
TPU-MLIR 快速入门指南

发行版本 1.6

SOPHGO

2024 年 02 月 24 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 tpu_mlir 安装	6
3 编译 ONNX 模型	7
3.1 安装 tpu-mlir	7
3.2 准备工作目录	7
3.3 ONNX 转 MLIR	8
3.4 MLIR 转 F16 模型	9
3.5 MLIR 转 INT8 模型	10
3.5.1 生成校准表	10
3.5.2 编译为 INT8 对称量化模型	11
3.6 效果对比	11
3.7 模型性能测试	12
3.7.1 安装 libsophon 环境	12
3.7.2 检查 BModel 的性能	12
4 编译 TORCH 模型	14
4.1 安装 tpu-mlir	14
4.2 准备工作目录	14
4.3 TORCH 转 MLIR	15
4.4 MLIR 转 F16 模型	15
4.5 MLIR 转 INT8 模型	15
4.5.1 生成校准表	15
4.5.2 编译为 INT8 对称量化模型	16
4.6 效果对比	16
5 编译 Caffe 模型	18
5.1 安装 tpu-mlir	18
5.2 准备工作目录	18
5.3 Caffe 转 MLIR	19
5.4 MLIR 转 F32 模型	19
5.5 MLIR 转 INT8 模型	19
5.5.1 生成校准表	19
5.5.2 编译为 INT8 对称量化模型	20

6 编译 TFLite 模型	21
6.1 安装 tpu-mlir	21
6.2 准备工作目录	21
6.3 TFLite 转 MLIR	22
6.4 MLIR 转 INT8 模型	22
7 量化与量化调优	23
7.1 混精度使用方法	23
7.1.1 安装 tpu-mlir	23
7.1.2 准备工作目录	24
7.1.3 验证原始模型	24
7.1.4 转成 INT8 对称量化模型	24
7.1.5 转成混精度量化模型	28
7.2 敏感层搜索使用方法	30
7.2.1 安装 tpu-mlir	30
7.2.2 准备工作目录	32
7.2.3 测试 Float 和 INT8 对称量化模型分类效果	32
7.2.4 转成混精度量化模型	34
7.3 局部不量化	37
7.3.1 使用方法	37
7.3.2 参数说明	38
8 使用智能深度学习处理器做前处理	39
8.1 模型部署样例	40
8.1.1 BM1684X 部署	40
8.1.2 CV18xx 部署	41
9 使用智能深度学习处理器做后处理	42
9.1 安装 tpu-mlir	42
9.2 准备工作目录	42
9.3 ONNX 转 MLIR	43
9.4 MLIR 转换成 BModel	44
9.5 模型验证	44
10 附录 01：各框架模型转 ONNX 参考	46
10.1 PyTorch 模型转 ONNX	46
10.1.1 步骤 0：创建工作目录	46
10.1.2 步骤 1：搭建并保存模型	46
10.1.3 步骤 2：导出 ONNX 模型	47
10.2 TensorFlow 模型转 ONNX	48
10.2.1 步骤 0：创建工作目录	48
10.2.2 步骤 1：准备并转换模型	48
10.3 PaddlePaddle 模型转 ONNX	48
10.3.1 步骤 0：安装 openssl-1.1.1o	48
10.3.2 步骤 1：创建工作目录	49
10.3.3 步骤 2：准备模型	49
10.3.4 步骤 3：转换模型	49

11 附录 02: CV18xx 使用指南	50
11.1 编译 yolov5 模型	50
11.1.1 安装 tpu-mlir	50
11.1.2 准备工作目录	50
11.1.3 ONNX 转 MLIR	51
11.1.4 MLIR 转 BF16 模型	51
11.1.5 MLIR 转 INT8 模型	51
11.1.6 效果对比	52
11.2 合并 cvimodel 模型文件	53
11.2.1 步骤 0: 生成 batch 1 的 cvimodel	54
11.2.2 步骤 1: 生成 batch 2 的 cvimodel	54
11.2.3 步骤 2: 合并 batch 1 和 batch 2 的 cvimodel	55
11.2.4 步骤 3: runtime 接口调用 cvimodel	55
11.2.5 综述: 合并过程	56
11.3 编译和运行 runtime sample	56
11.3.1 在 EVB 运行 release 提供的 sample 预编译程序	57
11.3.2 交叉编译 samples 程序	59
11.3.3 docker 环境仿真运行的 samples 程序	62
11.4 FAQ	63
11.4.1 模型转换常见问题	63
11.4.2 模型评估常见问题	65
11.4.3 模型部署常见问题	66
11.4.4 其他常见问题	68
12 附录 03: BM168x 使用指南	69
12.1 合并 bmodel 模型文件	69
12.1.1 步骤 0: 生成 batch 1 的 bmodel	69
12.1.2 步骤 1: 生成 batch 2 的 bmodel	70
12.1.3 步骤 2: 合并 batch 1 和 batch 2 的 bmodel	71
12.1.4 综述: 合并过程	71
13 附录 04: BM168x 测试指南	72
13.1 配置系统环境	72
13.2 获取 model-zoo 模型	72
13.3 准备运行环境	73
13.4 准备数据集	74
13.4.1 ImageNet	74
13.4.2 COCO (可选)	74
13.5 在非 x86 环境运行性能与精度测试	74
13.6 获取 tpu-perf 工具	75
13.7 准备工具链编译环境	75
13.8 模型性能和精度测试流程	76
13.8.1 模型编译	76
13.8.2 性能测试	76
13.8.3 精度测试	77
13.9 FAQ	78
13.9.1 invalid command ‘bdist_wheel’	78

