tensor\_i0: Tensor 类型或 Scalar、int、float，表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int、float，表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 tensor\_i0, tensor\_i1, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 tensor\_i0, tensor\_i1, output 的 zero\_point。
- out\_dtype: string 类型或 None，表示输出 Tensor 的数据类型，为 None 时会与输入数据类型一致。可选参数为' float32' /' float16' /' int8' /' uint8' /' int16' /' uint16' /' int32' /' uint32'。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型由 `out_dtype` 指定，或与输入数据类型一致。当数据类型为 FLOAT16/FLOAT32 时，`tensor_i0` 与 `tensor_i1` 的数据类型必须一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT16/UINT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT16/UINT16/INT8/UINT8。

## min

### 接口定义

```
def min(tensor_i0: Union[Tensor, Scalar, int, float],
        tensor_i1: Union[Tensor, Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_dtype: str = None,
        out_name: str = None):
    #pass
```

### 功能描述

张量和张量的按元素取最小值。 $tensor_o = \min(tensor_i0, tensor_i1)$ 。该操作支持广播。该操作属于 **本地操作**。

### 参数说明

- `tensor_i0`: Tensor 类型或 Scalar、int、float，表示输入左操作 Tensor 或 Scalar。
- `tensor_i1`: Tensor 类型或 Scalar、int、float，表示输入右操作 Tensor 或 Scalar。`tensor_i0` 和 `tensor_i1` 至少有一个是 Tensor。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 `tensor_i0`, `tensor_i1`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 3，分别为 `tensor_i0`, `tensor_i1`, `output` 的 zero\_point。
- `out_dtype`: string 类型或 None，表示输出 Tensor 的数据类型，为 None 时会与输入数据类型一致。可选参数为 'float32' / 'float16' / 'int8' / 'uint8' / 'int16' / 'uint16' / 'int32' / 'uint32'。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型由 out\_dtype 指定, 或与输入数据类型一致。当数据类型为 FLOAT16/FLOAT32 时, tensor\_i0 与 tensor\_i1 的数据类型必须一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT16/UINT16/INT32/UINT32/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT16/UINT16/INT32/UINT32/INT8/UINT8。

## add\_shift

### 接口定义

```
def add_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
              is_saturate: bool=True,
              out_name: str = None):
    #pass
```

### 功能描述

运算公式  $tensor_o = (tensor_{i0} + tensor_{i1}) << shift$ 。张量和张量的按元素相加后再舍入算术移 shift 位, shift 为正时, 左移, shift 为负时, 右移。舍入模式由 round\_mode 确定。add\_shift 数据相加后, 以 INT64 为中间结果保存, 然后在 INT64 基础上做一次舍入的算数移位操作; 结果支持饱和处理; 当 tensor\_i0、tensor\_i1 为 signed, 且 tensor\_o 为 unsigned 时, 结果必须饱和处理。该操作支持广播。该操作属于 **本地操作**。

## 参数说明

- tensor\_i0: Tensor 类型或 Scalar、int，表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int，表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- shift: int 型，表示移位的位数。
- round\_mode: String 型，表示舍入模式。默认值为'half\_away\_from\_zero'。取值范围为“half\_away\_from\_zero”，“half\_to\_even”，“towards\_zero”，“down”，“up”。
- is\_saturate: Bool 型，表示结果是否需要饱和处理，默认饱和处理。
- out\_dtype: String 或 None，表示输出 Tensor 的数据类型，取默认值时则和 tensor\_i0 的类型一致。可选参数为'int8' /' uint8' /' int16' /' uint16' /' int32' /' uint32'。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 指定，或与输入数据类型一致。

## 处理器支持

- BM1688: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。
- BM1684X: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。

## sub\_shift

### 接口定义

```
def sub_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
              is_saturate: bool=True,
              out_name: str = None):
    #pass
```

## 功能描述

运算公式  $tensor\_o = (tensor\_i0 - tensor\_i1) << shift$ 。张量和张量的按元素相减后再舍入算术移 shift 位，shift 为正时，左移，shift 为负时，右移。舍入模式由 round\_mode 确定。sub\_shift 数据相减后，以 INT64 为中间结果保存，然后在 INT64 基础上做一次舍入的算数移位操作；结果支持饱和处理。该操作支持广播。该操作属于 **本地操作**。

## 参数说明

- tensor\_i0: Tensor 类型或 Scalar、int，表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int，表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- shift: int 型，表示移位的位数。
- round\_mode: String 型，表示舍入模式。默认值为 'half\_away\_from\_zero'。取值范围为 "half\_away\_from\_zero", "half\_to\_even", "towards\_zero", "down", "up"。
- is\_saturate: Bool 型，表示结果是否需要饱和处理，默认饱和处理。
- out\_dtype: String 或 None，表示输出 Tensor 的数据类型，取默认值时则和 tensor\_i0 的类型一致。可选参数为 'int8' / 'int16' / 'int32'。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 指定，或与输入数据类型一致。

## 处理器支持

- BM1688: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。
- BM1684X: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。

## mul\_shift

### 接口定义

```
def mul_shift(tensor_i0: Union[Tensor, Scalar, int],
              tensor_i1: Union[Tensor, Scalar, int],
              shift: int,
              out_dtype: str,
              round_mode: str='half_away_from_zero',
```

(续下页)

(接上页)

```
is_saturate: bool=True,
out_name: str = None):
#pass
```

## 功能描述

运算公式  $tensor\_o = (tensor\_i0 * tensor\_i1) << shift$ 。张量和张量的按元素相减后再舍入算术移 shift 位，shift 为正时，左移，shift 为负时，右移。舍入模式由 round\_mode 确定。mul\_shift 数据相乘后，以 INT64 为中间结果保存，然后在 INT64 基础上做一次舍入的算数移位操作；结果支持饱和处理；当 tensor\_i0、tensor\_i1 为 signed，且 tensor\_o 为 unsigned 时，结果必须饱和处理。该操作支持广播。该操作属于 **本地操作**。

## 参数说明

- tensor\_i0: Tensor 类型或 Scalar、int，表示输入左操作 Tensor 或 Scalar。
- tensor\_i1: Tensor 类型或 Scalar、int，表示输入右操作 Tensor 或 Scalar。tensor\_i0 和 tensor\_i1 至少有一个是 Tensor。
- shift: int 型，表示移位的位数。
- round\_mode: String 型，表示舍入模式。默认值为 ‘half\_away\_from\_zero’。取值范围为 “half\_away\_from\_zero”，“half\_to\_even”，“towards\_zero”，“down”，“up”。
- is\_saturate: Bool 型，表示结果是否需要饱和处理，默认饱和处理。
- out\_dtype: String 或 None，表示输出 Tensor 的数据类型，取默认值时则和 tensor\_i0 的类型一致。可选参数为 ‘int8’ / ‘uint8’ / ‘int16’ / ‘uint16’ / ‘int32’ / ‘uint32’。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 指定，或与输入数据类型一致。

## 处理器支持

- BM1688: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。
- BM1684X: 输入数据类型可以是 INT32/UINT32/INT16/UINT6/INT8/UINT8。

## copy

### 接口定义

```
def copy(input: Tensor, out_name: str = None):
    #pass
```

### 功能描述

copy，将输入数据复制到输出 Tensor 中。该操作属于 全局操作。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X：输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## clamp

### 接口定义

```
def clamp(input: Tensor, min: float, max: float, out_name: str = None):
    #pass
```

## 功能描述

将输入 Tensor 中所有元素的值都限定在设置的最大最小值范围内，大于最大值则截断为最大值，小于最大值则截断为最小值。要求所有输入 Tensor 及 Scalar 的 dtype 一致。该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor。
- min: float 类型，表示阶段的下限。
- max: float 类型，表示阶段的上限。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

### 24.5.3 Element-wise Compare Operator

gt

## 接口定义

```
def gt(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素大于比较运算。 $tensor\_o = tensor\_i0 > tensor\_i1 ? 1 : 0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于 **本地操作**。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

It

## 接口定义

```
def lt(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素小于比较运算。 $tensor\_o = tensor\_i0 < tensor\_i1 ? 1 : 0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于 **本地操作**。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

ge

## 接口定义

```
def ge(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素大于等于比较运算。 $tensor\_o = tensor\_i0 >= tensor\_i1 ? 1 : 0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于本地操作。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

le

## 接口定义

```
def le(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素小于等于比较运算。 $tensor\_o = tensor\_i0 <= tensor\_i1 ? 1 : 0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于本地操作。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

`eq`

## 接口定义

```
def eq(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素等于比较运算。 $tensor\_o = tensor\_i0 == tensor\_i1?1:0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于 **本地操作**。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

ne

## 接口定义

```
def ne(tensor_i0: Tensor,
       tensor_i1: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和张量的按元素不等于比较运算。 $tensor\_o = tensor\_i0! = tensor\_i1?1 : 0$ 。该操作支持广播。 $tensor\_i0$  或者  $tensor\_i1$  可以被指定为 COEFF\_TENSOR。该操作属于 **本地操作**。

## 参数说明

- $tensor\_i0$ : Tensor 类型, 表示输入左操作 Tensor。
- $tensor\_i1$ : Tensor 类型, 表示输入右操作 Tensor。
- $scale$ : List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 scale。 $tensor\_i0$  与  $tensor\_i1$  的 scale 必须一致。
- $zero\_point$ : List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 3, 分别为  $tensor\_i0$ ,  $tensor\_i1$ , output 的 zero\_point。 $tensor\_i0$  与  $tensor\_i1$  的 zero\_point 必须一致。
- $out\_name$ : string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。 $tensor\_i0$  与  $tensor\_i1$  的数据类型必须一致。

gts

## 接口定义

```
def gts(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素大于比较运算。 $tensor\_o = tensor\_i0 > scalar\_i1?1:0$ 。该操作属于本地操作。

## 参数说明

- `tensor_i0`: Tensor 类型，表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型，表示输入右操作数。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

`lts`

## 接口定义

```
def lts(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素小于比较运算。 $tensor\_o = tensor\_i0 < scalar\_i1?1:0$ 。该操作属于本地操作。

## 参数说明

- `tensor_i0`: Tensor 类型，表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型，表示输入右操作数。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

`ges`

## 接口定义

```
def ges(tensor_i0: Tensor,
       scalar_i1: Union[Scalar, int, float],
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素大于等于比较运算。 $tensor\_o = tensor\_i0 >= scalar\_i1 ? 1 : 0$ 。该操作属于 **本地操作**。

## 参数说明

- `tensor_i0`: Tensor 类型，表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型，表示输入右操作数。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

`les`

## 接口定义

```
def les(tensor_i0: Tensor,
       scalar_i1: Union[Scalar, int, float],
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素小于等于比较运算。 $tensor\_o = tensor\_i0 \leq scalar\_i1 ? 1 : 0$ 。该操作属于 **本地操作**。

## 参数说明

- `tensor_i0`: Tensor 类型, 表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型, 表示输入右操作数。
- `scale`: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

`eqs`

## 接口定义

```
def eqs(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素等于比较运算。 $tensor\_o = tensor\_i0 == scalar\_i1 ? 1 : 0$ 。该操作属于本地操作。

## 参数说明

- `tensor_i0`: Tensor 类型，表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型，表示输入右操作数。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

`nes`

## 接口定义

```
def nes(tensor_i0: Tensor,
        scalar_i1: Union[Scalar, int, float],
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

## 功能描述

张量和标量的按元素不等于比较运算。 $tensor\_o = tensor\_i0! = scalar\_i1?1 : 0$ 。该操作属于 **本地操作**。

## 参数说明

- `tensor_i0`: Tensor 类型, 表示输入左操作数。
- `scalar_i1`: Scalar, int 或 float 类型, 表示输入右操作数。
- `scale`: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。scalar\_i1 数据类型为 FLOAT32。

### 24.5.4 Activation Operator

`relu`

## 接口定义

```
def relu(input: Tensor, out_name: str = None):
    #pass
```

## 功能描述

relu 激活函数，逐元素实现功能  $y = \max(0, x)$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。若输入是 quantized 类型，输出的 scale 与 zero\_point 与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## prelu

## 接口定义

```
def prelu(input: Tensor, slope : Tensor, out_name: str = None):  
    #pass
```

## 功能描述

prelu 激活函数，逐元素实现功能  $y = \begin{cases} x & x > 0 \\ x * slope & x \leq 0 \end{cases}$ 。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入 Tensor。
- slope: Tensor 类型, 表示 slope Tensor。仅支持 slope 为 coeff Tensor。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## leaky\_relu

### 接口定义

```
def leaky_relu(input: Tensor,
               negative_slope: float = 0.01,
               out_name: str = None,
               round_mode : str="half_away_from_zero",):
    #pass
```

