

密级状态：绝密() 秘密() 内部() 公开(√)

RKNN SDK User Guide

(图形计算平台中心)

文件状态： [] 正在修改 [√] 正式发布	当前版本：	2.0.0-beta0
	作 者：	HPC
	完成日期：	2024-03-15
	审 核：	熊伟
	完成日期：	2024-03-15

瑞芯微电子股份有限公司

Rockchips Semiconductor Co., Ltd

(版本所有,翻版必究)

更新记录

版本	修改人	修改日期	修改说明	核定人
V1.6.0	HPC	2023-11-28	初始版本	熊伟
V2.0.0-beta0	HPC	2024-03-15	1. 增加 RK3576 内容； 2. 增加稀疏化推理； 3. Matmul API 接口改进，增加量化参数、动态 shape 输入、iommu_domain_id 功能	熊伟

目 录

1 RKNN 简介	1
1.1 RKNN 工具链介绍	1
1.1.1 RKNN 软件栈整体介绍	1
1.1.2 RKNN-Toolkit2 功能介绍	1
1.1.3 RKNN Runtime 功能介绍	2
1.2 RKNN 开发流程介绍	3
1.3 适用的硬件平台	5
1.4 关键字说明	5
2 开发环境准备	6
2.1 RKNN-Toolkit2 安装	6
2.1.1 通过 Docker 方式安装	6
2.1.2 通过 Pip 方式安装	8
2.2 设备 NPU 环境准备	10
2.2.1 NPU 驱动版本确认	10
2.2.2 NPU 连板环境确认	10
2.2.3 RKNN Server 安装和更新	11
2.2.4 查看 RKNN Server 详细日志	13
3 RKNN 使用说明	14
3.1 模型转换	14
3.1.1 RKNN 初始化及对象释放	15
3.1.2 模型 Config 配置	15
3.1.3 模型加载接口介绍	17
3.1.4 构建 RKNN 模型	18
3.1.5 导出 RKNN 模型	18

3.1.6 模型转换工具 RKNN_Convert	19
3.1.7 RKNN-Toolkit2 模型量化功能	22
3.2 模型评估	23
3.2.1 模型推理	23
3.2.2 模型精度分析	24
3.2.3 模型性能评估	26
3.2.4 模型内存评估	28
3.3 板端 C 推理	28
3.4 板端 Python 推理	31
3.4.1 系统依赖说明	31
3.4.2 工具安装	31
3.4.3 基本使用流程	32
3.4.4 运行参考示例	33
3.4.5 RKNN-Toolkit Lite2 API 详细说明	34
3.5 矩阵乘法接口	37
3.5.1 主要用途和特点	37
3.5.2 Matmul API 使用流程	39
3.5.3 矩阵乘法高级用法	40
3.5.4 高性能的数据排列方式	45
4 示例	52
4.1 MobileNet 模型部署示例	52
4.1.1 模型转换	52
4.1.2 模型连板运行	52
4.1.3 模型评估	53
4.1.4 板端部署	54
4.2 YOLOv5 模型部署示例	55

4.2.1 模型转换	55
4.2.2 模型连板运行	55
4.2.3 板端部署运行	56
5 RKNN 进阶使用说明	58
5.1 数据排列格式	58
5.2 RKNN Runtime 零拷贝调用	61
5.2.1 零拷贝介绍	61
5.2.2 C API 零拷贝整体流程	63
5.2.3 C API 零拷贝的用法	66
5.3 NPU 多核配置	70
5.3.1 多核运行配置方法	70
5.3.2 查看多核运行效果	73
5.3.3 多核性能提升技巧	74
5.4 动态 Shape	75
5.4.1 动态 Shape 功能介绍	75
5.4.2 RKNN SDK 版本和平台要求	76
5.4.3 生成动态 Shape 的 RKNN 模型	76
5.4.4 C API 部署	77
5.5 自定义算子	88
5.5.1 自定义算子介绍	88
5.5.2 整体流程介绍	89
5.5.3 Python 端处理	90
5.5.4 C API 部署	92
5.6 多 Batch 使用说明	105
5.6.1 多 Batch 原理	105
5.6.2 多 Batch 使用方式	106

5.6.3 多 Batch 输入输出设置	106
5.7 RK3588 NPU SRAM 使用说明	107
5.7.1 板端环境要求	107
5.7.2 使用方法	108
5.7.3 调试方法	109
5.8 模型剪枝	111
5.9 模型加密	112
5.10 Cacheable 内存一致性	112
5.10.1 Cacheable 内存同步的方向	113
5.10.2 同步 Cacheable 内存	113
5.11 模型稀疏化推理	114
5.11.1 稀疏化原理	114
5.11.2 训练稀疏化模型	115
5.11.3 RKNN 稀疏化推理使用方法	116
5.11.4 RKNN 稀疏化推理限制	117
5.12 生成部署 C 代码	118
5.13 ONNX 模型编辑	119
5.13.1 onnx_edit 接口说明	119
5.13.2 onnx_edit 变换公式说明	119
5.13.3 变换公式示例	120
6 量化说明	121
6.1 量化介绍	121
6.1.1 量化定义	121
6.1.2 量化计算原理	121
6.1.3 量化误差	121
6.1.4 线性对称量化和线性非对称量化	122

6.1.5 Per-Layer 量化和 Per-Channel 量化	122
6.1.6 量化算法	123
6.2 量化配置	123
6.2.1 量化数据类型	123
6.2.2 量化算法建议	123
6.2.3 量化校正集建议	124
6.2.4 量化配置方法	124
6.3 混合量化	125
6.3.1 混合量化用法	125
6.3.2 混合量化使用流程	125
6.4 量化感知训练	128
6.4.1 QAT 简介	128
6.4.2 QAT 原理	128
6.4.3 QAT 使用依据	128
6.4.4 QAT 实现简例及配置说明	129
6.4.5 QAT 支持的算子	130
6.4.6 QAT 模型中浮点算子的处理	131
6.4.7 QAT 经验总结	132
7 精度排查	133
7.1 模拟器精度排查	133
7.1.1 模拟器 FP16 精度	134
7.1.2 模拟器量化精度	135
7.2 Runtime 精度排查	138
7.2.1 连板精度	138
7.2.2 Runtime 精度	139
8 性能优化	140

