
TPU-MLIR 快速入门指南

发行版本 1.20-1-gcc08e8b8b

SOPHGO

2025 年 07 月 03 日

目录

1 TPU-MLIR 简介	3
2 开发环境配置	5
2.1 基础环境配置	5
2.2 安装 TPU-MLIR	6
2.3 安装 TPU-MLIR 依赖	6
3 编译 ONNX 模型	8
3.1 安装 TPU-MLIR	8
3.2 准备工作目录	9
3.3 ONNX 转 MLIR	9
3.4 MLIR 转 F16 模型	10
3.5 MLIR 转 INT8 模型	12
3.5.1 生成校准表	12
3.5.2 编译为 INT8 对称量化模型	12
3.6 效果对比	12
3.7 模型性能测试	14
3.7.1 安装 libraphon 环境	14
3.7.2 检查 BModel 的性能	14
4 编译 TORCH 模型	16
4.1 安装 TPU-MLIR	16
4.2 准备工作目录	16
4.3 TORCH 转 MLIR	17
4.4 MLIR 转 F16 模型	17
4.5 MLIR 转 INT8 模型	18
4.5.1 生成校准表	18
4.5.2 编译为 INT8 对称量化模型	18
4.6 效果对比	18
5 编译 Caffe 模型	20
5.1 安装 TPU-MLIR	20
5.2 准备工作目录	20
5.3 Caffe 转 MLIR	21
5.4 MLIR 转 F32 模型	21
5.5 MLIR 转 INT8 模型	22
5.5.1 生成校准表	22
5.5.2 编译为 INT8 对称量化模型	22

6 编译 TFLite 模型	23
6.1 安装 TPU-MLIR	23
6.2 准备工作目录	23
6.3 TFLite 转 MLIR	24
6.4 MLIR 转 INT8 模型	24
7 量化与量化调优	25
7.1 TPU-MLIR 全 int8 对称量化	25
7.1.1 run_calibration 流程介绍	26
7.1.2 run_calibration 参数介绍	26
7.1.3 run_calibration 参数使用介绍	27
7.2 TPU-MLIR 混合精度量化概述	31
7.3 pattern-match	31
7.3.1 YOLO 系列自动混精度方法	31
7.3.2 transformer 系列自动混精度方法	31
7.4 search_qtable	32
7.4.1 安装 TPU-MLIR	32
7.4.2 准备工作目录	32
7.4.3 测试 Float 和 INT8 对称量化模型分类效果	32
7.4.4 转成混精度量化模型	35
7.5 run_sensitive_layer	40
7.5.1 安装 TPU-MLIR	40
7.5.2 准备工作目录	40
7.5.3 测试 Float 和 INT8 对称量化模型分类效果	42
7.5.4 转成混精度量化模型	44
7.6 fp_forward	47
7.6.1 使用方法	47
7.6.2 参数说明	48
8 使用智能深度学习处理器做前处理	49
8.1 模型部署样例	50
8.1.1 BM1684X 部署	50
8.1.2 CV18xx 部署	51
9 使用智能深度学习处理器做后处理	52
9.1 检测模型后处理添加 (yolov5s)	52
9.1.1 准备工作目录	52
9.1.2 ONNX 转 MLIR	53
9.1.3 MLIR 转换成 BModel	54
9.1.4 模型验证	54
9.2 分割模型后处理添加 (yolov8s_seg)	55
9.2.1 准备工作目录	55
9.2.2 ONNX 转 MLIR	55
9.2.3 MLIR 转换成 BModel	56
9.2.4 模型验证	57
10 编译 LLM 模型	58
10.1 概述	58

10.2 命令行参数说明	58
10.3 示例用法	59
11 附录 01: 各框架模型转 ONNX 参考	60
11.1 PyTorch 模型转 ONNX	60
11.1.1 步骤 0: 创建工作目录	60
11.1.2 步骤 1: 搭建并保存模型	60
11.1.3 步骤 2: 导出 ONNX 模型	61
11.2 TensorFlow 模型转 ONNX	62
11.2.1 步骤 0: 创建工作目录	62
11.2.2 步骤 1: 准备并转换模型	62
11.3 PaddlePaddle 模型转 ONNX	62
11.3.1 步骤 0: 安装 openssl-1.1.1o	62
11.3.2 步骤 1: 创建工作目录	63
11.3.3 步骤 2: 准备模型	63
11.3.4 步骤 3: 转换模型	63
12 附录 02: CV18xx 使用指南	64
12.1 编译 yolov5 模型	64
12.1.1 安装 tpu-mlir	64
12.1.2 准备工作目录	64
12.1.3 ONNX 转 MLIR	65
12.1.4 MLIR 转 BF16 模型	65
12.1.5 MLIR 转 INT8 模型	66
12.1.6 效果对比	66
12.2 合并 cvimodel 模型文件	68
12.2.1 步骤 0: 生成 batch 1 的 cvimodel	68
12.2.2 步骤 1: 生成 batch 2 的 cvimodel	69
12.2.3 步骤 2: 合并 batch 1 和 batch 2 的 cvimodel	69
12.2.4 步骤 3: runtime 接口调用 cvimodel	69
12.2.5 综述: 合并过程	70
12.3 编译和运行 runtime sample	70
12.3.1 在 EVB 运行 release 提供的 sample 预编译程序	71
12.3.2 交叉编译 samples 程序	73
12.3.3 docker 环境仿真运行的 samples 程序	76
12.4 在开发板上进行模型测试及验证工作	77
12.5 FAQ	78
12.5.1 模型转换常见问题	78
12.5.2 模型评估常见问题	80
12.5.3 模型部署常见问题	81
12.5.4 其他常见问题	82
13 附录 03: BM168x 使用指南	84
13.1 合并 bmodel 模型文件	84
13.1.1 步骤 0: 生成 batch 1 的 bmodel	84
13.1.2 步骤 1: 生成 batch 2 的 bmodel	85
13.1.3 步骤 2: 合并 batch 1 和 batch 2 的 bmodel	86

13.1.4	综述: 合并过程	86
14	附录 04: Model-zoo 测试	87
14.1	注意事项	87
14.2	配置系统环境	87
14.3	获取 model-zoo 模型	88
14.4	准备运行环境	88
14.5	配置 SoC 设备	89
14.6	准备数据集	89
14.6.1	ImageNet	90
14.6.2	COCO (可选)	90
14.6.3	Vid4 (可选)	90
14.7	准备工具链编译环境	90
14.8	模型性能和精度测试流程	91
14.8.1	模型编译	91
14.8.2	性能测试	92
14.8.3	精度测试	93
14.9	FAQ	94
14.9.1	invalid command ‘bdist_wheel’	94
14.9.2	not a supported wheel	94
14.9.3	no module named ‘xxx’	95
14.9.4	精度测试因为内存不足被 kill	95
15	附录 05: TPU Profile 工具使用指南	96
15.1	编译 bmodel	96
15.2	生成 Profile 原始数据	97
15.3	可视化 Profile 数据	98
16	附录 06: TDB 调试工具使用指南	99
16.1	准备工作	99
16.2	启动 TDB	100
16.3	TDB 命令汇总	100
16.4	TDB 使用流程	101
16.5	TDB 功能说明	101
16.5.1	next 功能	101
16.5.2	breakpoint 功能	102
16.5.3	info 功能	103
16.5.4	print 功能	104
16.5.5	watchpoint 功能	104
16.5.6	py 功能	106
16.6	BModel Disassembler	106
16.7	BModel Checker	107
17	附录 07: 已支持的算子	112
17.1	本章节主要提供目前 TPU-MLIR 支持的算子列表	112



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

电话

010-57590723

发布记录

目录

版本	发布日期	说明
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->abcdgfg 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	支持 Caffe, 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	Copyright 版本号: 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, 其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模型文件 bmodel/cvimodel。代码已经开源到 [github: https://github.com/sophgo/tpu-mlir](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 PyTorch、ONNX、TFLite 和 Caffe。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>。

转模型需要在指定的 docker 执行, 主要分两步, 一是通过 `model_transform` 将原始模型转换成 mlir 文件, 二是通过 `model_deploy` 将 mlir 文件转换成 bmodel/cvimodel。

如果要转 INT8 模型, 则需要调用 `run_calibration` 生成校准表, 然后传给 `model_deploy`。

如果 INT8 模型不满足精度需要, 可以调用 `search_qtable` 生成量化表, 用来决定哪些层采用浮点计算, 然后传给 `model_deploy` 生成混精度模型。

本文通过简单的例子介绍 TPU-MLIR 是如何使用的。

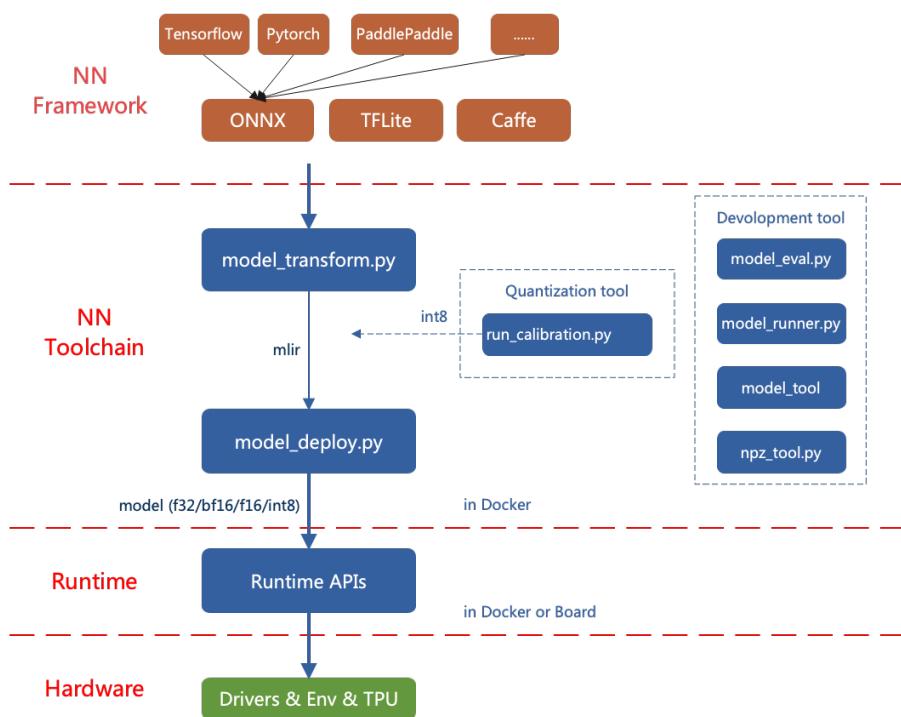


图 1.1: TPU-MLIR 的整体架构

CHAPTER 2

开发环境配置

首先检查当前系统环境是否满足 Ubuntu 22.04 和 Python 3.10。如不满足，请进行下一节基础环境配置；如满足，直接跳至[TPU-MLIR 安装](#)。

2.1 基础环境配置

如不满足上述系统环境，则需要使用 Docker，从 DockerHub https://hub.docker.com/r/sophgo/tpuc_dev 下载所需的镜像文件，或使用下方命令直接拉取镜像：

```
$ 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
```

若下载镜像文件，则需要确保镜像文件在当前目录，并在当前目录创建容器如下：

```
# 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  
$ docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.4
```

其中, myname 为容器名称, 可以自定义; \$PWD 为当前目录, 与容器的 /workspace 目录同步。

后文假定用户已经处于 docker 里面的 /workspace 目录。

2.2 安装 TPU-MLIR

目前支持 2 种安装方法, 分别是在线安装和离线安装。

在线安装

直接从 pypi 下载并安装, 默认安装最新版:

```
$ pip install tpu_mlir
```

离线安装

从 Github 的 Assets 处下载最新的 tpu_mlir-*-py3-none-any.whl, 然后使用 pip 安装:

```
$ pip install tpu_mlir-*py3-none-any.whl
```

2.3 安装 TPU-MLIR 依赖

TPU-MLIR 在对不同框架模型处理时所需的依赖不同, 在线安装和离线安装方式都需要安装额外依赖。

在线安装

在线安装方式对于 onnx 或 torch 生成的模型文件, 可使用下方命令安装额外的依赖环境:

```
# 安装onnx依赖  
$ pip install tpu_mlir[onnx]  
# 安装torch依赖  
$ pip install tpu_mlir[torch]
```

目前支持 5 种配置:

```
onnx, torch, tensorflow, caffe, paddle
```

可使用一条命令安装多个配置, 也可直接安装全部依赖环境:

```
# 同时安装onnx, torch, caffe依赖  
$ pip install tpu_mlir[onnx,torch,caffe]  
# 安装全部依赖  
$ pip install tpu_mlir[all]
```

离线安装

同理，离线安装方式可使用下方命令安装额外的依赖环境：

```
# 安装onnx依赖  
$ pip install tpu_mlir-*‑py3‑none‑any.whl[onnx]  
# 安装全部依赖  
$ pip install tpu_mlir-*‑py3‑none‑any.whl[all]
```

CHAPTER 3

编译 ONNX 模型

本章以 yolov5s.onnx 为例，介绍如何编译迁移一个 onnx 模型至深度学习处理器平台运行。

该模型来自 yolov5 的官网：<https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx>

本章需要安装 TPU-MLIR。

平台	文件名	说明
cv183x/cv182x/cv181x/cv180x	xxx.cvimodel	请参考: CV18xx 使用指南
其它	xxx.bmodel	继续本章节

3.1 安装 TPU-MLIR

进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[onnx]  
# or  
$ pip install tpu_mlir-* -py3-none-any.whl[onnx]
```

3.2 准备工作目录

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar
$ mv regression/ tpu-mlir-resource/
```

建立 model_yolov5s 目录，并把模型文件和图片文件都放入 model_yolov5s 目录中。

操作如下：

```
1 $ mkdir model_yolov5s && cd model_yolov5s
2 $ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ cp -rf tpu_mlir_resource/image .
5 $ mkdir workspace && cd workspace
```

3.3 ONNX 转 MLIR

如果模型是图片输入，在转模型之前我们需要了解模型的预处理。如果模型用预处理后的 npz 文件做输入，则不需要考虑预处理。

预处理过程用公式表达如下（ x 代表输入）：

$$y = (x - \text{mean}) \times \text{scale}$$

官网 yolov5 的图片是 rgb 格式，每个值会乘以 1/255，转换成 mean 和 scale 对应为 0.0,0.0,0.0 和 0.0039216,0.0039216,0.0039216。

模型转换命令如下：

```
$ model_transform \
--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
```

model_transform 主要参数说明如下（完整介绍请参见 TPU-MLIR 开发参考手册用户界面章节）：

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
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
input_types	否	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
resize_dims	否	原始图片需要 resize 之后的尺寸; 如果不指定, 则 resize 成模型的输入尺寸
keep_aspect_ratio	否	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 四种格式, 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, 默认为 nchw
output_names	否	指定输出的名称, 如果不指定, 则用模型的输出; 指定后用该指定名称做输出
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, 则不会进行正确性验证
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
mlir	是	指定输出的 mlir 文件名称和路径, .mlir 后缀

转成 mlir 文件后, 会生成一个 \${model_name}_in_f32.npz 文件, 该文件是模型的输入文件。

3.4 MLIR 转 F16 模型

将 mlir 文件转换成 f16 的 bmodel, 操作方法如下:

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

model_deploy 的主要参数说明如下 (完整介绍请参见 TPU-MLIR 开发参考手册用户界面章节) :

表 3.2: model_deploy 参数功能

参数名	必选 ?	说明
mlir	是	指定 mlir 文件
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。
processor	是	指定模型将要用到的平台, 支持 bm1690, bm1688, bm1684x, bm1684, cv186x, cv183x, cv182x, cv181x, cv180x
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
tolerance	否	表示 MLIR 量化后的结果与 MLIR fp32 推理结果相似度的误差容忍度
test_input	否	指定输入文件用于验证, 可以是 jpg 或 npy 或 npz; 可以不指定, 则不会进行正确性验证
test_reference	否	用于验证模型正确性的参考数据 (使用 npz 格式)。其为各算子的计算结果
compare_all	否	验证正确性时是否比较所有中间结果, 默认不比较中间结果
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
num_core	否	当 target 选择为 bm1688 时, 用于选择并行计算的 tpu 核心数量, 默认设置为 1 个 tpu 核心
skip_validation	否	跳过验证 bmodel 正确性环节, 用于提升模型部署的效率, 默认执行 bmodel 验证

对于不同处理器和支持的 quantize 类型对应关系如下表所示:

表 3.3: 不同处理器支持的 quantize 量化类型

处理器	支持的 quantize
BM1688	F32/F16/BF16/INT8/INT4
BM1684X	F32/F16/BF16/INT8
BM1684	F32/INT8
CV186X	F32/F16/BF16/INT8/INT4
CV183X/CV182X/CV181X/CV180X	BF16/INT8
BM1690	F32/F16/BF16/INT8/F8E4M3/F8E5M2

编译完成后, 会生成名为 yolov5s_1684x_f16.bmodel 的文件。

3.5 MLIR 转 INT8 模型

3.5.1 生成校准表

转 INT8 模型前需要跑 calibration，得到校准表；输入数据的数量根据情况准备 100~1000 张左右。

然后用校准表，生成对称或非对称 bmodel。如果对称符合需求，一般不建议用非对称，因为非对称的性能会略差于对称模型。

这里用现有的 100 张来自 COCO2017 的图片举例，执行 calibration：

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

运行完成后会生成名为 yolov5s_cali_table 的文件，该文件用于后续编译 INT8 模型的输入文件。

3.5.2 编译为 INT8 对称量化模型

转成 INT8 对称量化模型，执行如下命令：

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

编译完成后，会生成名为 yolov5s_1684x_int8_sym.bmodel 的文件。

3.6 效果对比

在本发布包中有用 python 写好的 yolov5 用例，使用 detect_yolov5 命令，用于对图片进行目标检测。

该命令对应源码路径 {package/path/to/tpu_mlir}/python/samples/detect_yolov5.py。

阅读该代码可以了解模型是如何使用的：先预处理得到模型的输入，然后推理得到输出，最后做后处理。

用以下代码分别来验证 onnx/f16/int8 的执行结果。

onnx 模型的执行方式如下，得到 dog_onnx.jpg：

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

f16 bmodel 的执行方式如下，得到 dog_f16.jpg：

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

int8 对称 bmodel 的执行方式如下，得到 dog_int8_sym.jpg：

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

对比结果如下：

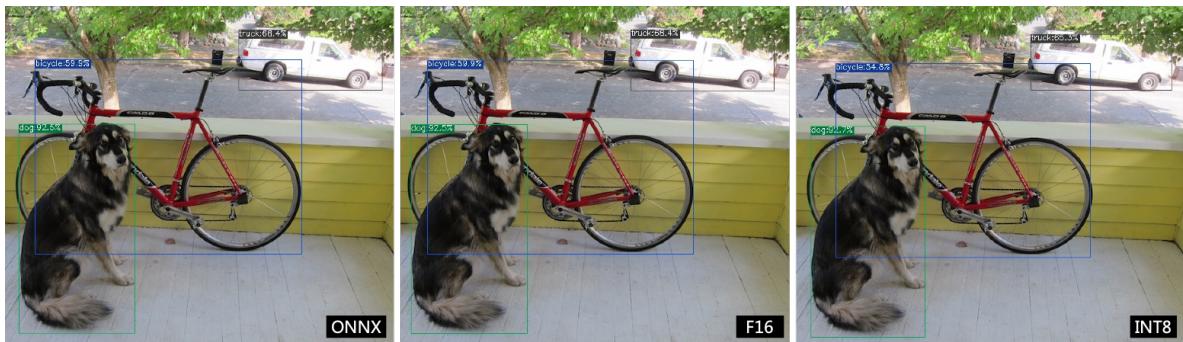


图 3.1: TPU-MLIR 对 YOLOv5s 编译效果对比

由于运行环境不同，最终的效果和精度与 图 3.1 会有些差异。

3.7 模型性能测试

以下操作需要在 Docker 外执行,

3.7.1 安装 libsophon 环境

请参考 [libsophon 使用手册](#)安装 libsophon。

3.7.2 检查 BModel 的性能

安装好 libsophon 后, 可以使用 `bmrt_test` 来测试编译出的 bmodel 的正确性及性能。可以根据 `bmrt_test` 输出的性能结果, 来估算模型最大的 fps, 来选择合适的模型。

```
# 下面测试上面编译出的bmodel
# --bmodel参数后面接bmodel文件,
$ cd path/to/model_yolov5s/workspace
$ bmrt_test --bmodel yolov5s_1684x_f16.bmodel
$ bmrt_test --bmodel yolov5s_1684x_int8_sym.bmodel
```

以最后一个命令输出为例 (此处对日志做了部分截断处理):

```
1 [BMRT][load_bmodel:983] INFO:pre net num: 0, load net num: 1
2 [BMRT][show_net_info:1358] INFO: #####
3 [BMRT][show_net_info:1359] INFO: NetName: yolov5s, Index=0
4 [BMRT][show_net_info:1361] INFO: ---- stage 0 ----
5 [BMRT][show_net_info:1369] INFO: Input 0) 'images' shape=[ 1 3 640 640 ] dtype=FLOAT32
6 [BMRT][show_net_info:1378] INFO: Output 0) '350_Transpose_f32' shape=[ 1 3 80 80 85 ] ...
7 [BMRT][show_net_info:1378] INFO: Output 1) '498_Transpose_f32' shape=[ 1 3 40 40 85 ] ...
8 [BMRT][show_net_info:1378] INFO: Output 2) '646_Transpose_f32' shape=[ 1 3 20 20 85 ] ...
9 [BMRT][show_net_info:1381] INFO: #####
10 [BMRT][bmrt_test:770] INFO:==> running network #0, name: yolov5s, loop: 0
11 [BMRT][bmrt_test:834] INFO:reading input #0, bytesize=4915200
12 [BMRT][print_array:702] INFO: --> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
13 [BMRT][bmrt_test:982] INFO:reading output #0, bytesize=6528000
14 [BMRT][print_array:702] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
15 [BMRT][bmrt_test:982] INFO:reading output #1, bytesize=1632000
16 [BMRT][print_array:702] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
17 [BMRT][bmrt_test:982] INFO:reading output #2, bytesize=408000
18 [BMRT][print_array:702] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
19 [BMRT][bmrt_test:1014] INFO:net[yolov5s] stage[0], launch total time is 4122 us (npu 4009[F]
→normal 113 us)
20 [BMRT][bmrt_test:1017] INFO:+++ The network[yolov5s] stage[0] output_data +++
21 [BMRT][print_array:702] INFO:output data #0 shape: [1 3 80 80 85 ] < 0.301003 ...
22 [BMRT][print_array:702] INFO:output data #1 shape: [1 3 40 40 85 ] < 0 0.228689 ...
23 [BMRT][print_array:702] INFO:output data #2 shape: [1 3 20 20 85 ] < 1.00135 ...
24 [BMRT][bmrt_test:1058] INFO:load input time(s): 0.008914
25 [BMRT][bmrt_test:1059] INFO:calculate time(s): 0.004132
```