### 功能描述

leaky\_relu 激活函数, 逐元素实现功能  $y = \begin{cases} x & x > 0 \\ x * params[0] & x \leq 0 \end{cases}$ 。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入 Tensor。
- negative\_slope: float 类型, 表示输入的负斜率, 默认值为 0.01。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: string 型, 表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

## 返回值

返回一个 Tensor, 该 Tensor 的形状和数据类型与输入 Tensor 相同。若输入是 quantized 类型, 输出的 scale 与 zero\_point 与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## abs

### 接口定义

```
def abs(input: Tensor, out_name: str = None):  
    #pass
```

### 功能描述

abs 绝对值激活函数, 逐元素实现功能  $y = |x|$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型, 表示输入 Tensor。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。若输入是 quantized 类型，输出的 scale 与 zero\_point 与输入一致。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X：输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## ln

### 接口定义

```
def ln(input: Tensor,  
       scale: List[float]=None,  
       zero_point: List[int]=None,  
       out_name: str = None):  
    #pass
```

### 功能描述

ln 激活函数，逐元素实现功能  $y = \log(x)$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## ceil

### 接口定义

```
def ceil(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

### 功能描述

ceil 向上取整激活函数，逐元素实现功能  $y = \lceil x \rceil$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## floor

### 接口定义

```
def floor(input: Tensor,  
         scale: List[float]=None,  
         zero_point: List[int]=None,  
         out_name: str = None):  
    #pass
```

### 功能描述

floor 向下取整激活函数，逐元素实现功能  $y = \lceil x \rceil$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## round

### 接口定义

```
def round(input: Tensor, out_name: str = None):
    #pass
```

### 功能描述

round 四舍五入整激活函数，逐元素实现功能  $y = \text{round}(x)$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## sin

### 接口定义

```
def sin(input: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

### 功能描述

sin 正弦激活函数，逐元素实现功能  $y = \sin(x)$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

**COS****接口定义**

```
def cos(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

**功能描述**

cos 余弦激活函数，逐元素实现功能  $y = \cos(x)$ 。该操作属于 **本地操作**。

**参数说明**

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

**返回值**

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

**处理器支持**

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

exp

## 接口定义

```
def exp(input: Tensor,  
        scale: List[float]=None,  
        zero_point: List[int]=None,  
        out_name: str = None):  
    #pass
```

## 功能描述

exp 指数激活函数，逐元素实现功能  $y = e^x$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## tanh

### 接口定义

```
def tanh(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None,
         round_mode : str="half_away_from_zero"):
    #pass
```

### 功能描述

tanh 双曲正切激活函数，逐元素实现功能  $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- round\_mode: string 型，表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## sigmoid

### 接口定义

```
def sigmoid(input: Tensor,
            scale: List[float]=None,
            zero_point: List[int]=None,
            out_name: str = None,
            round_mode : str="half_away_from_zero"):
    #pass
```

### 功能描述

sigmoid 激活函数，逐元素实现功能  $y = 1/(1 + e^{-x})$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- round\_mode: string 型，表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## log\_sigmoid

### 接口定义

```
def log_sigmoid(input: Tensor,
                 scale: List[float]=None,
                 zero_point: List[int]=None,
                 out_name: str = None):
    #pass
```

### 功能描述

log\_sigmoid 激活函数，逐元素实现功能  $y = \log(1/(1 + e^{-x}))$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## elu

### 接口定义

```
def elu(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### 功能描述

elu 激活函数，逐元素实现功能  $y = \begin{cases} x & x \geq 0 \\ e^x - 1 & x < 0 \end{cases}$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## square

### 接口定义

```
def square(input: Tensor,
           scale: List[float]=None,
           zero_point: List[int]=None,
           out_name: str = None):
    #pass
```

### 功能描述

square 平方激活函数，逐元素实现功能  $y = \square x$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## sqrt

### 接口定义

```
def sqrt(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

### 功能描述

`sqrt` 平方根激活函数，逐元素实现功能  $y = \sqrt{x}$ 。该操作属于 **本地操作**。

### 参数说明

- `tensor`: `Tensor` 类型，表示输入 `Tensor`。
- `scale`: `List[float]` 类型或 `None`，量化参数。取 `None` 代表非量化计算。若为 `List`，长度为 2，分别为 `tensor_i0`, `output` 的 `scale`。
- `zero_point`: `List[int]` 类型或 `None`，量化参数。取 `None` 代表非量化计算。若为 `List`，长度为 2，分别为 `tensor_i0`, `output` 的 `zero_point`。
- `out_name`: `string` 类型或 `None`，表示输出 `Tensor` 的名称，为 `None` 时内部会自动产生名称。

### 返回值

返回一个 `Tensor`，该 `Tensor` 的形状和数据类型与输入 `Tensor` 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 `FLOAT32/INT8/UINT8`。`FLOAT16` 数据会自动转换为 `FLOAT32`。
- BM1684X: 输入数据类型可以是 `FLOAT32/INT8/UINT8`。`FLOAT16` 数据会自动转换为 `FLOAT32`。

## rsqrt

### 接口定义

```
def rsqrt(input: Tensor,
          scale: List[float]=None,
          zero_point: List[int]=None,
          out_name: str = None):
    #pass
```

### 功能描述

rsqrt 平方根取反激活函数，逐元素实现功能  $y = 1/(sqrt{x})$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## silu

### 接口定义

```
def silu(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### 功能描述

silu 激活函数，逐元素实现功能  $y = x * (1/(1 + e^{-x}))$ 。该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## swish

### 接口定义

```
def swish(input: Tensor,
          beta: float,
          scale: List[float]=None,
          zero_point: List[int]=None,
          round_mode: str = "half_away_from_zero",
          out_name: str = None):
    #pass
```

### 功能描述

swish 激活函数，逐元素实现功能  $y = x * (1/(1 + e^{-x*beta}))$ 。该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型，表示输入 Tensor。
- beta: Scalar 或 float 类型，表示 beta 值。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- round\_mode: string 型，表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

`erf`

## 接口定义

```
def erf(input: Tensor,
       scale: List[float]=None,
       zero_point: List[int]=None,
       out_name: str = None):
    #pass
```

## 功能描述

`erf` 激活函数，对于输入输出 Tensor 对应位置的元素  $x$  和  $y$ ，逐元素实现功能  $y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\eta^2} d\eta$ 。该操作属于 **本地操作**。

## 参数说明

- `tensor`: Tensor 类型，表示输入 Tensor。
- `scale`: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 scale。
- `zero_point`: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 `tensor_i0`, `output` 的 zero\_point。
- `out_name`: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

**tan**

## 接口定义

```
def tan(input: Tensor, out_name: str = None):  
    #pass
```

## 功能描述

tan 正切激活函数，逐元素实现功能  $y = \tan(x)$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。

## softmax

### 接口定义

```
def softmax(input: Tensor,
            axis: int,
            out_name: str = None):
    #pass
```

### 功能描述

softmax 激活函数，实现功能  $tensor\_o = \exp(tensor\_i) / \sum(\exp(tensor\_i), axis)$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- axis: int 型，表示进行运算的轴。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## softmax\_int

### 接口定义

```
def softmax_int(input: Tensor,
                axis: int,
                scale: List[float],
                zero_point: List[int] = None,
                out_name: str = None,
```

(续下页)

(接上页)

```
round_mode : str="half_away_from_zero"):
#pass
```

## 功能描述

softmax 定点运算。可参考各框架下的 softmax 定义。

```
for i in range(256)
    table[i] = exp(scale[0] * i)

for n,h,w in N,H,W
    max_val = max(input[n,c,h,w] for c in C)
    sum_exp = sum(table[max_val - input[n,c,h,w]] for c in C)
    for c in C
        prob = table[max_val - input[n,c,h,w]] / sum_exp
        output[n,c,h,w] = saturate(int(round(prob * scale[1])) + zero_point[1]), F
    ↵ 其中saturate饱和到output数据类型
```

其中 table 表示查表。

## 参数说明

- tensor: Tensor 类型, 表示输入 Tensor。
- axis: int 型, 表示进行运算的轴。
- scale: List[float] 型, 表示输入和输出的量化系数。长度必须时 2。
- zero\_point: List[int] 型或 None 型, 表示输入和输出偏移, 长度与 scale 一致。如果为 None, 则取 [0, 0]。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: string 型, 表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

## 返回值

返回一个 Tensor, 该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 INT8/UINT8。
- BM1684X: 输入数据类型可以是 INT8/UINT8。

## mish

### 接口定义

```
def mish(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

### 功能描述

mish 激活函数，逐元素实现功能  $y = x * \tanh(\ln(1 + e^x))$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## hswish

### 接口定义

```
def hswish(input: Tensor,
           scale: List[float]=None,
           zero_point: List[int]=None,
           out_name: str = None):
    #pass
```

### 功能描述

hswish 激活函数，逐元素实现功能  $y = \begin{cases} 0 & x \leq -3 \\ x & x \geq 3 \\ x * ((x + 3)/6) & -3 < x < 3 \end{cases}$ 。该操作属于 **本地** 操作。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

arccos

## 接口定义

```
def arccos(input: Tensor, out_name: str = None):
    #pass
```

## 功能描述

arccos 反余弦激活函数，逐元素实现功能  $y = \arccos(x)$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。

## arctanh

### 接口定义

```
def arctanh(input: Tensor, out_name: str = None):
    #pass
```

### 功能描述

arctanh 反双曲正切激活函数，逐元素实现功能  $y = \operatorname{arctanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$ 。该操作属于 **本地操作**。

### 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。FLOAT16 数据会自动转换为 FLOAT32。

## sinh

### 接口定义

```
def sinh(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

## 功能描述

$\sinh$  双曲正弦激活函数，逐元素实现功能  $y = \sinh(x) = \frac{e^x - e^{-x}}{2}$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## cosh

## 接口定义

```
def cosh(input: Tensor,
         scale: List[float]=None,
         zero_point: List[int]=None,
         out_name: str = None):
    #pass
```

## 功能描述

cosh 双曲余弦激活函数，逐元素实现功能  $y = \cosh(x) = \frac{e^x + e^{-x}}{2}$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## sign

## 接口定义

```
def sign(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None):
    #pass
```

## 功能描述

sign 激活函数，逐元素实现功能  $y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## gelu

## 接口定义

```
def gelu(input: Tensor,
        scale: List[float]=None,
        zero_point: List[int]=None,
        out_name: str = None,
        round_mode : str="half_away_from_zero"):
    #pass
```

## 功能描述

gelu 激活函数，逐元素实现功能  $y = x * 0.5 * (1 + erf(\frac{x}{\sqrt{2}}))$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- round\_mode: string 型，表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X：输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

## hsigmoid

### 接口定义

```
def hsigmoid(input: Tensor,
             scale: List[float]=None,
             zero_point: List[int]=None,
             out_name: str = None):
    #pass
```

## 功能描述

hsigmoid 激活函数，逐元素实现功能  $y = \min(1, \max(0, \frac{x}{6} + 0.5))$ 。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型，表示输入 Tensor。
- scale: List[float] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 scale。
- zero\_point: List[int] 类型或 None，量化参数。取 None 代表非量化计算。若为 List，长度为 2，分别为 tensor\_i0, output 的 zero\_point。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的形状和数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32/INT8/UINT8。FLOAT16 数据会自动转换为 FLOAT32。

### 24.5.5 Data Arrange Operator

#### permute

## 接口定义

```
def permute(input:tensor,
           order:Union[List[int], Tuple[int]],
           out_name:str=None):
    #pass
```

## 功能描述

根据置换参数对输入 Tensor 进行重排。例如：输入 shape 为 (6, 7, 8, 9)，置换参数 order 为 (1, 3, 2, 0)，则输出的 shape 为 (7, 9, 8, 6)。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型，表示输入操作 Tensor。
- order: List[int] 或 Tuple[int] 型，表示置换参数。要求 order 长度和 tensor 维度一致。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

tile

## 接口定义

```
def tile(tensor_i: Tensor,  
        reps: Union[List[int], Tuple[int]],  
        out_name: str = None):  
    #pass
```

## 功能描述

在指定的维度重复复制数据。该操作属于 **受限本地操作**。

## 参数说明

- tensor\_i: Tensor 类型, 表示输入操作 Tensor。
- reps: List[int] 或 Tuple[int] 型, 表示每个维度的复制份数。要求 order 长度和 tensor 维度一致。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

## broadcast

## 接口定义

```
def broadcast(input: Tensor,
             reps: Union[List[int], Tuple[int]],
             out_name: str = None):
    #pass
```

## 功能描述

在指定的维度重复复制数据。该操作属于 **受限本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。
- reps: List[int] 或 Tuple[int] 型, 表示每个维度的复制份数。要求 order 长度和 tensor 维度一致。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8/INT16/UINT16。