8.1 模型性能优化前期分析流程	141
8.1.1 环境条件与配置检查	141
8.1.2 部署过程耗时分析	142
8.2 模型性能分析	142
8.2.1 获取 Profile 信息	142
8.2.2 分析逐层耗时	143
8.2.3 分析 CPU 算子影响	144
8.2.4 分析 NPU 算子性能瓶颈	145
8.3 量化加速	145
8.4 图级别优化	146
8.4.1 非 NPU OP 通过图变换实现 NPU 化	146
8.4.2 利用硬件 Fuse 特性设计网络或图优化	147
8.4.3 算法等效变换或者子图单 OP 化	148
8.4.4 算子等效进行“同类项合并”、“提取公因式”	149
8.5 算子级别优化	150
8.5.1 面向 DDR 性能优化的 OP 尺寸设计（非强制）	150
8.5.2 高利用率模型算子的设计	152
8.5.3 子图融合的匹配	153
9 内存使用优化	155
9.1 模型运行时内存组成及分析方法介绍	155
9.1.1 RKNN 模型运行时内存组成	155
9.1.2 模型内存分析方法	155
9.2 如何使用外部分配内存	155
9.2.1 输入输出内存外部分配	155
9.2.2 模型内存的外部分配	158
9.3 Internal 内存复用	159

9.4 多线程复用上下文	160
9.5 多种分辨率模型共享相同权重	161
10 常见问题	163
10.1 NPU 环境准备问题	163
10.2 工具安装问题	164
10.3 模型转换常用参数说明	165
10.4 模型加载问题	170
10.4.1 RKNN-Toolkit2 支持的深度学习框架和对应版本	170
10.4.2 各框架的 OP 支持列表	170
10.4.3 ONNX 模型转换常见问题	170
10.4.4 Pytorch 模型转换常见问题	171
10.4.5 TensorFlow 模型转换常见问题	172
10.5 模型量化问题	175
10.6 模型转换问题	175
10.7 模拟器推理及连板推理的说明	180
10.8 模型评估常见问题	182
10.9 C API 使用常见问题	187
11 相关资源	189

1 RKNN 简介

1.1 RKNN 工具链介绍

1.1.1 RKNN 软件栈整体介绍

RKNN 软件栈可以帮助用户快速的将 AI 模型部署到 Rockchip 芯片。整体的框架如下：

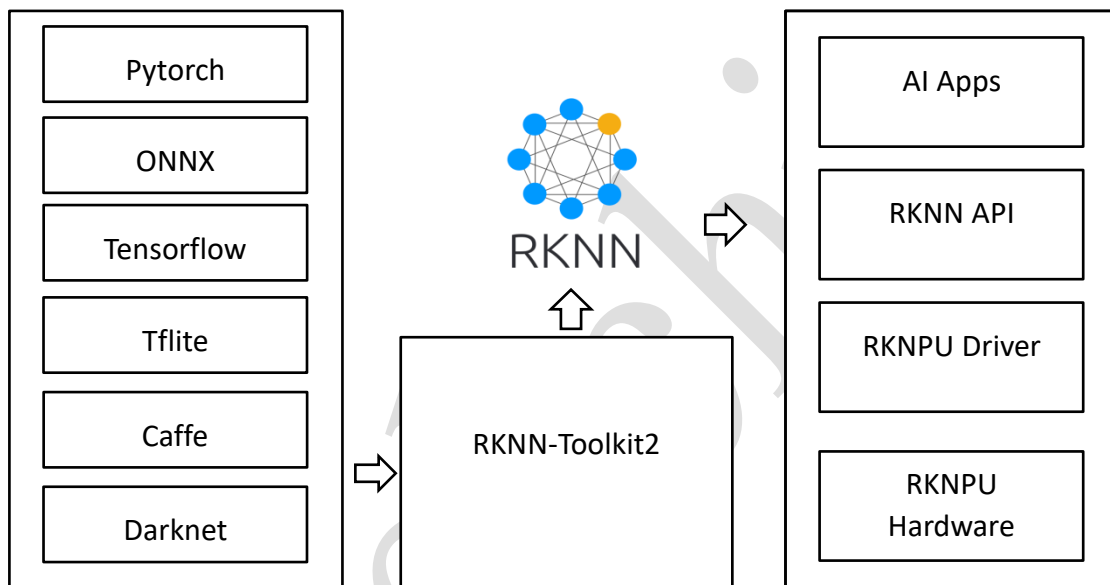


图 1-1 RKNN 软件栈

为了使用 RKNPU，用户需要首先在计算机上运行 RKNN-Toolkit2 工具，将训练好的模型转换为 RKNN 格式模型，之后使用 RKNN C API 或 Python API 在开发板上进行部署。