13.9.2	not a supported wheel	78
13.9.3	no module named ‘xxx’	79
13.9.4	精度测试因为内存不足被 kill	79
14	附录 05: TPU Profile 工具使用	80
14.1	编译 bmodel	80
14.2	生成 Profile 原始数据	81
14.3	可视化 Profile 数据	82
15	附录 06: 已支持的算子	83
15.1	本章节主要提供目前 TPU-MLIR 支持的算子列表	83



法律声明

版权所有 © 算能 2022. 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

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

发布记录

目录

版本	发布日期	说明
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	初版发布, 支持 resnet/mobilenet/vgg/ssd/yolov5s , 并用 yolov5s 作为用例。

CHAPTER 1

TPU-MLIR 简介

TPU-MLIR 是算能深度学习处理器的编译器工程。该工程提供了一套完整的工具链, 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 `run_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。

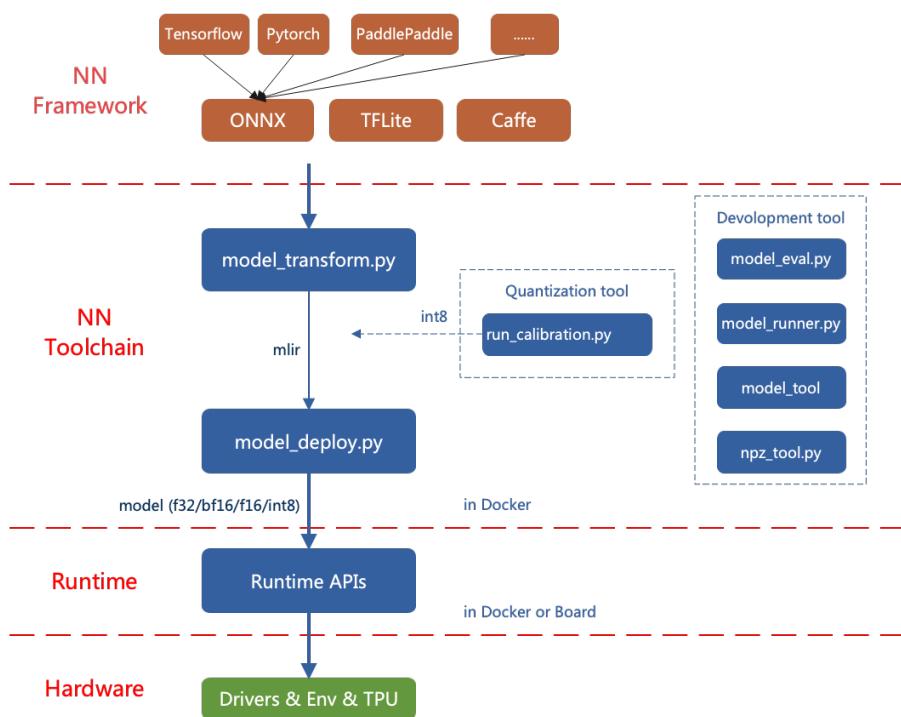


图 1.1: TPU-MLIR 的整体架构

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

```
$ docker pull sophgo/tpuc_dev:v3.2
```

如果是首次使用 Docker，可执行下述命令进行安装和配置（仅首次执行）：

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

确保安装包在当前目录，然后在当前目录创建容器如下：

```
$ docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.2
# myname只是举个名字的例子，请指定成自己想要的容器的名字
```

后文假定用户已经处于 docker 里面的/workspace 目录。

2.2 tpu_mlir 安装

目前支持 2 种安装方法：

- (1) 直接从 pypi 下载并安装：

```
$ pip install tpu_mlir
```

- (2) 从 Github 的 [Assets](#) 处下载最新 tpu_mlir-*[-py3-none-any.whl](#), 然后使用 pip 安装：

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

tpu_mlir 在对不同框架模型处理时所需的依赖不同，对于 onnx 或 torch 生成的模型文件，使用下面命令安装额外的依赖环境：

```
$ pip install tpu_mlir[onnx]
```

```
$ 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：

```
$ pip install tpu_mlir[onnx,torch,caffe]
```

```
$ pip install tpu_mlir[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/cv180t 其它	xxx.cvimodel xxx.bmodel	请参考: CV18xx 使用指南 继续本章节