### concat

#### 接口定义

```
def concat(inputs: List[Tensor],
           scales: Optional[Union[List[float], List[int]]] = None, zero_points:
           Optional[List[int]] = None, axis: int = 0, out_name: str = None,
           dtype="float32", round_mode: str="half_away_from_zero"):

    #pass
```

#### 功能描述

对多个张量在指定的轴上进行拼接，以及支持不同量纲输入、输出。

该操作属于 **受限本地操作**。

#### 参数说明

- inputs: List[Tensor] 类型，存放多个 Tensor，所有的 Tensor 要求数据格式一致并具有相同的 shape 维度数，且除了待拼接的那一维，shape 其他维度的值应该相等。
- scales: Optional[Union[List[float], List[int]]] 类型，存放多个输入和一个输出 scale，最后一个为输出的 scale。
- zero\_points: Optional[List[int]] 类型，存放多个输入和一个输出的 zero\_point，最后一个为输出的 zero\_point。
- axis: int 型，表示进行拼接运算的轴。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。
- dtype: string 类型，默认是“float32”。

- round\_mode: string 型, 表示舍入模式。默认为 “half\_away\_from\_zero”。round\_mode 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

## split

### 接口定义

```
def split(input:tensor,
          axis:int=0,
          num:int=1,
          size:Union[List[int], Tuple[int]]=None,
          out_name:str=None):
    #pass
```

### 功能描述

对输入 Tensor 在指定的轴上拆成多个 Tensor。如果 size 不为空, 则由分裂后的大小由 size 决定, 反之则会根据 tensor 尺寸和 num 计算平均分裂后的大小。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示将要进行切分的 Tensor。
- axis: int 型, 表示进行切分运算的轴。
- num: int 型, 表示切分的份数;
- size: List[int] 或 Tuple[int] 型, 非平均分裂时, 指定每一份大小, 平均分裂时, 设置为空即可。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 List[Tensor]，其中每个 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

## pad

### 接口定义

```
def pad(input:tensor,
       method='constant',
       value:Union[Scalar, Variable, None]=None,
       padding:Union[List[int], Tuple[int], None]=None,
       out_name:str=None):
    #pass
```

### 功能描述

对输入 Tensor 进行填充。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型，表示将要进行填充的 Tensor。
- method: string 类型，表示填充方法，可选方法” constant”, ” reflect”, ” symmetric”, ” edge”。
- value: Saclar 或 Variable 型或 None，表示待填充的数值。数据类型和 tensor 一致；
- padding: List[int] 或 Tuple[int] 型或 None。padding 为 None 时使用一个长度为  $2*\text{len}(\text{tensor.shape})$  的全 0 list。例如，一个 hw 的二维 Tensor 对应的 padding 是 [h\_top, w\_left, h\_bottom, w\_right]。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

## repeat

### 接口定义

```
def repeat(tensor_i:Tensor,  
          reps:Union[List[int], Tuple[int]],  
          out_name:str=None):  
    #pass
```

### 功能描述

在指定的维度重复复制数据。功能同 tile。该操作属于 受限本地操作。

### 参数说明

- tensor\_i: Tensor 类型，表示输入操作 Tensor。
- reps: List[int] 或 Tuple[int] 型，表示每个维度的复制份数。要求 order 长度和 tensor 维度一致。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

### extract

#### 接口定义

```
def extract(input: Tensor,  
           start: Union[List[int], Tuple[int]] = None,  
           end: Union[List[int], Tuple[int]] = None,  
           stride: Union[List[int], Tuple[int]] = None,  
           out_name: str = None)
```

#### 功能描述

对输入 tensor 进行切片提取操作。

#### 参数说明

- input: Tensor 类型, 表示输入张量。
- start: 整数的列表或者元组或 None, 表示切片的起始位置, 为 None 时表示全为 0。
- end: 整数的列表或者元组或 None, 表示切片的终止位置, 为 None 时表示输出张量的形状。
- stride: 整数的列表或者元组或 None, 表示切片的步长, 为 None 时表示全为 1。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

#### 返回值

返回一个 Tensor, 数据类型与输入 Tensor 的数据类型相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8/INT16/UINT16。

`roll`

## 接口定义

```
def roll(input:Tensor,
        shifts: Union[int, List[int], Tuple[int]],
        dims: Union[int, List[int], Tuple[int]] = None,
        out_name:str=None):
    #pass
```

## 功能描述

沿给定维度滚动输入张量。移出最后一个位置的元素将在第一个位置重新引入。如果 `dims` 为 `None`，则张量将在滚动之前展平，然后恢复到原始形状。该操作属于 **本地操作**。

## 参数说明

- `input`: Tensor 类型，表示输入操作 Tensor。
- `shifts`: int, List[int] 或 Tuple[int] 型，张量元素移动的位数。如果 `shifts` 是元组/列表，则 `dims` 必须是相同大小的元组/列表，并且每个维度将按相应的值滚动。
- `dims`: int, List[int], Tuple[int] 型或 `None`, 滚动的轴。
- `out_name`: string 类型或 `None`，表示输出 Tensor 的名称，为 `None` 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8/INT16/UINT16。

### 24.5.6 Sort Operator

`arg`

#### 接口定义

```
def arg(input: Tensor,
        method: str = "max",
        axis: int = 0,
        keep_dims: bool = True,
        out_name: str = None):
    #pass
```

#### 功能描述

对输入 tensor 的指定的 axis 求最大或最小值，输出对应的 index，并将该 axis 的 dim 设置为 1。该操作属于 **受限本地操作**。

#### 参数说明

- input: Tensor 类型，表示输入的操作 Tensor。
- method: string 类型，表示操作的方法，可选' max'，' min'。
- axis: int 型，表示指定的轴。默认值为 0。
- keep\_dims: bool 型，表示是否保留运算后的指定轴，默认值为 True 表示保留（此时该轴长度为 1）。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回两个 Tensor，第一个 Tensor 表示 indices，类型为 int32；第二个 Tensor 表示 values，类型会和 input 的类型一致。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32。
- BM1684X：输入数据类型可以是 FLOAT32。

## topk

### 接口定义

```
def topk(input: Tensor,  
         axis: int,  
         k: int,  
         out_name: str = None):
```

### 功能描述

按某个轴排序后前 K 个数。

### 参数说明

- input：Tensor 类型，表示输入 Tensor。
- axis：int 型，表示排序所使用的轴。
- k：int 型，表示沿着轴排序靠前的数的个数。
- out\_name：string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回两个 Tensor，第一个 Tensor 表示前几个数，其数据类型与输入类型相同，第二个 Tensor 表示前几个数在输入中的索引。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。

**sort**

## 接口定义

```
def sort(input: Tensor,
         axis: int = -1,
         descending : bool = True,
         out_name = None)
```

## 功能描述

沿某个轴的输入张量进行排序，输出排序后的张量以及该张量的数据在输入张量中的索引。

## 参数说明

- input: Tensor 类型，表示输入张量。
- axis: int 类型，表示指定的轴。(暂时只支持 axis== -1)
- descending: bool 类型，表示是否按从大到小排列。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回两个 Tensor，第一个张量的数据类型与输入张量的数据类型相同，第二个张量的数据类型为 INT32。

## 处理器支持

- BM1688: 输入张量的数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入张量的数据类型可以是 FLOAT32/FLOAT16。

**argsort****接口定义**

```
def argsort(input: Tensor,
            axis: int = -1,
            descending : bool = True,
            out_name : str = None)
```

**功能描述**

沿某个轴的输入张量进行排序，输出排序后的张量的数据在输入张量中的索引。

**参数说明**

- input: Tensor 类型，表示输入张量。
- axis: int 类型，表示指定的轴。(暂时只支持 axis== -1)
- descending: bool 类型，表示是否按从大到小排列。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

**返回值**

返回一个 Tensor，其数据类型为 INT32。

**处理器支持**

- BM1688: 输入张量的数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入张量的数据类型可以是 FLOAT32/FLOAT16。

**sort\_by\_key****接口定义**

```
def sort_by_key(input: Tensor,
                key: Tensor,
                axis: int = -1,
                descending : bool = True,
                out_name = None)
```

## 功能描述

沿某个轴按键对输入张量进行排序，输出排序后的张量以及相应的键。

## 参数说明

- input: Tensor 类型，表示输入。
- key: Tensor 类型，表示键。
- axis: int 类型，表示指定的轴。
- descending: bool 类型，表示是否按从大到小排列。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回两个 Tensor，第一个张量的数据类型与输入的数据类型相同，第二个张量的数据类型与键的数据类型相同。

## 处理器支持

- BM1688: 输入和键的数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入和键的数据类型可以是 FLOAT32/FLOAT16。

### 24.5.7 Shape About Operator

#### squeeze

## 接口定义

```
def squeeze(tensor_i: Tensor, axis: Union[Tuple[int], List[int]], out_name: str = None):  
    #pass
```

## 功能描述

降维操作，去掉输入 shape 指定的某些 1 维的轴，如果没有指定轴 (axis) 则去除所有是 1 维的轴。该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入操作 Tensor。
- axis: List[int] 或 Tuple[int] 型，表示指定的轴。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## reshape

## 接口定义

```
def reshape(tensor: Tensor, new_shape: Union[Tuple[int], List[int], Tensor], out_
           ↵name: str = None):
    #pass
```

## 功能描述

对输入 tensor 做 reshape 的操作。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型, 表示输入操作 Tensor。
- new\_shape: List[int] 或 Tuple[int] 或 Tensor 类型, 表示转化后的形状。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## shape\_fetch

### 接口定义

```
def shape_fetch(tensor_i: Tensor,
               begin_axis: int = None,
               end_axis: int = None,
               step: int = 1,
               out_name: str = None):
    #pass
```

### 功能描述

对输入 tensor 取指定轴 (axis) 之间的 shape 信息。该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型, 表示输入操作 Tensor。
- begin\_axis: int 型, 表示指定开始的轴。
- end\_axis: int 型, 表示指定结束的轴。
- step: int 型, 表示步长。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor, 该 Tensor 的数据类型为 INT32。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

### unsqueeze

### 接口定义

```
def unsqueeze(input: Tensor, axes: List[int] = [1,2], out_name: str = None):
    #pass
```

### 功能描述

增维操作。在 axis 指定的位置增加 1。该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。
- axis: int 型, 表示指定的轴, 设 tensor\_i 的维度长度是 D, 则 axis 范围 [-D,D-1)。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

### 24.5.8 Quant Operator

`requant_fp_to_int`

#### 接口定义

```
def requant_fp_to_int(tensor_i,
                      scale,
                      offset,
                      requant_mode, #unused
                      out_dtype,
                      out_name = None,
                      round_mode='half_away_from_zero'):
```

#### 功能描述

对输入 tensor 进行量化处理。

该操作对应的计算式为

```
output = saturate(int(round(input * scale)) + offset),  
其中saturate为饱和到output的数据类型
```

该操作属于 **本地操作**。

#### 参数说明

- `tensor_i`: Tensor 类型, 表示输入 Tensor, 3-5 维。
- `scale`: List[float] 型或 float 型, 表示量化系数。
- `offset`: List[int] 型或 int 型; 表示输出偏移。
- `requant_mode`: int 型, 表示量化模式。废弃。
- `round_mode`: string 型, 表示舍入模式。默认为 “half\_away\_from\_zero”。`round_mode` 取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。(TODO)
- `out_dtype`: string 类型, 表示输入 Tensor 的类型. 数据类型可以是” int16 ” /” uint16 ” /” int8 ” /” uint8 ”。

- `out_name`: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor。该 Tensor 的数据类型由 `out_dtype` 确定。

### 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。

### `requant_fp`

#### 接口定义

```
def requant_fp(tensor_i: Tensor,
               scale: Union[float, List[float]],
               offset: Union[float, List[float]],
               out_dtype: str,
               out_name: str=None,
               round_mode: str='half_away_from_zero',
               first_round_mode: str='half_away_from_zero'):
```

#### 功能描述

对输入 tensor 进行量化处理。

该操作对应的计算式为：

```
output = saturate(int(round(float(input) * scale + offset))),  
其中 saturate 为饱和到 output 的数据类型
```

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型, 表示输入 Tensor, 3-5 维。
- scale: List[float] 型或 float 型, 表示量化系数。
- offset: List[int] 型或 int 型。表示输出偏移。
- out\_dtype: string 类型, 表示输入 Tensor 的类型。数据类型可以是“int16” / “uint16” / “int8” / “uint8”
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: string 型, 表示舍入模式。默认为“half\_away\_from\_zero”。round\_mode 取值范围为“half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。
- first\_round\_mode: string 型, 表示之前量化 tensor\_i 时使用的舍入模式。默认为“half\_away\_from\_zero”。first\_round\_mode 取值范围为“half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1688: 输入数据类型可以是 INT32/INT16/UINT16。
- BM1684X: 输入数据类型可以是 INT32/INT16/UINT16。