1.1.2 RKNN-Toolkit2 功能介绍

RKNN-Toolkit2 是为用户提供在计算机上进行模型转换、推理和性能评估的开发套件，RKNN-Toolkit2 的主要框图如下：

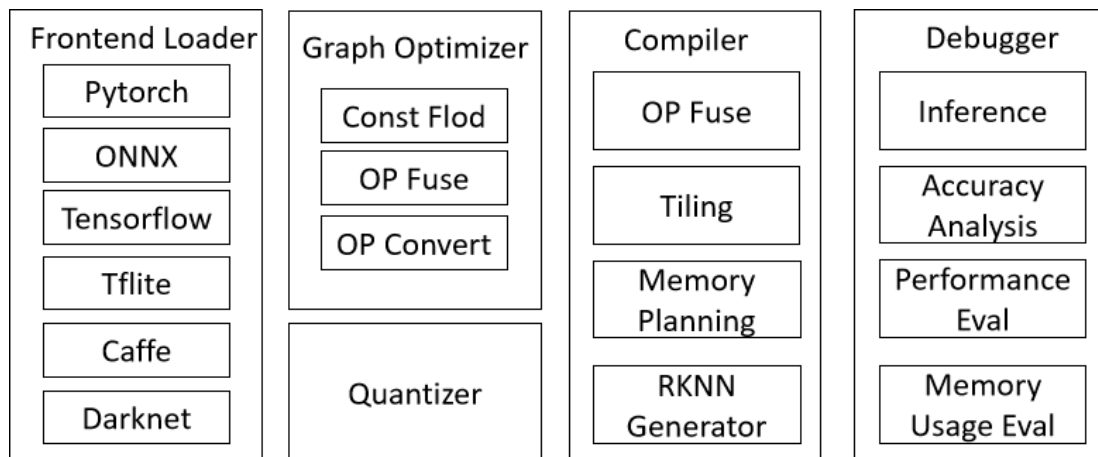


图 1-2 RKNN-Toolkit2 软件框图

通过该工具提供的 Python 接口可以便捷地完成以下功能：

1. 模型转换：支持 PyTorch、ONNX、TensorFlow、TensorFlow Lite、Caffe、DarkNet 等模型转为 RKNN 模型。
2. 量化功能：支持将浮点模型量化为定点模型，并支持混合量化。
3. 模型推理：将 RKNN 模型分发到指定的 NPU 设备上推理并获取推理结果；或在计算机上仿真 NPU 运行 RKNN 模型并获取推理结果。
4. 性能和内存评估：将 RKNN 模型分发到指定 NPU 设备上运行，以评估模型在实际设备上运行时的性能和内存占用情况。
5. 量化精度分析：该功能将给出模型量化后每一层推理结果与浮点模型推理结果的余弦距离和欧氏距离，以便于分析量化误差是如何出现的，为提高量化模型的精度提供思路。
6. 模型加密功能：使用指定的加密等级将 RKNN 模型整体加密。

1.1.3 RKNN Runtime 功能介绍

RKNN Runtime 负责加载 RKNN 模型，并调用 NPU 驱动实现在 NPU 上推理 RKNN 模型。推理 RKNN 模型时，包括原始数据输入预处理、NPU 运行模型、输出后处理三项流程。根据不同模型输入格式和量化方式，RKNN Runtime 提供通用 API 和零拷贝 API 两种处理流程。

- 通用 API 推理：提供一套简洁、无门槛的推理 API，易于使用，流程如图 1-3 所示。其中对数据的归一化、量化、数据排布格式转换、反量化等均在 CPU 上运行，模型本身的推理在 NPU 上运行。

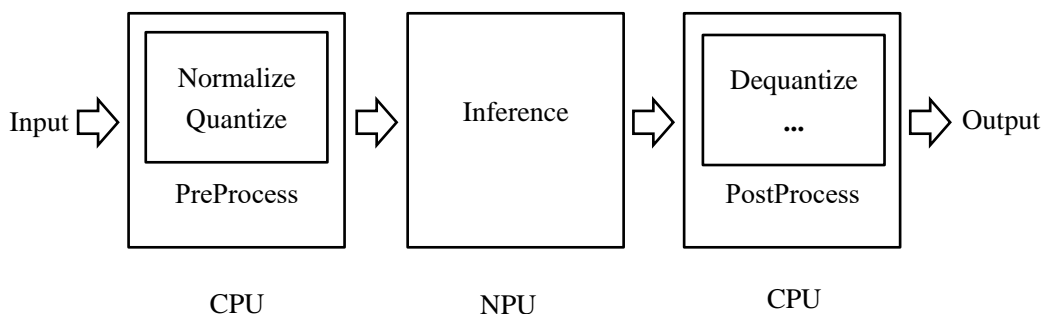


图 1-3 通用 API 的数据处理流程

- 零拷贝 API 推理：流程如图 1-4 所示。优化了通用 API 的数据处理流程，归一化、量化和模型推理都会在 NPU 上运行，NPU 输出的数据排布格式和反量化过程在 CPU 或者 NPU 上运行。零拷贝 API 对于输入数据流程的处理效率会比通用 API 高。支持数据在不同的 IP 核之间流动，没有数据拷贝，减少 CPU 及 DDR 带宽消耗。比如通过 camera 或者解码出来的数据，支持零拷贝导入到 NPU 中使用。

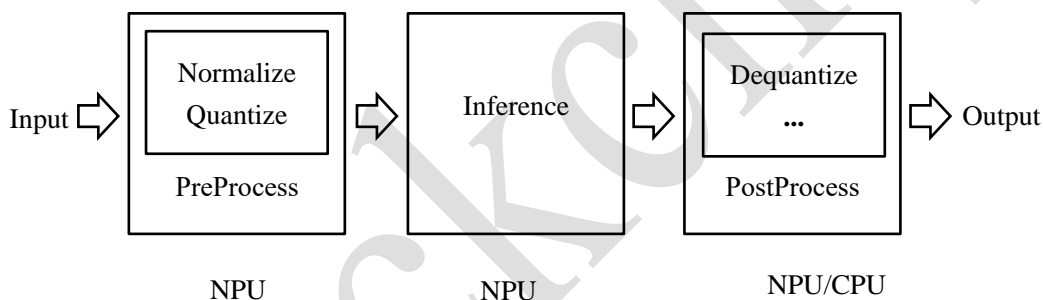


图 1-4 零拷贝 API 数据处理流程

当用户输入数据只有虚拟地址时，只能使用通用 API 接口；当用户输入数据有物理地址或 fd 时，两组接口都可以使用。

1.2 RKNN 开发流程介绍

用户可参考流程图了解 RKNN 的整体开发步骤，流程主要分为 3 个部分：模型转换、模型评估和板端部署运行。



图 1-5 RKNN 开发流程图

1. 模型转换：

在这一阶段，原始的深度学习模型会被转化为 RKNN 格式，以便在 RKNPU 平台上进行高效的推理。这一步骤包括：

- a. 获取原始模型：获取或训练深度学习模型，通常使用主流框架，如 ONNX、PyTorch 或 TensorFlow。
- b. 模型配置：在 RKNN-Toolkit2 中进行必要的配置，如均值、归一化、量化参数和目标平台等。
- c. 模型加载：使用适当的加载接口将模型导入 RKNN-Toolkit2，根据模型类型选择正确的加载方法。
- d. 模型构建：通过 `rknn.build()` 接口构建 RKNN 模型，可选择是否进行量化，提高模型部署在硬件上的性能。
- e. 模型导出：通过 `rknn.export_rknn()` 接口将 RKNN 模型导出为一个文件（.rknn 格式），用于后续部署。