(续下页)

(接上页)

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

从上面输出可以看到以下信息：

1. 05-08 行是 bmodel 的网络输入输出信息
2. 19 行是运行时间，其中深度学习处理器用时 4009us，非加速用时 113us。这里非加速用时主要是指在 HOST 端调用等待时间
3. 24 行是加载数据到 NPU 的 DDR 的时间
4. 25 行相当于 19 行的总时间
5. 26 行是输出数据取回时间

CHAPTER 4

编译 TORCH 模型

本章以 yolov5s.pt 为例，介绍如何编译迁移一个 pytorch 模型至 BM1684X 平台运行。

本章需要安装 TPU-MLIR。

4.1 安装 TPU-MLIR

进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[torch]  
# or  
$ pip install tpu_mlir-*py3-none-any.whl[torch]
```

4.2 准备工作目录

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar  
$ mv regression/ tpu-mlir-resource/
```

建立 model_yolov5s_pt 目录，并把模型文件和图片文件都放入 model_yolov5s_pt 目录中。
操作如下：

```

1 $ mkdir model_yolov5s_pt && cd model_yolov5s_pt
2 $ wget -O yolov5s.pt "https://github.com/sophgo/tpu-mlir/raw/master/regression/model/yolov5s.
  ↪pt"
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ cp -rf tpu_mlir_resource/image .
5 $ mkdir workspace && cd workspace

```

4.3 TORCH 转 MLIR

本例中的模型是 RGB 输入，mean 和 scale 分别为 0.0,0.0,0.0 和 0.0039216,0.0039216,0.0039216。

模型转换命令如下：

```

$ model_transform \
--model_name yolov5s_pt \
--model_def ./yolov5s.pt \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--test_input ./image/dog.jpg \
--test_result yolov5s_pt_top_outputs.npz \
--mlir yolov5s_pt.mlir

```

转成 mlir 文件后，会生成一个 \${model_name}_in_f32.npz 文件，该文件是模型的输入文件。值得注意的是，目前仅支持静态模型，模型在编译前需要调用 `torch.jit.trace()` 以生成静态模型。

4.4 MLIR 转 F16 模型

将 mlir 文件转换成 f16 的 bmodel，操作方法如下：

```

$ model_deploy \
--mlir yolov5s_pt.mlir \
--quantize F16 \
--processor bm1684x \
--test_input yolov5s_pt_in_f32.npz \
--test_reference yolov5s_pt_top_outputs.npz \
--model yolov5s_pt_1684x_f16.bmodel

```

编译完成后，会生成名为 `yolov5s_pt_1684x_f16.bmodel` 的文件。

4.5 MLIR 转 INT8 模型

4.5.1 生成校准表

转 INT8 模型前需要跑 calibration，得到校准表；这里用现有的 100 张来自 COCO2017 的图片举例，执行 calibration：

```
$ run_calibration yolov5s_pt.mlir \
--dataset ./COCO2017 \
--input_num 100 \
-o yolov5s_pt_cali_table
```

运行完成后会生成名为 `yolov5s_pt_cali_table` 的文件，该文件用于后续编译 INT8 模型的输入文件。

4.5.2 编译为 INT8 对称量化模型

转成 INT8 对称量化模型，执行如下命令：

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

编译完成后，会生成名为 `yolov5s_pt_1684x_int8_sym.bmodel` 的文件。

4.6 效果对比

利用 `detect_yolov5` 命令，对图片进行目标检测。

用以下代码分别来验证 pytorch/f16/int8 的执行结果。

pytorch 模型的执行方式如下，得到 `dog_torch.jpg`：

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

f16 bmodel 的执行方式如下，得到 `dog_f16.jpg`：

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

int8 对称 bmodel 的执行方式如下，得到 dog_int8_sym.jpg：

```
$ detect_yolov5 \
--input ..../image/dog.jpg \
--model yolov5s_pt_1684x_int8_sym.bmodel \
--output dog_int8_sym.jpg
```

对比结果如下：



图 4.1: TPU-MLIR 对 YOLOv5s 编译效果对比

由于运行环境不同，最终的效果和精度与 图 4.1 会有些差异。

CHAPTER 5

编译 Caffe 模型

本章以 mobilenet_v2_deploy.prototxt 和 mobilenet_v2.caffemodel 为例，介绍如何编译迁移一个 caffe 模型至 BM1684X 平台运行。

本章需要安装 TPU-MLIR。

5.1 安装 TPU-MLIR

进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[caffe]  
# or  
$ pip install tpu_mlir-*py3-none-any.whl[caffe]
```

5.2 准备工作目录

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar  
$ mv regression/ tpu-mlir-resource/
```

建立 mobilenet_v2 目录，并把模型文件和图片文件都放入 mobilenet_v2 目录中。

操作如下：

```

1 $ mkdir mobilenet_v2 && cd mobilenet_v2
2 $ wget https://raw.githubusercontent.com/shicai/MobileNet-Caffe/master/mobilenet_v2_deploy.
   ↪prototxt
3 $ wget https://github.com/shicai/MobileNet-Caffe/raw/master/mobilenet_v2.caffemodel
4 $ cp -rf tpu_mlir_resource/dataset/ILSVRC2012 .
5 $ cp -rf tpu_mlir_resource/image .
6 $ mkdir workspace && cd workspace

```

5.3 Caffe 转 MLIR

本例中的模型是 BGR 输入, mean 和 scale 分别为 103.94,116.78,123.68 和 0.017,0.017,0.017。
模型转换命令如下:

```

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

```

转成 mlir 文件后, 会生成一个 \${model_name}_in_f32.npz 文件, 该文件是模型的输入文件。

5.4 MLIR 转 F32 模型

将 mlir 文件转换成 f32 的 bmodel, 操作方法如下:

```

$ model_deploy \
--mlir mobilenet_v2.mlir \
--quantize F32 \
--processor bm1684x \
--test_input mobilenet_v2_in_f32.npz \
--test_reference mobilenet_v2_top_outputs.npz \
--model mobilenet_v2_1684x_f32.bmodel

```

编译完成后, 会生成名为 \${model_name}_1684x_f32.bmodel 的文件。

5.5 MLIR 转 INT8 模型

5.5.1 生成校准表

转 INT8 模型前需要跑 calibration, 得到校准表; 输入数据的数量根据情况准备 100~1000 张左右。

然后用校准表, 生成对称或非对称 bmodel。如果对称符合需求, 一般不建议用非对称, 因为非对称的性能会略差于对称模型。

这里用现有的 100 张来自 ILSVRC2012 的图片举例, 执行 calibration:

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

运行完成后会生成名为 \${model_name}_cali_table 的文件, 该文件用于后续编译 INT8 模型的输入文件。

5.5.2 编译为 INT8 对称量化模型

转成 INT8 对称量化模型, 执行如下命令:

```
$ model_deploy \
--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 0.96,0.70 \
--model mobilenet_v2_1684x_int8.bmodel
```

编译完成后, 会生成名为 \${model_name}_1684x_int8.bmodel 的文件。

CHAPTER 6

编译 TFLite 模型

本章以 lite-model_mobilebert_int8_1.tflite 模型为例，介绍如何编译迁移一个 TFLite 模型至 BM1684X 平台运行。

本章需要安装 TPU-MLIR。

6.1 安装 TPU-MLIR

进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[tensorflow]
# or
$ pip install tpu_mlir-*py3-none-any.whl[tensorflow]
```

6.2 准备工作目录

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar
$ mv regression/ tpu-mlir-resource/
```

建立 mobilebert_tf 目录，注意是与 tpu-mlir 同级目录；并把测试图片文件放入 mobilebert_tf 目录中。

操作如下：

```

1 $ mkdir mobilebert_tf && cd mobilebert_tf
2 $ wget -O lite-model_mobilebert_int8_1.tflite https://storage.googleapis.com/tfhub-lite-models/
   ↪iree/lite-model/mobilebert/int8/1.tflite
3 $ cp -rf tpu_mlir_resource/npz_input/squad_data.npz .
4 $ mkdir workspace && cd workspace

```

6.3 TFLite 转 MLIR

模型转换命令如下:

```

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

```

转成 mlir 文件后, 会生成一个 mobilebert_tf_in_f32.npz 文件, 该文件是模型的输入文件。

6.4 MLIR 转 INT8 模型

该模型是 tflite int8 模型, 可以按如下参数转成模型:

```

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

```

编译完成后, 会生成名为 mobilebert_tf_bm1684x_int8.bmodel 的文件。

量化与量化调优

神经网络在大规模部署时候，往往对吞吐量也就是推理时间有较高要求，硬件也专门对低比特计算进行了优化，其算力更加突出。所以以尽量高的精度进行低比特量化就显得尤为重要。但是要保持高精度和高吞吐率，网络往往需要以混合精度方式运行，即大部分算子以低比特定点计算，少部分以浮点进行计算。如何决定哪些算子使用浮点往往与网络和网络权重有直接关系，需要根据网络特点来选择。

TPU-MLIR 所采用的混合精度方式为搜索网络中不适于低比特量化的层生成 `quantize_table`，用以在 `model_deploy` 阶段指定这些层采用较高比特的量化方式。

本章首先会对 TPU-MLIR 当前的全 int8 对称量化进行介绍，然后对 TPU-MLIR 现有的 `quantize_table` 自动生成工具使用方式进行介绍。

7.1 TPU-MLIR 全 int8 对称量化

TPU-MLIR 默认采用全 int8 对称量化，全 int8 是指除编译器默认执行浮点运算的算子（如 `layernorm`）外，其余算子均进行 int8 量化。本节介绍如何使用 TPU-MLIR 全 int8 对称量化工具。

当按照之前教程通过 `model_transform` 命令将模型生成对应 mlir 文件之后，若要对模型进行 int8 对称量化，还需要通过 `run_calibration` 命令生成校准表 `cali_table`，对于不同类型模型该如何使用 `run_calibration` 命令的参数，从而使得生成的量化模型精度较好，下面将给出详细指导。

7.1.1 run_calibration 流程介绍

下图 (`run_calibration` 整体流程) 量化部分展示了当前 `run_calibration` 整体流程, 其中包括了自动混精模块 `search_qtable`, 自动校准方法选择模块 `search_threshold`, 跨层权重均衡模块 `weight_equalization` 以及偏置修正模块 `bias_correction` 等, 后面小节我们将结合实际情况给出上述方法的使用细节。

7.1.2 run_calibration 参数介绍

下表给出了 `run_calibration` 命令的参数介绍。

表 7.1: `run_calibration.py` 参数

参数	描述
<code>mlir_file</code>	<code>mlir</code> 文件
<code>sq</code>	开启 SmoothQuant
<code>we</code>	开启 <code>weight_equalization</code>
<code>bc</code>	开启 <code>bias_correction</code>
<code>dataset</code>	校准数据集
<code>data_list</code>	样本列表
<code>input_num</code>	校准样本数量
<code>inference_num</code>	<code>search_qtable</code> 和 <code>search_threshold</code> 推理过程所需图片数量, 默认为 30
<code>bc_inference_num</code>	<code>bias_correction</code> 推理过程所需图片数量, 默认为 30
<code>tune_list</code>	tuning 用到的样本列表
<code>tune_num</code>	tuning 的图像数量
<code>histogram_bin_num</code>	指定 kld 计算的直方图 bin 数量, 默认为 2048
<code>expected_cos</code>	期望 <code>search_qtable</code> 混精模型输出与浮点模型输出的相似度, 取值范围 [0,1], 默认为 0.99
<code>min_layer_cos</code>	<code>bias_correction</code> 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], 默认为 0.99
<code>max_float_layers</code>	<code>search_qtable</code> 设置浮点层数量, 默认为 5
<code>processor</code>	处理器类型, 默认为 <code>bm1684x</code>
<code>cali_method</code>	选择校准模式; 不添加该参数默认为 KLD 校准。“ <code>use_percentile9999</code> ”采用 99.99 分位作为门限。“ <code>use_max</code> ”采用绝对值最大值作为门限。“ <code>use_torch_observer_for_cali</code> ”采用 torch 的 observer 进行校准。” <code>use_mse</code> ”采用 octav 进行校准。
<code>fp_type</code>	<code>search_qtable</code> 浮点层数据类型
<code>post_process</code>	后处理路径
<code>global_compare_layers</code>	指定全局对比层, 例如 <code>layer1,layer2</code> 或 <code>layer1:0.3,layer2:0.7</code>
<code>search</code>	指定搜索类型, 其中包括 <code>search_qtable,search_threshold,false</code> 。其中默认为 <code>false</code> , 不开启搜索
<code>transformer</code>	是否是 transformer 模型, <code>search_qtable</code> 中如果是 transformer 模型可分配指定加速策略, 默认为 False

续下页

表 7.1 – 接上页

参数	描述
quantize_method_list	search_qtable 用来搜索的门限方法, 默认为 MSE, 可选择范围为 MSE, KL, MAX, Percentile9999
benchmark_method	指定 search_threshold 中相似度计算方法, 默认为 cos
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
debug_log	日志输出级别

7.1.3 run_calibration 参数使用介绍

根据用户需求以及用户对模型本身和量化的了解程度, 本节也针对性的给出了不同情况下 run_calibration 参数使用的方式。

表 7.2: run_calibration 参数适用情况

场景	描述	量化速度	校准方法	推荐方法
case1	模型初次量化	不敏感	不清楚	search_threshold
case2	模型初次量化	/	清楚	cali_method 直接选择对应校准方法
case3	模型初次量化	敏感	不清楚	cali_method 选择固定校准方法, 具体校准方法选择细节可看后续章节
case4	模型量化后在 bm1684 处理器上部署精度无法满足需求	/	/	开启 sq、we 和 bc 方法

case1: 当对您的模型进行初步量化时, 也就是第一次使用 run_calibration 命令, 此时您对当前模型所适应的校准方法并不清楚, 并且对量化速度并不敏感, 这里推荐您使用 search_threshold 方法, 该方法可以自动选择对应您当前模型最适合的校准方法, 并且输出该种方法生成的校准表 cali_table 到您指定的输出路径。同时也会生成一个 log 日志文件 Search_Threshold, 里面记录了不同校准方法的量化信息。具体操作如下:

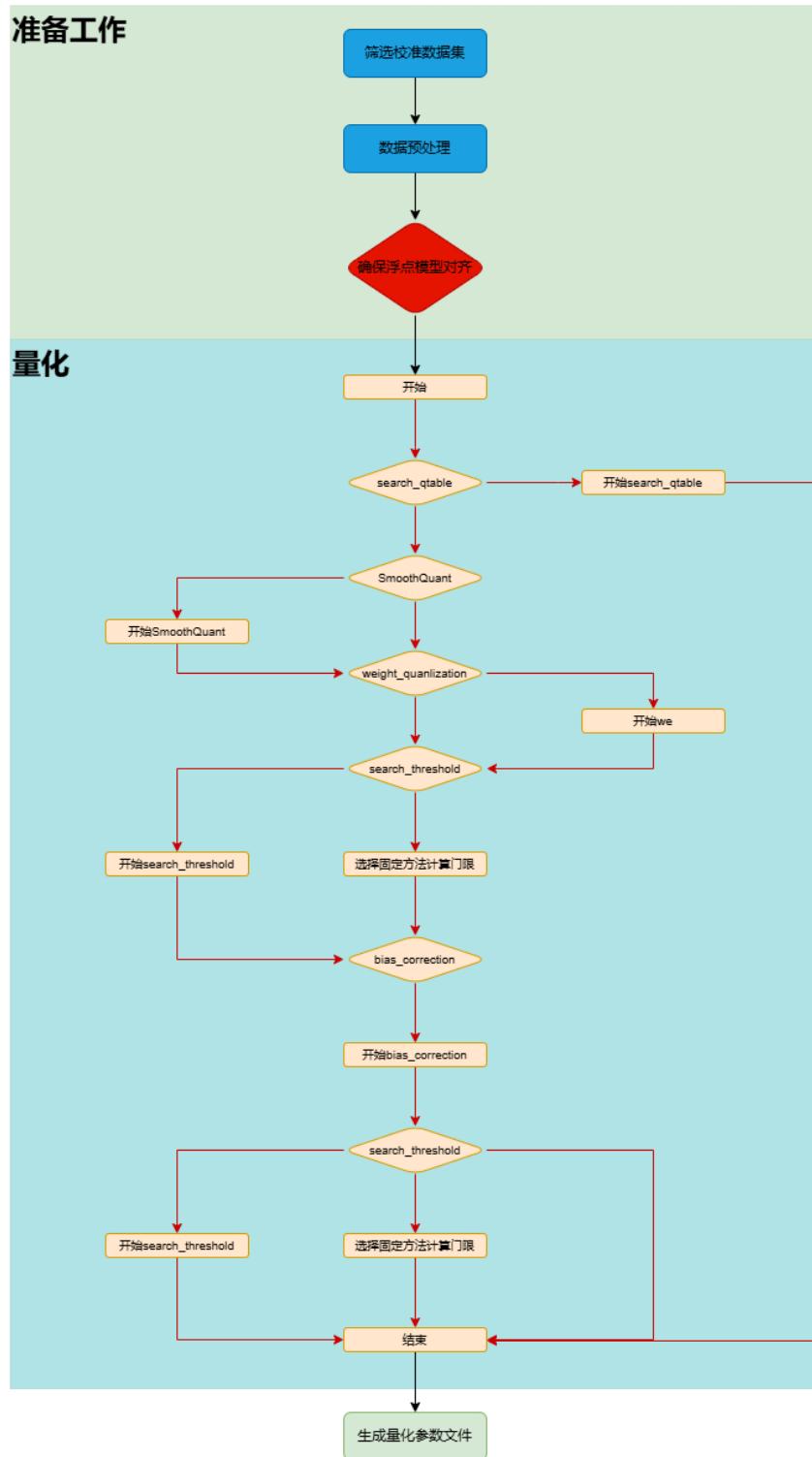


图 7.1: run_calibration 整体流程

```
$ run_calibration mlir.file \
--dataset data_path \
--input_num 100 \
--processor bm1684x \
--search search_threshold \
--inference_num 30 \
-o cali_table
```

注意事项:1. 此时需要选择 processor 参数, 该参数对应模型想要部署的处理器平台, 当前默认是 bm1684x。2. inference_num 对应 search_threshold 过程所需的推理数据数量(该数据将从您给定的 dataset 中抽取)。inference_num 越大, search_threshold 结果也更加准确, 但所需的量化时间也更长, 这里默认 inference_num 等于 30, 可根据实际情况自定义。

case2: 当对您的模型进行初步量化时您已经清楚该模型适合于何种校准方法, 此时可以直接根据 cali_method 参数去选择固定的校准方法。具体操作如下:

```
$ run_calibration mlir.file \
--dataset data_path \
--input_num 100 \
--cali_method use_mse \
-o cali_table
```

注意事项:1. 当不添加 cali_method 参数时, 此时将采用默认的 KLD 校准方法。2. 目前 cali_method 支持五种选择, 包括 use_mse, use_max, use_percentile9999, use_aciq_gauss 以及 use_aciq_laplace。

case3: 当您对量化时间比较敏感, 希望尽可能快的生成校准表 cali_table, 但您不清楚该如何选择校准方法时, 这里推荐您直接根据 cali_method 参数去选择固定的校准方法, 相比于 TPU-MLIR V1.8 版本的量化速度, V1.9 版本的单个校准方法量化速度提升 100%, 因此所需时间也平均降低到之前的 50% 左右, 加速效果明显。在 V1.9 版本校准方法中, use_mse 是平均量化速度最快的。对于校准方法的选择, 可以参考以下几点经验性的结论: 1. 当您的模型是不带有 attention 结构的非 transformer 模型, 可以选择 use_mse 校准方法。具体操作如下:

```
$ run_calibration mlir.file \
--dataset data_path \
--input_num 100 \
--cali_method use_mse \
-o cali_table
```

或者也可选择默认的 KLD 校准方法。具体操作如下:

```
$ run_calibration mlir.file \
--dataset data_path \
--input_num 100 \
-o cali_table
```

如果上述两种方法精度均不满足需求, 可能需要采取混合精度策略或者混合门限方法, 具体介绍可看后面小节。

2. 当您的模型是带有 attention 结构的 transformer 模型, 可以选择 use_mse 校准方法, 如果 use_mse 校准方法效果略差, 则可以尝试 use_max 校准方法, 具体操作如下:

```
$ run_calibration mlir.file \
--dataset data_path \
--input_num 100 \
--cali_method use_max \
-o cali_table
```

如果 `use_max` 效果也无法满足需求, 此时需要采取混合精度策略, 可依据后续介绍的混精方法进行尝试。

除去上面总体的选择规则, 也提供一些选择校准方法的细节:1. 如果您的模型是 yolo 系列的检测模型, 建议采取默认的 KLD 校准方法。2. 如果您的模型是有多个输出的分类模型, 建议采取默认的 KLD 校准方法。

case4: 当您的模型是部署在 bm1684 处理器上时, 如果通过上述方法获得的全 int8 量化模型精度较差, 可以尝试开启 SmoothQuant(sq)、跨层权重均衡 (we) 和偏置修正 (bc), 具体操作就是在原先的命令上面添加 `sq`、`we` 和 `bc` 参数。如果使用了 `search_threshold` 进行搜索, 添加 `sq`、`we` 和 `bc` 操作如下:

```
$ run_calibration mlir.file \
--sq \
--we \
--bc \
--dataset data_path \
--input_num 100 \
--processor bm1684 \
--search search_threshold \
--inference_num 30 \
--bc_inference_num 100 \
-o cali_table
```

如果使用 `cali_method` 选择固定校准方法, 下面以 `use_mse` 为例添加 `sq`、`we` 和 `bc` 方法, 具体操作如下:

```
$ run_calibration mlir.file \
--sq \
--we \
--bc \
--dataset data_path \
--input_num 100 \
--processor bm1684 \
--cali_method use_mse \
--bc_inference_num 100 \
-o cali_table
```

如果您采用的是默认的 KLD 校准方法, 去掉 `cali_method` 参数即可。

注意事项:1. 这里需要指定 `processor` 参数为 `bm1684`。2. `bc_inference_num` 参数是使用 `bc` 量化方法时所需的推理数据数量 (该数据将从您给定的 `dataset` 中抽取), 这里图片数量不应太少。3. `sq`、`we` 和 `bc` 方法可单独使用, 可以仅仅选择 `we` 方法, 在操作上直接去掉 `sq` 和 `bc` 参数即可。4. `run_calibration` 过程中会检查每个算子, 找到进行 `shape` 计算的算子在当前目录生成名为 `net_name_shape_ops` 的 `qtable`, 将这些算子设置为不量化, 里面内容可以手动和

下面混精的配置合并作为 qtable 用在 model_deploy 中。

7.2 TPU-MLIR 混合精度量化概述

TPU-MLIR 支持模型混合精度量化，其核心步骤在于获得记录算子名称及其量化类型的 quantize_table，后称 qtable。

TPU-MLIR 支持两种获取 qtable 的获取路径，对于典型模型，TPU-MLIR 提供基于经验的 pattern-match 方法。对于特殊模型或非典型模型，TPU-MLIR 提供三种基于检索的方法，分别为 search_qtable，run_sensitive_layer 和 fp_forward。在后续四个章节中会详细介绍上述四种方法工具。

7.3 pattern-match

pattern-match 方法集成于 run_calibration 中，不需要显示指定参数，当前共有两类模型提供经验 qtable，一类为 YOLO 系列，另一类为 BERT 等 Transformer 系列。在获得 cali_table 后，如果模型匹配上现有 pattern，则会在 path/to/cali_table/ 文件夹下生成 qtable。

7.3.1 YOLO 系列自动混合精度方法

当前共支持 YOLOV5,V6,V7,V8,V9,V10,11,12 系列模型。

YOLO 系列模型较为经典，使用广泛，在官方支持的模型导出时，通常会将数值差异较大的不同后处理分支合并输出，导致模型量化为全 INT8 精度损失大。由于 YOLO 系列模型通常具有相似结构特征，即三级 maxpool 结构，pattern-match 会自动判断模型是否属于 YOLO 系列，如是，进一步识别后处理部分算子，将这些算子设置为不量化，生成 qtable，该 qtable 可以手动和下面混精的配置合并作为 qtable 用在 model_deploy 中。以 yolov8 模型输出为例：

```

1  ['top.MaxPool', 'top.MaxPool', 'top.MaxPool', 'top.Concat'] (Name: yolo_block) is a subset of the [E]
  ↪main list. Count: 1
2  The [yolov6_8_9_11_12] post-processing pattern matches this model. Block count: 1
3  The [yolov6_8_9_11_12] post-processing pattern is: ['top.Sub', 'top.Add', 'top.Add', 'top.Sub',
  ↪'top.MulConst', 'top.Concat', 'top.Mul', 'top.Concat']
4  The qtable has been generated in: path/to/cali_table/qtable !!!