## requant\_int

对输入 tensor 进行量化处理。

```
def requant_int(tensor_i: Tensor,
    mul: Union[int, List[int]],
    shift: Union[int, List[int]],
    offset: Union[int, List[int]],
    requant_mode: int,
    out_dtype: str="int8",
    out_name=None,
    round_mode='half_away_from_zero', rq_axis:int = 1, fuse_rq_to_matmul:[F
    ↵bool = False):
```

## 功能描述

对输入 tensor 进行量化处理。

当 requant\_mode==0 时，该操作对应的计算式为：

```
output = shift > 0 ? (input << shift) : input
output = saturate((output * multiplier) >> 31), 其中 >> 为 round_half_up, [F]
↳ saturate 饱和到 INT32
output = shift < 0 ? (output >> -shift) : output, 其中 >> 的舍入模式由 round_
↳ mode 确定。
output = saturate(output + offset), 其中 saturate 饱和到 output 数据类型
```

- BM1684X: input 数据类型可以是 INT32, output 数据类型可以是 INT32/INT16/INT8
- BM1688: input 数据类型可以是 INT32, output 数据类型可以是 INT32/INT16/INT8

当 requant\_mode==1 时，该操作对应的计算式为：

```
output = saturate((input * multiplier) >> 31), 其中 >> 为 round_half_up, [F]
↳ saturate 饱和到 INT32
output = saturate(output >> -shift + offset), 其中 >> 的舍入模式由 round_
↳ mode 确定, saturate 饱和到 output 数据类型
```

- BM1684X: input 数据类型可以是 INT32, output 数据类型可以是 INT32/INT16/INT8
- BM1688: input 数据类型可以是 INT32, output 数据类型可以是 INT32/INT16/INT8

当 requant\_mode==2 时，该操作对应的计算式为（建议使用）：

```
output = input * multiplier
output = shift > 0 ? (output << shift) : (output >> -shift), 其中 >> [F]
↳ 的舍入模式由 round_mode 确定
output = saturate(output + offset), 其中 [F]
↳ saturate 饱和到 output 数据类型
```

- BM1684X: input 数据类型可以是 INT32/INT16/UINT16, output 数据类型可以是 INT16/UINT16/INT8/UINT8
- BM1688: input 数据类型可以是 INT32/INT16/UINT16, output 数据类型可以是 INT16/UINT16/INT8/UINT8

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor，3-5 维。
- mul: List[int] 型或 int 型，表示量化乘子系数。
- shift: List[int] 型或 int 型，表示量化移位系数。右移为负，左移为正。
- offset: List[int] 型或 int 型，表示输出偏移。
- requant\_mode: int 型，表示量化模式。
- round\_mode: string 型，表示舍入模式。默认为 “half\_away\_from\_zero”，范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。
- out\_dtype: string 类型或 None，表示输出 Tensor 的类型。None 代表输出数据类型为 “int8”
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- rq\_axis: int 型，表示在 rq\_axis 维度做 requant。
- fuse\_rq\_to\_matmul: bool 类型，表示是否将 requant 融合到 matmul，默认是 False。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1684X
- BM1688

## dequant\_int\_to\_fp

### 接口定义

```
def dequant_int_to_fp(tensor_i: Tensor,
                      scale: Union[float, List[float]],
                      offset: Union[int, List[int], float, List[float]],
                      out_dtype: str="float32",
                      out_name: str=None,
                      round_mode: str='half_away_from_zero'):
```

## 功能描述

对输入 tensor 进行反量化处理。

该操作对应的计算式为：

```
output = (input - offset) * scale
```

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor，3-5 维。
- scale: List[float] 型或 float 型，表示量化系数。
- offset: List[int] 型或 int 型，表示输出偏移。
- out\_dtype: string 类型，表示输出 Tensor 的类型。默认输出数据类型为“float32”。当输入数据类型为 int8/uint8 时，取值范围为“float16”，“float32”。当输入类型为 int16/uint16 时，输出类型只能为“float32”。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- round\_mode: string 型，表示舍入模式。默认为“half\_away\_from\_zero”。round\_mode 取值范围为“half\_away\_from\_zero”，“half\_to\_even”，“towards\_zero”，“down”，“up”。(TODO)

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 指定。

## 处理器支持

- BM1684X：input 数据类型可以是 INT16/UINT16/INT8/UINT8。

## dequant\_int

### 接口定义

```
def dequant_int(tensor_i: Tensor,
    mul: Union[int, List[int]],
    shift: Union[int, List[int]],
    offset: Union[int, List[int]],
    lshift: int,
```

(续下页)

(接上页)

```
requant_mode: int,
out_dtype: str="int8",
out_name=None,
round_mode='half_up');
```

## 功能描述

对输入 tensor 进行反量化处理。

当 requant\_mode==0 时，该操作对应的计算式为：

```
output = (intpu - offset) * multiplier
output = saturate(output >> -shift),           其中 >> 的舍入模式由round_
↪mode确定, saturate饱和到INT32
```

- BM1684X: input 数据类型可以是 INT16/UINT16/INT8/UINT8, output 数据类型可以是 INT32/INT16/UINT16

当 requant\_mode==1 时，该操作对应的计算式为：

```
output = ((input - offset) * multiplier) << lshift
output = saturate(output >> 31),                 其中 >> 为round_half_up, F
↪saturate饱和到INT32
output = saturate(output >> -shift),           其中 >> 的舍入模式由round_
↪mode确定, saturate饱和到output数据类型
```

- BM1684X: input 数据类型可以是 INT16/UINT16/INT8/UINT8, output 数据类型可以是 INT32/INT16/INT8

该操作属于 **本地操作**。

## 参数说明

- tensor\_i: Tensor 类型，表示输入 Tensor, 3-5 维。
- mul: List[int] 型或 int 型，表示量化乘子系数。
- shift: List[int] 型或 int 型，表示量化移位系数。右移为负，左移为正。
- offset: List[int] 型或 int 型，表示输出偏移。
- lshift: int 型，表示左移位系数。
- requant\_mode: int 型，表示量化模式。取值为 0 和 1，0 表示 “Normal”，1 表示 “TFLite”。
- round\_mode: string 型，表示舍入模式。默认为 “half\_up”，范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。
- out\_dtype: string 类型，表示输入 Tensor 的类型。默认为 “int8”。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor。该 Tensor 的数据类型由 out\_dtype 确定。

## 处理器支持

- BM1684X

## cast

## 接口定义

```
def cast(tensor_i: Tensor,  
        out_dtype: str = 'float32',  
        out_name: str = None,  
        round_mode: str = 'half_away_from_zero'):
```

## 功能描述

将输入张量 tensor\_i 转换为指定的数据类型 out\_dtype, 并根据指定的舍入模式 round\_mode 对数据进行舍入。注意本算子不能单独使用, 必须配合其他算子。

## 参数说明

- tensor\_i: Tensor 类型, 表示输入操作 Tensor。
- out\_dtype: str = ‘float32’, 输出张量的数据类型, 默认为 float32。
- out\_name: str = None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: str = ‘half\_away\_from\_zero’ , 舍入模式, 默认为 half\_away\_from\_zero。取值范围为 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”。注意, 此函数 round\_mode 不支持 “half\_up” 与 “half\_down”。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型由输入的 out\_dtype 决定。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/UINT8/INT8。

### 24.5.9 Up/Down Scaling Operator

`maxpool2d`

## 接口定义

```
def maxpool2d(input: Tensor,
              kernel: Union[List[int], Tuple[int], None] = None,
              stride: Union[List[int], Tuple[int], None] = None,
              pad: Union[List[int], Tuple[int], None] = None,
              ceil_mode: bool = False,
              scale: List[float] = None,
              zero_point: List[int] = None,
              out_name: str = None,
              round_mode: str = "half_away_from_zero"):
    #pass
```

## 功能描述

对输入 Tensor 进行 Max 池化处理。请参考各大框架下的池化操作。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型，表示输入操作 Tensor。
- kernel: List[int] 或 Tuple[int] 型或 None，输入 None 表示使用 global\_pooling，不为 None 时要求该参数长度为 2。
- stride: List[int] 或 Tuple[int] 型或 None，表示步长尺寸，输入 None 使用默认值 [1,1]，不为 None 时要求该参数长度为 2。
- pad: List[int] 或 Tuple[int] 型或 None，表示填充尺寸，输入 None 使用默认值 [0,0,0,0]，不为 None 时要求该参数长度为 4。
- ceil: bool 型，表示计算 output shape 时是否向上取整。

- scale: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 scale。
- zero\_point: List[int] 类型或 None, 偏移参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 zero\_point。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示舍入模式。默认值为'half\_away\_from\_zero'。round\_mode 取值范围为"half\_away\_from\_zero", "half\_to\_even", "towards\_zero", "down", "up"。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## maxpool2d\_with\_mask

### 接口定义

```
def maxpool2d_with_mask(input: Tensor,
                        kernel: Union[List[int], Tuple[int], None] = None,
                        stride: Union[List[int], Tuple[int], None] = None,
                        pad: Union[List[int], Tuple[int], None] = None,
                        ceil_mode: bool = False,
                        out_name: str = None,
                        mask_name: str = None):
    #pass
```

### 功能描述

对输入 Tensor 进行 Max 池化处理, 并输出其 mask index。请参考各大框架下的池化操作。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。
- kernel: List[int] 或 Tuple[int] 型或 None, 输入 None 表示使用 global\_pooling, 不为 None 时要求该参数长度为 2。
- pad: List[int] 或 Tuple[int] 型或 None, 表示填充尺寸, 输入 None 使用默认值 [0,0,0,0], 不为 None 时要求该参数长度为 4。
- stride: List[int] 或 Tuple[int] 型或 None, 表示步长尺寸, 输入 None 使用默认值 [1,1], 不为 None 时要求该参数长度为 2。
- ceil: bool 型, 表示计算 output shape 时是否向上取整。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- mask\_name: string 类型或 None, 表示输出 Mask 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回两个 Tensor, 一个 Tensor 的数据类型与输入 Tensor 相同。另一个返回一个坐标 Tensor, 该 Tensor 是记录使用比较运算池化时所选择的坐标。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。

## maxpool3d

### 接口定义

```
def maxpool3d(input: Tensor,
              kernel: Union[List[int], int, Tuple[int, ...]] = None,
              stride: Union[List[int], int, Tuple[int, ...]] = None,
              pad: Union[List[int], int, Tuple[int, ...]] = None,
              ceil_mode: bool = False,
              scale: List[float] = None,
              zero_point: List[int] = None,
              out_name: str = None,
              round_mode: str = "half_away_from_zero"):
    #pass
```

## 功能描述

对输入 Tensor 进行 Max 池化处理。请参考各大框架下的池化操作。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。
- kernel: List[int] 或 Tuple[int] 型或 int 或 None, 输入 None 表示使用 global\_pooling, 不为 None 时若输入单个整数, 表示在 3 个维度上的 kernel 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 3。
- stride: List[int] 或 Tuple[int] 型或 int 或 None, 表示步长尺寸, 输入 None 使用默认值 [1,1,1], 不为 None 时若输入单个整数, 表示在 3 个维度上的 stride 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 3。
- pad: List[int] 或 Tuple[int] 型或 int 或 None, 表示填充尺寸, 输入 None 使用默认值 [0,0,0,0,0,0], 不为 None 时若输入单个整数, 表示在 3 个维度上的 pad 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 6。
- ceil\_mode: bool 型, 表示计算 output shape 时是否向上取整。
- scale: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 scale。
- zero\_point: List[int] 类型或 None, 偏移参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 zero\_point。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示舍入模式。默认值为 'half\_away\_from\_zero'。round\_mode 取值范围为 "half\_away\_from\_zero", "half\_to\_even", "towards\_zero", "down", "up"。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## avgpool2d

### 接口定义

```
def avgpool2d(input: Tensor,
    kernel: Union[List[int], Tuple[int], None] = None,
    stride: Union[List[int], Tuple[int], None] = None,
    pad: Union[List[int], Tuple[int], None] = None,
    ceil_mode: bool = False,
    scale: List[float] = None,
    zero_point: List[int] = None,
    out_name: str = None,
    count_include_pad: bool = False,
    round_mode: str="half_away_from_zero",
    first_round_mode: str="half_away_from_zero"):

#pass
```