具体内容请见 [3.1](#) 和 [4.2.1](#) 章节内容。

2. 模型评估：

模型评估阶段帮助用户量化和分析模型性能，包括精度、连板推理性能和内存占用等关键指标。评估的主要步骤包括以下操作：

- a. 模型连板调试：模型连板调试阶段是通过使用 Python 连接 RKNPU 平台进行推理并验证模型，这个阶段涵盖了输入数据的前处理和输出结果的后处理，以确保模型在板上运行正确。
- b. 精度评估：通过 `rknn.accuracy_analysis()` 接口，比较量化模型与浮点模型之间的输出分析量化误差。
- c. 性能评估：通过 `rknn.eval_perf()` 接口，了解模型在平台上的推理性能，帮助用户进一步优化模型结构加快推理性能。
- d. 内存评估：通过 `rknn.eval_memory()` 接口，了解模型在板上的内存使用情况，帮助用户进一步优化模型结构最小化内存占用。

具体内容请见 [3.2](#)、[3.3](#)、[4.2.2](#) 和 [4.2.3](#) 章节内容。

3. 板端部署运行：

这个阶段涵盖了模型的实际部署和运行。它通常包括以下步骤：

- a. 模型初始化：加载 RKNN 模型到 RKNPU 平台，准备进行前处理。

- b. 模型前处理：加载待推理数据到 RKNPU 平台，准备进行推理。
- c. 模型推理：执行推理操作，将输入数据传递给模型并获取推理结果。
- d. 模型后处理：取出推理结果进行后处理，后处理结果传给应用端。
- e. 模型释放：在完成推理流程后，释放模型资源，以便其他任务使用 RKNN 模型。

具体内容请见 [3.4](#) 和 [4.2](#) 章节内容。

这三个步骤构成了完整的 RKNN 开发流程，确保人工智能模型能够成功转换、调试、评估和最终在 RKNPU 上高效部署。

1.3 适用的硬件平台

本文档适用如下硬件平台：

RK3576、RK3562、RK3566 系列、RK3568 系列、RK3588 系列、RV1103、RV1106

注：文档部分地方使用 RK3566_RK3568 来统一表示 RK3566/RK3568 系列，使用 RK3588 来统一表示 RK3588/RK3588S 系列。

1.4 关键字说明

RKNN 模型：指运行在 RKNPU 上的文件，后缀名为.rknn。

连板推理：指通过 USB 口连接 PC 和开发板，调用 RKNN-Toolkit2 的接口运行模型。模型实际运行在开发板的 NPU 上。

HIDL：用于指定 Android HAL 和其用户之间的接口的一种接口描述语言。

CTS：全名兼容性测试套件，是谷歌提供的一个 Andorid 平台自动化测试套件。

VTS：全名供应商测试套件，是谷歌提供的一个 Andorid 平台自动化测试套件。

DRM：英文全名 Direct Rendering Manager，是一个主流的图形显示框架。

NATIVE_LAYOUT：指对于 NPU 运行时而言，通常性能表现最佳的计算机内存排列格式。

tensor：张量，在深度学习中，用它表示高阶数组的数据。

fd：文件描述符，被用来标识一块内存空间。

i8 模型：量化的 RKNN 模型，即以 8 位有符号整型数据运行的模型。

fp16 模型：非量化的 RKNN 模型，即以 16 位半精度浮点型数据运行的模型。

2 开发环境准备

2.1 RKNN-Toolkit2 安装

本章节提供两种 RKNN-Toolkit2 安装方式，通过 Docker 方式安装和通过 pip 方式安装，用户可自行选择任意一种方式进行安装。

2.1.1 通过 Docker 方式安装

2.1.1.1 安装 Docker 工具

已安装 Docker 工具的用户可跳过此步骤，未安装的用户请根据官方手册进行安装。

Docker 安装官方手册链接：<https://docs.docker.com/install/linux/docker-ce/ubuntu/>。

注意事项：需要将用户添加到 docker 用户组。

```
# 创建 docker 用户组
sudo groupadd docker
# 把当前用户加入 docker 用户组
sudo usermod -aG docker $USER
# 更新激活 docker 用户组
newgrp docker
# 验证不需要 sudo 执行 docker 命令
docker run hello-world
```

正确运行结果展示：

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest:
sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5a
f48e8d
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

2.1.1.2 镜像准备

本节介绍两种 RKNN-Toolkit2 镜像环境准备方式，可任选一种方式进行操作。

1. 通过 Dockerfile 创建镜像环境

在 RKNN-Toolkit2 工程中 docker/docker_file 文件夹下，提供了构建 RKNN-Toolkit2 开发环境的 Dockerfile 文件，用户通过 Docker 命令创建镜像，如下所示。

```
cd docker/docker_file/ubuntu_xx_xx_cpxx
docker build -f Dockerfile_ubuntu_xx_xx_for_cpxx -t rknn-
toolkit2:x.x.x-cpxx .
```

2. 加载已打包所有开发环境的 Docker 镜像

下载对应版本的 RKNN-Toolkit2 工程文件，解压后在 docker/docker_image 文件夹下提供了已打包所有开发环境的 Docker 镜像。

网盘下载链接：<https://console.zbox.filez.com/l/I00fc3>。提取码：rknn

执行以下命令加载对应 Python 版本的镜像文件。

```
docker load --input rknn-toolkit2-x.x.x-cpxx-docker.tar.gz
```

2.1.1.3 查询镜像信息

创建或加载镜像成功后，查看 Docker 的镜像信息。

```
docker images
```

相应的 RKNN-Toolkit2 镜像信息显示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
rknn-toolkit2	x.x.x-cpxx	xxxxxxxxxxxxx	1 hours ago	5.89GB