3.1 安装 tpu-mlir

```
$ pip install tpu_mlir[onnx]
```

3.2 准备工作目录

建立 model_yolov5s 目录, 注意是与 tpu-mlir 同级目录; 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

```

请从 Github 的 [Assets](#) 处下载 tpu-mlir-resource.tar 并解压，解压后的文件夹的路径即是 tpu_mlir_resource。

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	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, 可以是图片或 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
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

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
processor	是	指定模型将要用到的平台, 支持 bm1688, bm1684x, bm1684, cv186x, cv183x, cv182x, cv181x, cv180x
calibration_table	否	指定校准表路径, 当存在 INT8 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, 可以是图片或 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	cv186x/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 或 cv186x 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。

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:

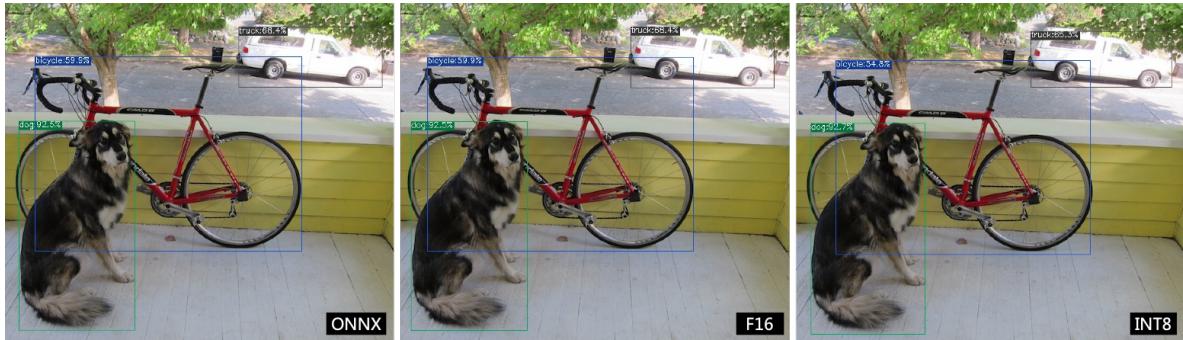


图 3.1: TPU-MLIR 对 YOLOv5s 编译效果对比

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: #####
```

(续下页)

(接上页)

```

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 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 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 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
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

```
$ pip install tpu_mlir[torch]
```

4.2 准备工作目录

建立 model_yolov5s_pt 目录, 注意是与 tpu-mlir 同级目录; 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
```

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压, 解压后的文件夹的路径即是 tpu_mlir_resource。

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, 目前我们仅支持静态模型, 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 \
```

(续下页)

(接上页)

```
--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：

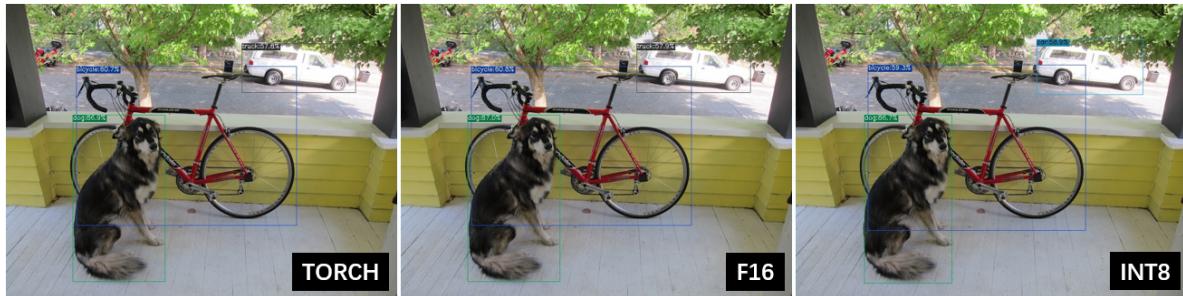


图 4.1: TPU-MLIR 对 YOLOv5s 编译效果对比

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

```
$ pip install tpu_mlir[caffe]
```

5.2 准备工作目录

建立 mobilenet_v2 目录, 注意是与 tpu-mlir 同级目录; 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
```

请从 Github 的 [Assets](#) 处下载 tpu-mlir-resource.tar 并解压, 解压后的文件夹的路径即是 tpu_mlir_resource。

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

```
$ pip install tpu_mlir[tensorflow]
```

6.2 准备工作目录

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
```

请从 Github 的 [Assets](#) 处下载 tpu-mlir-resource.tar 并解压, 解压后的文件夹的路径即是 tpu_mlir_resource。

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。本章节主要以 yolo 系列的两个模型为例，说明了混合精度网络的工作原理和设置方法，并介绍了敏感层搜索和局部不量化两个工具的使用方法。

7.1 混精度使用方法

本章以检测网络 yolov3 tiny 网络模型为例，介绍如何使用混精度。该模型来自 https://github.com/ONNX/ONNX-Model-Zoo/tree/main/vision/object_detection_segmentation/tiny-yolov3。

本章需要安装 tpu_mlir。

7.1.1 安装 tpu-mlir

```
$ pip install tpu_mlir[all]
```

7.1.2 准备工作目录

建立 yolov3_tiny 目录, 注意是与 tpu-mlir 同级目录; 并把模型文件和图片文件都放入 yolov3_tiny 目录中。

操作如下:

```

1 $ mkdir yolov3_tiny && cd yolov3_tiny
2 $ wget https://github.com/onnx/models/raw/main/vision/object_detection_segmentation/tiny-
   →yolov3/model/tiny-yolov3-11.onnx
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .
4 $ mkdir workspace && cd workspace

```

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压, 解压后的文件夹的路径即是 tpu_mlir_resource。

注意如果 tiny-yolov3-11.onnx 用 wget 下载失败, 请用其他方式下载后放到 yolov3_tiny 目录。

7.1.3 验证原始模型

detect_yolov3 是已经写好的验证命令, 可以用来对 yolov3_tiny 网络进行验证。执行过程如下:

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

执行完后打印检测到的结果如下:

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

并得到图片 yolov3_onnx.jpg, 如下 (yolov3_tiny ONNX 执行效果):