### 功能描述

对输入 Tensor 进行 Avg 池化处理。请参考各大框架下的池化操作。该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。
- kernel: List[int] 或 Tuple[int] 型或 None, 输入 None 表示使用 global\_pooling, 不为 None 时要求该参数长度为 2。
- stride: List[int] 或 Tuple[int] 型或 None, 表示步长尺寸, 输入 None 使用默认值 [1,1], 不为 None 时要求该参数长度为 2。
- pad: List[int] 或 Tuple[int] 型或 None, 表示填充尺寸, 输入 None 使用默认值 [0,0,0,0], 不为 None 时要求该参数长度为 4。
- ceil\_mode: bool 型, 表示计算 output shape 时是否向上取整。
- scale: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 scale。
- zero\_point: List[int] 类型或 None, 偏移参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 zero\_point。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- count\_include\_pad: Bool 类型, 表示在计算平均值时, 是否将 pad 值计算在内, 默认值为 False。

- round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示舍入模式。默认值为'half\_away\_from\_zero'。round\_mode 取值范围为"half\_away\_from\_zero" , "half\_to\_even" , "towards\_zero" , "down" , "up"。
- first\_round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示第一次的舍入模式。默认值为'half\_away\_from\_zero'。round\_mode 取值范围为"half\_away\_from\_zero" , "half\_to\_even" , "towards\_zero" , "down" , "up"。

## 返回值

返回一个 Tensor, 该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## avgpool3d

### 接口定义

```
def avgpool3d(input: Tensor,
    kernel: Union[List[int], int, Tuple[int, ...]] = None,
    stride: Union[List[int], int, Tuple[int, ...]] = None,
    pad: Union[List[int], int, Tuple[int, ...]] = None,
    ceil_mode: bool = False,
    scale: List[float] = None,
    zero_point: List[int] = None,
    out_name: str = None,
    count_include_pad: bool = False,
    round_mode: str = "half_away_from_zero",
    first_round_mode: str = "half_away_from_zero"):
    #pass
```

### 功能描述

对输入 Tensor 进行 Avg 池化处理。请参考各大框架下的池化操作。该操作属于 **本地操作**。

## 参数说明

- tensor: Tensor 类型, 表示输入操作 Tensor。
- kernel: List[int] 或 Tuple[int] 型或 int 或 None, 输入 None 表示使用 global\_pooling, 不为 None 时若输入单个整数, 表示在 3 个维度上的 kernel 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 3。
- pad: List[int] 或 Tuple[int] 型或 int 或 None, 表示填充尺寸, 输入 None 使用默认值 [0,0,0,0,0,0], 不为 None 时若输入单个整数, 表示在 3 个维度上的 pad 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 6。
- stride: List[int] 或 Tuple[int] 型或 int 或 None, 表示步长尺寸, 输入 None 使用默认值 [1,1,1], 不为 None 时若输入单个整数, 表示在 3 个维度上的 stride 大小相同, 若输入 List 或 Tuple, 要求该参数长度为 3。
- ceil\_mode: bool 型, 表示计算 output shape 时是否向上取整。
- scale: List[float] 类型或 None, 量化参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 scale。
- zero\_point: List[int] 类型或 None, 偏移参数。取 None 代表非量化计算。若为 List, 长度为 2, 分别为 input, output 的 zero\_point。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- count\_include\_pad: Bool 类型, 表示在计算平均值时, 是否将 pad 值计算在内, 默认值为 False。
- round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示第二次的舍入模式。默认值为' half\_away\_from\_zero'。round\_mode 取值范围为" half\_away\_from\_zero" , "half\_to\_even" , "towards\_zero" , "down" , "up"。
- first\_round\_mode: String 型, 当输入输出 Tensor 为量化时, 表示第一次的舍入模式。默认值为' half\_away\_from\_zero'。round\_mode 取值范围为" half\_away\_from\_zero" , "half\_to\_even" , "towards\_zero" , "down" , "up"。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。
- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8/UINT8。

## upsample

### 接口定义

```
def upsample(tensor_i: Tensor,
            scale: int = 2,
            out_name: str = None):
    #pass
```

### 功能描述

在 h 和 w 维度对输入 tensor 数据进行 scale 倍重复扩展输出。该操作属于 **本地操作**。

### 参数说明

- tensor\_i: Tensor 类型，表示输入操作 Tensor。
- scale: int 型，表示扩展倍数。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 Tensor 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16/INT8。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16/INT8。

`reduce`

## 接口定义

```
def reduce(tensor_i: Tensor,
          method: str = 'ReduceSum',
          axis: Union[List[int], Tuple[int], int] = None,
          keep_dims: bool = False,
          out_name: str = None):
    #pass
```

## 功能描述

依据 `axis_list`, 对输入的 `tensor` 做 `reduce` 操作。该操作属于 **受限本地操作**; 仅当输入数据类型为 FLOAT32 时是 **本地操作**。

## 参数说明

- `tensor_i`: `Tensor` 类型, 表示输入操作 `Tensor`。
- `method`: `string` 类型, 表示 `reduce` 方法, 目前可选”`ReduceMin`”, “`ReduceMax`”, “`ReduceMean`”, “`ReduceProd`”, “`ReduceL2`”, “`ReduceL1`”, “`ReduceSum`”。
- `axis`: `List[int]` 或 `Tuple[int]` 或 `int`, 表示需要 `reduce` 的轴。
- `keep_dims`: `bool` 型, 表示是否要保留原先的维度。
- `out_name`: `string` 类型或 `None`, 表示输出 `Tensor` 的名称, 为 `None` 时内部会自动生成名称。

## 返回值

返回一个 `Tensor`, 该 `Tensor` 的数据类型与输入 `Tensor` 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

### 24.5.10 Normalization Operator

#### batch\_norm

##### 接口定义

```
def batch_norm(input: Tensor,
               mean: Tensor,
               variance: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               out_name: str = None):
    #pass
```

##### 功能描述

该 batch\_norm 算子先完成输入值的批归一化，完成归一化之后再进行缩放和平移。批归一化运算过程可参考各框架的 batch\_norm 算子。

该操作属于 **本地操作**。

##### 参数说明

- input: Tensor 类型，表示输入待归一化的 Tensor，维度不限，如果 x 只有 1 维，c 为 1，否则 c 等于 x 的 shape[1]。
- mean: Tensor 类型，表示输入的均值，shape 为 [c]。
- variance: Tensor 类型，表示输入的方差值，shape 为 [c]。
- gamma: Tensor 类型或 None，表示批归一化之后进行的缩放，不为 None 时要求 shape 为 [c]，取 None 时相当于 shape 为 [c] 的全 1Tensor。
- beta: Tensor 类型或 None，表示批归一化和缩放之后进行的平移，不为 None 时要求 shape 为 [c]，取 None 时相当于 shape 为 [c] 的全 0Tensor。
- epsilon: FLOAT 类型，表示为了除法运算数值稳定加在分母上的值。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回 Tensor 类型，表示输出归一化后的 Tensor，数据类型与输入一致。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X：输入数据类型可以是 FLOAT32/FLOAT16。

## layer\_norm

### 接口定义

```
def layer_norm(input: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               axis: int,
               out_name: str = None):
    #pass
```

### 功能描述

该 layer\_norm 算子先完成输入值的归一化，完成归一化之后再进行缩放和平移。批归一化运算过程可参考各框架的 layer\_norm 算子。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型，表示输入待归一化的 Tensor，维度不限，如果  $x$  只有 1 维， $c$  为 1，否则  $c$  等于  $x$  的  $shape[1]$ 。
- gamma: Tensor 类型或 None，表示批归一化之后进行的缩放，不为 None 时要求  $shape$  为  $[c]$ ，取 None 时相当于  $shape$  为  $[c]$  的全 1 Tensor。
- beta: Tensor 类型或 None，表示批归一化和缩放之后进行的平移，不为 None 时要求  $shape$  为  $[c]$ ，取 None 时相当于  $shape$  为  $[c]$  的全 0 Tensor。
- epsilon: FLOAT 类型，表示为了除法运算数值稳定加在分母上的值。
- axis: int 型，第一个标准化的维度。如果  $rank(X)$  为  $r$ ，则 axis 的允许范围为  $[-r, r]$ 。负值表示从后面开始计算维度。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回 Tensor 类型, 表示输出归一化后的 Tensor, 数据类型与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## group\_norm

### 接口定义

```
def group_norm(input: Tensor,
               gamma: Tensor = None,
               beta: Tensor = None,
               epsilon: float = 1e-5,
               num_groups: int,
               out_name: str = None):
    #pass
```

### 功能描述

该 group\_norm 算子先完成输入值的归一化, 完成归一化之后再进行缩放和平移。批归一化运算过程可参考各框架的 group\_norm 算子。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示输入待归一化的 Tensor, 维度不限, 如果 x 只有 1 维, c 为 1, 否则 c 等于 x 的 shape[1]。
- gamma: Tensor 类型或 None, 表示批归一化之后进行的缩放, 不为 None 时要求 shape 为 [c], 取 None 时相当于 shape 为 [c] 的全 1Tensor。
- beta: Tensor 类型或 None, 表示批归一化和缩放之后进行的平移, 不为 None 时要求 shape 为 [c], 取 None 时相当于 shape 为 [c] 的全 0Tensor。
- epsilon: FLOAT 类型, 表示为了除法运算数值稳定加在分母上的值。

- num\_groups: int 型, 表示分组的数量。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回 Tensor 类型, 表示输出归一化后的 Tensor, 数据类型与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## rms\_norm

### 接口定义

```
def rms_norm(input: Tensor,
             gamma: Tensor = None,
             epsilon: float = 1e-5,
             axis: int = -1,
             out_name: str = None):
    #pass
```

### 功能描述

该 rms\_norm 算子先完成输入值最后一个维度的归一化, 完成归一化之后再进行缩放。运算过程可参考各框架的 RMSNorm 算子。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型, 表示输入待归一化的 Tensor, 维度不限。
- gamma: Tensor 类型或 None, 表示批归一化之后进行的缩放, 不为 None 时要求 shape 与 input 最后一维 w 相等, 取 None 时相当于 shape 为 [w] 的全 1Tensor。
- epsilon: FLOAT 类型, 表示为了除法运算数值稳定加在分母上的值。
- axis: int 型, 第一个标准化的维度。如果 rank(X) 为 r, 则 axis 的允许范围为 [-r, r]。负值表示从后面开始计算维度。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回 Tensor 类型, 表示输出归一化的后的 Tensor, 数据类型与输入一致。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## normalize

### 接口定义

```
def normalize(input: Tensor,
             p: float = 2.0,
             axes: Union[List[int], int] = 1,
             eps : float = 1e-12,
             out_name: str = None):
```

### 功能描述

对输入张量 input 的指定维度进行  $L_p$  归一化。

对于大小为  $(n_0, \dots, n_{dim}, \dots, n_k)$  的张量输入, 每个  $n_{dim}$  ‘元素向量’ :  $v$  ‘沿维度’ : attr : ‘axes’ 的变换为

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}$$

在默认参数下, 它对向量的维度 1 上进行 L2 归一化。

该操作属于 **本地操作**。

### 参数说明

- input: Tensor 类型。表示输入 Tensor。数据类型为 float32, float16。
- p: float 类型, 默认值为 2.0。表示是归一化过程中的指数值。
- axes: Union[List[int], int] 类型, 默认为 1。表示要归一化的维度。如果是 list, 那么 list 内的值必须是连续的。注意, axes = [0,-1] 并不是连续的。

- eps : float 类型，默认值为 1e-12。一个极小值，用来避免除以 0。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回 Tensor 类型，表示输出归一化的后的 Tensor，数据类型与输入一致。

## 处理器支持

- BM1688：输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X：输入数据类型可以是 FLOAT32/FLOAT16。

### 24.5.11 Vision Operator

#### nms

##### 接口定义

```
def nms(boxes: Tensor,  
        scores: Tensor,  
        format: str = 'PYTORCH',  
        max_box_num_per_class: int = 1,  
        out_name: str = None)
```

##### 功能描述

对输入 tensor 进行非极大值抑制处理。

##### 参数说明

- boxes: Tensor 类型，表示输入框的列表。必须是三维张量，第一维为批的个数，第二维为框的个数，第三维为框的 4 个坐标。
- scores: Tensor 类型，表示输入得分的列表。必须是三维张量，第一维为批的个数，第二维为类的个数，第三维为框的个数。
- format: string 类型，'TENSORFLOW' 表示 Tensorflow 格式 [y1, x1, y2, x2]，'PYTORCH' 表示 Pytorch 格式 [x\_center, y\_center, width, height]，默认值为 'PYTORCH'。