2.1.1.4 运行镜像

执行以下命令运行 Docker 镜像，运行后将进入镜像的 bash 环境。

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb rknn-
toolkit2:x.x.x-cpxx /bin/bash
```

将文件夹 examples 代码映射进 Docker 环境可通过附加 “-v <host src folder>:<image dst folder>” 参数。

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb -v
/your/rknn-toolkit2-x.x.x/examples:/examples rknn-toolkit2:x.x.x-
cpxx /bin/bash
```

2.1.1.5 运行 Demo

```
cd examples/onnx/yolov5
python test.py
```

脚本运行成功后结果如下。

class	score	xmin, ymin, xmax, ymax
person	0.884	[208, 244, 286, 506]
person	0.868	[478, 236, 559, 528]
person	0.825	[110, 238, 230, 534]
person	0.339	[79, 353, 122, 516]
bus	0.705	[92, 128, 554, 467]

2.1.2 通过 Pip 方式安装

2.1.2.1 安装 Python 环境

若已安装 Python 环境，则可省略此步骤。

```
sudo apt-get update
sudo apt-get install python3 python3-dev python3-pip
sudo apt-get install libxslt1-dev zlib1g zlib1g-dev libglib2.0-0
libsm6 libgl1-mesa-glx libprotobuf-dev gcc
```

若想安装 RKNN-Toolkit2 工具在本地环境（非 Conda 虚拟环境）中，请在安装好 Python 环境后跳至步骤 2.1.2.4。

2.1.2.2 安装 Conda 工具

如果系统中同时有多个版本的 Python 环境，建议使用 Conda 管理 Python 环境。

检查是否安装 Conda 和版本信息，若已安装则可省略此小节步骤。

```
conda -V
# 提示 conda: command not found 则表示未安装 conda
# 提示 例如版本 conda 23.9.0
```

下载 Conda 安装包

```
wget -c
https://mirrors.bfsu.edu.cn/anaconda/miniconda/Miniconda3-latest-
Linux-x86_64.sh
```

安装 Conda

```
chmod 777 Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

2.1.2.3 创建 RKNN-Toolkit2 Conda 环境

进入 Conda base 环境

```
source ~/miniconda3/bin/activate # miniconda3 为安装目录
# (base) xxx@xxx-pc:~$
```

创建一个 Python3.8 版本（建议版本）名为 RKNN-Toolkit2 的 Conda 环境

```
conda create -n RKNN-Toolkit2 python=3.8
```

进入 RKNN-Toolkit2 Conda 环境

```
conda activate RKNN-Toolkit2
# (RKNN-Toolkit2) xxx@xxx-pc:~$
```

2.1.2.4 安装 RKNN-Toolkit2 依赖库

可以在本地环境或 RKNN-Toolkit2 Conda 环境下安装。根据不同的 Python 版本，安装选择对应的依赖包。

```
pip3 install -r rknn-toolkit2/packages/requirements_cpxx.txt
```

表 2-1 不同 Python 版本对应的依赖包

Python 版本	RKNN-Toolkit2 依赖包
3.6	requirements_cp36.txt
3.7	requirements_cp37.txt
3.8	requirements_cp38.txt
3.9	requirements_cp39.txt
3.10	requirements_cp310.txt
3.11	requirements_cp311.txt

2.1.2.5 安装 RKNN-Toolkit2

可以在本地环境或 RKNN-Toolkit2 Conda 环境下安装。请根据不同的 Python 版本，选择 packages 文件夹下不同的安装包文件。

```
pip3 install rknn-toolkit2/packages/rknn_toolkit2-x.x.x+xxxxxxxx-cp36-cp36m-linux_x86_64.whl
```

包名格式为：rknn_toolkit2-{版本号}+{commit 号}-cp{Python 版本}-cp{Python 版本}-linux_x86_64.whl，例如：rknn_toolkit2-1.5.2+b642f30c-cp36-cp36m-linux_x86_64.whl。

表 2-2 不同 Python 版本对应的安装包

Python 版本	RKNN-Toolkit2 安装包
3.6	rknn_toolkit2-{版本号}+{commit 号}-cp36-cp36m-linux_x86_64.whl
3.7	rknn_toolkit2-{版本号}+{commit 号}-cp37-cp37m-linux_x86_64.whl
3.8	rknn_toolkit2-{版本号}+{commit 号}-cp38-cp38-linux_x86_64.whl
3.9	rknn_toolkit2-{版本号}+{commit 号}-cp39-cp39-linux_x86_64.whl
3.10	rknn_toolkit2-{版本号}+{commit 号}-cp310-cp310-linux_x86_64.whl
3.11	rknn_toolkit2-{版本号}+{commit 号}-cp311-cp311-linux_x86_64.whl

若执行以下命令没有报错，则安装成功。

```
from rknn.api import RKNN
```

2.2 设备 NPU 环境准备

2.2.1 NPU 驱动版本确认

查询命令：

```
dmesg | grep -i rknpu
# 或
cat /sys/kernel/debug/rknpu/version
# 或
cat /sys/kernel/debug/rknpu/driver_version
# 或
cat /proc/debug/rknpu/driver_version
```

查询结果：

```
RKNPU driver: vX.X.X
```

X.X.X 表示版本号，例如 0.9.2。

Rockchip 的固件均自带 RKNPU 驱动。若以上命令均查询不到 NPU 驱动版本，则可能为第三方固件未安装 RKNPU 驱动，需要打开 kernel config 文件的“CONFIG_ROCKCHIP_RKNPU=y”选项，重新编译内核驱动并烧录。建议 RKNPU 驱动版本 $\geq 0.9.2$ 。

2.2.2 NPU 连板环境确认

RKNN-Toolkit2 的连板调试功能需要板端的 RKNN Server，其是一个运行在开发板上的后台代理服务，用于接收电脑通过 USB 传输过来的命令和数据，然后执行对应的接口，并返回结果给电脑。所以需要确认开发板是否启动 RKNN Server。

进入开发板终端，查询是否有 RKNN Server 进程。

```
adb shell
ps | grep rknn_server
```

查询结果：

```
702 root      1192 S      grep rknn_server
```

若查询到 RKNN Server 进程 ID，则说明 RKNN Server 启动正常。若没有查询到 RKNN Server 进程 ID，则执行以下命令手动启动 RKNN Server 后再进行查询。

Android 系统手动启动 RKNN Server：