7.1.4 转成 INT8 对称量化模型

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。



图 7.1: yolov3_tiny ONNX 执行效果

第一步: 转成 F32 mlir

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

第二步: 生成 calibartion table

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

第三步: 转对称量化模型

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

第四步: 验证模型

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

执行完后有如下打印信息，表示检测到一个目标：

```
orange:72.9%
```

得到图片 `yolov3_int8.jpg`, 如下 (`yolov3_tiny int8 对称量化执行效果`)：

可以看出 int8 对称量化模型相对原始模型, 在这张图上效果不佳, 只检测到一个目标。



图 7.2: yolov3_tiny int8 对称量化执行效果

7.1.5 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。

第一步: 生成混精度量化表

使用 run_qtable 生成混精度量化表, 相关参数说明如下:

表 7.1: run_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
calibration_table	是	输入校准表
processor	是	指定模型将要用到的平台, 支持 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	指定输入样本数量, 默认用 10 个
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
min_layer_cos	否	指定期望每层输出 cos 的最小值, 低于该值会尝试设置浮点计算, 一般默认为 0.99 即可
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' or 'layer1:0.3,layer2:0.7'
fp_type	否	指定混合精度的浮点类型
loss_table	否	指定保存所有被量化成浮点类型的层的损失值的文件名, 默认为 full_loss_table.txt

本例中采用默认 10 张图片校准, 需要首先安装 Graphviz 工具:

```
$ sudo apt-get install graphviz
```

然后执行如下命令 (对于 CV18xx 系列的处理器, 将 processor 设置为对应的名称即可) :

```
$ run_qtable yolov3_tiny.mlir \
--dataset ./COCO2017 \
--calibration_table yolov3_cali_table \
--processor bm1684x \
--min_layer_cos 0.999 \ #若这里使用默认的0.99时, 程序会检测到原始int8模型已满足0.
→99的cos, 从而直接不再搜索
--expected_cos 0.9999 \
-o yolov3_qtable
```

执行完后最后输出如下打印:

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

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 表示部分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 44 秒, 另外, 生成的混精度量化表 yolov3_qtable, 内容如下:

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

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。另外同时也会生成一个 loss 表文件 full_loss_table.txt, 内容如下:

```
1 # platform: bm1684x mix_mode: F16
2 ####
3 No.0 : Layer: model_1/leaky_re_lu_3/LeakyRelu:0_LeakyRelu Cos: 0.994022
4 No.1 : Layer: model_1/leaky_re_lu_5/LeakyRelu:0_LeakyRelu Cos: 0.997445
5 No.2 : Layer: model_1/leaky_re_lu_2/LeakyRelu:0_LeakyRelu Cos: 0.997487
6 No.3 : Layer: model_1/leaky_re_lu_4/LeakyRelu:0_LeakyRelu Cos: 0.997978
7 No.4 : Layer: model_1/leaky_re_lu_2/LeakyRelu:0_MaxPool Cos: 0.998159
8 No.5 : Layer: convolution_output11_Conv Cos: 0.998307
9 No.6 : Layer: model_1/leaky_re_lu_1/LeakyRelu:0_LeakyRelu Cos: 0.999249
10 No.7 : Layer: convolution_output9_Conv Cos: 0.999292
11 No.8 : Layer: convolution_output8_Conv Cos: 0.999427
12 No.9 : Layer: model_1/leaky_re_lu_1/LeakyRelu:0_MaxPool Cos: 0.999580
13 No.10 : Layer: convolution_output12_Conv Cos: 1.000004
```

该表按 cos 从小到大顺利排列, 表示该层的前驱 Layer 根据各自的 cos 已换成相应的浮点模式后, 该层计算得到的 cos, 若该 cos 仍小于前面 min_layer_cos 参数, 则会将该层及直接后继层设置为浮点计算。run_qtable 会在每次设置某相邻 2 层为浮点计算后, 接续计算整个网络的输出 cos, 若该 cos 大于指定的 expected_cos, 则退出搜索。因此, 若设置更大的 expected_cos, 会尝试将更多层设为浮点计算

第二步：生成混精度量化模型

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

第三步：验证混精度模型

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

执行完后打印结果为：

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

得到图片 `yolov3_mix.jpg`, 如下 (`yolov3_tiny` 混精度对称量化执行效果)：

可以看出混精度后, 检测结果更接近原始模型的结果。

需要说明的是, 除了使用 `run_qtable` 生成量化表外, 也可根据模型中每一层的相似度对比结果, 自行设置量化表中需要做混精度量化的 OP 的名称和量化类型。

7.2 敏感层搜索使用方法

本章以检测网络 `mobilenet-v2` 网络模型为例, 介绍如何使用敏感层搜索。该模型来自 `nnmodels/pytorch_models/accuracy_test/classification/mobilenet_v2.pt`。

本章需要安装 `tpu_mlir`。

7.2.1 安装 tpu-mlir

```
$ pip install tpu_mlir[all]
```



图 7.3: yolov3_tiny 混精度对称量化执行效果

7.2.2 准备工作目录

建立 mobilenet-v2 目录，注意是与 tpu-mlir 同级目录；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 $ cp -rf tpu_mlir_resource/dataset/ILSVRC2012 .
3 $ wget https://github.com/sophgo/tpu-mlir/releases/download/v1.4-beta.0/mobilenet_v2.pt
4 $ mkdir workspace && cd workspace

```

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后的文件夹的路径即是 tpu_mlir_resource。

7.2.3 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。

第一步：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

```

第二步：生成 calibartion table

```

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

```

第三步：转 FP32 bmodel

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

第四步：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
```

第五步：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_f32.bmodel \
--input .../ILSVRC2012/n01440764_9572.JPEG \
--output mobilenet_v2_fp32_bmodel.jpeg \
--category_file .../ILSVRC2012/synset_words.txt
```