- max\_box\_num\_per\_class: int 型, 表示每个类中的输出框的最大个数。必须大于 0, 默认值为 1。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor, 表示从框列表中选出的框的索引的列表, 它是一个 2 维张量, 格式为 [num\_selected\_indices, 3], 其中每个索引的格式为 [batch\_index, class\_index, box\_index]。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32。
- BM1684X: 输入数据类型可以是 FLOAT32。

## interpolate

### 接口定义

```
def interpolate(input: Tensor,
               scale_h: float,
               scale_w: float,
               method: str = 'nearest',
               coord_mode: str = "pytorch_half_pixel",
               out_name: str = None)
```

### 功能描述

对输入 tensor 进行插值。

### 参数说明

- input: Tensor 类型, 表示输入的 Tensor。必须是至少 2 维的张量。
- scale\_h: float 型, 表示高度方向的缩放系数, 必须大于 0。
- scale\_w: float 型, 表示宽度方向的缩放系数, 必须大于 0。
- method: string 类型, 表示插值方法, 可选项为” nearest” 或” linear”。默认值为” nearest”。
- coord\_mode: string 类型, 表示输出坐标的计算方法, 可选项为” align\_corners”、” pytorch\_half\_pixel”、” half\_pixel”、” asymmetric”。默认值为” pytorch\_half\_pixel”。

- out\_name: string 类型或 None，表示输出 Tensor 的名称。如果为 None，内部会自动生成名称。

其中，coord\_mode 的意义跟 onnx 的 Resize 算子的参数 coordinate\_transformation\_mode 的意义是一样的。若 h/w 方向的放缩因子为 scale，输入坐标为 x\_in，输入尺寸为 l\_in，输出坐标为 x\_out，输出尺寸为 l\_out，则逆映射定义如下：

- “half\_pixel”:

```
x_in = (x_out + 0.5) / scale - 0.5
```

- “pytorch\_half\_pixel”:

```
x_in = len > 1 ? (x_out + 0.5) / scale - 0.5 : 0
```

- “align\_corners”:

```
x_in = x_out * (l_in - 1) / (l_out - 1)
```

- “asymmetric”:

```
x_in = x_out / scale
```

## 返回值

返回一个 Tensor，表示插值后的结果。数据类型与输入类型相同，形状根据缩放系数进行调整。

## 处理器支持

- BM1688: 支持的输入数据类型为 FLOAT32/FLOAT16/INT8。
- BM1684X: 支持的输入数据类型为 FLOAT32/FLOAT16/INT8。

## yuv2rgb

### 接口定义

```
def yuv2rgb(
    inputs: Tensor,
    src_format: int,
    dst_format: int,
    ImageOutFormatAttr: str,
    formula_mode: str,
    round_mode: str,
    out_name: str = None,
):
```

## 功能描述

对输入 tensor 进行 yuv 转 rgb 格式，输入的 Tensor 要求 shape=[n,h\*3/2,w]，其中 n 为批个数，h 为图像的像素高，w 为图像的像素宽。

## 参数说明

- inputs: Tensor 类型，表示输入的 yuv 矩阵。必须是三维张量，第一维为批的个数，第二维为输入矩阵的高，第三维为输入矩阵的宽。
- src\_format: Int 类型，表示输入的格式。FORMAT\_MAPPING\_YUV420P\_YU12=0, FORMAT\_MAPPING\_YUV420P\_YV12=1, FORMAT\_MAPPING\_NV12=2, FORMAT\_MAPPING\_NV21=3。
- dst\_format: Int 类型，表示输出的格式。FORMAT\_MAPPING\_RGB=4, FORMAT\_MAPPING\_BGR=5。
- ImageOutFormatAttr: str 型，目前只支持”UINT8”。
- formula\_mode: string 类型，表示使用的 yuv2rgb 转换公式，目前支持”\_601\_limited”、”\_601\_full”。
- round\_mode: string 类型，表示使用的舍入模式，目前支持”HalfAwayFromZero”，”HalfToEven”。
- out\_name: string 类型，表示输出 Tensor 的名称，非必选，默认为 None。

## 返回值

返回一个 Tensor，表示转换出的 rgb 格式 Tensor，shape=[n,3,h,w]，其中 n 为批个数，h 为图像的像素高，w 为图像的像素宽。

## 处理器支持

- BM1684X: 输入数据类型是 INT8/UINT8，输出 UINT8。
- BM1688: 输入数据类型是 INT8/UINT8，输出 UINT8。

## roiExtractor

## 接口定义

```
def roiExtractor(rois: Tensor,
                 target_lvls: Tensor,
                 feats: List[Tensor],
                 PH: int,
```

(续下页)

(接上页)

```
PW: int,
sampling_ratio: int,
list_spatial_scale: Union[int, List[int], Tuple[int]],
mode:str=None,
out_name:str=None)
```

## 功能描述

给定 4 个 feature map，根据 target\_lvls 索引从 rois 中抽取对应的 roi，并与对应的 feature map 做 roi align，得到最终输出。该操作属于 **本地操作**。

## 参数说明

- rois: Tensor 类型，表示所有的 rois。
- target\_lvls: Tensor 类型，表示 roi 对应哪层 feature map。
- feats: List[Tensor] 型，表示多层 feature map。
- PH: int 类型，表示输出的 height。
- PW: int 类型，表示输出的 width。
- sampling\_ratio: int 类型，表示每层 feature map 的 sample ratio。
- list\_spatial\_scale: int, List[int] 或 Tuple[int] 型，表示每层 feature map 对应的 spatial scale。  
请注意，spatial scale 遵循 mmdetection 风格，最初给定一个整数值，但其浮点倒数最终被用于 RoIAlign。
- mode: string 类型，表示 Op 执行模式，目前支持 DynNormal, DynFuse。  
请注意，在 DynFuse 模式下，输入 rois 的坐标支持 2 类风格，1) 遵循 mmdetection 的风格，即 5 长度 [batch\_id, x0, y0, x1, y1]。  
2) 自定义的 7 长度 [a, b, x0, y0, x1, y1, c]，特别注意如果 batch\_id 和 a,b,c 难以匹配，建议另外重新生成 batch\_id。  
在 DynNormal 模式下，输入 rois 的坐标风格是一种自定义的 7 长度 [a, b, x0, y0, x1, y1, c] 风格，以便应用客户独特的模型。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动产生名称。

## 返回值

返回一个 Tensor，该 Tensor 的数据类型与输入 rois 相同。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

### 24.5.12 Select Operator

#### nonzero

## 接口定义

```
def nonzero(tensor_i:Tensor,  
           dtype: str = 'int32',  
           out_name: str = None):  
    #pass
```

## 功能描述

抽取输入 Tensor data 为 true 时对应的位置信息信息。该操作属于 全局操作。

## 参数说明

- tensor\_i: Tensor 类型，表示输入操作 Tensor。
- dtype: string 型，表示输出数据类型，目前仅可使用默认值” int32”。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

## 返回值

返回一个 Tensor，数据类型为 INT32。

## 处理器支持

- BM1688: 输入数据类型可以是 FLOAT32/FLOAT16。
- BM1684X: 输入数据类型可以是 FLOAT32/FLOAT16。

## lut

### 接口定义

```
def lut(input: Tensor,  
        table: Tensor,  
        out_name: str = None):  
    #pass
```

### 功能描述

对输入 tensor 进行查找表查找操作。

### 参数说明

- input: Tensor 类型, 表示输入。
- table: Tensor 类型, 表示查找表。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor, 数据类型与张量 table 的数据类型相同。

## 处理器支持

- BM1688: input 的数据类型可以是 INT8/UINT8, table 的数据类型可以是 INT8/UINT8。
- BM1684X: input 的数据类型可以是 INT8/UINT8, table 的数据类型可以是 INT8/UINT8。

## select

### 接口定义

```
def select(lhs: Tensor,  
          rhs: Tensor,  
          tbrn: Tensor,  
          fbrn: Tensor,  
          type: str,  
          out_name = None):  
    #pass
```

### 功能描述

根据 lhs 与 rhs 的数值比较结果来选择，条件为真时，选择 tbrn，条件为假时，选择 fbrn。

### 参数说明

- lhs: Tensor 类型，表示左边的张量。
- rhs: Tensor 类型，表示右边的张量。
- tbrn: Tensor 类型，表示条件为真时取的值。
- fbrn: Tensor 类型，表示条件为假时取的值。
- type: string 类型，表示比较符。可选项为” Greater” /” Less” /” GreaterOrEqual” /” LessOrEqual” /” Equal” /” NotEqual”。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

约束条件：要求 lhs 与 rhs 的形状和数据类型相同，tbrn 与 fbrn 的形状和数据类型相同。

### 返回值

返回一个 Tensor，数据类型与张量 ‘tbrn’ 的数据类型相同。

## 处理器支持

- BM1688: lhs / rhs / tbrn / fbrn 的数据类型可以是 FLOAT32/FLOAT16(TODO)。
- BM1684X: lhs / rhs / tbrn / fbrn 的数据类型可以是 FLOAT32/FLOAT16(TODO)。

## cond\_select

### 接口定义

```
def cond_select(cond: Tensor,
                tbrn: Union[Tensor, Scalar],
                fbrn: Union[Tensor, Scalar],
                out_name:str = None):
    #pass
```

### 功能描述

根据条件 cond 来选择，条件为真时，选择 tbrn，条件为假时，选择 fbrn。

### 参数说明

- cond: Tensor 类型，表示条件。
- tbrn: Tensor 类型或 Scalar 类型，表示条件为真时取的值。
- fbrn: Tensor 类型或 Scalar 类型，表示条件为假时取的值。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。

约束条件：若 tbrn 和 fbrn 皆为张量，则要求 tbrn 与 fbrn 的形状和数据类型相同。

### 返回值

返回一个 Tensor，数据类型与张量 tbrn 的数据类型相同。

## 处理器支持

- BM1688: cond / tbrn / fbrn 的数据类型可以是 FLOAT32/FLOAT16(TODO)。
- BM1684X: cond / tbrn / fbrn 的输入数据类型可以是 FLOAT32/FLOAT16(TODO)。

## bmodel\_inference\_combine

### 接口定义

```
def bmodel_inference_combine(
    bmodel_file: str,
    final_mlir_fn: str,
    input_data_fn: Union[str, dict],
    tensor_loc_file: str,
    reference_data_fn: str,
    dump_file: bool = True,
    save_path: str = "",
    out_fixed: bool = False,
    dump_cmd_info: bool = True,
    skip_check: bool = True, # disable data_check to increase processing speed
    run_by_op: bool = False, # enable to run_by_op, may cause timeout error[F]
    ↵when some OPs contain too many atomic cmds
    desire_op: list = [], # set ["A","B","C"] to only dump tensor A/B/C, dump all[F]
    ↵tensor as default
    is_soc: bool = False, # soc mode ONLY support {reference_data_fn=xxx.npz,[F]
    ↵dump_file=True}
    using_memory_opt: bool = False, # required when is_soc=True
    enable_soc_log: bool = False, # required when is_soc=True
    soc_tmp_path: str = "/tmp", # required when is_soc=True
    hostname: str = None, # required when is_soc=True
    port: int = None, # required when is_soc=True
    username: str = None, # required when is_soc=True
    password: str = None, # required when is_soc=True
):

```

## 功能描述

根据生成的 bmodel 进行推理和逐层 Tensor 数据打印，配合 npz\_tool.py 进行 bmodel 正确性验证。

## 参数说明

- bmodel\_file: String 类型，表示 bmodel 绝对路径。
- final\_mlir\_fn: String 类型，表示 bmodel 对应的 final.mlir 的绝对路径。
- input\_data\_fn: String 类型或 dict 类型，表示输入数据的格式，支持字典格式、.dat 格式、.npz 格式。
- tensor\_loc\_file: String 类型，表示 bmodel 对应的 tensor\_location.json 文件的绝对路径。
- reference\_data\_fn: String, 类型，表示 ‘module.state = “TPU\_LOWERED”’ 的 mlir 文件或对应的.npz 推理结果的绝对路径。bmodel 推理时会将原本一个算子的 shape 拆散，该参数用于恢复原本的 shape。
- dump\_file: Bool 类型，表示逐层 Tensor 数据是否以.npz 文件形式保存，或直接返回字典。
- save\_path: String 类型，表示 dump\_file=True 时的主机 (host) 端保存逐层推理的.npz 文件的绝对路径。
- out\_fixed: Bool 类型，表示逐层 Tensor 数据输出是否保持为定点格式。
- dump\_cmd\_info: Bool 类型，表示将当前 bmodel 中包含的所有原子指令对应的 final.mlir 的信息保存成 txt 文件，保存路径在 save\_path 下。
- skip\_check: Bool 类型，启用此项可禁用数据对比，提高推理速度。soc 模式下默认不进行数据对比。
- run\_by\_op: Bool 类型，启用后按 OP 粒度运行，禁用时为按原子指令粒度运行。按 OP 粒度运行速度较快，但当一个 OP 中包含过多原子指令时可能会引发 timeout 错误。
- desire\_op: List 类型，其中当传入多个 String 类型的名字时，只会 dump 出给定名字的 tensor。默认 dump 所有层 tensor。
- is\_soc: Bool 类型，表示是否启用 soc 模式进行推理。
- using\_memory\_opt: Bool 类型，启用后会减小在 device 端的内存消耗，但会增加耗时。推荐在大模型时启用。
- enable\_soc\_log: Bool 类型，启用此项打印并在 save\_path 下保存 log 日志。
- soc\_tmp\_path: String 类型，表示 soc 模式下，板卡 (device) 端存放临时文件与推理工具的绝对路径。
- hostname: String 类型，表示 soc 模式下，device 端的 ip 地址。
- port: Int 类型，表示 soc 模式下，device 端的端口号。