```
su
setenforce 0
/vendor/bin/rknn_server &
```

Linux 系统手动启动 RKNN Server：

```
restart_rknn.sh
```

正常情况下 Rockchip 固件均集成 RKNN Server 并自启动，若无自启动或无相关文件用

于手动启动，请手动安装或更新 RKNN Server。

2.2.3 RKNN Server 安装和更新

RKNN-Toolkit2 的连板调试功能要求板端已安装 RKNPU2 环境，并且启动 RKNN Server 服务。以下是 RKNPU2 环境中的 2 个基本概念：

1. RKNN Server：一个运行在开发板上的后台代理服务，用于接收电脑通过 USB 传输过来的数据，然后执行板端 Runtime 对应的接口，并返回结果给 PC。

2. RKNPU2 Runtime 库

- librknnrt.so：是用于 RK3562/RK3566/RK3568/RK3588/RK3576 板端的 Runtime 库。
- librknnmrt.so：是用于 RV1103/RV1106 板端的 Runtime 库。

如果板端没有安装 RKNN Server，Runtime 库不存在，或者 RKNN Server 和 Runtime 库的版本不一致，都需要重新安装或更新。

注意：

1. 若使用动态维度输入的 RKNN 模型，则要求 RKNN Server 和 RKNPU2 Runtime 库版本 $\geq 1.5.0$ 。

2. 要保证 RKNN Server、Runtime 库的版本、RKNN-Toolkit2 的版本是一致的，建议都安装最新的版本。

2.2.3.1 RK356X/RK3588/RK3576 平台

1. Android 系统

查询 RKNN Server 服务和 librknnrt.so 库版本，若版本号不一致请更新至同一版本。

```
# 查询 rknn_server 版本
strings /vendor/bin/rknn_server | grep -i "rknn_server version"
# 显示 rknn_server 版本为 X.X.X
# rknn_server version: X.X.X

# 查询 librknnrt.so 库版本
# 64 位系统
strings /vendor/lib64/librknnrt.so | grep -i "librknnrt version"
# 32 位系统
strings /vendor/lib/librknnrt.so | grep -i "librknnrt version"
# 显示 librknnrt 库版本为 X.X.X
# librknnrt version: X.X.X
```

更新 RKNN Server 服务和 librknnrt.so 库。

```
adb root
```

```
adb remount
```

64 位系统:

```
adb push runtime/Android/rknn_server/arm64/rknn_server
/vendor/bin/
adb push runtime/Android/librknn_api/arm64-v8a/librknnrt.so
/vendor/lib64
```

32 位系统:

```
adb push runtime/Android/rknn_server/arm/rknn_server /vendor/bin/
adb push runtime/Android/librknn_api/armeabi-v7a/librknnrt.so
/vendor/lib
```

重启 RKNN Server 服务:

```
adb shell
su
chmod +x /vendor/bin/rknn_server
sync
reboot
```

2. Linux 系统

查询 RKNN Server 服务和 librknnrt.so 库版本, 若不一致请更新至同一版本。

```
# 查询 rknn_server 版本
strings /usr/bin/rknn_server | grep -i "rknn_server version"
# 显示 rknn_server 版本为 X.X.X
# rknn_server version: X.X.X

# 查询 librknnrt.so 库版本
strings /usr/lib/librknnrt.so | grep -i "librknnrt version"
# 显示 librknnrt 库版本为 X.X.X
# librknnrt version: X.X.X
```

更新 RKNN Server 服务和 librknnrt.so 库。

64 位系统:

```
adb push runtime/Linux/rknn_server/aarch64/usr/bin/* /usr/bin
adb push runtime/Linux/librknn_api/aarch64/librknnrt.so /usr/lib
```

32 位系统:

```
adb push runtime/Linux/rknn_server/armhf/usr/bin/* /usr/bin
adb push runtime/Linux/librknn_api/armhf/librknnrt.so /usr/lib
```

重启 RKNN Server 服务:

```
adb shell
chmod +x /usr/bin/rknn_server
chmod +x /usr/bin/start_rknn.sh
chmod +x /usr/bin/restart_rknn.sh
restart_rknn.sh
```

2.2.3.2 RV1103/RV1106 平台

查询 RKNN Server 和 librknnmrt.so 库版本, 若不一致请更新至同一版本。

```
# 查询 rknn_server 版本
strings /oem/usr/bin/rknn_server | grep -i "rknn_server version"
# 显示 rknn_server 版本为 X.X.X
# rknn_server version: X.X.X

# 查询 librknmmrt.so 库版本
strings /oem/usr/lib/librknmmrt.so | grep -i "librknmmrt version"
# 显示 librknmmrt 库版本为 X.X.X
# librknmmrt version: X.X.X
```

更新 RKNN Server 服务和 librknmmrt.so 库。RV1103 与 RV1106 使用同一份 RKNN Server 服务和 librknmmrt.so 库。

```
adb push runtime/Linux/rknn_server/armhf-uclibc/usr/bin/*
/oem/usr/bin
adb push runtime/Linux/librknn_api/armhf-uclibc/librknmmrt.so
/oem/usr/lib
```

重启 RKNN Server 服务：

```
adb shell
chmod +x /oem/usr/bin/rknn_server
chmod +x /oem/usr/bin/start_rknn.sh
chmod +x /oem/usr/bin/restart_rknn.sh
restart_rknn.sh
```

2.2.4 查看 RKNN Server 详细日志

2.2.4.1 Android 系统

进入开发板终端，设置日志等级。

```
adb shell
su
setenforce 0
setprop persist.vendor.rknn.server.log.level 5
```

关闭当前 RKNN Server 服务进程。