在输出结果图片上可以看到如下分类信息，正确结果 tench 排在第一名：

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

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
```

在输出结果图片上可以看到如下分类信息，正确结果 tench 排在第一名：

```
Top-5
n01440764 tench, Tinca tinca
n02749479 assault 日 file, assau
n02536864 coho, cohoe, coho
n02916936 bulletproof vest
n04336792 stretcher
```

7.2.4 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。

第一步：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 7.2: `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 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' or 'layer1:0.3,layer2:0.7'
fp_type	否	指定混合精度的浮点类型

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
```

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：

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

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
6 INFO:root:run int8 mode: mobilenet_v2.mlir
7 INFO:root:outputs_cos_loss = 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_loss = 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。

第二步：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

```

第三步：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

```

(续下页)

(接上页)

```
n02536864 coho, cohoe, coho salmon, blue jack, silver salmon, Oncorhynchus kisutch
n04090263 rifle
```

7.3 局部不量化

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。

7.3.1 使用方法

本章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 FP32 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
  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。

7.3.2 参数说明

表 7.3: fp_forward 参数功能

参数名	必选？	说明
无	是	指定 mlir 文件
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
processor	是	指定模型将要用到的平台，支持 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，默認為 auto，表示由程序内部自动选择
o	是	输出混精度量化表

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 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 即可以直接受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：

表 8.2: customization_format 格式和说明

customization_format	说明	BM1684X	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	False	True
YUV_NV21	yuv420 的 NV21 格式, 来自 vpss 的输入	False	True
YUV_NV12	yuv420 的 NV12 格式, 来自 vpss 的输入	False	True
RGBA_PLANAR	rgba 格式, 按照 nchw 摆放	False	True

其中“YUV*”类格式为 CV18xx 系列特有的输入格式。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 BM1684X 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。

CHAPTER 9

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。

本章将 yolov5s 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。

9.1 安装 tpu-mlir

```
$ pip install tpu_mlir[onnx]
```

9.2 准备工作目录

建立 model_yolov5s 目录，注意是与 tpu-mlir 同级目录；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  
3 $ cp -rf tpu_mlir_resource/dataset/COCO2017 .  
4 $ cp -rf tpu_mlir_resource/image .  
5 $ mkdir workspace && cd workspace
```

请从 Github 的 [Assets](#) 处下载 tpu-mlir-resource.tar 并解压，解压后的文件夹的路径即是 tpu_mlir_resource。

9.3 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：

```
1 [0., 16., 0.924488, 184.21094, 401.21973, 149.66412, 268.50336 ]
```

9.4 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 chip: 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。

9.5 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 \
```

(续下页)

(接上页)

```
--net_input_dims 640,640 \
--fuse_preprocess \
--fuse_postprocess \
--output dog_out.jpg
```

CHAPTER 10

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。

10.1 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

10.1.1 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
```

10.1.2 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")

```

运行完后我们会在当前目录下获得一个 weight.pth 的权重文件。

10.1.3 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。

10.2 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。

10.2.1 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
```

10.2.2 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。

10.3 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)。

10.3.1 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。

10.3.2 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
```

10.3.3 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/PaddlePaddle/Paddle2ONNX/develop/tools/paddle/
2   ↪paddle_infer_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，并在该目录下有一个 new_model 的目录。

10.3.4 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 11

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。

11.1 编译 yolov5 模型

11.1.1 安装 tpu-mlir

```
$ pip install tpu_mlir[all]
```

11.1.2 准备工作目录

建立 model_yolov5s 目录，注意是与 tpu-mlir 同级目录；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
```

请从 Github 的 Assets 处下载 tpu-mlir-resource.tar 并解压，解压后的文件夹的路径即是 tpu_mlir_resource。

11.1.3 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。

11.1.4 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。

11.1.5 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。

11.1.6 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 \
```

(续下页)