```

7.3.2 transformer 系列自动混合精度方法

当前共支持 BERT, EVA, DeiT, Swin, CSWin, ViT, DETR 系列模型。

如识别到上述模块，会将 Add 后的 LayerNorm,SiLU,GELU 算子设置为不量化。同时，ViT 会识别 Softmax/GELU 后的 MatMul 算子；EVA 会识别 Add,SiLU->Mul 后的 MatMul 算子；Swin 会识别 Add,Depth2Space 和 Reshape->LayerNorm 前的 Permute 算子；DeiT 会识别非 Conv,Scale,Reshape 及非 LayerNorm/Reshape 后的 MatMul 外所有算子。将这些算子设置为不量化，生成 qtable。

7.4 search_qtable

`search_qtable` 是集成于 `run_calibration` 中的混精功能, 当全 `int8` 量化精度无法满足需求时, 需要采用混合精度方法, 也就是将部分算子设置为浮点运算。`search_qtable` 是对于 `run_sensitive_layer` 的优化版本。相比 `run_sensitive_layer`, `search_qtable` 速度更快, 支持更多自定义参数。本节以检测网络 mobilenet-v2 网络模型为例, 介绍如何使用 `search_qtable`。本节需要安装 TPU-MLIR。

7.4.1 安装 TPU-MLIR

```
$ pip install tpu_mlir[all]
# or
$ pip install tpu_mlir-*-py3-none-any.whl[all]
```

7.4.2 准备工作目录

请从 Github 的 `Assets` 处下载 `tpu-mlir-resource.tar` 并解压, 解压后将文件夹重命名为 `tpu_mlir_resource` :

```
$ tar -xvf tpu-mlir-resource.tar
$ mv regression/ tpu-mlir-resource/
```

建立 `mobilenet-v2` 目录, 并把模型文件和图片文件都放入 `mobilenet-v2` 目录中。

操作如下:

```
1 $ mkdir mobilenet-v2 && cd mobilenet-v2
2 $ wget https://github.com/sophgo/tpu-mlir/releases/download/v1.4-beta.0/mobilenet_v2.pt
3 $ cp -rf tpu_mlir_resource/dataset/ILSVRC2012 .
4 $ mkdir workspace && cd workspace
```

7.4.3 测试 Float 和 INT8 对称量化模型分类效果

如上述章节介绍的转模型方法, 这里不做参数说明, 只有操作过程。

步骤 1: 转成 FP32 mlir

```
$ model_transform.py \
--model_name mobilenet_v2 \
--model_def ./mobilenet_v2.pt \
--input_shapes [[1,3,224,224]] \
--resize_dims 256,256 \
--mean 123.675,116.28,103.53 \
--scale 0.0171,0.0175,0.0174 \
```

(续下页)

(接上页)

```
--pixel_format rgb \
--mlir mobilenet_v2.mlir
```

步骤 2: 生成 calibartion table

这里我们采用 use_mse 方法进行校准。

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

步骤 3: 转 FP32 bmodel

```
$ model_deploy.py \
--mlir mobilenet_v2.mlir \
--quantize F32 \
--processor bm1684 \
--model mobilenet_v2_bm1684_f32.bmodel
```

步骤 4: 转对称量化模型

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

步骤 5: 验证 FP32 模型和 INT8 对称量化模型

classify_mobilenet_v2 是已经写好的验证程序，可以用来对 mobilenet_v2 网络进行验证。执行过程如下，FP32 模型：

```
$ classify_mobilenet_v2.py \
--model_def mobilenet_v2_bm1684_f32.bmodel \
--input ./ILSVRC2012/n02090379_7346.JPEG \
--output mobilenet_v2_fp32_bmodel.jpeg \
--category_file ./ILSVRC2012/synset_words.txt
```

在输出结果图片 mobilenet_v2_fp32_bmodel_1.jpeg 中，正确结果 sleeping bag 排在第一名：

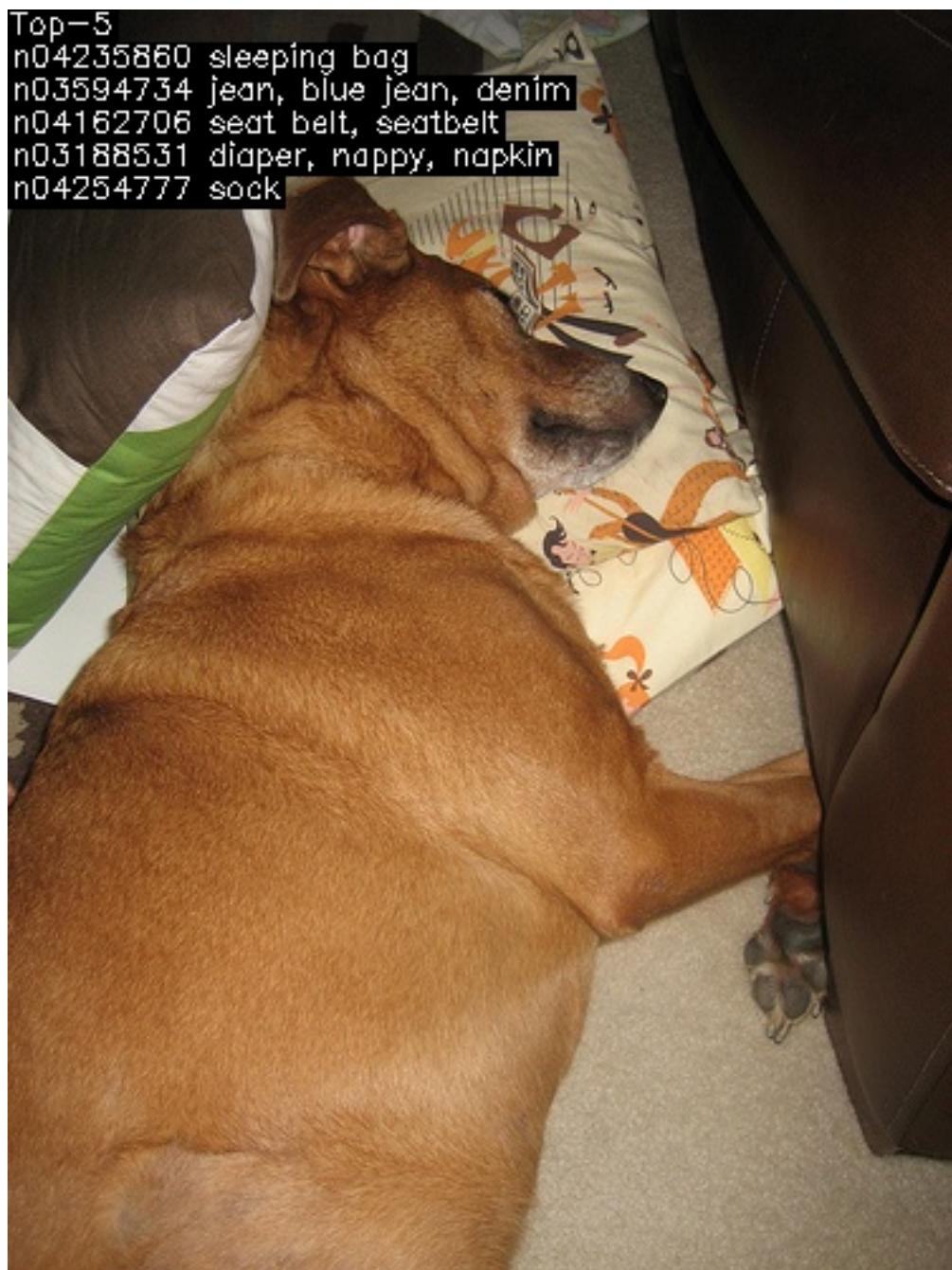


图 7.2: classify_mobilenet_v2 fp32 执行效果

INT8 对称量化模型：

```
$ classify_mobilenet_v2.py \
--model_def mobilenet_v2_bm1684_int8_sym.bmodel \
--input ..../ILSVRC2012/n02090379_7346.JPEG \
--output mobilenet_v2_INT8_sym_bmodel.jpeg \
--category_file ..../ILSVRC2012/synset_words.txt
```

在输出结果图片 `mobilenet_v2_INT8_sym_bmodel_1.jpeg` 中，正确结果 `sleeping bag` 排在第二名：

7.4.4 转成混精度量化模型

在转 int8 对称量化模型的基础上，执行如下步骤。

步骤 1：执行 `search_qtable` 命令

`search_qtable` 功能目前集成于 `run_calibration` 流程中，因此在使用时只需要在 `run_calibration` 命令中添加相关参数即可。`run_calibration` 中与 `search_qtable` 相关参数说明如下：



图 7.3: classify_mobilenet_v2 int8 执行效果

表 7.3: search_qtable 参数功能

参数名	必选 ?	说明
无	是	指定 mlir 文件
dataset	否	指定输入样本的目录, 该路径放对应的图片, 或 npz, 或 npy
data_list	否	指定样本列表, 与 dataset 必须二选一
processor	是	指定模型将要用到的平台, 支持 bm1690, bm1688, bm1684x, bm1684, cv186x, cv183x, cv182x, cv181x, cv180x
fp_type	否	指定混精度使用的 float 类型, 支持 auto,F16,F32,BF16, 默认为 auto, 表示由程序内部自动选择
input_num	是	指定用于量化的输入样本数量
inference_num	否	指定用于推理的输入样本数量, 默认用 30 个
max_float_layers	否	指定用于生成 qtable 的 op 数量, 默认用 5 个
tune_list	否	指定用于调整 threshold 的样本路径
tune_num	否	指定用于调整 threshold 的样本数量, 默认为 5
post_process	否	用户自定义后处理文件路径, 默认为空
expected_cos	否	指定期望网络最终输出层的最小 cos 值, 一般默认为 0.99 即可, 越小时可能会设置更多层为浮点计算
debug_cmd	否	指定调试命令字符串, 开发使用, 默认为空
global_compare_layers	否	指定用于替换最终输出层的层, 并用于全局比较, 例如: layer1,layer2 或 layer1:0.3,layer2:0.7
search	是	指定搜索类型, 其中包括 search_qtable, search_threshold ,false。这里需要选择 search_qtable
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, 默认是 False
quantize_method_list	否	search_qtable 用来搜索的校准方法, 默认仅使用 MSE 校准方法, 可选择 MSE, KL, MAX, Percentile9999
quantize_table	是	输出混精度量化表
calibration_table	是	校准表输出路径

search_qtable 支持用户自定义的后处理方法 post_process_func.py, 可以放在当前工程目录下, 也可以放在其他位置, 如果放在其他位置需要在 post_process 中指明文件的完整路径。后处理方法函数名称需要定义为 PostProcess, 输入数据为网络的输出, 输出数据为后处理结果。创建 post_process_func.py 文件, 其示例内容如下:

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

search_qtable 可以自定义混合门限的校准方法, 由参数 quantize_method_list 控制, 默认仅采用 MSE 校准方法进行搜索。当你想要使用 KLD 和 MSE 混合搜索时, 参数 quantize_method_list 输入 KL,MSE 即可。search_qtable 针对 transformer 模型设置了加速策略, 如果模型是带有 attention 结构的 transformer 模型, 可以设置参数 transformer 为 True。

使用 `search_qtable` 搜索损失较大的 layer, 注意尽量使用 bad cases 进行搜索。

本例中采用 100 张图片做量化,30 张图片做推理, 使用 KLD 和 MSE 校准方法混合搜索, 执行命令如下:

```
$ run_calibration.py mobilenet_v2.mlir \
--dataset ./ILSVRC2012 \
--input_num 100 \
--inference_num 30 \
--expected_cos 0.99 \
--quantize_method_list KL,MSE \
--search search_qtable \
--transformer False \
--processor bm1684 \
--post_process post_process_func.py \
--quantize_table mobilenet_v2_qtable \
--calibration_table mobilenet_v2_cali_table \
```

执行完后最后输出如下打印:

```
the layer input3.1 is 0 sensitive layer, loss is 0.004858517758037473, type is top.Conv
the layer input5.1 is 1 sensitive layer, loss is 0.002798812150635266, type is top.Scale
the layer input11.1 is 2 sensitive layer, loss is 0.0015642610676610547, type is top.Conv
the layer input13.1 is 3 sensitive layer, loss is 0.0009357141882855302, type is top.Scale
the layer input6.1 is 4 sensitive layer, loss is 0.0009211346574943269, type is top.Conv
the layer input2.1 is 5 sensitive layer, loss is 0.0007767164275293004, type is top.Scale
the layer input0.1 is 6 sensitive layer, loss is 0.0006842551513905892, type is top.Conv
the layer input128.1 is 7 sensitive layer, loss is 0.0003780628201499603, type is top.Conv
.....
run result:
int8 outputs_cos:0.986809 old
mix model outputs_cos:0.993372
Output mix quantization table to mobilenet_v2_qtable
total time:667.644282579422
success search qtable
```

上面 `int8 outputs_cos` 表示 int8 模型网络输出和 fp32 的 cos 相似度, `mix model outputs_cos` 表示前五个敏感层使用混精度后网络输出的 cos 相似度, `total time` 表示搜索时间为 667 秒, 另外, 生成的混精度量化表 `mobilenet_v2_qtable`, 内容如下:

```
# op_name  quantize_mode
input3.1 F32
input5.1 F32
input11.1 F32
input13.1 F32
input6.1 F32
```

该表中, 第一列表示相应的 layer, 第二列表示类型, 支持的类型有 F32/F16/BF16/INT8。`search_qtable` 会根据用户自定义的 `expected_cos` 参数值来确定混精度量化表中混精度层的数量, 举例来说, 如果 `expected_cos` 参数值等于 0.99, 那么混精度量化表中混精度层个数对应着混精度模型输出比对达到该水平的最小混精度层数, 当然混精度量化表中混精度层数会根据模型算子数量设置上限, 如果最小混精度层数超过上限, 那么只会取该上限对应的混精度层。与此同时, 也会生成

一个 log 日志文件 Search_Qtable, 内容如下:

```

1 INFO:root:quantize_method_list =['KL', 'MSE']
2 INFO:root:run float mode: mobilenet_v2.mlir
3 INFO:root:run int8 mode: mobilenet_v2.mlir
4 INFO:root:all_int8_cos=0.9868090914371674
5 INFO:root:run int8 mode: mobilenet_v2.mlir
6 INFO:root:layer name check pass !
7 INFO:root:all layer number: 117
8 INFO:root:all layer number no float: 116
9 INFO:root:transformer model: False, all search layer number: 116
10 INFO:root:Global metrics layer is : None
11 INFO:root:start to handle layer: input0.1, type: top.Conv
12 INFO:root:adjust layer input0.1 th, with method KL, and threshlod 9.442267236793155
13 INFO:root:run int8 mode: mobilenet_v2.mlir
14 INFO:root:outputs_cos_los = 0.0006842551513905892
15 INFO:root:adjust layer input0.1 th, with method MSE, and threshlod 9.7417731
16 INFO:root:run int8 mode: mobilenet_v2.mlir
17 INFO:root:outputs_cos_los = 0.0007242344141149548
18 INFO:root:layer input0.1, layer type is top.Conv, best_th = 9.442267236793155, best_method = F
  ↳KL, best_cos_loss = 0.0006842551513905892
19 .....

```

日志文件首先给出了自定义的参数, 包括混合门限所使用的校准方法 quantize_method_list, 要搜索的 op 数量 all search layer number 以及是否是 transformer model 等信息。然后记录了每个 op 在给定校准方法 (此处是 MSE 和 KL) 下得到的 threshold, 同时给出了在只对该 op 使用对应 threshold 做 int8 计算后的混精度模型与原始 float 模型输出的相似度的 loss(1-余弦相似度)。此外, 日志还包含了屏幕端输出的每个 op 的 loss 信息以及最后的混精度模型与原始 float 模型的余弦相似度。用户可以使用程序输出的 qtable, 也可以根据 loss 信息对 qtable 进行修改, 然后生成混精度模型。在 search_qtable 结束后, 最优的 threshold 会被更新到一个新的量化表 new_cali_table.txt, 该量化表存储在当前工程目录下, 在生成混精度模型时需要调用新量化表。

步骤 2: 生成混精度量化模型

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

步骤 3: 验证混精度模型

```
$ classify_mobilenet_v2 \
--model_def mobilenet_v2_bm1684_int8_mix.bmodel \
--input ..../ILSVRC2012/n02090379_7346.JPEG \
--output mobilenet_v2_INT8_mix_bmodel_1.jpeg \
--category_file ..../ILSVRC2012/synset_words.txt
```

在输出结果图片 `mobilenet_v2_INT8_mix_bmodel_1.jpeg` 中，正确结果 `sleeping bag` 排在第一名：

7.5 run_sensitive_layer

本节以检测网络 mobilenet-v2 网络模型为例，介绍如何使用敏感层搜索。

本节需要安装 TPU-MLIR。

7.5.1 安装 TPU-MLIR

```
$ pip install tpu_mlir[all]
# or
$ pip install tpu_mlir-* -py3-none-any.whl[all]
```

7.5.2 准备工作目录

请从 Github 的 Assets 处下载 `tpu-mlir-resource.tar` 并解压，解压后将文件夹重命名为 `tpu_mlir_resource`：

```
$ tar -xvf tpu-mlir-resource.tar
$ mv regression/ tpu-mlir-resource/
```

建立 `mobilenet-v2` 目录，并把模型文件和图片文件都放入 `mobilenet-v2` 目录中。

操作如下：

```
1 $ mkdir mobilenet-v2 && cd mobilenet-v2
2 $ wget https://github.com/sophgo/tpu-mlir/releases/download/v1.4-beta.0/mobilenet_v2.pt
3 $ cp -rf tpu_mlir_resource/dataset/ILSVRC2012 .
4 $ mkdir workspace && cd workspace
```



图 7.4: classify_mobilenet_v2 混精模型执行效果

7.5.3 测试 Float 和 INT8 对称量化模型分类效果

如前面章节介绍的转模型方法, 这里不做参数说明, 只有操作过程。

步骤 0: 转成 FP32 mlir

```
$ model_transform \
--model_name mobilenet_v2 \
--model_def ./mobilenet_v2.pt \
--input_shapes [[1,3,224,224]] \
--resize_dims 256,256 \
--mean 123.675,116.28,103.53 \
--scale 0.0171,0.0175,0.0174 \
--pixel_format rgb \
--mlir mobilenet_v2.mlir
```

步骤 1: 生成 calibartion table

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

步骤 2: 转 FP32 bmodel

```
$ model_deploy \
--mlir mobilenet_v2.mlir \
--quantize F32 \
--processor bm1684 \
--model mobilenet_v2_bm1684_f32.bmodel
```

步骤 3: 转对称量化模型

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

步骤 4: 验证 FP32 模型和 INT8 对称量化模型

classify_mobilenet_v2 是已经写好的验证程序，可以用来对 mobilenet_v2 网络进行验证。执行过程如下，FP32 模型：

```
$ classify_mobilenet_v2 \
--model_def mobilenet_v2_bm1684_fp32.bmodel \
--input ..../ILSVRC2012/n01440764_9572.JPEG \
--output mobilenet_v2_fp32_bmodel.jpeg \
--category_file ..../ILSVRC2012/synset_words.txt
```

在输出结果图片 mobilenet_v2_fp32_bmodel.jpeg 中，正确结果 tench 排在第一名：



图 7.5: classify_mobilenet_v2 fp32 执行效果

INT8 对称量化模型：

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

在输出结果图片 mobilenet_v2_INT8_sym_bmodel.jpeg 中，正确结果 tench 排在第一名：



图 7.6: classify_mobilenet_v2 int8 执行效果

7.5.4 转成混精度量化模型

在转 int8 对称量化模型的基础上，执行如下步骤。

步骤 0：进行敏感层搜索

使用 run_sensitive_layer 搜索损失较大的 layer，注意尽量使用 bad cases 进行敏感层搜索，相关参数说明如下：

表 7.4: run_sensitive_layer 参数功能

参数名	必选？	说明
无	是	指定 mlir 文件
dataset	否	指定输入样本的目录，该路径放对应的图片，或 npz，或 npy
data_list	否	指定样本列表，与 dataset 必须二选一
calibration_table	是	输入校准表
processor	是	指定模型将要用到的平台，支持 bm1690, bm1688, bm1684x, bm1684, cv186x, cv183x, cv182x, cv181x, cv180x
fp_type	否	指定混精度使用的 float 类型，支持 auto,F16,F32,BF16， 默认为 auto，表示由程序内部自动选择
input_num	否	指定用于量化的输入样本数量，默认用 10 个
inference_num	否	指定用于推理的输入样本数量，默认用 10 个
max_float_layers	否	指定用于生成 qtable 的 op 数量，默认用 5 个
tune_list	否	指定用于调整 threshold 的样本路径
tune_num	否	指定用于调整 threshold 的样本数量， 默认为 5
histogram_bin_num	否	指定用于 kld 方法中使用的 bin 数量， 默认为 2048
post_process	否	用户自定义后处理文件路径， 默认为空
expected_cos	否	指定期望网络最终输出层的最小 cos 值，一般默认为 0.99 即可，越小时可能会设置更多层为浮点计算
debug_cmd	否	指定调试命令字符串，开发使用， 默认为空
o	是	输出混精度量化表
global_compare_layers	否	指定用于替换最终输出层的层，并用于全局比较，例如：layer1,layer2 或 layer1:0.3,layer2:0.7
fp_type	否	指定混合精度的浮点类型