```
kill -9 `pgrep rknn_server`
```

若没有自动重启 RKNN Server 服务，可以手动启动，查看详细日志。

```
/vendor/bin/rknn_server &
logcat
```

2.2.4.2 Linux 系统

进入开发板终端，设置日志等级。

```
adb shell
export RKNN_SERVER_LOGLEVEL=5
```

重启 RKNN Server 服务可查看详细日志。

```
restart_rknn.sh
```

3 RKNN 使用说明

3.1 模型转换

RKNN-Toolkit2 提供了丰富的功能，包括模型转换、性能分析、部署调试等。本节将重点介绍 RKNN-Toolkit2 的模型转换功能。模型转换是 RKNN-Toolkit2 的核心功能之一，它允许用户将各种深度学习模型从不同的框架转换为 RKNN 格式以在 RKNPU 上运行，用户可以参考模型转换流程图以帮助理解如何进行模型转换。

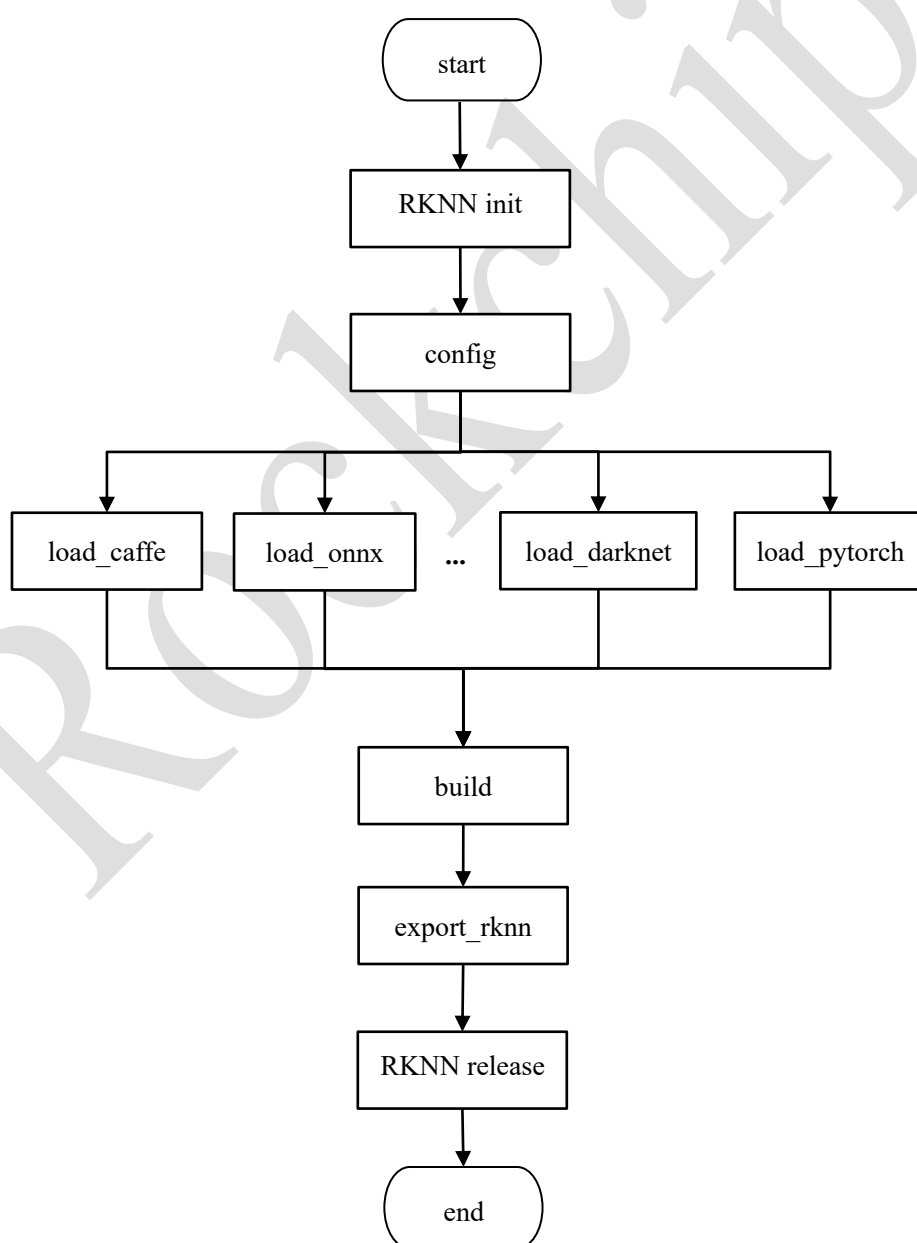


图 3-1 模型转换流程图

目前 RKNN-Toolkit2 支持多个主流深度学习框架的模型转换，包括：

- Caffe（推荐版本为 1.0）
- TensorFlow（推荐版本为 1.12.0~2.8.0）
- TensorFlow Lite（推荐版本为 Schema version = 3）
- ONNX（推荐版本为 1.7.0~1.10.0）
- PyTorch（推荐版本为 1.6.0~1.13.1）
- Darknet（推荐版本为 Commit ID = 810d7f7）

用户可以使用上述框架训练或获取预训练模型并将它们转换为 RKNN 格式，更高效地在 RKNPU 平台上进行部署和推理。

3.1.1 RKNN 初始化及对象释放

在这一部分，用户需要先初始化 RKNN 对象，这是整个工作流程的第一步：

- 初始化 RKNN 对象：
 - 使用 RKNN()构造函数来初始化 RKNN 对象，用户可以传入参数 verbose 和 verbose_file。
 - verbose 决定是否在屏幕上显示详细日志信息。
 - 如果设置了 verbose_file 参数并且 verbose 为 True，日志信息还将写入到指定的文件中。

示例代码：

```
rknn = RKNN(verbose=True, verbose_file='./mobilenet_build.log')
```

当完成所有的 RKNN 相关的操作后，用户需要释放资源，这是整个工作流程的最后一步：

- 释放资源：
 - 使用 release()方法来释放 RKNN 对象占用的资源。

示例代码：