- username: String 类型, 表示 soc 模式下, device 端的用户名。
- password: String 类型, 表示 soc 模式下, device 端的密码。

注意:

- 当使用 pcie 或 soc 模式进行逐层 dump 时, 需先使用 /tpu-mlir/envsetup.sh 中的 use\_chip 切换环境变量。当使用 cmodel 模式时, 使用 use\_cmodel。
- 当使用 soc 模式时: reference\_data\_fn 必须是.npz 格式。

## 返回值

- cmodel/pcie 模式下: 如果 dump\_file=True, 则在 save\_path 下生成 bmodel\_infer\_xxx.npz 文件, 否则返回 python 字典。
- soc 模式下: 在 save\_path 下生成 soc\_infer\_xxx.npz 文件。

## 处理器支持

- BM1688: cmodel 模式。
- BM1684X: cmodel/pcie/soc 模式。

### scatter

#### 接口定义

```
def scatter(input: Tensor,
           index: Tensor,
           updates: Tensor,
           axis: int = 0,
           out_name: str = None):
    #pass
```

#### 功能描述

根据指定的索引, 将输入数据写入目标 Tensor 的特定位置。该操作允许将更新输入 Tensor 的元素散布到输出 Tensor 的指定位置。请参考各大框架下的 ScatterElements 操作。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor, 即需要更新的目标 Tensor。
- index: Tensor 类型, 表示指定更新位置的索引 Tensor。
- updates: Tensor 类型, 表示要写入目标 Tensor 的值。
- axis: int 型, 表示更新的轴。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个新的 Tensor, 该 Tensor 在指定位置上进行了更新操作, 其他位置保持了原始输入的 Tensor 值。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16,INT8。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16,INT8。

## scatterND

### 接口定义

```
def scatterND(input: Tensor,  
             indices: Tensor,  
             updates: Tensor,  
             out_name: str = None):  
    #pass
```

### 功能描述

根据指定的索引, 将输入数据写入目标 Tensor 的特定位置。该操作允许将更新输入 Tensor 的元素散布到输出 Tensor 的指定位置。请参考 ONNX 11 下的 ScatterND 操作。该操作属于 **本地操作**。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor, 即需要更新的目标 Tensor。
- indices: Tensor 类型, 表示指定更新位置的索引 Tensor。数据类型必须是 uint32。
- updates: Tensor 类型, 表示要写入目标 Tensor 的值。Rank(updates) = Rank(input) + Rank(indices) - shape(indices)[-1] - 1。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回一个新的 Tensor, 该 Tensor 在指定位置上进行了更新操作, 其他位置保持了原始输入的 Tensor 值。形状与数据类型和 input 一致。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16,INT8。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16,INT8。

### 24.5.13 Preprocess Operator

#### mean\_std\_scale

##### 接口定义

```
def mean_std_scale(input: Tensor,
                   std: List[float],
                   mean: List[float],
                   scale: Optional[Union[List[float],List[int]]] = None,
                   zero_points: Optional[List[int]] = None,
                   out_name: str = None,
                   dtype="float16",
                   round_mode: str = "half_away_from_zero"):

#pass
```

## 功能描述

对输入 Tensor 进行预处理操作。该操作属于 全局操作。

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。必须是 4 维或 5 维。
- std: List[float] 类型, 表示数据集的标准差。mean,std 维度必须和 input 的 channel 维度一致, 即 input 的第二维。
- mean: List[float] 类型, 表示数据集的均值。mean,std 维度必须和 input 的 channel 维度一致, 即 input 的第二维。
- scale: Optional[Union[List[float],List[int]]] 类型或 None, 缩放系数。
- zero\_points: Optional[List[int]] 类型或 None, 表示零点。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。
- odtype: String 类型, 表示接口输出 Tensor 数据类型。默认值为” float16”。目前支持 float16, int8。
- round\_mode: String 类型, 表示取整方法。默认值为” half\_away\_from\_zero”, 范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”

## 返回值

返回一个 Tensor, 数据类型为 odtype。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32/UINT8/INT8, 输出类型可以为 INT8/FLOAT16。

### 24.5.14 Transform Operator

rope

## 接口定义

```
def rope( input: Tensor,
          weight0: Tensor,
          weight1: Tensor,
```

(续下页)

(接上页)

```

is_permute_optimize: bool = False,    # unused
mul1_round_mode: str = 'half_up',
mul2_round_mode: str = 'half_up',
add_round_mode: str = 'half_up',
mul1_shift: int = None,
mul2_shift: int = None,
add_shift: int = None,
mul1_saturation: bool = True,
mul2_saturation: bool = True,
add_saturation: bool = True,
out_name: str = None):
#pass

```

## 功能描述

对输入 Tensor 进行旋转编码 (RoPE) 操作。该操作属于 全局操作

## 参数说明

- input: Tensor 类型, 表示输入操作 Tensor。必须是 4 维。
- weight0: Tensor, 表示输入操作 Tensor。
- weight1: Tensor, 表示输入操作 Tensor。
- is\_permute\_optimize: bool 类型, 表示是否做 permute 下沉, 进行 permute 下沉 shape 的检查。# unused
- mul1\_round\_mode: String 类型, 表示 RoPE 中 mul1 的取整方法。默认值为”half\_away\_from\_zero”，范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”。
- mul2\_round\_mode: String 类型, 表示 RoPE 中 mul2 的取整方法。默认值为”half\_away\_from\_zero”，范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”。
- add\_round\_mode: String 类型, 表示 RoPE 中 add 的取整方法。默认值为”half\_away\_from\_zero”，范围是 “half\_away\_from\_zero”, “half\_to\_even”, “towards\_zero”, “down”, “up”, “half\_up”, “half\_down”。
- mul1\_shift: int 型, 表示 RoPE 中 mul1 的移位的位数。
- mul2\_shift: int 型, 表示 RoPE 中 mul2 的移位的位数。
- add\_shift: int 型, 表示 RoPE 中 add 的移位的位数。
- mul1\_saturation: bool 类型, 表示 RoPE 中的 mul1 计算结果是否需要饱和处理, 默认为 True 饱和处理, 非必要不修改。

- mul2\_saturation: bool 类型, 表示 RoPE 中的 mul2 计算结果是否需要饱和处理, 默认为 True 饱和处理, 非必要不修改。
- add\_saturation: bool 类型, 表示 RoPE 中的 add 计算结果是否需要饱和处理, 默认为 True 饱和处理, 非必要不修改。
- out\_name: output name, string 类型, 默认为 None。

## 返回值

返回一个 Tensor, 数据类型为 odtype。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16 和 INT 类型。

## multi\_scale\_deformable\_attention

### 接口定义

```
def multi_scale_deformable_attention(  
    query: Tensor,  
    value: Tensor,  
    key_padding_mask: Tensor,  
    reference_points: Tensor,  
    sampling_offsets_weight: Tensor,  
    sampling_offsets_bias_ori: Tensor,  
    attention_weights_weight: Tensor,  
    attention_weights_bias_ori: Tensor,  
    value_proj_weight: Tensor,  
    value_proj_bias_ori: Tensor,  
    output_proj_weight: Tensor,  
    output_proj_bias_ori: Tensor,  
    spatial_shapes: List[List[int]],  
    embed_dims: int,  
    num_heads: int = 8,  
    num_levels: int = 4,  
    num_points: int = 4,  
    out_name: str = None):  
  
    #pass
```

## 功能描述

对输入进行多尺度可变形注意力机制，具体功能可参考 [https://github.com/open-mmlab/mmcv/blob/main/mmcv/ops/multi\\_scale\\_deform\\_attn.py](https://github.com/open-mmlab/mmcv/blob/main/mmcv/ops/multi_scale_deform_attn.py):MultiScaleDeformableAttention:forward。该操作的实现方式与官方有所不同。目前只支持 batch\_size=1 的情况。该操作属于 **本地操作**。

## 参数说明

- query: Tensor 类型, Transformer 的查询张量, 形状为 (1, num\_query, embed\_dims)。
- value: Tensor 类型, 值投影张量, 形状为 (1, num\_key, embed\_dims)。
- key\_padding\_mask: Tensor 类型, 查询张量的 mask, 形状为 (1, num\_key)。
- reference\_points: Tensor 类型, 归一化的参考点, 形状为 (1, num\_query, num\_levels, 2), 所有元素的范围在 [0, 1] 之间, 左上角为 (0,0), 右下角为 (1,1), 包括填充区域。
- sampling\_offsets\_weight: Tensor 类型, 计算采样偏移量全连接层的权重, 形状为 (embed\_dims, num\_heads\*num\_levels\*num\_points\*2)。
- sampling\_offsets\_bias\_ori: Tensor 类型, 计算采样偏移量全连接层的偏置, 形状为 (num\_heads\*num\_levels\*num\_points\*2)。
- attention\_weights\_weight: Tensor 类型, 计算注意力权重全连接层的权重, 形状为 (embed\_dims, num\_heads\*num\_levels\*num\_points)。
- attention\_weights\_bias\_ori: Tensor 类型, 计算注意力权重全连接层的偏置, 形状为 (num\_heads\*num\_levels\*num\_points)。
- value\_proj\_weight: Tensor 类型, 计算值投影全连接层的权重, 形状为 (embed\_dims, embed\_dims)。
- value\_proj\_bias\_ori: Tensor 类型, 计算值投影全连接层的偏置, 形状为 (embed\_dims)。
- output\_proj\_weight: Tensor 类型, 计算输出投影全连接层的权重, 形状为 (embed\_dims, embed\_dims)。
- output\_proj\_bias\_ori: Tensor 类型, 计算输出投影全连接层的偏置, 形状为 (embed\_dims)。
- spatial\_shapes: List[List[int]] 类型, 不同层级特征的空间形状, 形状为 (num\_levels, 2), 最后一个维度表示 (h, w)。
- embed\_dims: int 类型, 查询、键、值的 hidden\_size。
- num\_heads: int 类型, 注意力头数, 默认值为 8。
- num\_levels: int 类型, 多尺度注意力的层级数, 默认值为 4。
- num\_points: int 类型, 每个层级的采样点数, 默认值为 4。

- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

### 返回值

返回一个 Tensor, 数据类型为 query.dtype。

### 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16 类型。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16 类型。

#### 24.5.15 Transform Operator

##### a16matmul

###### 接口定义

```
def a16matmul(input: Tensor,
               weight: Tensor,
               scale: Tensor,
               zp: Tensor,
               bias: Tensor = None,
               right_transpose=True,
               out_dtype: str = 'float16',
               out_name: str = None,
               group_size: int = 128,
               bits: int = 4,
               g_idx: Tensor = None,
               ):
    #pass
```

###### 功能描述

对输入进行 W4A16/W8A16 MatMul。该操作属于 全局操作。

## 参数说明

- input: Tensor 类型，表示输入 tensor。
- weight: Tensor 类型，表示 4bits/8bits 量化后权重，以 int32 类型存储。
- scale: Tensor 类型，表示权重量化缩放因子，以 float32 类型存储。
- zp: Tensor 类型，表示权重量化零点，以 int32 类型存储。
- bias: Tensor 类型，表示偏置，以 float32 类型存储。
- right\_transpose: Bool 类型，表示权重矩阵是否转置，目前仅支持为 True。
- out\_dtype: string 类型，表示输出张量的数据类型。
- out\_name: string 类型或 None，表示输出 Tensor 的名称，为 None 时内部会自动生成名称。
- group\_size: int 类型，表示量化的 group 大小。
- bits: int 类型，表示量化位宽，仅支持 4bits/8bits。
- g\_idx: Tensor 类型，量化重排系数，目前不支持。

## 返回值

返回一个 Tensor，数据类型为 out\_dtype。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16 类型。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16 类型。