(接上页)

```
--model yolov5s_cv183x_int8_sym.cvimodel \
--output dog_int8.jpg
```

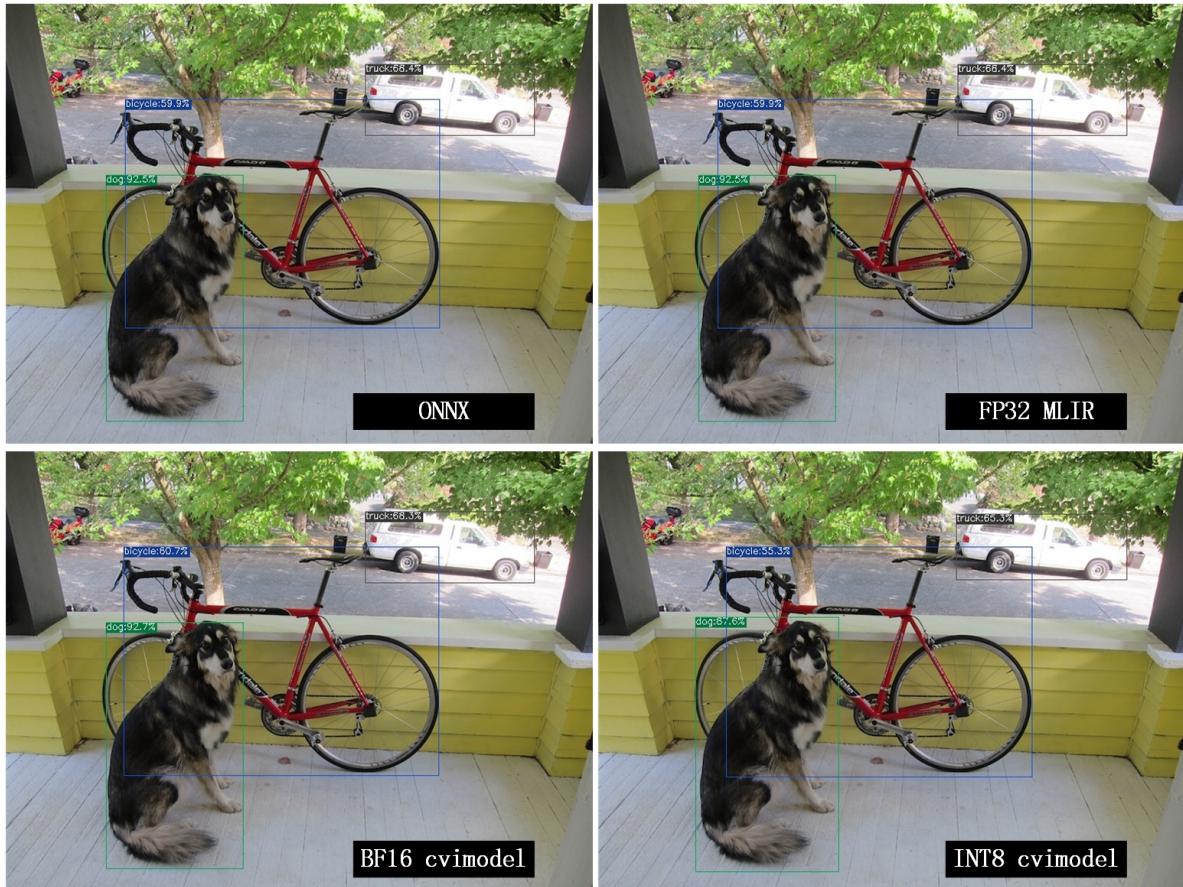


图 11.1: 不同模型效果对比

四张图片对比如 图 11.1，由于运行环境不同，最终的效果和精度与 图 11.1 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。

11.2 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：

11.2.1 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
```

11.2.2 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]] \
```

(续下页)

(接上页)

```
--mean 0.0,0.0,0.0 \
--scale 0.0039216,0.0039216,0.0039216 \
--keep_aspect_ratio \
--pixel_format rgb \
--output_names 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
```

11.2.3 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
```

11.2.4 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, ...);
....
```

(续下页)

(接上页)

```

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);

```

11.2.5 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

11.3 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)

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

需要如下文件:

- cvitek_tpu_sdk_[cv186x | cv183x | cv182x | cv182x_uclibc | cv181x_glibc32 | cv181x_musl_riscv64_rvv | cv180x_musl_riscv64_rvv | cv181x_glibc_riscv64].tar.gz
- cvimodel_samples_[cv186x | 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
```

11.3.2 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_[cv186x | 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 \
-DOENCV_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

```

11.3.3 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_[cv186x|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

11.4 FAQ

11.4.1 模型转换常见问题

1 模型转换问题

1.1 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。

1.2 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。

1.3 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
sqnr_similarity = 21.209481
154 compared
153 passed
1 equal, 0 close, 152 similar
1 failed
0 not equal, 1 not similar
min_similariy = (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 --ATLESD
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("!!Error: {}".format(cmd_str))
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)。

1.4 `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。

1.5 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。

2 量化问题

2.1 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。

2.2 跑量化如何处理多输入问题?

多输入模型跑 `run_calibration` 时, 需要多输入模型跑 `run_calibration` 时, 可使用 `.npz` 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。

2.3 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。

2.4 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
```

2.5 是否支持手动修改 calibration table?

支持, 但是不建议修改。

3 其它常见问题

3.1 转换后的模型是否支持加密?

暂时不支持。

3.2 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。

3.3 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](#)

11.4.2 模型评估常见问题

1 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

2 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

3 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。

11.4.3 模型部署常见问题

1 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。

2 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。

3 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。

4 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。

5 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。

6 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。

7 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

11.4.4 其他常见问题

1 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

2 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 12

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。

12.1 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:

12.1.1 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
```

12.1.2 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
```

12.1.3 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
```

12.1.4 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 13

附录 04：BM168x 测试指南

13.1 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 可执行下述命令进行安装和配置（仅首次执行，同时该配置是在用户自己系统中，并非 Docker container 中）：

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

13.2 获取 model-zoo 模型

在欲使用的工作目录下，使用以下命令克隆 model-zoo 工程：

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

model-zoo 的目录结构如下：

```
|── config.yaml  
|── requirements.txt  
|── data  
|── dataset  
|── harness
```

(续下页)

(接上页)

```

└── output
└── ...