敏感层搜索支持用户自定义的后处理方法 post_process_func.py，可以放在当前工程目录下，也可以放在其他位置，如果放在其他位置需要在 post_process 中指明文件的完整路径。后处

本例中采用 100 张图片做量化, 30 张图片做推理, 执行命令如下:

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

执行完后最后输出如下打印:

```
the layer input3.1 is 0 sensitive layer, loss is 0.008808857469573828, type is top.Conv
the layer input11.1 is 1 sensitive layer, loss is 0.0016958347875666302, type is top.Conv
the layer input128.1 is 2 sensitive layer, loss is 0.0015641432811860367, type is top.Conv
the layer input130.1 is 3 sensitive layer, loss is 0.0014325751094084183, type is top.Scale
the layer input127.1 is 4 sensitive layer, loss is 0.0011817314259702227, type is top.Add
the layer input13.1 is 5 sensitive layer, loss is 0.001018420214596527, type is top.Scale
the layer 787 is 6 sensitive layer, loss is 0.0008603856180608993, type is top.Scale
the layer input2.1 is 7 sensitive layer, loss is 0.0007558935451825732, type is top.Scale
the layer input119.1 is 8 sensitive layer, loss is 0.000727441637624282, type is top.Add
the layer input0.1 is 9 sensitive layer, loss is 0.0007138056757098887, type is top.Conv
the layer input110.1 is 10 sensitive layer, loss is 0.000662179506136229, type is top.Conv
.....
run result:
int8 outputs_cos:0.978803 old
mix model outputs_cos:0.989258
Output mix quantization table to mobilenet_v2_qtable
total time:402.15848112106323
success sensitive layer search
```

上面 int8 outputs_cos 表示 int8 模型原本网络输出和 fp32 的 cos 相似度, mix model outputs_cos 表示前五个敏感层使用混精度后网络输出的 cos 相似度, total time 表示搜索时间为 402 秒, 另外, 生成的混精度量化表 mobilenet_v2_qtable, 内容如下:

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

该表中, 第一列表示相应的 layer, 第二列表示类型, 支持的类型有 F32/F16/BF16/INT8。与此同时, 也会生成一个 log 日志文件 SensitiveLayerSearch, 内容如下:

```
1 INFO:root:start to handle layer: input3.1, type: top.Conv
2 INFO:root:adjust layer input3.1 th, with method MAX, and threshlod 5.5119305
3 INFO:root:run int8 mode: mobilenet_v2.mlir
4 INFO:root:outputs_cos_loss = 0.014830573787862011
5 INFO:root:adjust layer input3.1 th, with method Percentile9999, and threshlod 4.1202815
```

(续下页)

(接上页)

```

6 INFO:root:run int8 mode: mobilenet_v2.mlir
7 INFO:root:outputs_cos_los = 0.011843443367980822
8 INFO:root:adjust layer input3.1 th, with method KL, and threshlod 2.6186381997094728
9 INFO:root:run int8 mode: mobilenet_v2.mlir
10 INFO:root:outputs_cos_los = 0.008808857469573828
11 INFO:root:layer input3.1, layer type is top.Conv, best_th = 2.6186381997094728, best_method = F
   ↳KL, best_cos_loss = 0.008808857469573828

```

日志文件记录了每个 Op 在不同量化方法 (MAX/Percentile9999/KL) 下得到的 threshold, 同时给出了在只对该 Op 使用对应 threshold 做 int8 计算后的混精度模型与原始 float 模型输出的相似度的 loss (1-余弦相似度)。此外, 日志还包含了屏幕端输出的每个 op 的 loss 信息以及最后的混精度模型与原始 float 模型的余弦相似度。用户可以使用程序输出的 qtable, 也可以根据 loss 信息对 qtable 进行修改, 然后生成混精度模型。在敏感层搜索结束后, 最优的 threshold 会被更新到一个新的量化表 new_cali_table.txt, 该量化表存储在当前工程目录下, 在生成混精度模型时需要调用新量化表。在本例中, 根据输出的 loss 信息, 观察到 input3.1 的 loss 比其他 op 高很多, 可以在 qtable 中只设置 input3.1 为 FP32。

步骤 1: 生成混精度量化模型

```
$ model_deploy \
--mlir mobilenet_v2.mlir \
--quantize INT8 \
--processor bm1684 \
--calibration_table new_cali_table.txt \
--quantize_table mobilenet_v2_qtable \
--model mobilenet_v2_bm1684_mix.bmodel
```

步骤 2: 验证混精度模型

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

在输出结果图片上可以看到如下分类信息, 可以看出混精度后, 正确结果 tench 排到了第一名。

```
Top-5
n01440764 tench, Tinca tinca
n02749479 assault rifle, assault gun
n02916936 bulletproof vest
n02536864 coho, cohoe, coho salmon, blue jack, silver salmon, Oncorhynchus kisutch
n04090263 rifle
```

7.6 fp_forward

对于特定网络，部分层由于数据分布差异大，量化成 INT8 会大幅降低模型精度，使用局部不量化功能，可以一键将部分层之前、之后、之间添加到混精度表中，在生成混精度模型时，这部分层将不被量化。

7.6.1 使用方法

本节将沿用第三章提到的 yolov5s 网络的例子，介绍如何使用局部不量化功能，快速生成混精度模型。

生成 FP32 和 INT8 模型的过程与第三章相同，下面仅介绍精度测试方案与混精度流程。

对于 yolo 系列模型来说，最后三个卷积层由于数据分布差异较大，常常手动添加混精度表以提升精度。使用局部不量化功能，从 model_transform 生成的 Top 层 mlir 文件搜索到对应的层。快速添加混精度表。

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

点开 yolov5s_qtable 可以看见相关层都被加入到 qtable 中。

生成混精度模型

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

验证 FP32 模型和混精度模型的精度 model-zoo 中有对目标检测模型进行精度验证的程序 yolo，可以在 mlir.config.yaml 中使用 harness 字段调用 yolo：

相关字段修改如下

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

harness:
  type: yolo
```

(续下页)

(接上页)

```
args:
- name: FP32
  bmodel: ${workdir}/${name}_bm1684_f32.bmodel
- name: INT8
  bmodel: ${workdir}/${name}_bm1684_int8_sym.bmodel
- name: mix
  bmodel: ${workdir}/${name}_bm1684_mix.bmodel
```

切换到 model-zoo 顶层目录，使用 tpu_perf.precision_benchmark 进行精度测试，命令如下：

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

执行完后，精度测试的结果存放在 output/yolo.csv 中：

FP32 模型 mAP 为：37.14%

INT8 模型 mAP 为：34.70%

混精度模型 mAP 为：36.18%

在 yolov5 以外的检测模型上，使用混精度的方式常会有更明显的效果。

7.6.2 参数说明

表 7.5: fp_forward 参数功能

参数名	必选？	说明
无	是	指定 mlir 文件
processor	是	指定模型将要用到的平台，支持 bm1690, bm1688, bm1684x, bm1684, cv186x, cv183x, cv182x, cv181x, cv180x
fpfwd_inputs	否	指定层（包含本层）之前不执行量化，多输入用，间隔
fpfwd_outputs	否	指定层（包含本层）之后不执行量化，多输入用，间隔
fpfwd_blocks	否	指定起点和终点之间的层不执行量化，起点和终点之间用 : 间隔，多个 block 之间用空格间隔
fp_type	否	指定混精度使用的 float 类型，支持 auto,F16,F32,BF16，默认为 auto，表示由程序内部自动选择
o	是	输出混精度量化表

CHAPTER 8

使用智能深度学习处理器做前处理

目前 TPU-MLIR 支持的两个主要系列 BM168x (除 BM1684 外) 与 CV18xx 均支持将图像常见的预处理加入到模型中进行计算。开发者可以在模型编译阶段，通过编译选项传递相应预处理参数，由编译器直接在模型运算前插入相应前处理算子，生成的 bmodel 或 cvimodel 即可以直接以预处理前的图像作为输入，随模型推理过程使用深度学习处理器处理前处理运算。

表 8.1: 预处理类型支持情况

预处理类型	BM168x	CV18xx
图像裁剪	True	True
归一化计算	True	True
NHWC to NCHW	True	True
BGR/RGB 转换	True	True

其中图像裁剪会先将图片按使用 `model_transform` 工具时输入的 “`-resize_dims`” 参数将图片调整为对应的大小，再裁剪成模型输入的尺寸。而归一化计算支持直接将未进行预处理的图像数据 (即 `unsigned int8` 格式的数据) 做归一化处理。

若要将预处理融入到模型中，则需要在使用 `model_deploy` 工具进行部署时使用 “`-fuse_preprocess`” 参数。如果要做验证，则传入的 `test_input` 需要是图像原始格式的输入 (即 jpg, jpeg 和 png 格式)，相应地会生成原始图像输入对应的 npz 文件，名称为 `${model_name}_in_ori.npz`。

此外，当实际外部输入格式与模型的格式不相同时，用 “`-customization_format`” 指定实际的外部输入格式，支持的格式说明如下 (以下支持情况不包含 BM1684)：

表 8.2: customization_format 格式和说明

customization_format	说明	BM168X	CV18xx
None	与原始模型输入保持一致, 不做处理。默认	True	True
RGB_PLANAR	rgb 顺序, 按照 nchw 摆放	True	True
RGB_PACKED	rgb 顺序, 按照 nhwc 摆放	True	True
BGR_PLANAR	bgr 顺序, 按照 nchw 摆放	True	True
BGR_PACKED	bgr 顺序, 按照 nhwc 摆放	True	True
GRAYSCALE	仅有一个灰色通道, 按 nchw 摆放	True	True
YUV420_PLANAR	yuv420 planner 格式, 来自 vpss 的输入	True	True
YUV_NV21	yuv420 的 NV21 格式, 来自 vpss 的输入	True	True
YUV_NV12	yuv420 的 NV12 格式, 来自 vpss 的输入	True	True
RGBA_PLANAR	rgba 格式, 按照 nchw 摆放	False	True

注意, BM168X 模型中 YUV 格式的输入数据形状是 (n, resize_dim_h, resize_dim_w) , resize_dim_h,resize_dim_w 为 model_transform `` 阶段的 ``resize_dim 参数。

当 customization_format 中颜色通道的顺序与模型输入不同时, 将会进行通道转换操作。若指令中未设置 customization_format 参数, 则根据使用 model_transform `` 工具时定义的 ``pixel_format 和 channel_format 参数自动获取对应的 customization_format 。

8.1 模型部署样例

以 mobilenet_v2 模型为例, 参考“编译 Caffe 模型”章节, 使用 model_transform 工具生成原始 mlir, 并通过 run_calibration 工具生成校准表。

8.1.1 BM168X 部署

生成融合预处理的 INT8 对称量化 bmodel 模型指令如下:

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

8.1.2 CV18xx 部署

生成融合预处理的 INT8 对称量化 cvimodel 模型的指令如下:

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

VPSS 作为输入

当输入数据是来自于 CV18xx 提供的视频后处理模块 VPSS 时 (使用 VPSS 进行预处理的详细使用方法请参阅《CV18xx 媒体软件开发参考》, 本文档不做介绍), 则会有数据对齐要求, 比如 w 按照 32 字节对齐, 此时 fuse_preprocess, aligned_input 需要同时被设置, 生成融合预处理的 cvimodel 模型的指令如下:

```
$ model_deploy \
--mlir mobilenet_v2.mlir \
--quantize INT8 \
--calibration_table mobilenet_v2_cali_table \
--processor cv183x \
--test_input ..image/cat.jpg \
--test_reference mobilenet_v2_top_outputs.npz \
--tolerance 0.96,0.70 \
--fuse_preprocess \
--customization_format YUV_NV21 \
--aligned_input \
--model mobilenet_v2_cv183x_int8_sym_fuse_preprocess_aligned.cvimodel
```

上述指令中, aligned_input 指定了模型需要做输入的对齐。

值得注意的是: vpss 做输入, runtime 可以使用 CVI_NN_SetTensorPhysicalAddr 减少数据的拷贝。

使用智能深度学习处理器做后处理

目前 TPU-MLIR 支持将 yolo 系列和 ssd 网络模型的后处理集成到模型中，目前支持该功能的处理器有 BM1684X、BM1688、CV186X、BM1690。

本章将 yolov5s 和 yolov8s_seg 转成为 F16 模型为例，介绍该功能如何被使用。

本章需要安装 TPU-MLIR 进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[onnx]  
# or  
$ pip install tpu_mlir-*~py3-none-any.whl[onnx]
```

请从 Github 的 [Assets](#) 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar  
$ mv regression/ tpu-mlir-resource/
```

9.1 检测模型后处理添加 (yolov5s)

9.1.1 准备工作目录

建立 model_yolov5s 目录，并把模型文件和图片文件都放入 model_yolov5s 目录中。

操作如下：

```
1 $ mkdir yolov5s_onnx && cd yolov5s_onnx  
2 $ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
```

(续下页)

(接上页)

```

3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ cp -rf tpu_mlir_resource/image .
5 $ mkdir workspace && cd workspace

```

9.1.2 ONNX 转 MLIR

模型转换命令如下：

```

$ model_transform \
--model_name yolov5s \
--model_def ./yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 326,474,622 \
--add_postprocess yolov5 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s.mlir

```

这里要注意两点，一是命令中需要加入 `--add_postprocess` 参数；二是指定的 `--output_names` 对应最后的卷积操作。

生成后的 `yolov5s.mlir` 文件最后被插入了一个 `top.YoloDetection`，如下：

```

1 %260 = "top.Weight"() : () -> tensor<255x512x1x1xf32> loc(#loc261)
2 %261 = "top.Weight"() : () -> tensor<255xf32> loc(#loc262)
3 %262 = "top.Conv"(%253, %260, %261) {dilations = [1, 1], do_relu = false, group = 1 : i64, F
  ↪ kernel_shape = [1, 1], pads = [0, 0, 0, 0], relu_limit = -1.000000e+00 : f64, strides = [1, 1]} : F
  ↪ (tensor<1x512x20x20xf32>, tensor<255x512x1x1xf32>, tensor<255xf32>) -> tensor
  ↪ <1x255x20x20xf32> loc(#loc263)
4 %263 = "top.YoloDetection"(%256, %259, %262) {agnostic_nms = false, anchors = [10, 13, 16, 30,
  ↪ 33, 23, 30, 61, 62, 45, 59, 119, 116, 90, 156, 198, 373, 326], class_num = 80 : i64, keep_topk = F
  ↪ 200 : i64, net_input_h = 640 : i64, net_input_w = 640 : i64, nms_threshold = 5.000000e-01 : F
  ↪ f64, num_boxes = 3 : i64, obj_threshold = 5.000000e-01 : f64, version = "yolov5"} : (tensor
  ↪ <1x255x80x80xf32>, tensor<1x255x40x40xf32>, tensor<1x255x20x20xf32>) -> tensor
  ↪ <1x1x200x7xf32> loc(#loc264)
5 return %263 : tensor<1x1x200x7xf32> loc(#loc)

```

这里看到 `top.YoloDetection` 包括了 `anchors`、`num_boxes` 等等参数，如果并非标准的 `yolo` 后处理，需要改成其他参数，可以直接修改 `mlir` 文件的这些参数。

另外输出也变成了 1 个，`shape` 为 `1x1x200x7`，其中 200 代表最大检测框数，当有多个 batch 时，它的数值会变为 `batch x 200`；7 分别指 `[batch_number, class_id, score, center_x, center_y, width, height]`。其中坐标是相对模型输入长宽的坐标，比如本例中 `640x640`，数值参考如下：

```
[0., 16., 0.924488, 184.21094, 401.21973, 149.66412, 268.50336 ]
```

9.1.3 MLIR 转换成 BModel

将 mlir 文件转换成 F16 的 bmodel, 操作方法如下:

```
$ model_deploy \
--mlir yolov5s.mlir \
--quantize F16 \
--processor bm1684x \
--fuse_preprocess \
--test_input ./image/dog.jpg \
--test_reference yolov5s_top_outputs.npz \
--model yolov5s_1684x_f16.bmodel
```

这里加上参数 --fuse_preprocess, 是为了将前处理也合并到模型中。这样转换后的模型就是包含了前后处理的模型, 用 model_tool 查看模型信息如下:

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

```
1 bmodel version: B.2.2
2 processor: BM1684X
3 create time: Wed Jan 3 07:29:14 2024
4
5 kernel_module name: libbm1684x_kernel_module.so
6 kernel_module size: 2677600
7 =====
8 net 0: [yolov5s] static
9 -----
10 stage 0:
11 subnet number: 2
12 input: images_raw, [1, 3, 640, 640], uint8, scale: 1, zero_point: 0
13 output: yolo_post, [1, 1, 200, 7], float32, scale: 1, zero_point: 0
14
15 device mem size: 31238060 (coeff: 14757888, instruct: 124844, runtime: 16355328)
16 host mem size: 0 (coeff: 0, runtime: 0)
```

这里的 [1, 1, 200, 7] 是最大 shape, 实际输出根据检测的框数有所不同。

9.1.4 模型验证

在本发布包中有用 python 写好的 yolov5 用例, 使用 detect_yolov5 命令, 用于对图片进行目标检测。该命令对应源码路径 {package/path/to/tpu_mlir}/python/samples/detect_yolov5.py。阅读该代码可以了解最终输出结果是怎么转换画框的。

命令执行如下:

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

9.2 分割模型后处理添加 (yolov8s_seg)

9.2.1 准备工作目录

建立 model_yolov8s_seg 目录，使用官方模型导出 onnx 模型文件，并将图片文件放入 model_yolov8s_seg 目录中。

操作如下：

```
1 $ mkdir yolov8s_seg_onnx && cd yolov8s_seg_onnx
2 $ python -c "import torch; from ultralytics import YOLO; model = YOLO('yolov8s-seg.pt'); model.
   →export(format='onnx')"
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ cp -rf tpu_mlir_resource/image .
5 $ mkdir workspace && cd workspace
```

9.2.2 ONNX 转 MLIR

模型转换命令如下：

```
$ model_transform \
--model_name yolov8s_seg \
--model_def ../yolov8s-seg.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 \
--add_postprocess yolov8_seg \
--test_input ../image/dog.jpg \
--test_result yolov8s_seg_top_outputs.npz \
--mlir yolov8s_seg.mlir
```

生成后的 yolov8s_seg.mlir 文件最后插入了若干命名为 yolo_seg_post* 的算子，用于执行后处理中的坐标变换、nms、矩阵乘等运算。

```
1 %429 = "top.Sigmoid"(%428) {bias = 0.000000e+00 : f64, log = false, round_mode =
   →"HalfAwayFromZero", scale = 1.000000e+00 : f64} : (tensor<8400x25600xf32>) -> tensor
   →<8400x25600xf32> loc(#loc431)
```

(续下页)

(接上页)

```

2 %430 = "top.Reshape"(%429) {flatten_start_dim = -1 : i64} : (tensor<8400x25600xf32>) -> F
3   ↳ tensor<8400x160x160xf32> loc(#loc432)
4 %431 = "top.Slice"(%430, %0, %0, %0) {axes = [], ends = [100, 160, 160], hasparamConvert_
5   ↳ axes = [], offset = [0, 0, 0], steps = [1, 1, 1]} : (tensor<8400x160x160xf32>, none, none, none) ->
6   ↳ tensor<100x160x160xf32> loc(#loc433)
7 %432 = "top.Slice"(%425, %0, %0, %0) {axes = [], ends = [100, 6], hasparamConvert_ axes = [], F
8   ↳ offset = [0, 0], steps = [1, 1]} : (tensor<8400x38xf32>, none, none, none) -> tensor<100x6xf32>
9   ↳ loc(#loc434)
10 return %431, %432 : tensor<100x160x160xf32>, tensor<100x6xf32> loc(#loc)

```

模型的输出共有 2 个，其中 masks_uncrop_uncompare 是原始的分割掩码，shape 为 100x160x160，其中 100 代表最大检测框数，160x160 代表了最后一层特征图的像素大小。

seg_out 是检测框，shape 为 100x6，其中 100 代表最大检测框数，6 分别指 [x_left, y_up, x_right, y_bottom, score, class_id]。目前暂不支持多 batch 的分割模型后处理添加，因此没有 batch 信息。其中坐标是相对模型输入长宽的坐标，比如本例中 640x640，数值参考如下：

```
[-74.06776, 263.67566, 74.06777, 531.1172, 0.9523437, 16.]
```

从原始掩码到最后的掩码输出，还需要将其进行 resize 运算，放大回原始图片大小；并根据 seg_out 中检测框对 mask 多余部分进行 crop；最后对掩码进行阈值过滤，得到全分辨率的掩码。以上的处理代码可参考该源码路径 {package/path/to/tpu_mlir}/python/samples/segment_yolo.py 中的流程。

9.2.3 MLIR 转换成 BModel

将 mlir 文件转换成 F16 的 bmodel，操作方法如下：

```

$ fp_forward.py yolov8s_seg.mlir \
--fpfwd_outputs yolo_seg_post_mulconst3 \
--chip bm1684x \
--fp_type F32 \
-o yolov8s_seg_qtable

$ model_deploy \
--mlir yolov8s_seg.mlir \
--quantize F16 \
--processor bm1684x \
--fuse_preprocess \
--quantize_table yolov8s_seg_qtable \
--model yolov8s_seg_1684x_f16.bmodel

```

这里加上参数 --fuse_preprocess，是为了将前处理也合并到模型中；还使用了 yolov8s_seg_qtable，这是因为后处理中会对 box 施加偏移运算，将框图的坐标数值乘以一个大整数，F16 的特性会导致涉及大整数的运算存在偏差，最终会导致产生过多的 mask，影响后处理性能，因此需要将这一部分算子使用 F32 混合精度进行运算。

9.2.4 模型验证

在本发布包中有用 python 写好的 yolov8s_seg 用例, 使用 segment_yolo 命令, 用于对图片进行分割。该命令对应源码路径 {package/path/to/tpu_mlir}/python/samples/segment_yolo.py。阅读该代码可以了解最终输出结果; 并且如何产生分割掩膜和框图。

命令执行如下:

```
$ segment_yolo \
--input ..../image/dog.jpg \
--model yolov8s_seg_1684x_f16.bmodel \
--net_input_dims 640,640 \
--fuse_preprocess \
--fuse_postprocess \
--output dog_out.jpg
```

CHAPTER 10

编译 LLM 模型

10.1 概述

`llm_convert.py` 是一个用于将大语言模型（LLM）转换成 bmodel 的工具，将原始模型权重转换为 bmodel 格式，以便在 BM1684X、BM1688 和 CV186AH 等芯片平台上进行高效推理。目前支持的 LLM 类型包括 qwen2 和 llama，比如 Qwen2-7B-Instruct、Llama-2-7b-chat-hf 等。

10.2 命令行参数说明

下面介绍该工具支持的各个命令行参数：

- `-m, --model_path` (string, 必填) 指定原始模型权重的路径。例如：`./Qwen2-7B-Instruct`
- `-s, --seq_length` (integer, 必填) 指定转换时使用的序列长度。
- `-q, --quantize` (string, 必填) 指定 bmodel 的量化类型，必须从下面的选项中选择：
 - `bf16`
 - `w8bf16`
 - `w4bf16`
 - `f16`
 - `w8f16`
 - `w4f16`

- `-g, --q_group_size` (integer, 默认值: 64) 如果使用 W4A16 量化模式, 则用于设置每组量化的组大小。
- `-c, --chip` (string, 默认值: `bm1684x`) 指定生成 bmodel 的芯片平台, 支持如下选项:
 - `bm1684x`
 - `bm1688`
 - `cv186ah`
- `--num_device` (integer, 默认值: 1) 指定 bmodel 部署的设备数。
- `--num_core` (integer, 默认值: 0) 指定 bmodel 部署使用的核数, 0 表示采用最大核数。
- `--symmetric` 如果设置该标志, 则使用对称量化。
- `--embedding_disk` 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。
- `--max_pixels` (integer) 对于多模态模型如 `qwen2.5vl`, 用于指定最大像素尺寸, 比如可以指定为 672,896, 表示 672x896 的图片; 也可以是 602112, 表示最大像素。
- `-o, --out_dir` (string, 默认值: `./tmp`) 指定输出的 bmodel 文件保存路径。

10.3 示例用法

假设需要将位于 `/workspace/Qwen2-7B-Instruct` 的大模型转换为 `bm1684x` 平台的 bmodel, 同时使用 384 的序列长度和 “w4bf16” 的量化类型, 设置 group size 为 128, 并将输出文件存放在目录 `qwen2_7b` 下, 可以执行以下命令:

首先需要将 Qwen2-7B-Instruct 从 `huggingface` 下载到本地, 然后执行如下命令:

```
llm_convert.py -m /workspace/Qwen2-7B-Instruct -s 384 -q w4bf16 -g 128 -c bm1684x -o qwen2_
→7b
```

注意如果提示 `transformers` 找不到, 则需要安装, 命令如下 (其他 pip 包同理):

```
pip3 install transformers --upgrade
```

另外该命令也支持 AWQ 和 GPTQ 模型, 参考命令如下:

```
llm_convert.py -m /workspace/Qwen2.5-0.5B-Instruct-AWQ -s 384 -q w4bf16 -c bm1684x -o qwen2.
→5_0.5b
llm_convert.py -m /workspace/Qwen2.5-0.5B-Instruct-GPTQ-Int4 -s 384 -q w4bf16 -c bm1684x -
→o qwen2.5_0.5b
```

CHAPTER 11

附录 01：各框架模型转 ONNX 参考

本章节主要提供了将 PyTorch, TensorFlow 与 PaddlePaddle 模型转为 ONNX 模型的方式参考，读者也可以参考 ONNX 官方仓库提供的转模型教程：<https://github.com/onnx/tutorials>。本章节中的所有操作均在 Docker 容器中进行，具体的环境配置方式请参考第二章的内容。

11.1 PyTorch 模型转 ONNX

本节以一个自主搭建的简易 PyTorch 模型为例进行 onnx 转换

11.1.1 步骤 0：创建工作目录

在命令行中创建并进入 torch_model 目录。

```
1 $ mkdir torch_model  
2 $ cd torch_model
```

11.1.2 步骤 1：搭建并保存模型

在该目录下创建名为 simple_net.py 的脚本并运行，脚本的具体内容如下：

```
1 #!/usr/bin/env python3  
2 import torch  
3  
4 # Build a simple nn model  
5 class SimpleModel(torch.nn.Module):
```

(续下页)

(接上页)

```

6 def __init__(self):
7     super(SimpleModel, self).__init__()
8     self.m1 = torch.nn.Conv2d(3, 8, 3, 1, 0)
9     self.m2 = torch.nn.Conv2d(8, 8, 3, 1, 1)
10
11
12 def forward(self, x):
13     y0 = self.m1(x)
14     y1 = self.m2(y0)
15     y2 = y0 + y1
16     return y2
17
18 # Create a SimpleModel and save its weight in the current directory
19 model = SimpleModel()
20 torch.save(model.state_dict(), "weight.pth")

```

运行命令如下：

```
$ python simple_net.py
```

运行完后我们会在当前目录下获得一个 weight.pth 的权重文件。

11.1.3 步骤 2：导出 ONNX 模型

在该目录下创建另一个名为 export_onnx.py 的脚本并运行，脚本的具体内容如下：

```

1 #!/usr/bin/env python3
2 import torch
3 from simple_net import SimpleModel
4
5 # Load the pretrained model and export it as onnx
6 model = SimpleModel()
7 model.eval()
8 checkpoint = torch.load("weight.pth", map_location="cpu")
9 model.load_state_dict(checkpoint)
10
11 # Prepare input tensor
12 input = torch.randn(1, 3, 16, 16, requires_grad=True)
13
14 # Export the torch model as onnx
15 torch.onnx.export(model,
16     input,
17     'model.onnx', # name of the exported onnx model
18     opset_version=13,
19     export_params=True,
20     do_constant_folding=True)

```

运行完脚本后，我们即可在当前目录下得到名为 model.onnx 的 onnx 模型。

11.2 TensorFlow 模型转 ONNX

本节以 TensorFlow 官方仓库中提供的 mobilenet_v1_0.25_224 模型作为转换样例。

11.2.1 步骤 0：创建工作目录

在命令行中创建并进入 tf_model 目录。

```
1 $ mkdir tf_model
2 $ cd tf_model
```

11.2.2 步骤 1：准备并转换模型

命令行中通过以下命令下载模型并利用 tf2onnx 工具将其导出为 ONNX 模型：

```
1 $ wget -nc http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_0.
2   ↪25_224.tgz
3 # tar to get "*.pb" model def file
4 $ tar xzf mobilenet_v1_0.25_224.tgz
5 $ python -m tf2onnx.convert --graphdef mobilenet_v1_0.25_224_frozen.pb \
6   --output mnet_25.onnx --inputs input:0 \
7   --inputs-as-nchw input:0 \
     --outputs MobilenetV1/Predictions/Reshape_1:0
```

运行以上所有命令后我们即可在当前目录下得到名为 mnet_25.onnx 的 onnx 模型。

11.3 PaddlePaddle 模型转 ONNX

本节以 PaddlePaddle 官方仓库中提供的 SqueezeNet1_1 模型作为转换样例。本节需要额外安装 openssl-1.1.1o (ubuntu 22.04 默认提供 openssl-3.0.2)。

11.3.1 步骤 0：安装 openssl-1.1.1o

```
1 wget http://nz2.archive.ubuntu.com/ubuntu/pool/main/o/openssl/libssl1.1_1.1.1f-1ubuntu2.19_
2   ↪amd64.deb
3 sudo dpkg -i libssl1.1_1.1.1f-1ubuntu2.19_amd64.deb
```

如果上述链接失效，请参考 <http://nz2.archive.ubuntu.com/ubuntu/pool/main/o/openssl/?C=M;O=D> 更换有效链接。

11.3.2 步骤 1：创建工作目录

在命令行中创建并进入 pp_model 目录。

```
1 $ mkdir pp_model
2 $ cd pp_model
```

11.3.3 步骤 2：准备模型

在命令行中通过以下命令下载模型：

```
1 $ wget https://bj.bcebos.com/paddlehub/fastdeploy/SqueezeNet1_1_infer.tgz
2 $ tar xzf SqueezeNet1_1_infer.tgz
3 $ cd SqueezeNet1_1_infer
```

并用 PaddlePaddle 项目中的 paddle_infer_shape.py 脚本对模型进行 shape 推理，此处将输入 shape 以 NCHW 的格式设置为 [1,3,224,224]：

```
1 $ wget https://raw.githubusercontent.com/jiangjiajun/PaddleUtils/main/paddle/paddle_infer_
2   ↪shape.py
3 $ python paddle_infer_shape.py --model_dir . \
4   --model_filename inference.pdmodel \
5   --params_filename inference.pdiparams \
6   --save_dir new_model \
    --input_shape_dict="{'inputs':[1,3,224,224]}"
```

运行完以上所有命令后我们将处于 SqueezeNet1_1_infer 目录下，并在该目录下生成 new_model 的目录。

11.3.4 步骤 3：转换模型

在命令行中通过以下命令安装 paddle2onnx 工具，并利用该工具将 PaddlePaddle 模型转为 ONNX 模型：

```
1 $ pip install paddle2onnx
2 $ paddle2onnx --model_dir new_model \
3   --model_filename inference.pdmodel \
4   --params_filename inference.pdiparams \
5   --opset_version 13 \
6   --save_file squeezenet1_1.onnx
```

运行完以上所有命令后我们将获得一个名为 squeezenet1_1.onnx 的 onnx 模型。

CHAPTER 12

附录 02: CV18xx 使用指南

CV18xx 支持 ONNX 系列和 Caffe 模型，目前不支持 TFLite 模型。在量化数据类型方面，CV18xx 支持 BF16 格式的量化和 INT8 格式的对称量化。本章节以 CV183X 为例，介绍 CV18xx 系列编译模型和运行 runtime sample。

12.1 编译 yolov5 模型

12.1.1 安装 tpu-mlir

进入 Docker 容器，并执行以下命令安装 TPU-MLIR：

```
$ pip install tpu_mlir[all]
# or
$ pip install tpu_mlir-*py3-none-any.whl[all]
```

12.1.2 准备工作目录

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后将文件夹重命名为 tpu_mlir_resource：

```
$ tar -xvf tpu-mlir-resource.tar
$ mv regression/ tpu-mlir-resource/
```

建立 model_yolov5s 目录，并把模型文件和图片文件都放入 model_yolov5s 目录中。

操作如下：

```

1 $ mkdir model_yolov5s && cd model_yolov5s
2 $ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ cp -rf tpu_mlir_resource/image .
5 $ mkdir workspace && cd workspace

```

12.1.3 ONNX 转 MLIR

如果模型是图片输入，在转模型之前我们需要了解模型的预处理。如果模型用预处理后的 npz 文件做输入，则不需要考虑预处理。预处理过程用公式表达如下（ x 代表输入）：

$$y = (x - \text{mean}) \times \text{scale}$$

官网 yolov5 的图片是 rgb，每个值会乘以 1/255，转换成 mean 和 scale 对应为 0.0,0.0,0.0 和 0.0039216,0.0039216,0.0039216。

模型转换命令如下：

```
$ model_transform \
--model_name yolov5s \
--model_def ./yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 326,474,622 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s.mlir
```

model_transform 的相关参数说明参考[model_transform 参数说明](#) 部分。

12.1.4 MLIR 转 BF16 模型

将 mlir 文件转换成 bf16 的 cvimodel，操作方法如下：

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

model_deploy 的相关参数说明参考[model_deploy 参数说明](#) 部分。

12.1.5 MLIR 转 INT8 模型

转 INT8 模型前需要跑 calibration, 得到校准表; 输入数据的数量根据情况准备 100~1000 张左右。然后用校准表, 生成 INT8 对称 cvimodel

这里用现有的 100 张来自 COCO2017 的图片举例, 执行 calibration:

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

运行完成后会生成名为 \${model_name}_cali_table 的文件, 该文件用于后续编译 INT8 模型的输入文件。

转成 INT8 对称量化 cvimodel 模型, 执行如下命令:

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

编译完成后, 会生成名为 \${model_name}_cv183x_int8_sym.cvimodel 的文件。

12.1.6 效果对比

onnx 模型的执行方式如下, 得到 dog_onnx.jpg :

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

FP32 mlir 模型的执行方式如下, 得到 dog_mlir.jpg :

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

BF16 cvimodel 的执行方式如下, 得到 dog_bf16.jpg :

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

INT8 cvimodel 的执行方式如下, 得到 dog_int8.jpg :

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

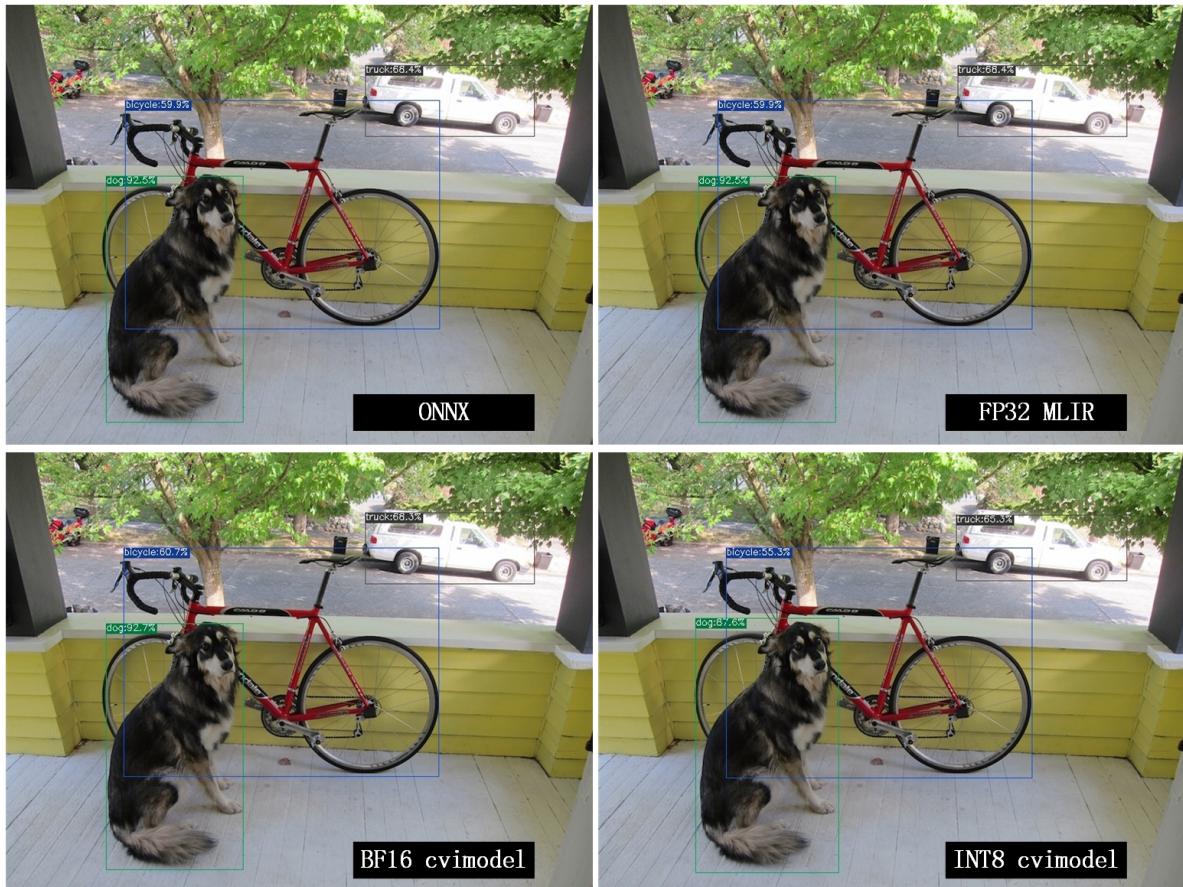


图 12.1: 不同模型效果对比

四张图片对比如 图 12.1 ,由于运行环境不同, 最终的效果和精度与 图 12.1 会有些差异。

上述教程介绍了 TPU-MLIR 编译 CV18xx 系列的 ONNX 模型的过程,caffe 模型的转换过程可参考“编译 Caffe 模型”章节, 只需要将对应的处理器名称换成实际的 CV18xx 名称即可。

12.2 合并 cvimodel 模型文件

对于同一个模型，可以依据输入的 batch size 以及分辨率（不同的 h 和 w）分别生成独立的 cvimodel 文件。不过为了节省外存和运存，可以选择将这些相关的 cvimodel 文件合并为一个 cvimodel 文件，共享其权重部分。具体步骤如下：

12.2.1 步骤 0: 生成 batch 1 的 cvimodel

请参考前述章节，新建 workspace 目录，通过 model_transform 将 yolov5s 转换成 mlir fp32 模型。

注意：

1. 需要合并的 cvimodel 使用同一个 workspace 目录，并且不要与不需要合并的 cvimodel 共用一个 workspace；
 2. 步骤 0、步骤 1 中--merge_weight 是必需选项。
-

```
$ model_transform \
--model_name yolov5s \
--model_def ./yolov5s.onnx \
--input_shapes [[1,3,640,640]] \
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 326,474,622 \
--test_input ./image/dog.jpg \
--test_result yolov5s_top_outputs.npz \
--mlir yolov5s_bs1.mlir
```

使用前述章节生成的 yolov5s_cali_table；如果没有，则通过 run_calibration 工具对 yolov5s.mlir 进行量化校验获得 calibration table 文件。然后将模型量化并生成 cvimodel：

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

12.2.2 步骤 1: 生成 batch 2 的 cvimodel

同步骤 0, 在同一个 workspace 中生成 batch 为 2 的 mlir fp32 文件:

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

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

12.2.3 步骤 2: 合并 batch 1 和 batch 2 的 cvimodel

使用 model_tool 合并两个 cvimodel 文件:

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

12.2.4 步骤 3: runtime 接口调用 cvimodel

可以通过以下命令查看 bs1 和 bs2 指令的 program id:

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

在运行时可以通过如下方式去运行不同的 batch 命令:

```

CVI_MODEL_HANDLE bs1_handle;
CVI_RC ret = CVI_NN_RegisterModel("yolov5s_cv183x_int8_sym_bs1_bs2.cvimodel", &bs1_
→handle);
assert(ret == CVI_RC_SUCCESS);
// 选择bs1的program id
CVI_NN_SetConfig(bs1_handle, OPTION_PROGRAM_INDEX, 0);
CVI_NN_GetInputOutputTensors(bs1_handle, ...);
...

CVI_MODEL_HANDLE bs2_handle;
// 复用已加载的模型
CVI_RC ret = CVI_NN_CloneModel(bs1_handle, &bs2_handle);
assert(ret == CVI_RC_SUCCESS);
// 选择bs2的program id
CVI_NN_SetConfig(bs2_handle, OPTION_PROGRAM_INDEX, 1);
CVI_NN_GetInputOutputTensors(bs2_handle, ...);
...

// 最后销毁bs1_handle, bs2_handle
CVI_NN_CleanupModel(bs1_handle);
CVI_NN_CleanupModel(bs2_handle);

```

12.2.5 综述: 合并过程

使用上面命令, 不论是相同模型还是不同模型, 均可以进行合并。合并的原理是: 模型生成过程中, 会叠加前面模型的 weight(如果相同则共用)。

主要步骤在于:

1. 用 model_deploy 生成模型时, 加上–merge_weight 参数
2. 要合并的模型的生成目录必须是同一个, 且在合并模型前不要清理任何中间文件 (叠加前面模型 weight 通过中间文件 _weight_map.csv 实现)
3. 用 model_tool –combine 将多个 cvimodel 合并

12.3 编译和运行 runtime sample

本章首先介绍 EVB 如何运行 sample 应用程序, 然后介绍如何交叉编译 sample 应用程序, 最后介绍 docker 仿真编译和运行 sample。具体包括 4 个 samples:

- Sample-1 : classifier (mobilenet_v2)
- Sample-2 : classifier_bf16 (mobilenet_v2)
- Sample-3 : classifier fused preprocess (mobilenet_v2)
- Sample-4 : classifier multiple batch (mobilenet_v2)

12.3.1 在 EVB 运行 release 提供的 sample 预编译程序

需要如下文件:

- cvitek_tpu_sdk_[cv183x | cv182x | cv182x_uclibc | cv181x_glibc32 | cv181x_musl_riscv64_rvv | cv180x_musl_riscv64_rvv | cv181x_glibc_riscv64].tar.gz
- cvimodel_samples_[cv183x | cv182x | cv181x | cv180x].tar.gz

将根据处理器类型选择所需文件加载至 EVB 的文件系统, 于 evb 上的 linux console 执行, 以 cv183x 为例:

解压 samples 使用的 model 文件 (以 cvimodel 格式交付), 并解压 TPU_SDK, 并进入 samples 目录, 执行测试, 过程如下:

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

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

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

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

# TOP_K[5]:
# 0.314453, idx 285, n02124075 Egyptian cat
# 0.040039, idx 331, n02326432 hare
# 0.018677, idx 330, n02325366 wood rabbit, cottontail, cottontail rabbit
# 0.010986, idx 463, n02909870 bucket, pail
```

(续下页)

(接上页)

```
# 0.010986, idx 852, n04409515 tennis ball

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

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

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

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

同时提供脚本作为参考, 执行效果与直接运行相同, 如下:

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

在 cvitek_tpu_sdk/samples/samples_extra 目录下有更多的 samples, 可供参考:

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

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

(续下页)

(接上页)

```
yolo_v5_out.jpg

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

./bin/cvi_sample_alpha_pose_fused_preprocess \
$MODEL_PATH/yolo_v3_416_fused_preprocess_with_detection.cvimodel \
$MODEL_PATH/alpha_pose_fused_preprocess.cvimodel \
./data/pose_demo_2.jpg \
alpha_pose_out.jpg

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

12.3.2 交叉编译 samples 程序

发布包有 samples 的源代码, 按照本节方法在 Docker 环境下交叉编译 samples 程序, 然后在 evb 上运行。

本节需要如下文件:

- cvitek_tpu_sdk_[cv183x | cv182x | cv182x_uclibc | cv181x_glibc32 | cv181x_musl_riscv64_rvv | cv180x_musl_riscv64_rvv].tar.gz
- cvitek_tpu_samples.tar.gz

aarch 64 位 (如 cv183x aarch64 位平台)

SDK 准备:

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

编译 samples, 安装至 install_samples 目录:

```
tar zxf cvitek_tpu_samples.tar.gz
cd cvitek_tpu_samples
mkdir build_soc
cd build_soc
```

(续下页)

(接上页)

```

cmake -G Ninja \
-DCMAKE_BUILD_TYPE=RELEASE \
-DCMAKE_C_FLAGS_RELEASE=-O3 \
-DCMAKE_CXX_FLAGS_RELEASE=-O3 \
-DCMAKE_TOOLCHAIN_FILE=$TPU_SDK_PATH/cmake/toolchain-aarch64-linux.cmake \
-DTPU_SDK_PATH=$TPU_SDK_PATH \
-DOPENCV_PATH=$TPU_SDK_PATH/opencv \
-DCMAKE_INSTALL_PREFIX=../install_samples \
..
cmake --build . --target install

```

arm 32 位 (如 cv183x 平台 32 位、cv182x 平台)

SDK 准备:

```

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

```

如果 docker 版本低于 1.7, 则需要更新 32 位系统库 (只需一次):

```

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

```

编译 samples, 安装至 install_samples 目录:

```

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

```

uclibc 32 位平台 (cv182x uclibc 平台)

SDK 准备:

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

如果 docker 版本低于 1.7, 则需要更新 32 位系统库 (只需一次):

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

编译 samples, 安装至 install_samples 目录:

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

riscv64 位 musl 平台 (如 cv181x、cv180x riscv64 位 musl 平台)

SDK 准备:

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

编译 samples, 安装至 install_samples 目录:

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

(续下页)

(接上页)

```

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

```

riscv64 位 glibc 平台 (如 cv181x、cv180x riscv64 位 glibc 平台)

SDK 准备:

```

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

```

编译 samples, 安装至 install_samples 目录:

```

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

```

12.3.3 docker 环境仿真运行的 samples 程序

需要如下文件:

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

TPU sdk 准备:

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

编译 samples, 安装至 install_samples 目录:

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

运行 samples 程序:

```
# envs
tar zxf cvimodel_samples_cv183x.tar.gz
export MODEL_PATH=$PWD/cvimodel_samples

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

其他 samples 运行命令参照 EVB 运行命令

12.4 在开发板上进行模型测试及验证工作

在板子上可以通过 cvitek_tpu_sdk/bin/下的 model_runner 程序进行模型验证；运行 model_runner 前需要将 cvitek_tpu_sdk 放到板子上，然后：

```
cd cvitek_tpu_sdk
source ./envs_tpu_sdk.sh
```

model_runner 支持以下选项：

表 12.1: model_runner 参数功能

参数名	说明
-model	指定模型文件
-input	指定输入 npz 文件
-output	指定输出 npz 文件
-pmu	打印性能数据
-count	循环运行次数
-reference	指定结果对比 npz 文件
-tolerances	指定结果对比相似度限制
-enable-timer	打印推理耗时信息

一般使用命令如下：

```
# 测试模型是否能正常推理
model_runner --model yolov5s.cvimodel

# 测试模型性能
model_runner --model yolov5s.cvimodel --pmu

# dump 模型结果
model_runner --model yolov5s.cvimodel --input input.npz --output output.npz

# 对比模型结果
model_runner --model yolov5s.cvimodel --input input.npz --reference ref.npz
```

12.5 FAQ

12.5.1 模型转换常见问题

模型转换问题

- pytorch,tensorflow 等是否能直接转换为 cvimodel?

pytorch: 支持通过 `jit.trace(torch_model.eval(), inputs).save(`model_name.pt`)` 静态化后的 pt 模型。

tensorflow / 其它: 暂不支持, 可以通过 onnx 间接支持 tf 模型。

- 执行 model_transform 报错

`model_transform` 命令作用是将 onnx,caffe 框架模型转化为 fp32 mlir 形式, 报错很大概率就是存在不支持的算子或者算子属性不兼容, 可以反馈给 tpu 团队解决。

- 执行 model_deploy 报错

`model_deploy` 作用是先将 fp32 mlir 通过量化转为 int8/bf16mlir 形式, 然后再将 int8/bf16mlir 转化为 cvimodel。在转化的过程中, 会涉及到两次相似度的对比: 一

次是 fp32 mlir 与 int8/bf16mlir 之间的量化对比, 一次是 int8/bf16mlir 与最终转化出来的 cvimodel 的相似度对比, 若相似度对比失败则会出现下列问题:

```
[437 Transpose      ]      SIMILAR [PASSED]
(1, 3, 20, 20, 85) float32
cosine_similarity   = 0.999616
euclidean_similarity = 0.972212
ssim_similarity     = 21.209481
154 compared
152 passed
1 equal, 0 close, 152 similar
1 failed
0 not equal, 1 not similar
min similarity = (0.9813582301139832, 0.7978442697003846, 13.49835753440857)
Target "yolo_v5_s_cv183x_int8_sym_tpu_outputs.npz"
Reference "yolo_v5_s_top_outputs.npz"
npz compare FAILED.
compare 437_Transpose: 100% | 154/154 [00:02<00:00, 53.68it/s]
Traceback (most recent call last):
File "/workspace/python/tools/model_deploy.py", line 286, in <module>
    tool.lowering()
File "/workspace/python/tools/model_deploy.py", line 103, in lowering
    tool.validate_tpu_mlir()
File "/workspace/python/tools/model_deploy.py", line 190, in validate_tpu_mlir
    f32_blobs_compare(self.tpu_npz, self.ref_npz, self.tolerance, self.excepts)
File "/workspace/python/utils/mlir_shell.py", line 172, in f32_blobs_compare
    os.system(cmd)
File "/workspace/python/utils/mlir_shell.py", line 50, in _os_system
    raise RuntimeError("[" + cmd + "]")
RuntimeError: [!Error]: npz_tool.py compare yolo_v5_s_cv183x_int8_sym_tpu_outputs.npz yolo_v5_s_top_outputs.npz --tolerance 0.96,0.80 --except -vv
```

解决方法: tolerance 参数不对。模型转换过程会对 int8/bf16 mlir 与 fp32 mlir 的输出计算相似度, 而 tolerance 作用就是限制相似度的最低值, 若计算出的相似度的最小值低于对应的预设的 tolerance 值则程序会停止执行, 可以考虑对 tolerance 进行调整。(如果相似度的最小值过低请反馈到 tpu 团队解决)。

- model_transform 的 pixel_format 参数和 model_deploy 的 customization_format 参数的差异?

channel_order 是原始模型的输入图片类型 (只支持 gray/rgb planar/bgr planar),customization_format 是转换成 cvimodel 后的输入图片类型, 由客户自行决定, 需与fuse_preprocess 共同使用 (如果输入图片是通过 VPSS 或者 VI 获取的 YUV 图片, 可以设置 customization_format 为 YUV 格式)。如果 pixel_format 与 customization_format 不一致, cvimodel 推理时会自动将输入转成 pixel_format 指定的类型。

- 是否支持多输入模型, 怎么进行预处理?

仅支持多输入图片使用同一种预处理方式的模型, 不支持多输入图片使用不同预处理方式的模型。

量化问题

- 跑 run_calibration 提示 KeyError: ‘images’

传入的 images 的路径不对, 请检查数据集的路径是否正确。

- 跑量化如何处理多输入问题?

多输入模型跑 run_calibration 时, 可使用.npy 存储多个输入, 或使用--data_list 参数, 且 data_list 中的每行的多个输入由 “,” 隔开。

- 跑量化输入会进行预处理吗?

会的, 根据 model_transform 的预处理参数保存到 mlir 文件中, 量化过程会进行加载预处理参数进行预处理。

- 跑量化输入程序被系统 kill 或者显示分配内存失败

需要先检查主机的内存是否足够, 常见的模型需要 8G 内存左右即可。如果内存不够, 可尝试在运行 run_calibration 时, 添加以下参数来减少内存需求。

--tune_num 2	#默认为5
--------------	-------

5. 是否支持手动修改 calibration table?

支持, 但是不建议修改。

其它常见问题

1. 转换后的模型是否支持加密?

暂时不支持。

2. bf16 的模型与 int8 模型的速度差异是多少?

大约是 3-4 倍时间差异, 具体的数据需要通过实验验证。

3. 是否支持动态 shape?

cvimodel 不支持动态 shape。如果是固定的几种 shape 可以依据输入的 batch_size 以及不同的 h 和 w 分别生成独立的 cvimodel 文件, 通过共享权重的形式合并为一个 cvimodel。详见: [合并 cvimodel 模型文件](#)

12.5.2 模型评估常见问题

模型的评估流程?

先转化为 bf16 模型, 通过 model_tool --info xxxx.cvimodel 命令来评估模型所需要的 ION 内存以及所占的存储空间, 接着在板子上执行 model_runner 来评估模型运行的时间, 之后根据提供的 sample 来评估业务场景下模型精度效果。模型输出的效果准确性符合预期之后, 再转化为 int8 模型再完成与 bf16 模型相同的流程

量化后精度与原来模型对不上, 如何调试?

1. 确保 model_deploy 的 --test_input, --test_reference, --compare_all, --tolerance 参数进行了正确设置。
2. 比较 bf16 模型与原始模型的运行结果, 确保误差不大。如果误差较大, 先确认预处理和后处理是否正确。
3. 如果 int8 模型精度差:
 - a. 确认 run_calibration 使用的数据集为训练模型时使用的验证集;
 - b. 可以增加 run_calibration 使用的业务场景数据集 (一般为 100-1000 张图片)。
4. 确认输入类型:
 - a. 若指定 --fuse_preprocess 参数, cvimodel 的 input 类型为 uint8;

- b. 若指定 `--quant_input`，一般情况下, `bf16_cvimodel` 的 `input` 类型为 `bf16,int8_cvimodel` 的 `input` 类型为 `int8`;
- c. `input` 类型也可以通过 `model_tool -info xxx.cvimodel` 查看

bf16 模型的速度比较慢,int8 模型精度不符合预期怎么办?

使用混精度量化方法, 可参考 `mix precision`。

12.5.3 模型部署常见问题

CVI_NN_Forward 接口调用多次后出错或者卡住时间过长?

可能驱动或者硬件问题, 需要反馈给 tpu 团队解决。

模型预处理速度比较慢?

1. 转模型的时候可以在运行 `model_deploy` 时加上 `fuse_preprocess` 参数, 将预处理放到深度学习处理器内部来处理。
2. 如果图片是从 `vpss` 或者 `vi` 获取, 那么可以在转模型时使用 `fuse_preprocess`、`aligned_input`, 然后使用 `CVI_NN_SetTensorPhysicalAddr` 等接口直接将 `input tensor` 地址设置为图片的物理地址, 减少数据拷贝耗时。

docker 的推理和 evb 推理的浮点和定点结果是否一样?

定点无差异, 浮点有差异, 但是相似度比较高, 误差可以忽略。

如果要跑多个模型支持多线程并行吗?

支持多线程, 但是多个模型在深度学习处理器上推理时是串行进行的。

填充 input tensor 相关接口区别

`CVI_NN_SetTensorPtr` : 设置 `input tensor` 的虚拟地址, 原本的 `tensor` 内存不会释放。推理时从用户设置的虚拟地址 **拷贝数据** 到原本的 `tensor` 内存上。

`CVI_NN_SetTensorPhysicalAddr` : 设置 `input tensor` 的物理地址, 原本的 `tensor` 内存会释放。推理时直接从新设置的物理地址读取数据, **无需拷贝数据**。从 `VPSS` 获取的 `Frame` 可以调用这个接口, 传入 `Frame` 的首地址。注意需要转模型的时候 `model_deploy` 设置 `--fused_preprocess --aligned_input` 才能调用此接口。

`CVI_NN_SetTensorWithVideoFrame` : 通过 `VideoFrame` 结构体来填充 `Input Tensor`。注意 `VideoFrame` 的地址为物理地址。如果转模型设置 `--fuse_preprocess --aligned_input`, 则等同于 `CVI_NN_SetTensorPhysicalAddr`, 否则会将 `VideoFrame` 的数据拷贝到 `Input Tensor`。

CVI_NN_SetTensorWithAlignedFrames : 与 CVI_NN_SetTensorWithVideoFrame 类似, 支持多 batch。

CVI_NN_FeedTensorWithFrames : 与 CVI_NN_SetTensorWithVideoFrame 类似。

模型载入后 ion 内存分配问题

1. 调用 CVI_NN_RegisterModel 后会为 weight 和 cmdbuf 分配 ion 内存 (从 model_tool 可以看到 weight 和 cmdbuf 大小)
2. 调用 CVI_NN_GetInputOutputTensors 后会为 tensor(包括 private_gmem, shared_gmem, io_mem) 分配 ion 内存
3. CVI_NN_CloneModel 可以共享 weight 和 cmdbuf 内存
4. 其他接口均不会再申请 ion 内存, 即除了初始化, 其他阶段模型都不会再申请内存。
5. 不同模型的 shared_gmem 是可以共享 (包括多线程情况), 因此优先初始化 shared_gmem 最大的模型可以节省 ion 内存。

加载业务程序后模型推理时间变长

设置环境变量 export TPU_ENABLE_PMU=1 后, 模型推理时会打印 tpu 日志, 记录 tdma_exe_ms、tiu_exe_ms、inference_ms 这 3 个耗时。一般加载业务后 tdma_exe_ms 会变长, tiu_exe_ms 不变, 这是因为 tdma_exe_ms 是内存搬运数据耗时, 如果内存带宽不够用了, tdma 耗时就会增加。

优化的方向:

- a. vpss/venc 等优化 chn, 降低分辨率
- b. 业务层减少内存拷贝, 如图片尽量保存引用, 减少拷贝等
- c. 模型填充 Input tensor 时, 使用无拷贝的方式

12.5.4 其他常见问题

在 cv182x/cv181x/cv180x 板端环境中出现: taz:invalid option -z 解压失败的情况

先在其他 linux 环境下解压, 再放到板子中使用, 因为 window 不支持软链接, 所以在 windows 环境下解压可能导致软链接失效导致报错

若 tensorflow 模型为 saved_model 的 pb 形式, 如何进行转化为 frozen_model 的 pb 形式

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

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

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

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

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

CHAPTER 13

附录 03: BM168x 使用指南

BM168x 支持 ONNX 系列、pytorch 模型、Caffe 模型和 TFLite 模型。本章节以 BM1684x 为例, 介绍 BM168x 系列 bmodel 文件的合并方法。

13.1 合并 bmodel 模型文件

对于同一个模型, 可以依据输入的 batch size 以及分辨率 (不同的 h 和 w) 分别生成独立的 bmodel 文件。不过为了节省外存和运存, 可以选择将这些相关的 bmodel 文件合并为一个 bmodel 文件, 共享其权重部分。具体步骤如下:

13.1.1 步骤 0: 生成 batch 1 的 bmodel

请参考前述章节, 新建 workspace 目录, 通过 model_transform 将 yolov5s 转换成 mlir fp32 模型。

注意:

1. 需要合并的 bmodel 使用同一个 workspace 目录, 并且不要与不需要合并的 bmodel 共用一个 workspace;
 2. 步骤 0、步骤 1 中--merge_weight 是必需选项。
-

```
$ model_transform \
  --model_name yolov5s \
  --model_def ./yolov5s.onnx \
```

(续下页)

(接上页)

```
--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_bs1.mlir
```

使用前述章节生成的 `yolov5s_cali_table`; 如果没有, 则通过 `run_calibration` 工具对 `yolov5s.mlir` 进行量化校验获得 `calibration table` 文件。然后将模型量化并生成 `bmodel`:

```
# 加上 --merge_weight参数
$ model_deploy \
  --mlir yolov5s_bs1.mlir \
  --quantize INT8 \
  --calibration_table yolov5s_cali_table \
  --processor bm1684x \
  --test_input yolov5s_in_f32.npz \
  --test_reference yolov5s_top_outputs.npz \
  --tolerance 0.85,0.45 \
  --merge_weight \
  --model yolov5s_bm1684x_int8_sym_bs1.bmodel
```

13.1.2 步骤 1: 生成 batch 2 的 bmodel

同步骤 0, 在同一个 workspace 中生成 batch 为 2 的 mlir fp32 文件:

```
$ model_transform \
  --model_name yolov5s \
  --model_def ./yolov5s.onnx \
  --input_shapes [[2,3,640,640]] \
  --mean 0.0,0.0,0.0 \
  --scale 0.0039216,0.0039216,0.0039216 \
  --keep_aspect_ratio \
  --pixel_format rgb \
  --output_names 350,498,646 \
  --test_input ./image/dog.jpg \
  --test_result yolov5s_top_outputs.npz \
  --mlir yolov5s_bs2.mlir
```

```
# 加上 --merge_weight参数
$ model_deploy \
  --mlir yolov5s_bs2.mlir \
  --quantize INT8 \
  --calibration_table yolov5s_cali_table \
  --processor bm1684x \
```

(续下页)

(接上页)

```
--test_input yolov5s_in_f32.npz \
--test_reference yolov5s_top_outputs.npz \
--tolerance 0.85,0.45 \
--merge_weight \
--model yolov5s_bm1684x_int8_sym_bs2.bmodel
```

13.1.3 步骤 2: 合并 batch 1 和 batch 2 的 bmodel

使用 model_tool 合并两个 bmodel 文件:

```
model_tool \
--combine \
yolov5s_bm1684x_int8_sym_bs1.bmodel \
yolov5s_bm1684x_int8_sym_bs2.bmodel \
-o yolov5s_bm1684x_int8_sym_bs1_bs2.bmodel
```

13.1.4 综述: 合并过程

使用上面命令, 不论是相同模型还是不同模型, 均可以进行合并。合并的原理是: 模型生成过程中, 会叠加前面模型的 weight(如果相同则共用)。

主要步骤在于:

1. 用 model_deploy 生成模型时, 加上--merge_weight 参数
2. 要合并的模型的生成目录必须是同一个, 且在合并模型前不要清理任何中间文件 (叠加前面模型 weight 通过中间文件 _weight_map.csv 实现)
3. 用 model_tool --combine 将多个 bmodel 合并

CHAPTER 14

附录 04：Model-zoo 测试

14.1 注意事项

每项测试耗时若超过以下时间限制则视为异常：

- 编译测试：48 小时
- 性能测试：24 小时
- 精度测试：24 小时（当前仅 BM1684X PCIE 需要进行精度测试）

14.2 配置系统环境

如果是首次使用 Docker，那么请使用[开发环境配置](#) 中的方法安装并配置 Docker。同时，本章中会使用到 git-lfs，如果首次使用 git-lfs，用户需要在自己系统中（并非 Docker 容器中）执行下述命令进行安装和配置：

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

14.3 获取 model-zoo 模型

在工作目录下，从 SOPHGO 提供的 SDK 包中获取 model-zoo 测试包，并执行以下操作创建并设置好 model-zoo：

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

model-zoo 的目录结构如下：

```
├── config.yaml
├── requirements.txt
├── dataset
├── harness
└── output
    └── ...
```

- config.yaml：包含通用的配置：数据集的目录、模型的根目录等，以及一些复用的参数和命令
- requirements.txt：model-zoo 的 python 依赖
- dataset：目录中包含 modelzoo 中模型的数据集，将作为 plugin 被 tpu_perf 调用
- output：目录将用于存放编译输出的 bmodel 和一些中间数据
- 其他目录包含各个模型的信息和配置。每个模型对应的目录都有一个 config.yaml 文件，该配置文件中配置了模型的名称、路径和 FLOPs、数据集制作参数，以及模型的量化编译命令。

14.4 准备运行环境

在系统中（Docker 容器外）安装运行 model-zoo 所需的依赖：

```
# for ubuntu 操作系统
$ sudo apt install build-essential
$ sudo apt install python3-dev
$ sudo apt install -y libgl1
# for centos 操作系统
$ sudo yum install make automake gcc gcc-c++ kernel-devel
$ sudo yum install python-devel
$ sudo yum install mesa-libGL
# 精度测试需要执行以下操作，性能测试可以不执行，推荐使用Anaconda等创建python3.
→7或以上的虚拟环境
$ cd path/to/model-zoo
$ pip3 install -r requirements.txt
```

另外，进行性能和精度测试时需要调用 TPU 硬件，请安装 TPU 硬件对应的 runtime 环境。

14.5 配置 SoC 设备

注意: 如果您的设备是 PCIE 板卡, 可以直接跳过该节内容。

性能测试只依赖于 TPU 硬件对应的 runtime 环境, 所以在工具链编译环境编译完的模型连同 model-zoo 整个打包, 就可以在 SoC 环境使用 tpu_perf 进行性能测试。但是, SoC 设备上存储有限, 完整的 model-zoo 与编译输出内容可能无法完整拷贝到 SoC 中。这里介绍一种通过 linux nfs 远程文件系统挂载来实现在 SoC 设备上运行测试的方法。

首先, 在工具链环境服务器『host 系统』安装 nfs 服务:

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

在 /etc/exports 中添加以下内容 (配置共享目录):

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

其中 * 表示所有人都可以访问该共享目录, 也可以配置成特定网段或 IP 可访问, 如:

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

然后执行如下命令使配置生效:

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

另外, 需要为 dataset 目录下的图片添加读取权限:

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

在 SoC 设备上安装客户端并挂载该共享目录:

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

这样便可以在 SoC 环境访问测试目录。SoC 测试其余的操作与 PCIE 基本一致, 请参考下文进行操作; 运行环境命令执行位置的差别, 已经在执行处添加说明。

14.6 准备数据集

注意: 由于 SoC 设备 CPU 资源有限, 不推荐进行精度测试, 因此 SoC 设备测试可以跳过数据集准备与精度测试部分

14.6.1 ImageNet

下载 ImageNet 2012 数据集。

解压后，将 Data/CLS_LOC/val 下的数据移动到 model-zoo 如下目录中：

```
$ cd path/to/sophon/model-zoo
$ mkdir -p dataset/ILSVRC2012/ILSVRC2012_img_val
$ mv path/to/imagenet-object-localization-challenge/Data/CLS_LOC/val dataset/ILSVRC2012/
→ILSVRC2012_img_val
# 也可以通过软链接 ln -s 将数据集目录映射到 dataset/ILSVRC2012/ILSVRC2012_img_val
```

14.6.2 COCO (可选)

如果精度测试用到了 coco 数据集（如 yolo 等用 coco 训练的网络），请按照如下步骤下载解压：

```
$ cd path/to/model-zoo/dataset/COCO2017/
$ wget http://images.cocodataset.org/annotations/annotations_trainval2017.zip
$ wget http://images.cocodataset.org/zips/val2017.zip
$ unzip annotations_trainval2017.zip
$ unzip val2017.zip
```

14.6.3 Vid4 (可选)

如果需要对 BasicVSR 进行精度测试，请按照如下步骤下载解压 Vid4 数据集：

```
$ pip3 install gdown
$ cd path/to/model-zoo/dataset/basicvsr/
$ gdown https://drive.google.com/open?id=1ZuvNNLgR85TV_whJoHM7uVb-XW1y70DW --fuzzy
$ unzip -o Vid4.zip -d eval
```

14.7 准备工具链编译环境

建议在 docker 环境使用工具链软件，可以参考[基础环境配置](#)安装 Docker。并在工作目录（即 model-zoo 所在目录）下执行以下命令创建 Docker 容器：

```
$ docker pull sophgo/tpuc_dev:v3.4
$ docker run --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.4
```

如果要让容器在退出后删除，可以添加 `--rm` 参数：

```
$ docker run --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.4 --rm
```

运行命令后会处于 Docker 的容器中，从 SOPHGO 提供的 SDK 包中获取最新的 tpu-mlir wheel 安装包，例如 `tpu_mlir-*~py3-none-any.whl`。在 Docker 容器中安装 TPU-MLIR：

```
$ pip install tpu_mlir-*-py3-none-any.whl[all]
```

14.8 模型性能和精度测试流程

14.8.1 模型编译

模型编译过程需要在 Docker 内进行，Docker 内需要按照上文要求安装 tpu_mlir。

model-zoo 的相关 config.yaml 配置了 SDK 的测试内容。以 resnet18-v2 为例，其配置文件为 model-zoo/vision/classification/resnet18-v2/mlir.config.yaml 。

执行以下命令，可以编译 resnet18-v2 模型：

```
$ cd ../model-zoo
$ python3 -m tpu_perf.build --target BM1684X --mlir vision/classification/resnet18-v2/mlir.config
→yaml
```

其中，--target 用于指定处理器型号，目前支持 BM1684、BM1684X、BM1688、BM1690、CV186X。

执行以下命令，可以编译全部高优先级测试样例：