### 24.5.16 Transform Operator

#### qwen2\_block

##### 接口定义

```
def qwen2_block(hidden_states: Tensor,
                position_ids: Tensor,
                attention_mask: Tensor,
                q_proj_weights: Tensor,
                q_proj_scales: Tensor,
                q_proj_zps: Tensor,
                q_proj_bias: Tensor,
                k_proj_weights: Tensor,
```

(续下页)

(接上页)

```
k_proj_scales: Tensor,  
k_proj_zps: Tensor,  
k_proj_bias: Tensor,  
v_proj_weights: Tensor,  
v_proj_scales: Tensor,  
v_proj_zps: Tensor,  
v_proj_bias: Tensor,  
o_proj_weights: Tensor,  
o_proj_scales: Tensor,  
o_proj_zps: Tensor,  
o_proj_bias: Tensor,  
down_proj_weights: Tensor,  
down_proj_scales: Tensor,  
down_proj_zps: Tensor,  
gate_proj_weights: Tensor,  
gate_proj_scales: Tensor,  
gate_proj_zps: Tensor,  
up_proj_weights: Tensor,  
up_proj_scales: Tensor,  
up_proj_zps: Tensor,  
input_layernorm_weight: Tensor,  
post_attention_layernorm_weight: Tensor,  
cos: List[Tensor],  
sin: List[Tensor],  
out_dtype: str = 'float16',  
group_size: int = 128,  
weight_bits: int = 4,  
hidden_size: int = 3584,  
rms_norm_eps: float = 1e-06,  
num_attention_heads: int = 28,  
num_key_value_heads: int = 4,  
mrope_section: List[int] = [16, 24, 24],  
quant_method: str = "gptq",  
out_name: str = None  
):  
  
#pass
```

## 功能描述

qwen2 在 prefill 阶段的一个 block layer。该操作属于 全局操作。

## 参数说明

- hidden\_states: Tensor 类型, 表示激活值, 形状为 (1, seq\_length, hidden\_size)。
- position\_ids: Tensor 类型, 表示位置索引, 形状为 (3, 1, seq\_length)。
- attention\_mask: Tensor 类型, 表示注意力掩码, 形状为 (1, 1, seq\_length, seq\_length)。
- q\_proj\_weights: Tensor 类型, 表示 query 量化后权重, 以 int32 类型存储。
- q\_proj\_scales: Tensor 类型, 表示 query 量化缩放因子, 以 float32 类型存储。
- q\_proj\_zps: Tensor 类型, 表示 query 量化零点, 以 int32 类型存储。
- q\_proj\_bias: Tensor 类型, 表示 query 偏置, 以 float32 类型存储。
- k\_proj\_weights: Tensor 类型, 表示 key 量化后权重, 以 int32 类型存储。
- k\_proj\_scales: Tensor 类型, 表示 key 量化缩放因子, 以 float32 类型存储。
- k\_proj\_zps: Tensor 类型, 表示 key 量化零点, 以 int32 类型存储。
- k\_proj\_bias: Tensor 类型, 表示 key 偏置, 以 float32 类型存储。
- v\_proj\_weights: Tensor 类型, 表示 value 量化后权重, 以 int32 类型存储。
- v\_proj\_scales: Tensor 类型, 表示 value 量化缩放因子, 以 float32 类型存储。
- v\_proj\_zps: Tensor 类型, 表示 value 量化零点, 以 int32 类型存储。
- v\_proj\_bias: Tensor 类型, 表示 value 偏置, 以 float32 类型存储。
- o\_proj\_weights: Tensor 类型, 表示输出投影层量化后权重, 以 int32 类型存储。
- o\_proj\_scales: Tensor 类型, 表示输出投影层量化缩放因子, 以 float32 类型存储。
- o\_proj\_zps: Tensor 类型, 表示输出投影层量化零点, 以 int32 类型存储。
- o\_proj\_bias: Tensor 类型, 表示输出投影层偏置, 以 float32 类型存储。
- down\_proj\_weights: Tensor 类型, 表示降维投影层量化后权重, 以 int32 类型存储。
- down\_proj\_scales: Tensor 类型, 表示降维投影层量化缩放因子, 以 float32 类型存储。
- down\_proj\_zps: Tensor 类型, 表示降维投影层量化零点, 以 int32 类型存储。
- gate\_proj\_weights: Tensor 类型, 表示门投影层量化后权重, 以 int32 类型存储。
- gate\_proj\_scales: Tensor 类型, 表示门投影层量化缩放因子, 以 float32 类型存储。
- gate\_proj\_zps: Tensor 类型, 表示门投影层量化零点, 以 int32 类型存储。
- up\_proj\_weights: Tensor 类型, 表示升维投影层量化后权重, 以 int32 类型存储。
- up\_proj\_scales: Tensor 类型, 表示升维投影层量化缩放因子, 以 float32 类型存储。
- up\_proj\_zps: Tensor 类型, 表示升维投影层量化零点, 以 int32 类型存储。
- input\_layernorm\_weight: Tensor 类型, 表示对 input 做 layernorm 的权重, 以 int32 类型存储。

- post\_attention\_layernorm\_weight: Tensor 类型, 表示对 attention 层输出做 layernorm 的权重, 以 int32 类型存储。
- cos: List[Tensor] 型类型, 表示 cos 位置编码。
- sin: List[Tensor] 型类型, 表示 sin 位置编码。
- out\_dtype: string 类型, 表示输出张量的数据类型。
- group\_size: int 类型, 表示量化的 group 大小。
- weight\_bits: int 类型, 表示量化位宽, 仅支持 4bits/8bits。
- hidden\_size: int 类型, 表示 query/key/value 的 hidden\_size。
- rms\_norm\_eps: float 类型, 表示 layernorm 中的 eps 参数。
- num\_attention\_heads: int 类型, 表示注意力头的个数。
- num\_key\_value\_heads: int 类型, 表示 key/value 头的个数。
- mrope\_section: List[int] 类型, 表示位置编码的三个维度大小。
- quant\_method: str 类型, 表示量化方式, 目前仅支持 GPTQ 量化。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动产生名称。

## 返回值

返回 3 个 Tensor, 分别为激活输出、key cache、value cache, 数据类型为 out\_dtype。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16 类型。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16 类型。

### 24.5.17 Transform Operator

#### qwen2\_block\_cache

##### 接口定义

```
def qwen2_block_cache(hidden_states: Tensor,
                      position_ids: Tensor,
                      attention_mask: Tensor,
                      k_cache: Tensor,
                      v_cache: Tensor,
                      q_proj_weights: Tensor,
```

(续下页)

(接上页)

```
q_proj_scales: Tensor,
q_proj_zps: Tensor,
q_proj_bias: Tensor,
k_proj_weights: Tensor,
k_proj_scales: Tensor,
k_proj_zps: Tensor,
k_proj_bias: Tensor,
v_proj_weights: Tensor,
v_proj_scales: Tensor,
v_proj_zps: Tensor,
v_proj_bias: Tensor,
o_proj_weights: Tensor,
o_proj_scales: Tensor,
o_proj_zps: Tensor,
o_proj_bias: Tensor,
down_proj_weights: Tensor,
down_proj_scales: Tensor,
down_proj_zps: Tensor,
gate_proj_weights: Tensor,
gate_proj_scales: Tensor,
gate_proj_zps: Tensor,
up_proj_weights: Tensor,
up_proj_scales: Tensor,
up_proj_zps: Tensor,
input_layernorm_weight: Tensor,
post_attention_layernorm_weight: Tensor,
cos: List[Tensor],
sin: List[Tensor],
out_dtype: str = 'float16',
group_size: int = 128,
weight_bits: int = 4,
hidden_size: int = 3584,
rms_norm_eps: float = 1e-06,
num_attention_heads: int = 28,
num_key_value_heads: int = 4,
mrope_section: List[int] = [16, 24, 24],
quant_method: str = "gptq",
out_name: str = None
):
```

#pass

## 功能描述

qwen2 在 decode 阶段的一个 block layer。该操作属于 全局操作。

## 参数说明

- hidden\_states: Tensor 类型，表示激活值，形状为 (1, 1, hidden\_size)。
- position\_ids: Tensor 类型，表示位置索引，形状为 (3, 1, 1)。
- attention\_mask: Tensor 类型，表示注意力掩码，形状为 (1, 1, 1, seq\_length + 1)。
- k\_cache: Tensor 类型，表示 key cache，形状为 (1, seq\_length, num\_key\_value\_heads, head\_dim)。
- v\_cache: Tensor 类型，表示 value cache，形状为 (1, seq\_length, num\_key\_value\_heads, head\_dim)。
- q\_proj\_weights: Tensor 类型，表示 query 量化后权重，以 int32 类型存储。
- q\_proj\_scales: Tensor 类型，表示 query 量化缩放因子，以 float32 类型存储。
- q\_proj\_zps: Tensor 类型，表示 query 量化零点，以 int32 类型存储。
- q\_proj\_bias: Tensor 类型，表示 query 偏置，以 float32 类型存储。
- k\_proj\_weights: Tensor 类型，表示 key 量化后权重，以 int32 类型存储。
- k\_proj\_scales: Tensor 类型，表示 key 量化缩放因子，以 float32 类型存储。
- k\_proj\_zps: Tensor 类型，表示 key 量化零点，以 int32 类型存储。
- k\_proj\_bias: Tensor 类型，表示 key 偏置，以 float32 类型存储。
- v\_proj\_weights: Tensor 类型，表示 value 量化后权重，以 int32 类型存储。
- v\_proj\_scales: Tensor 类型，表示 value 量化缩放因子，以 float32 类型存储。
- v\_proj\_zps: Tensor 类型，表示 value 量化零点，以 int32 类型存储。
- v\_proj\_bias: Tensor 类型，表示 value 偏置，以 float32 类型存储。
- o\_proj\_weights: Tensor 类型，表示输出投影层量化后权重，以 int32 类型存储。
- o\_proj\_scales: Tensor 类型，表示输出投影层量化缩放因子，以 float32 类型存储。
- o\_proj\_zps: Tensor 类型，表示输出投影层量化零点，以 int32 类型存储。
- o\_proj\_bias: Tensor 类型，表示输出投影层偏置，以 float32 类型存储。
- down\_proj\_weights: Tensor 类型，表示降维投影层量化后权重，以 int32 类型存储。
- down\_proj\_scales: Tensor 类型，表示降维投影层量化缩放因子，以 float32 类型存储。
- down\_proj\_zps: Tensor 类型，表示降维投影层量化零点，以 int32 类型存储。
- gate\_proj\_weights: Tensor 类型，表示门投影层量化后权重，以 int32 类型存储。

- gate\_proj\_scales: Tensor 类型, 表示门投影层量化缩放因子, 以 float32 类型存储。
- gate\_proj\_zps: Tensor 类型, 表示门投影层量化零点, 以 int32 类型存储。
- up\_proj\_weights: Tensor 类型, 表示升维投影层量化后权重, 以 int32 类型存储。
- up\_proj\_scales: Tensor 类型, 表示升维投影层量化缩放因子, 以 float32 类型存储。
- up\_proj\_zps: Tensor 类型, 表示升维投影层量化零点, 以 int32 类型存储。
- input\_layernorm\_weight: Tensor 类型, 表示对 input 做 layernorm 的权重, 以 int32 类型存储。
- post\_attention\_layernorm\_weight: Tensor 类型, 表示对 attention 层输出做 layernorm 的权重, 以 int32 类型存储。
- cos: List[Tensor] 型类型, 表示 cos 位置编码。
- sin: List[Tensor] 型类型, 表示 sin 位置编码。
- out\_dtype: string 类型, 表示输出张量的数据类型。
- group\_size: int 类型, 表示量化的 group 大小。
- weight\_bits: int 类型, 表示量化位宽, 仅支持 4bits/8bits。
- hidden\_size: int 类型, 表示 query/key/value 的 hidden\_size。
- rms\_norm\_eps: float 类型, 表示 layernorm 中的 eps 参数。
- num\_attention\_heads: int 类型, 表示注意力头的个数。
- num\_key\_value\_heads: int 类型, 表示 key/value 头的个数。
- mrope\_section: List[int] 类型, 表示位置编码的三个维度大小。
- quant\_method: str 类型, 表示量化方式, 目前仅支持 GPTQ 量化。
- out\_name: string 类型或 None, 表示输出 Tensor 的名称, 为 None 时内部会自动生成名称。

## 返回值

返回 3 个 Tensor, 分别为激活输出、key cache、value cache, 数据类型为 out\_dtype。

## 处理器支持

- BM1684X: 输入数据类型可以是 FLOAT32,FLOAT16 类型。
- BM1688: 输入数据类型可以是 FLOAT32,FLOAT16 类型。