```

- config.yaml 中包含通用的配置：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 为 model-zoo 的 python 依赖
- dataset 目录中包含模型的 imagenet 数据集预处理，将作为 plugin 被 tpu_perf 调用
- data 目录将用于存放 lmdb 数据集
- 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。

如果已经克隆过 model-zoo 可以执行以下命令同步模型到最新状态：

```

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

```

此过程会从 GitHub 上下载大量数据。由于具体网络环境的差异，此过程可能耗时较长。

注意：如果您获得了 SOPHGO 提供的 model-zoo 测试包，可以执行以下操作创建并设置好 model-zoo，完成此步骤后直接进入下一节。

```

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

```

13.3 准备运行环境

安装运行 model-zoo 所需的依赖：

```

# for ubuntu 操作系统
sudo apt-get install build-essential
sudo apt install python3-dev
sudo apt-get install -y libgl1 # For OpenCV
# 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，运行环境中调用 tpu 硬件进行性能和精度测试，请根据 libsophon 使用手册安装 libsophon。

13.4 准备数据集

13.4.1 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  
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 将目录映射为 dataset/ILSVRC2012/ILSVRC2012_img_val
```

13.4.2 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
```

13.5 在非 x86 环境运行性能与精度测试

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 libsophon 运行环境，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 进行性能与精度测试。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。

13.6 获取 tpu-perf 工具

从 <https://github.com/sophgo/tpu-perf/releases> 地址下载最新的 tpu-perf wheel 安装包。例如: tpu_perf-x.x.x-py3-none-manylinux2014_x86_64.whl。并将 tpu-perf 包放置到与 model-zoo 同一级目录下。此时的目录结构应该为如下形式:

```
├── tpu_perf-x.x.x-py3-none-manylinux2014_x86_64.whl
└── model-zoo
```

13.7 准备工具链编译环境

建议在 docker 环境使用工具链软件, 最新版本的 docker 可以参考 [官方教程](#) 进行安装。安装完成后, 执行下面的脚本将当前用户加入 docker 组, 获得 docker 执行权限。

```
$ sudo usermod -aG docker $USER
$ newgrp docker
```

然后, 在欲使用的工作目录 (即 model-zoo 所在目录) 下执行以下命令:

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

运行命令后会处于 Docker 的容器中。在 Docker 容器中安装 tpu_mlir:

```
$ pip install tpu_mlir[all]
```

13.8 模型性能和精度测试流程

13.8.1 模型编译

使用以下命令完成设置运行测试所需的环境变量:

```
$ source $(which envsetup.sh)
```

该过程结束后不会有任何提示。之后使用以下命令安装 tpu-perf:

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

model-zoo 的相关 config.yaml 配置了 SDK 的测试内容。例如: resnet18 的配置文件为 model-zoo/vision/classification/resnet18-v2/config.yaml。

执行以下命令, 运行全部测试样例:

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

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

此时会编译以下模型 (由于 model-zoo 的模型在持续添加中, 这里只列出部分模型; 同时该过程也编译了用于测试精度的模型, 后续精度测试部分无需再编译模型。) :

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

命令正常结束后, 会看到新生成的 output 文件夹 (测试输出内容都在该文件夹中)。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
```

13.8.2 性能测试

运行测试需要在 Docker 外面的环境 (此处假设您已经安装并配置好了 1684X 设备和驱动) 中进行, 可以退出 Docker 环境:

```
$ exit
```

1. PCIE 板卡下运行以下命令, 测试生成的 bmodel 性能。

```
$ pip3 install ./tpu_perf-*py3-none-manylinux2014_x86_64.whl
$ cd model-zoo
$ python3 -m tpu_perf.run --target BM1684X --mlir -l full_cases.txt
```

--target 用于指定处理器型号，目前支持 BM1684、BM1684X、BM1688``和``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.run --target BM1684X --devices 2 --mlir -l full_cases.txt
```

2. SOC 设备使用以下步骤，测试生成的 bmodel 性能。

从 <https://github.com/sophgo/tpu-perf/releases> 地址下载最新的 tpu-perf tpu_perf-x.x.x-py3-none-manylinux2014_aarch64.whl 文件到 SOC 设备上并执行以下操作：

```
$ pip3 install ./tpu_perf-x.x.x-py3-none-manylinux2014_aarch64.whl
$ cd model-zoo
$ python3 -m tpu_perf.run --target BM1684X --mlir -l full_cases.txt
```

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。

13.8.3 精度测试

运行测试需要在 Docker 外面的环境（此处假设您已经安装并配置好了 1684X 设备和驱动）中进行，可以退出 Docker 环境：

```
$ exit
```

PCIE 板卡下运行以下命令，测试生成的 bmodel 精度。

```
$ pip3 install ./tpu_perf-*py3-none-manylinux2014_x86_64.whl
$ cd model-zoo
$ python3 -m tpu_perf.precision_benchmark --target BM1684X --mlir -l full_cases.txt
```

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

各类精度数据在 output 目录中的各个 csv 文件可以获得。

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
```

具体参数说明可以通过以下命令获得：

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

13.9 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。

13.9.1 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
[100%] Installing the project stripped...
-- Install configuration: ""
-- Installing: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.libpipeline.so
-- Set runtime path of "/1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.libpipeline.so" to ""
-- Up-to-date: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.blob_pb2.py
-- Up-to-date: /1684x/nntc/1001rc1/RC1/tpu-perf_v1.0.11/tpu-perf-1.0.11/python/tpu_perf/.blob_pb.cc
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
      or: setup.py --help [cmd1 cmd2 ...]
      or: setup.py --help-commands
      or: setup.py cmd --help

1 error: invalid command 'bdist_wheel'
CMake 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

13.9.2 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

13.9.3 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:~/dig/hy_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/40/93/655af887bafece2a655998f53b9bd21ad94b627d81d44acf35c79f40de6/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

13.9.4 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 14

附录 05：TPU Profile 工具使用

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。

14.1 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.py 加上--debug 选项。如果在编译时未开启该选项，运行时开启 Profile 得到的数据在可视化时，会有部分数据缺失。

```
# 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 # 记录profile数据
```