```
$ cd ../model-zoo
$ python3 -m tpu_perf.build --target BM1684X --mlir -l full_cases.txt --priority_filter high
```

完整编译可能需要提前预留 2T 以上的空间，请根据实际情况调整。其中 --clear_if_success 参数可用于在编译成功后删除中间文件，节省空间。

此时会编译以下高优先级模型（由于 model-zoo 的模型在持续添加中，这里只列出部分模型）：

```
* efficientnet-lite4
* mobilenetv2
* resnet18-v2
* resnet50-v2
* shufflenet_v2
* squeezenet1.0
* vgg16
* yolov5s
* ...
```

编译结束后，会看到新生成的 output 文件夹，编译输出内容都在该文件夹中，此编译结果可以用于性能测试和精度测试，无需重新编译。但需要修改 output 文件夹的属性，以保证其可以被 Docker 外系统访问：

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

14.8.2 性能测试

性能测试需要在 Docker 外面的环境中进行，此处假设已经安装并配置好了 TPU 硬件对应的 runtime 环境。退出 Docker 环境：

```
$ exit
```

PCIE 板卡

PCIE 板卡下运行以下命令，测试生成的高优先级模型的 bmodel 性能：

```
$ cd model-zoo
$ python3 -m tpu_perf.run --target BM1684X --mlir -l full_cases.txt --priority_filter high
```

其中，--target 用于指定处理器型号，目前支持 BM1684、BM1684X、BM1688、BM1690、CV186X。

注意：如果主机上安装了多块 SOPHGO 的加速卡，可以在使用 tpu_perf 的时候，通过添加 --devices id 来指定 tpu_perf 的运行设备：

```
$ python3 -m tpu_perf.run --target BM1684X --devices 2 --mlir -l full_cases.txt --priority_filter high
```

SoC 设备

SoC 设备使用以下步骤，测试生成的高优先级模型的 bmodel 性能。

```
$ cd model-zoo
$ python3 -m tpu_perf.run --target BM1684X --mlir -l full_cases.txt --priority_filter high
```

输出结果

运行结束后，性能数据在 output/stats.csv 中可以获得。该文件中记录了相关模型的运行时间、计算资源利用率和带宽利用率。下方为 resnet18-v2 的性能测试结果：

name,prec,shape,gops,time(ms),mac_utilization,ddr_utilization,processor_usage
resnet18-v2,FP32,1x3x224x224,3.636,6.800,26.73%,10.83%,3.00%
resnet18-v2,FP16,1x3x224x224,3.636,1.231,18.46%,29.65%,2.00%
resnet18-v2,INT8,1x3x224x224,3.636,0.552,20.59%,33.20%,3.00%
resnet18-v2,FP32,4x3x224x224,14.542,26.023,27.94%,3.30%,3.00%
resnet18-v2,FP16,4x3x224x224,14.542,3.278,27.73%,13.01%,2.00%
resnet18-v2,INT8,4x3x224x224,14.542,1.353,33.59%,15.46%,2.00%

14.8.3 精度测试

注意：由于 SoC 设备 CPU 资源有限，不推荐进行精度测试，因此 SoC 设备测试可以跳过精度测试部分

精度测试需要在 Docker 外面的环境中进行，此处假设已经安装并配置好了 TPU 硬件对应的 runtime 环境。退出 Docker 环境：

```
$ exit
```

PCIE 板卡下运行以下命令，测试生成的高优先级模型的 bmodel 精度：

```
$ cd model-zoo
$ python3 -m tpu_perf.precision_benchmark --target BM1684X --mlir -l full_cases.txt --priority_
→filter high
```

其中，--target 用于指定处理器型号，目前支持 BM1684、BM1684X、BM1688、BM1690、CV186X。

注意：

- 如果主机上安装了多块 SOPHGO 的加速卡，可以在使用 tpu_perf 的时候，通过添加 --devices id 来指定 tpu_perf 的运行设备。如：

```
$ python3 -m tpu_perf.precision_benchmark --target BM1684X --devices 2 --mlir -l full_cases.txt --
→priority_filter high
```

- BM1688、BM1690、CV186X 精度测试需要额外配置以下环境变量：

```
$ export SET_NUM_SAMPLES_YOLO=200
$ export SET_NUM_SAMPLES_TOPK=100
$ export SET_NUM_SAMPLES_BERT=200
```

具体参数说明可以通过以下命令获得：

```
$ python3 -m tpu_perf.precision_benchmark --help
```

输出的精度数据在 output/topk.csv 中可以获得。下方为 resnet18-v2 的精度测试结果：

```
name,top1,top5
resnet18-v2-FP32,69.68%,89.23%
resnet18-v2-INT8,69.26%,89.08%
```

14.9 FAQ

此章节列出一些 tpu_perf 安装、使用中可能会遇到的问题及解决办法。

14.9.1 invalid command ‘bdist_wheel’

tpu_perf 编译之后安装，如提示如下图错误，由于没有安装 wheel 工具导致。

```
[root@localhost build]# ls
bdist.sh blob_pb2.py blob.pb.cc blob.pb.h CMakeCache.txt CMakeFiles cmake_install.cmake libpipeline.so Makefile
[root@localhost build]# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/build
[root@localhost build]# make install/strip -j4
[ 9%] Built target proto
[100%] Built target pipeline
Installing the project stripped...
3-- Install configuration: ""
3-- Installing: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.libpipeline.so
3-- Set runtime path of "/1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.libpipeline.so" to ""
1-- Up-to-date: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.blob_pb2.py
1-- Up-to-date: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.blob_pb.cc
1usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
1  or: setup.py --help [cmd1 cmd2 ...]
1  or: setup.py --help-commands
1  or: setup.py cmd --help

1error: invalid command 'bdist_wheel'
4CMake Error at /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/cmake/postinst.cmake:6 (message):
  Failed to build python wheel
Call Stack (most recent call first):
  cmake_install.cmake:71 (include)
```

则先运行：

```
$ pip3 install wheel
```

再安装 whl 包

14.9.2 not a supported wheel

tpu_perf 编译之后安装，如提示如下图错误，由于 pip 版本导致。

```
[cpu_perf-9.9.9-py3-none-manylinux2014_aarch64.whl
[root@localhost dist]# pip3 install tpu_perf-9.9.9-py3-none-manylinux2014_aarch64.whl
WARNING: Running pip install with root privileges is generally not a good idea. Try `pip3 install --user` instead.
tpu_perf-9.9.9-py3-none-manylinux2014_aarch64.whl is not a supported wheel on this platform.
```

则先运行：

```
$ pip3 install --upgrade pip
```

再安装 whl 包

14.9.3 no module named ‘xxx’

安装运行 model-zoo 所需的依赖时，如提示如下图错误，由于 pip 版本导致。

```
test@test:~/big/lhy_test/040irci-tpu-nntc-internal-model-zoo-1684x-full/sophon/model-zoo$ pip3 install -r requirements.txt
Collecting opencv-python (from -r requirements.txt (line 1))
  Using cached https://files.pythonhosted.org/packages/40/93/655af887bafce2a655998f53b9bd21ad94b0627d81d44aef35c79f40de6/opencv-python-4.7.0.72.tar.gz
    Complete output from command python setup.py egg_info:
    Traceback (most recent call last):
      File "<string>", line 1, in <module>
        File "/tmp/pip-build-9hcok9tk/opencv-python/setup.py", line 10, in <module>
          import skbuild
    ModuleNotFoundError: No module named 'skbuild'

    ----------------------------------------
Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build-9hcok9tk/opencv-python/
```

则先运行：

```
$ pip3 install --upgrade pip
```

再安装运行 model-zoo 所需的依赖

14.9.4 精度测试因为内存不足被 kill

对于 YOLO 系列的模型精度测试，可能需要 4G 左右的内存空间。SoC 环境如果存在内存不足被 kill 的情况，可以参考 SOPHON BSP 开发手册的板卡预制内存布局章节扩大内存。

CHAPTER 15

附录 05：TPU Profile 工具使用指南

本章节主要是介绍如何利用 Profile 数据及 TPU Profile 工具，可视化模型的完整运行流程，以便于进行模型性能分析。当前 Profile 工具支持 BM1684 , BM1684X , BM1688 , CV186X , BM1690 。

15.1 编译 bmodel

TPU Profile 是将 Profile 数据转换为可视化网页的工具。首先先生成 bmodel，下面以 tpu-mlir 工程中的 yolov5s 模型来演示。

由于 Profile 数据会把编译中的一些 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` 时加上 `--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。在 Docker 内生成 bmodel 的命令如下：

```
# 生成 top mlir
$ model_transform \
  --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 \
  --mlir yolov5s.mlir \
  --quantize F16 \
  --processor BM1684X \
  --test_input yolov5s_in_f32.npz \
  --test_reference yolov5s_top_outputs.npz \
  --model yolov5s_1684x_f16.bmodel \
  --debug
```

通过以上命令，将 yolov5s.onnx 编译成了 yolov5s_bm1684x_f16.bmodel，其中，--debug 参数将记录 profile 数据。

15.2 生成 Profile 原始数据

将生成的 `yolov5s_bm1684x_f16.bmodel` 拷贝到已安装 `libsophon` 的运行环境上。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 `bmodel` 前设置环境变量 `BMRUNTIME_ENABLE_PROFILE=1` 即可。然后用 `libsophon` 中提供的模型测试工具 `bmrt test` 运行 `bmodel`，生成 `profile` 数据。在 Docker 外执行如下命令：

```
export BMRUNTIME_ENABLE_PROFILE=1  
bmrt test --bmodel yolov5s 1684x f16.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: 1
[BMRT][show_net_info:1520] INFO: #####
[BMRT][show_net_info:1521] INFO: NetName: yolov5s, Index=0
[BMRT][show_net_info:1523] INFO: ---- stage 0 ----
[BMRT][show_net_info:1532] INFO: Input 0) 'images' shape=[ 1 3 640 640 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 0) '350_Transpose_f32' shape=[ 1 3 80 80 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 1) '498_Transpose_f32' shape=[ 1 3 40 40 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1542] INFO: Output 2) '646_Transpose_f32' shape=[ 1 3 20 20 85 ] dtype=FLOAT32 scale=1 zero_point=0
[BMRT][show_net_info:1545] INFO: #####
[BMRT][bmrt_test:782] INFO:==== running network #0, name: yolov5s, loop: 0
[BMRT][bmrt_test:868] INFO:reading input #0, bytessize=4915200
[BMRT][print_array:706] INFO: --> input_data: < 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:78] INFO:bdu 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 BMRTIME_ENABLE_PROFILE=1 *****
[BMRT][print_note:96] INFO: Note: BMRTime will collect time data during running      *
[BMRT][print_note:97] INFO: that will cost extra time.                                * Profile Mode Tips to avoid misuse during formal deploying
[BMRT][print_note:98] INFO: Close PROFILE Mode by "unset BMRTIME_ENABLE_PROFILE"
[BMRT][print_note:99] INFO:*****
[BMRT][bmrt_test:1005] INFO:reading output #0, bytessize=6528000
[BMRT][print_array:706] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1632000
[BMRT][bmrt_test:1005] INFO:reading output #1, bytessize=1632000
[BMRT][print_array:706] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=408000
[BMRT][bmrt_test:1005] INFO:reading output #2, bytessize=408000
[BMRT][print_array:706] INFO: --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=102000
[BMRT][bmrt_test:1039] INFO:net[yolov5s] stage[0], launch total time is 375609 us (npu 5650 us, cpu 369959 us) CPU Time is not accuracy on Profile Mode
[BMRT][bmrt_test:1042] INFO:++ The network[yolov5s] stage[0] output data ++
[BMRT][print_array:706] INFO:output data #0 shape: [1 3 80 80 85 ] < 0.30957 -0.289551 0.0744629 -0.203003 -11.9375 -1.54297 -5.25391 -3.05859 -5.39453 -5.64844 -9.26172 ... > 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 -4.6.63672 ... > 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
```

图 15.1: 开启 Profile 后运行输出的目录

运行完成后会在当前目录生成 `bmprofile_data-1` 文件夹, 为全部的 Profile 数据。

15.3 可视化 Profile 数据

将 `bmprofile_data-1` 目录拷贝回 Docker 内的 tpu-mlir 工程环境。tpu-mlir 提供了 `tpu_profile` 脚本, 来把生成的二进制 profile 数据转换成网页文件并进行可视化。在 Docker 内执行如下命令:

```
# 将bmprofile_data-1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 bmprofile_data-1 bmprofile_out --arch BM1684X
ls bmprofile_out
# echarts.min.js profile_data.js result.html
```

对于 BM1688 或 CV186X 的模型, 若要使 profile 中带有 layer 信息, 需要额外拷贝 `yolov5s_1684x_f16` (与 bmodel 同名) 目录下的 `tensor_location.json` 和 `final.mlir` 到 `bmprofile_data-1` 目录中。BM1690 暂不支持显示模型 layer 信息。

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, 可通过如下命令进行查看:

```
tpu_profile --help
```

在开发手册中有更详细的 Profile 说明, 可以参考。

CHAPTER 16

附录 06：TDB 调试工具使用指南

本章节主要介绍 TDB(Tensor Debugger) 工具的使用方法，TDB 提供了一个和 gdb、pdb 界面类似的调试窗口，可用于调试 BModel 运行流程，具有添加断点、单步执行、查看内存数据、数据比对、等功能。

此工具目前支持 BM1684、BM1684X、BM1688。

16.1 准备工作

环境配置

首先需要参考[开发环境配置章节](#) 完成环境配置，进入 TPU-MLIR 的 Docker 容器，并在其中安装 tpu_mlir。

若已完成环境配置可忽略此步骤。

生成 bmodel

在使用 TDB 之前需要先通过 TPU-MLIR 生成 bmodel 文件，可参考[编译 ONNX 模型章节](#)中的命令从模型生成 bmodel 文件。

需要使用以下 2 个命令：

```
# 将ONNX模型转换为top_mlir
$ model_transform
# 将top_mlir转换为bmodel
$ model_deploy
```

其中，model_deploy 命令需要添加 --debug 和 --compare_all 参数，用于保存 tpu_output.npz 文件并保留中间数据。

生成 bmodel 时，会自动产生带有 compilation.bmodel 和 final.mlir 文件的目录，此目录称为 Context 目录。

16.2 启动 TDB

```
$ tdb [-h]
  [--inputs [INPUTS]]
  [--ref_data [REF_DATA ...]]
  [--plugins [PLUGINS]]
  [--ddr_size [DDR_SIZE]] [-v]
  [context_dir]
```

tdb 命令的主要参数说明如下：

表 16.1: tdb 参数功能

参数名	必选？	说明
context_dir	是	bmodel 文件所在目录，默认为当前目录
-h, --help	否	显示帮助信息
--inputs	否	指定 bmodel 文件的输入数据
--ref_data	否	指定 bmodel 文件的参考数据
--plugins	否	添加额外的插件
--ddr_size	否	指定 cmodel 的 ddr_size
-v, --verbose	否	使用进度条

启动 TDB 示例：

```
$ tdb
# 等效于
$ tdb ./
```

16.3 TDB 命令汇总

在进入 TDB 后，按下两次 tab 可以获取命令提示。显示效果如下：

```
(tdb)
EOF      continue    enable     n          print      r          skip       w
b        delete      help       next      py         reload    start      watch
break    disable     info       p          q          run       static-check
c        display    list      plugin   quit      s          status
```

进入 TDB 后，可使用的命令如下：

表 16.2: TDB 命令汇总

命令	说明
s/start	加载 bmodel 并进行初始化
r/run	从头执行到结束, run 指令包含初始化功能
b/break	在 final.mlir 中添加断点
delete	删除断点
n/next	执行下一条指令, 可以使用 n [num] 执行多条指令
c/continue	继续执行指令, 直至断点或运行结束
info	打印断点信息或不同格式的指令
p/print	打印当前指令或指令对应的数据
w/watch	监视当前或上一条原子指令的某个输入/输出, 当其所在地址的数据变化时返回提示
q/quit	退出 TDB
py [py_cmd]	在 TDB 中执行 python 命令, 集成了 pdb 的代码补全功能

其中, num 为数字; py_cmd 为 python 命令。

16.4 TDB 使用流程

```
# 在context目录启动TDB
$ cd path/to/context_dir
$ tdb
# 初始化
$ s
# 逐条执行
$ n
# 添加断点
$ b
# 继续运行
$ c
# 继续调试
$ info/p/w
# 退出
$ q
```

16.5 TDB 功能说明

16.5.1 next 功能

```
# 使用next单步执行
(tdb) n
# 使用next执行多条指令
```

(续下页)

(接上页)

```
(tdb) n [num]
# 使用next执行3条指令
(tdb) n 3
```

n 命令后显示的指令为下一条未执行指令。

16.5.2 breakpoint 功能

breakpoint 功能包含查看断点、添加/删除断点、开启/关闭断点功能。使用方法如下：

表 16.3: breakpoint 功能

命令	说明	示例
info b/break	查看断点信息	info b; info break
b/break	添加断点	b 1
enable	开启断点	enable 1; enable 1,2
disable	关闭断点	disable 1; disable 1,2
delete	删除断点	delete 1

目前支持的断点类型如下：

value-id

bmodel 对应的 final.mlir 中的 Operation 前缀，例如：

```
%140 = "tpu.Load"(%6) {do_bcast = false ...}
```

其中，%140 和 %6 即为 value-id，添加此类型断点示例如下：

```
(tdb) b %140
(tdb) b %6
```

op-name

final.mlir 中的 Operation 的名称，上述例子中，tpu.Load 即为 Op 名称，添加此类型断点示例如下：

```
(tdb) b tpu.Load
```

cmd-id

解析出的 asm 的 cmd-id，上述例子中，D1 和 B0 即为 cmd-id，添加此类型断点示例如下：

```
(tdb) b D2
(tdb) b B4
```

16.5.3 info 功能

info 功能可以打印断点信息或不同格式的指令，使用方法如下：

info b

查看断点信息。

```
(tdb) info b
index  type enable  text hit
  1 dialect      y tpu.load  0
  2 addr        y     R0  3
  3 cmd-id      y     D1  0
  4 value-id    y    %7  0
```

info asm

查看当前的 asm 指令。

```
(tdb) info asm
%R0, %B15 = "arith.add"(%R13, %C1.0, %D3) {round_mode = 0} : (memref<1x32x54x160xf32, F
→ strides: [8640, 8640, 160, 1] >, f32, none) -> (memref<1x32x54x160xf32, strides: [8640, 8640, 160,
→ 1] >, none)
```

info mlir

查看当前指令对应在 final.mlir 中的 Operation。

```
(tdb) info mlir
%137 = "tpu.Active"(%134) {ginfo = #tpu.lg<out_addr = 212992, out_size = 35456, buffer_
→addr = 0, buffer_size = 71040, eu_align = true, n_idx = [0], n_slice = [1], c_idx = [0], c_
→slice = [32], d_idx = [0], d_slice = [1], h_idx = [0, 53, 107, 161, 215, 267], h_slice = [54, 55, 55,
→ 55, 53, 53], w_idx = [0, 159], w_slice = [160, 161], id = 6, stage = 1, group_type = 0>, mode[F
→= #tpu<active_mode SILU>} : (tensor<1x32x320x320xf32>) -> tensor<1x32x320x320xf32>[F
→loc(#loc19)
```

info reg

查看当前指令解析后各字段的值。

```
(tdb) info reg
{'cmd_short': 1, 'cmd_id': 15, 'cmd_id_dep': 3, 'tsk_typ': 3, 'tsk_eu_typ': 2, 'opd0_const': 0,
→'opd1_const': 1, 'opd2_const': 0, 'tsk_opd_num': 2, 'cmd_id_en': 1, 'pwr_step': 0, 'intr_en': 0,
→'res0_prec': 2, 'opd0_prec': 2, 'opd1_prec': 2, 'opd2_prec': 0, 'opd0_sign': 1, 'opd1_sign': 1,
→'res0_str': 0, 'opd0_str': 0, 'opd1_str': 0, 'opd2_n_str': 0, 'rsvd0': 0, 'res0_n': 1, 'res0_c': 32,
→'res0_h': 54, 'res0_w': 160, 'res0_addr': 0, 'opd0_addr': 212992, 'opd1_addr': 1065353216,
→'opd2_addr': 0, 'res0_n_str': 0, 'res0_c_str': 0, 'opd0_n_str': 0, 'opd0_c_str': 0, 'opd1_n_str':
→: 0, 'opd1_c_str': 0, 'res0_h_str': 0, 'res0_w_str': 0, 'opd0_h_str': 0, 'opd2_sign': 0, 'rsvd1': 0,
→'opd0_w_str': 0, 'opd1_h_str': 0, 'opd1_w_str': 0, 'rsvd2': 0}
```

info loc

查看 Context 目录中， tensor_location.json 中对应的 Operation 信息。

```
(tdb) info loc
{'core_id': 0,
'file_line': 27,
'loc_index': 4,
'opcode': 'tpu.Active',
'operands': [@163840({name=122_Conv, layout=eu_align, slice=[0:1, 0:32, 0:1, 0:54, 0:160], mlir_
→type=tensor<1x32x320x320xf32>, memory_type=<1x32x54x160xf32>})],
'results': [@212992({name=124_Mul, layout=eu_align, slice=[0:1, 0:32, 0:1, 0:54, 0:160], mlir_
→type=tensor<1x32x320x320xf32>, memory_type=<1x32x54x160xf32>})],
'slice_all': False,
'subnet_id': 0,
'tiu_dma_id_after': [17, 3],
'tiu_dma_id_before': [1, 3]}
```

16.5.4 print 功能

print 功能不仅可以打印当前的 asm 指令，还可以打印指令的输入和输出数据，使用方法如下：

表 16.4: print 功能

命令	说明	示例
p op	查看即将执行的指令	p op
p pre/next	查看上一条或下一条指令	p pre; p next
p in	查看下一条未执行指令的输入数据	p in; p in 0
p out	查看上一条已执行指令的输出数据	p out; p out 0

16.5.5 watchpoint 功能

watchpoint 功能可以监视指令的输入/输出数据，当某个监视变量的数据发生变化时会返回提示，使用方法如下：

w

查看当前已添加的 watchpoint，示例如下：

```
(tdb) w
index cmd_type cmd_id core_id enabled value
1 CMDType.dma 2 0 y %G0: memref<1x32x3x36xf32, strides: [3456, 108, 36, 1]>
```

w in

将下一条待执行指令的某个输入添加为 watchpoint，示例如下：

```
(tdb) n
%R15.2688, %D2 = "dma.tensor"(%G0, %B0) {decompress = False} : (memref<1x32x3x36xf32, [F
→strides: [3456, 108, 36, 1]>, none) -> (memref<1x32x3x36xf32, strides: [108, 108, 36, 1]>, none)
```

(续下页)

(接上页)

```
(tdb) w in 0
(tdb) w
index cmd_type cmd_id core_id enabled value
 1 CMDType.dma   2   0     y %G0: memref<1x32x3x36xf32, strides: [3456, 108, 36, 1]>
```

可以看到，w in 0 将下一条待执行指令的第一个输入%G0 添加为 watchpoint。

w out

将上一条已执行指令的某个输出添加为 watchpoint，示例如下：