通过以上命令，将 yolov5s.onnx 编译成了 yolov5s_bm1684x_f16.bmodel。

14.2 生成 Profile 原始数据

将生成的 yolov5s_bm1684x_f16.bmodel 拷贝到运行环境。同编译过程，运行时的 Profile 功能默认是关闭的，防止在做 profile 保存与传输时产生额外时间消耗。需要开启 profile 功能时，在运行编译好的应用前设置环境变量 BMRUNTIME_ENABLE_PROFILE=1 即可。然后用 libsophon 中提供的模型测试工具 bmrt_test 来作为应用，生成 profile 数据。

```
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) 'lImages' 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:888] INFO:reading input #0, bytesize=4915209
[BMRT][print_array:706] INFO: -> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1228000
[BMRT][write_block:295] INFO:write_block: type=1, len=36
[BMRT][write_block:295] INFO:write_block: type=8, len=44
[BMRT][end:76] INFO:bdc record_num=1838, max_record_num=1048576
[BMRT][write_block:295] INFO:write_block: type=3, len=58816
[BMRT][end:89] INFO:gdma record_num=196, max_record_num=1048576
[BMRT][write_block:295] INFO:write_block: type=4, len=37632
[BMRT][write_block:295] INFO:write_block: type=5, len=256
[BMRT][write_block:295] INFO:write_block: type=6, len=1072
[BMRT][print_note:94] INFO:*****
[BMRT][print_note:95] INFO:/* PROFILE MODE due to BMRUNTIME_ENABLE_PROFILE=1 */
[BMRT][print_note:96] INFO:/* Note: BMRunTime will collect time data during running           */
[BMRT][print_note:97] INFO:/* that will cost extra time.                                     * */
[BMRT][print_note:98] INFO:/* Close PROFILE Mode by "unset BMRUNTIME_ENABLE_PROFILE" */
[BMRT][print_note:99] INFO:*****
[BMRT][bmrt_test:1095] INFO:reading output #0, bytesize=6528000
[BMRT][print_array:706] INFO: -> output.ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1632000
[BMRT][bmrt_test:1095] INFO:reading output #1, bytesize=1632000
[BMRT][print_array:706] INFO: -> output.ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=408000
[BMRT][bmrt_test:1095] INFO:reading output #2, bytesize=408000
[BMRT][print_array:706] INFO: -> output.ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > 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
95 -6.26172 ... > len=1632000
[BMRT][print_array:706] INFO:output data #1 shape: [1 3 40 40 85 ] < -0.0398254 0.253174 -0.383057 -0.520996 -11.0391 -1.3125 -5.11328 -3.32031 -5.46484 -5.97656
812 -6.29684 ... > len=408000
[BMRT][print_array:706] INFO:output data #2 shape: [1 3 20 20 85 ] < 0.713379 0.654297 -0.534668 -0.241577 -9.82031 -1.23047 -5.77344 -3.35547 -5.92578 -5.12109
44 -6.63672 ... > len=102000
[BMRT][bmrt_test:1083] INFO:load input time(s): 0.003650
[BMRT][bmrt_test:1084] INFO:calculate time(s): 0.375613
[BMRT][bmrt_test:1085] INFO:get output time(s): 0.003838
[BMRT][bmrt_test:1086] INFO:compare time(s): 0.000827
```

图 14.1: 开启 Profile 后运行输出的日志

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

14.3 可视化 Profile 数据

将 bmprofile_data-1 目录拷贝回 tpu-mlir 工程环境。tpu-mlir 提供了 tpu_profile.py 脚本, 来把生成的二进制 profile 数据转换成网页文件, 来进行可视化。命令如下:

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

用浏览器打开 bmprofile_out/result.html 可以看到 profile 的图表。此外, 该工具还有其他用法, 可以通过 tpu_profile.py -help 来查看。更多的 Profile 工具使用分析说明请参考 <https://tpumlir.org/zh-cn/2023/09/18/analyse-tpu-performance-with-tpu-profile.html>

CHAPTER 15

附录 06：已支持的算子

15.1 本章节主要提供目前 TPU-MLIR 支持的算子列表

表 15.1: A

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

表 15.2: B

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

表 15.3: C

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

表 15.4: 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.DequantizeLinear
			top.DetectionOutput
			top.Div

表 15.5: E

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

表 15.6: F

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

表 15.7: G

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

表 15.8: H

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

表 15.9: I

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

表 15.10: 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::linear		top.LayerNormBwd
Log	aten::log		top.LayerNormTrain
LogSoftmax	aten::log2		top.LeakyRelu
Loop	aten::log_sigmoid		top.List
LRN	aten::log_softmax		top.Log
LSTM	aten::lstm		top.LogB
	aten::lt		top.Loop

表 15.11: M

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

表 15.12: N

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

表 15.13: O

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

表 15.14: P

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

表 15.15: Q

Onnx	Pytorch	Caffe	TOP
QuantizeLinear			top.QuantizeLinear

表 15.16: R

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

表 15.17: S

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

表 15.18: 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::topk		top.Trilu
	aten::transpose		top.Tuple
aten::type_as			

表 15.19: U

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

表 15.20: V

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

表 15.21: W

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

表 15.22: Y

Onnx	Pytorch	Caffe	TOP
		YoloDetection	top.Yield
			top.YoloDetection

表 15.23: Z

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