```
(tdb) w out 0
(tdb) w
index cmd_type cmd_id core_id enabled value
 1 CMDType.dma   2   0     y %G0: memref<1x32x3x36xf32, strides: [3456, 108, 36, F
→ 1]>
 2 CMDType.dma   1   0     y %R0: memref<1x3x110x322xf32, strides: [35424, 35424, 322,
→ 1]>
```

p w idx old/now

打印已添加 watchpoint 的值，示例如下：

其中 idx 是使用 w 命令返回的 watchpoint 的 index，old 表示查看该 watchpoint 最初被添加时的数据，now 表示查看 watchpoint 当前数据。

old/now 可省略，默认为 now，即查看 watchpoint 当前数据。

```
(tdb) w
index cmd_type cmd_id core_id enabled value
 1 CMDType.dma   2   0     y %G0: memref<1x32x3x36xf32, strides: [3456, 108, 36, F
→ 1]>
 2 CMDType.dma   1   0     y %R0: memref<1x3x110x322xf32, strides: [35424, 35424, 322,
→ 1]>
(tdb) p w 1
(tdb) p w 1 old
```

w delete [idx]

删除已添加的 watchpoint，示例如下：

当输入 idx 时表示删除对应的 watchpoint，不输入 idx 时表示删除全部的 watchpoint。

```
(tdb) w
index cmd_type cmd_id core_id enabled value
 1 CMDType.dma   2   0     y %G0: memref<1x32x3x36xf32, strides: [3456, 108, 36, F
→ 1]>
 2 CMDType.dma   1   0     y %R0: memref<1x3x110x322xf32, strides: [35424, 35424, 322,
→ 1]>
 3 CMDType.tiu   11   0     y %R13: memref<1x32x54x160xsi16, strides: [8640, 8640, 160,
→ 1]>
(tdb) w delete 1
```

(续下页)

(接上页)

```
(tdb) w
index cmd_type cmd_id core_id enabled
  2 CMDType.dma   1    0      y %R0: memref<1x3x110x322xf32, strides: [35424, 35424, 322,
→ 1]>
  3 CMDType.tiu  11    0      y %R13: memref<1x32x54x160xi16, strides: [8640, 8640, 160,
→ 1]>
(tdb) w delete
(tdb) w
index cmd_type cmd_id core_id enabled value
```

16.5.6 py 功能

py 功能可以在 TDB 环境下直接执行 python 命令，使用方法如下：

```
(tdb) py a = 2
(tdb) py b = a + 2
(tdb) py print(b)
4
```

16.6 BModel Disassembler

BModel Disassembler 可以对 bmodel 文件进行反汇编得到 MLIR 格式的 atomic 指令的汇编代码，即 asm 指令，用于分析模型的最终运行命令。

使用时首先需要进入 Context 目录，使用方法如下：

```
$ bmodel_dis [-h] [--format {mlir,reg,bits,bin,reg-set}] bmodels [bmodels ...]
```

其中，--format 可以指定输出格式，默认使用 mlir 格式，bmodels 表示要解析的 bmodel 文件。使用示例如下：

```
$ bmodel_dis compilation.bmodel
$ bmodel_dis --format reg compilation.bmodel
```

可将输出结果保存至文件，方法如下：

```
$ bmodel_dis compilation.bmodel > dis_bmodel.mlir
$ bmodel_dis --format reg compilation.bmodel > dis_reg.json
```

16.7 BModel Checker

BModel Checker 用于查找 bmodel 中的错误 (codegen 错误)，如果在 model_deploy 时发现生成的 bmodel 无法与 tpu 的参考数据对齐，则可以使用该工具来定位错误。目前支持 BM1684、BM1684X、BM1688 处理器的 BModel。

在生成 bmodel 文件时，model_deploy 命令需要添加 --debug 和 --compare_all 参数，用于保存 tpu_output.npz 文件并保留中间数据。

使用方法如下：

```
$ bmodel_checker [-h]
    [--tolerance TOLERANCE]
    [--report REPORT] [--fail_fast]
    [--quiet] [--no_interactive]
    [--dump_mode {failed,all,never}]
    context_dir reference_data
```

bmodel_checker 的主要参数说明如下：

表 16.5: bmodel_checker 参数功能

参数名	必选？	说明
context_dir	是	bmodel 文件所在目录
reference_data	是	tpu_output.npz 文件位置
quiet	否	不显示执行进度条
fail_fast	否	在发现第一个错误的时候就停下来
dump_mode	否	指定 dump 命令下载的数据，默认为 failed，还可以是 all 或 never
tolerance	否	指定比较容差，默认为“0.99,0.90”
report	否	将错误结果输出成文件，默认为 failed_bmodel_outputs.npz
no_interactive	否	运行完 bmodel_checker 会直接退出 TDB 模式
cache_mode	否	缓存模式，有 online, offline, generate 三种选项，默认为 online

使用 bmodel_checker 需要进入 Context 目录，使用示例如下：

```
$ bmodel_checker ./..../yolov5s_bm1684x_f32_tpu_outputs.npz
$ bmodel_checker ./..../yolov5s_bm1684x_f32_tpu_outputs.npz --fail_fast
$ bmodel_checker ./..../yolov5s_bm1684x_f32_tpu_outputs.npz --tolerance 0.99,0.90
```

执行 bmodel_checker 命令后，会输出检查报告，并将错误的输出结果保存到 failed_bmodel_outputs.npz 文件中，下面对检查报告进行说明：

其中，“对勾”表示通过，即该数据被检查，且其相似度符合 $\cos>0.99$, $eul>0.9$ (默认阈值，可通过 tolerance 参数修改)；“叉号”表示错误，即该数据没有达到要求的相似度；“问号”表示未知，即没有找到对应的参考数据，无法确定此数据的正确性。一个 yolov5s 模型的完整检查报告如下图所示：

Index	Check-Line-Operand-Result[✓] Summary										通过标识
	检查输入状态					检查输出状态					
0	1	2	3	4	6	7	8	9			
0	(15✓ ?)	(16? ✓)	(17✓ ?)	(18? ✓)	(19✓✓ ✓)	(20✓ ?)	(21? ✓)	(22✓✓ ✓)	(23✓ ✓)	(24✓ ✓)	

输出检查报告后会自动进入交互模式。交互模式可提供对错误的详细浏览，而且还可以快速在不同行之间跳转，下面以一个 `cswin_tiny` 模型为例展示其使用方法。

check summary

使用 `check summary` 命令可以重新打印检查报告，使用示例如下：

值得一提的是，使用 `check summary reduce` 命令可以聚合相同行号的输入和输出。

check data

(tdb) check data [file-line]

其中，file-line 为检查报告中的行号，对应 final.mlir 的行号。此命令可以给出 file-line 对应指令的所有输入输出数据的描述信息，使用示例如下：

(tdb) check data [file-line] [index]

其中，index 为 check data [file-line] 命令输出数据的 index。此命令可以给出对应 index 数据的详细信息。对比正确的数据示例如下：

对比错误的数据示例如下：

SeC 设备

当在 SoC 设备上执行时，为了在不引入 mlir 依赖的情况下执行比对，需要先在 Docker 环境

```
(tdb) check summary
(13?|?) (14?|?) (23?|?) (26?|?) (24?|?) (25?|?) (27?|?) (28?|?) (29?|?) (32?|?) (31?|?) (35?|?) (33?|?) (34?|?) (36?|?) (37?|?) (38?|?)
(42?|?) (43?|?) (45?|?) (47?|?) (49?|?) (52?|?) (53?|?) (56?|?) (58?|?) (60?|?) (63?|?) (65?|?) (67?|?) (68?|?) (70?|?) (71?|?) (73?|?) (74?|?)
(75?|?) (76?|?) (77?|?) (88?|?) (81?|?) (84?|?) (86?|?) (87?|?) (88?|?) (89?|?) (92?|?) (95?|?) (98?|?) (100?|?) (102?|?) (103?|?) (105?|?)
(106?|?) (108?|?) (109?|?) (110?|?) (111?|?) (112?|?) (115?|?) (116?|?) (131?|?) (132?|?) (135?|?) (133?|?) (134?|?) (137?|?) (136?|?) (138?|?)
(141?|?) (142?|?) (142?|?) (145?|?) (143?|?) (144?|?) (148?|?) (146?|?) (147?|?) (150?|?) (149?|?) (151?|?) (152?|?) (156?|?)
(154?|?) (155?|?) (157?|?) (158?|?) (159?|?) (160?|?) (163?|?) (166?|?) (168?|?) (167?|?) (169?|?) (170?|?) (171?|?) (174?|?) (182?|?) (185?|?)
(183?|?) (184?|?) (186?|?) (189?|?) (187?|?) (188?|?) (191?|?) (190?|?) (194?|?) (192?|?) (193?|?) (195?|?) (196?|?) (197?|?) (201?|?) (202?|?)
(204?|?) (206?|?) (208?|?) (211?|?) (213?|?) (216?|?) (218?|?) (221?|?) (224?|?) (226?|?) (228?|?) (230?|?) (232?|?) (231?|?) (233?|?) (234?|?)
(238?|?) (239?|?) (240?|?) (241?|?) (242?|?) (246?|?) (247?|?) (248?|?) (250?|?) (253?|?) (255?|?) (256?|?) (257?|?) (261?|?) (264?|?) (267?|?)
(269?|?) (271?|?) (273?|?) (275?|?) (274?|?) (276?|?) (277?|?) (281?|?) (282?|?) (283?|?) (284?|?) (285?|?) (289?|?) (290?|?) (291?|?) (293?|x)
(312?|?) (314?|x) (313?|x) (317?|x) (315?|?) (316?|?) (319?|?) (318?|?) (320?|_) (323|_) (321?|?) (322?|?) (324?|?) (327|_) (325?|?) (326?|?)
(330?|x|?) (328?|?) (329?|?) (332?|?) (331?|?) (333?|?) (335?|?) (334?|?) (338?|?) (336?|?) (337?|?) (339?|?) (340?|?) (341?|x) (344?|x)
(342?|?) (343?|?) (346?|?) (345?|_) (349?|_) (347?|?) (348?|?) (350?|?) (351?|_) (352?|_) (356?|x) (357?|x) (359?|x) (361?|x) (363?|x) (366?|x)
(368?|x) (371?|x) (373?|x) (376?|x) (379?|x) (381?|x) (383?|x) (385?|x) (387?|?) (386?|?) (388?|?) (389?|?) (393?|?) (394?|?) (395?|?) (396?|?)
(397?|?) (401?|?) (402?|x) (403?|x) (405?|x) (408?|x) (410?|x) (411?|x) (412?|x) (413?|x) (417?|x) (420?|x) (423?|x) (425?|x) (427?|x) (429?|x)
(431?|?) (430?|x) (432?|?) (433?|?) (437?|?) (438?|?) (439?|?) (440?|?) (441?|?) (445?|?) (446?|x) (447?|x) (449?|x) (464?|?) (466?|x) (465?|x)
(469?|x) (467?|?) (468?|?) (471?|?) (470?|x) (472?|_) (475?|_) (473?|?) (474?|?) (476?|_) (479?|_) (477?|?) (478?|?) (482?|?) (480?|?) (481?|?)
(484?|?) (483?|?) (485?|?) (487?|?) (486?|?) (490?|?) (488?|?) (489?|?) (491?|?) (492?|?) (493?|x) (494?|x) (497?|x) (500?|x) (502?|x) (501?|x)
(503?|x) (504?|_) (505?|_) (508?|_) (516?|?) (519?|?) (517?|?) (518?|?) (520?|?) (523?|x) (521?|?) (522?|?) (525?|?) (524?|?) (527?|x) (526?|?)
```

(tdb) check data 291							
291	loc_index	owner	value name	value type	compare	index	
0	205	tpu.Add	/stage2.0/attns.1/MatMul_1_output_0_MatMul	operand	✓	0	
1	205	tpu.Add	/stage2.0/attns.1/Transpose_7_output_0_Transpose	operand	✓	1	
2	205	tpu.Add	/stage2.0/attns.1/Add_output_0_Add	result	✗	2	

(tdb) check data 291 0	
value-Info	
opcode	tpu.Add
core_id	0
value	{ "address": 687195987968, "layout": "continuous", "memory_type": "<14x2x56x32xsi8>", "name": "/stage2.0/attns.1/MatMul_1_output_0_MatMul", "reshape": "", "slice": "[...]", "type": "tensor<14x2x56x32x!quant.uniform<i8:f32, 0.033072317322834645>, 687195987968 : i64>" }
tiu_dma_id (before codegen)	[7959, 553]
tiu_dma_id (after codegen)	[7971, 565]
compare	✓
value type	operand

```
(tdb) check data 291 2
      value-Info
opcode          tpu.Add
core_id          0
value           {
    "address": 687195934720,
    "layout": "continuous",
    "memory_type": "<14x2x56x32x1b>",
    "name": "/stage0.0/attns.1/Add_output_0_Add",
    "reshape": "",
    "slice": "[...,]",
    "type": "tensor<14x2x56x32x1b!quant.uniform<8:f32, 0.018248182677165353>, 687195934720 : 164>"
}
tiu_dma_id (before codegen) [7959, 555]
tiu_dma_id (after codegen) [7971, 565]
compare         x
value type      result
      data-error
Not equal to tolerance cos=0.99, euc=0.9
cosine similarity: 0.089906
euclidean similarity: 0.897553
top10 diff:
x: [ 0.310219, 0.474453, 0.474453, 0.310219, 0.474453, 0.474453, 0.474453, -0.529197, 0.474453]
y: [ 0.693431, 0.20073 , 0.20073 , 0.583942, 0.20073 , 0.20073 , 0.20073 , -0.273723, 0.237226]
0:10 data:
x: [-0.127737, -0.054745, 0.86292 , -0.127737, 0.328467, -0.273723, -0.036496, 1.478103, 0.        , -0.237226]
y: [ -0.20073 , -0.018248, 0.766424, -0.01241 , 0.291971, 0.291971, -0.072993, 1.423358, 0.        , -0.237226]

Not equal to tolerance rtol=0.001, atol=0.1
Mismatched elements: 641 / 50176 (1.28%)
Max absolute difference: 0.3832182
Max relative difference: 11.000001
      asm
%G2.1196032, %D564C0 = "dma.tensor"(%R8, %B7968C0) : (memref<4x2x56x32x1b, strides: [1792, 1792, 32, 1]>, none) -> (memref<4x2x56x32x1b, strides: [3584, 1792, 32, 1]>, none)
%G2.1210368, %D565C0 = "dma.tensor"(%R14, %B7971C0) : (memref<2x2x56x32x1b, strides: [1792, 1792, 32, 1]>, none) -> (memref<2x2x56x32x1b, strides: [3584, 1792, 32, 1]>, none)
=> %G2.0, %D566C0 = "dma.tensor"(%G2.1196032, %B7971C0) : (memref<14x2x56x32x1b, strides: [3584, 1792, 32, 1]>, none) -> (memref<14x2x56x32x1b, strides: [3584, 32, 64, 1]>, none)
%R4, %D567C0 = "dma.tensor"(%G2.0, %B7971C0) : (memref<1x784x4x1x1b, strides: [15016, 64, 1, 1]>, none) -> (memref<1x784x64x1x1b, strides: [1600, 64, 1, 1]>, none)
%R2, %B7972C0 = "arith.nul_satu"(%R4, %C17, %C252, %D567C0) : (round_mode = 5) : (memref<1x784x64x1x1b, strides: [1600, 64, 1, 1]>, none) -> (memref<1x784x64x1x1b, strides: [1600, 64, 1, 1]>, none)
```

内生成缓存，随后在 SoC 设备环境下使用缓存模型比对模型。

```
$ bmodel_checker ./../yolov5s_bm1684x_f32_tpu_outputs.npz --cache_mode generate # on TF
→ docker
$ bmodel_checker ./../yolov5s_bm1684x_f32_tpu_outputs.npz --cache_mode offline # on soc
```

CHAPTER 17

附录 07：已支持的算子

17.1 本章节主要提供目前 TPU-MLIR 支持的算子列表

表 17.1: A

Onnx	Pytorch	Caffe	TOP
Abs	aten::abs	AbsVal	top.A16MatMul
Acos	aten::acos	ArgMax	top.Abs
Add	aten::adaptive_avg_pool1		top.AdaptiveAvgPool
And	aten::adaptive_avg_pool2		top.Add
ArgMax	aten::add		top.AddConst
ArgMin	aten::addmm		top.Arange
Atan	aten::arange		top.Arccos
Atanh	aten::argmax		top.Arctanh
AveragePool	aten::argmin		top.Arg
	aten::atan		top.Attention
	aten::atanh		top.AvgPool
	aten::avg_pool1d		
	aten::avg_pool2d		
	aten::avg_pool3d		

表 17.2: B

Onnx	Pytorch	Caffe	TOP
BatchNormaliza- tion	aten::baddbmm	BatchNorm	top.BatchNorm
	aten::batch_norm	BN	top.BatchNormBwd
	aten::bmm		top.BatchNormTrain top.BinaryConstShift top.BinaryShift

表 17.3: C

Onnx	Pytorch	Caffe	TOP
Cast	aten::cat	Concat	top.Cast
Ceil	aten::ceil	ContinuationIndi- cator	top.Ceil
Clip	aten::channel_shuffle	Convolution	top.Clip
Concat	aten::chunk	ConvolutionDepth- wise	top.Compare
Constant	aten::clamp	Crop	top.CompareConst
ConstantOfShape	aten::clone		top.Concat
Conv	aten::constant_pad_nd		top.ConstantFill
ConvTranspose	aten::contiguous		top.Conv
Correlation	aten::_convolution		top.ConvBwd_Weight
Cos	aten::_convolution_mode		top.Convbwd
CumSum	aten::copy		top.Copy
	aten::cos		top.Correlation
	aten::cosh		top.Cos
	aten::cumsum		top.Cosh
			top.Csc
			top.CumSum
			top.Custom

表 17.4: D

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

表 17.5: E

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

表 17.6: F

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

表 17.7: G

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

表 17.8: H

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

表 17.9: I

Onnx	Pytorch	Caffe	TOP
Identity	aten::index	ImageData	top.If
If	aten::index_put_	InnerProduct	top.IndexPut
InstanceNormal-	aten::index_put	Input	top.Input
ization	aten::index_select	Interp	top.InstanceNorm
	aten::instance_norm		top.Interp

表 17.10: L

Onnx	Pytorch	Caffe	TOP
LayerNormalization	aten::layer_norm	LRN	top.LRN
LeakyRelu	aten::leaky_relu	LSTM	top.LSTM
Less	aten::less	Lstm	top.LayerNorm
LessOrEqual	aten::linalg_norm		top.LayerNormBwd
Log	aten::linear		top.LayerNormTrain
LogSoftmax	aten::log		top.LeakyRelu
Loop	aten::log2		top.List
LRN	aten::log_sigmoid		top.Log
LSTM	aten::log_softmax		top.LogB
	aten::lstm		top.LogicalAnd
	aten::lt		top.Loop
			top.Lut

表 17.11: M

Onnx	Pytorch	Caffe	TOP
MatMul	aten::masked_fill	MatMul	top.MaskRCNN_BboxPooler
Max	aten::matmul	Mish	top.MaskRCNN_GetBboxB
MaxPool	aten::max		top.MaskRCNN_MaskPooler
Min	aten::max_pool1d		top.MaskRCNN_RPNGetBboxes
Mod	aten::max_pool2d		top.MaskedFill
Mul	aten::max_pool2d_with_		top.MatMul
	aten::max_pool3d		top.MatchTemplate
	aten::mean		top.Max
	aten::meshgrid		top.MaxConst
	aten::min		top.MaxPool
	aten::mish		top.MaxPoolWithMask
	aten::mm		top.MaxPoolingIndicesBwd
	aten::mul		top.MaxUnpool
	aten::mv		top.MeanRstd
			top.MeanStdScale
			top.MeshGrid
			top.Min
			top.MinConst
			top.Mish
			top.Mmap2Rgbmap
			top.Mod
			top.Mul
			top.MulConst

表 17.12: N

Onnx	Pytorch	Caffe	TOP
Neg	aten::native_batch_norm	Normalize	top.Nms
NonMaxSuppression	aten::native_group_norm		top.NonZero
NonZero	aten::native_layer_norm		top.None
Not	aten::ne		top.Normalize
	aten::neg		
	aten::new_full		
	aten::new_ones		
	aten::new_zeros		
	aten::nonzero		

表 17.13: O

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

表 17.14: P

Onnx	Pytorch	Caffe	TOP
Pad	aten::pad	Padding	top.PRelu
PixelNormalization	aten::permute	Permute	top.Pack
Pow	aten::pixel_shuffle	Pooling	top.Pad
PRelu	aten::pixel_unshuffle	Power	top.Permute
	aten::pow	PReLU	top.PixelNorm
	aten::prelu	PriorBox	top.PoolMask
		Proposal	top.Pow
			top.Pow2
			top.Pow3
			top.Preprocess
			top.PriorBox
			top.Proposal

表 17.15: Q

Onnx	Pytorch	Caffe	TOP
QuantizeLinear			top.QuantizeLinear

表 17.16: R

Onnx	Pytorch	Caffe	TOP
RandomNormalLike	aten::reflection_pad1d	Reduction	top.RMSNorm
Range	aten::reflection_pad2d	ReLU	top.ROIPooling
Reciprocal	aten::relu	ReLU6	top.RandnLike
ReduceL1	aten::remainder	Reorg	top.Range
ReduceL2	aten::repeat	Reshape	top.Reciprocal
ReduceLogSumExp	aten::replication_pad1d	RetinaFaceDetection	top.Reduce
ReduceMax	aten::replication_pad2d	Reverse	top.Relu
ReduceMean	aten::reshape	ROIPooling	top.Remainder
ReduceMin	aten::roll		top.Repeat
ReduceProd	aten::rsqrt		top.RequantFp
ReduceSum	aten::rsub		top.RequantInt
Relu			top.Reshape
Reshape			top.RetinaFaceDetection
Resize			top.Reverse
ReverseSequence			top.RoiAlign
RoiAlign			top.RoiExtractor
Round			top.Rope
			top.Round
			top.Rsqrt

表 17.17: S

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

表 17.18: T

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

表 17.19: U

Onnx	Pytorch	Caffe	TOP
Unsqueeze	aten::unbind	Upsample	top.UnTuple
Upsample	aten::_unsafe_view aten::unsqueeze aten::upsample_bilinear2d aten::upsample_linear1d aten::upsample_nearest1d aten::upsample_nearest2d aten::upsample_nearest3d		top.Unpack top.Unsqueeze top.Upsample

表 17.20: V

Onnx	Pytorch	Caffe	TOP
	aten::view		top.Variance top.View

表 17.21: W

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

表 17.22: X

Onnx	Pytorch	Caffe	TOP
Xor			

表 17.23: Y

Onnx	Pytorch	Caffe	TOP
		YoloDetection	top.Yield top.YoloDetection top.Yuv2rgbFormula

表 17.24: Z

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