```
rknn.release()
```

3.1.2 模型 Config 配置

在模型转换之前，用户需要进行一些配置以确保模型转换的正确性和性能。配置参数可以通过 rknn.config()接口进行设置，包括设置输入均值、归一化值、是否量化等等，下面列出了一些常用的配置参数：

- `mean_values` 和 `std_values` 用于设置输入的均值和归一化值。这些值在量化过程中使用，且 C API 推理阶段图片不需再做均值和归一化值减小部署耗时。
- `quant_img_RGB2BGR` 用于控制量化时加载量化校正图像时是否需要先进行 RGB 到 BGR 的转换，默认值为 `False`。该配置只在量化数据集时生效，实际部署模型时，模型推理阶段不会生效，需要用户在输入前处理里预先处理好。注：`quant_img_RGB2BGR = True` 时模型的推理顺序为先做 RGB2BGR 转换再做 `mean_values` 和 `std_values` 操作，详细注意事项请见 [10.3](#) 章节。
- `target_platform` 用于指定 RKNN 模型的目标平台，支持 RK3576、RK3568、RK3566、RK3562、RK3588、RV1106 和 RV1103。
- `quantized_algorithm` 用于指定计算每一层的量化参数时采用的量化算法，可以选择 `normal`、`mmse` 或 `kl_divergence`，默认算法为 `normal`，详细说明见 [3.1.7](#)、[6.1](#) 和 [6.2](#) 章节。
- `quantized_method` 支持 `layer` 或 `channel`，用于每层的权重是否共享参数，默认为 `channel`，详细说明见 [3.1.7](#)、[6.1](#) 和 [6.2](#) 章节。
- `optimization_level` 通过修改模型优化等级，可以关掉部分或全部模型转换过程中使用的优化规则。该参数的默认值为 3，打开所有优化选项，值为 2 或 1 时关闭一部分可能会对部分模型精度产生影响的优化选项，值为 0 时关闭所有优化选项。
- `quantized_dtype` 用于指定量化类型，目前支持线性非对称量化的 `INT8`，默认为 `'asymmetric_quantized-8'`。
- `custom_string` 添加自定义字符串信息到 RKNN 模型，可以在 `runtime` 时通过 `query` 查询 `RKNN_QUERY_CUSTOM_STRING` 得到该信息，方便部署时根据不同的 RKNN 模型做特殊的处理。默认值为 `None`。
- `dynamic_input` 支持动态输入，根据用户指定的多组输入 `shape`，来模拟动态输入的功能，默认值为 `None`，详细说明见 [5.4](#) 章节。
- 更具体的 `rknn.config()` 接口配置请参考 API 手册，上述仅列出部分常用参数。

示例代码：

```
rknn.config(  
    mean_values=[[103.94, 116.78, 123.68]],  
    std_values=[[58.82, 58.82, 58.82]],  
    quant_img_RGB2BGR=False,  
    target_platform='rk3566')
```

3.1.3 模型加载接口介绍

根据不同类型的模型，用户需要使用相应的加载接口加载模型文件。RKNN-Toolkit2 提供了不同的加载接口，包括 Caffe、TensorFlow、TensorFlow Lite、ONNX、PyTorch 等，下面是各种类型的模型加载接口的简要介绍：

- Caffe 模型加载接口：

- 使用 `rknn.load_caffe()` 接口加载 Caffe 模型。
- 需要提供模型文件（.prototxt 后缀）路径和权重文件（.caffemodel 后缀）路径。
- 如果模型有多个输入层，可以指定输入层名称的顺序。

示例代码：

```
ret = rknn.load_caffe(model='./mobilenet_v2.prototxt',
                      blobs='./mobilenet_v2.caffemodel')
```

- TensorFlow 模型加载接口：

- 使用 `rknn.load_tensorflow()` 接口加载 TensorFlow 模型。
- 需要提供 TensorFlow 模型文件（.pb 后缀）路径、输入节点名、输入节点的形状以及输出节点名。

示例代码：

```
ret = rknn.load_tensorflow(
    tf_pb='./ssd_mobilenet_v1_coco_2017_11_17.pb',
    inputs=['Preprocessor/sub'],
    outputs=['concat', 'concat_1'],
    input_size_list=[[1, 300, 300, 3]])
```

- TensorFlow Lite 模型加载接口：

- 使用 `rknn.load_tflite()` 接口加载 TensorFlow Lite 模型。
- 需要提供 TensorFlow Lite 模型文件（.tflite 后缀）路径。

示例代码：

```
ret = rknn.load_tflite(model='./mobilenet_v1.tflite')
```

- ONNX 模型加载接口：

- 使用 `rknn.load_onnx()` 接口加载 ONNX 模型。
- 需要提供 ONNX 模型文件（.onnx 后缀）路径。

示例代码：

```
ret = rknn.load_onnx(model='./arcface.onnx')
```

- DarkNet 模型加载接口：

- 使用 `rknn.load_darknet()` 接口加载 DarkNet 模型。
- 需要提供 DarkNet 模型文件（.cfg 后缀）路径和权重文件（.weights 后缀）路径。

示例代码：

```
ret = rknn.load_darknet(model='./yolov3-tiny.cfg',
weight='./yolov3.weights')
```

- PyTorch 模型加载接口：

- 使用 rknn.load_pytorch()接口加载 PyTorch 模型。
- 需要提供 PyTorch 模型文件（.pt 后缀）路径，模型必须是 torchscript 格式的。

示例代码：

```
ret = rknn.load_pytorch(model='./resnet18.pt',
input_size_list=[[1, 3, 224, 224]])
```

用户可以根据不同类型的模型选择合适的接口进行加载，确保模型转换的正确性。

3.1.4 构建 RKNN 模型

用户加载原始模型后，下一步就是通过 rknn.build()接口构建 RKNN 模型。构建模型时，用户可以选择是否进行量化，量化助于减小模型的大小和提高在 RKNPU 上的性能。

rknn.build()接口参数如下：

- do_quantization 参数控制是否对模型进行量化，建议设置为 True。
- dataset 参数用于提供用于量化校准的数据集，数据集的格式是文本文件。

dataset.txt 示例：

```
./imgs/ILSVRC2012_val_00000665.JPEG
./imgs/ILSVRC2012_val_00001123.JPEG
./imgs/ILSVRC2012_val_00001129.JPEG
./imgs/ILSVRC2012_val_00001284.JPEG
./imgs/ILSVRC2012_val_00003026.JPEG
./imgs/ILSVRC2012_val_00005276.JPEG
```

示例代码：

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
```

3.1.5 导出 RKNN 模型

用户通过 rknn.build()接口构建了 RKNN 模型后，可以通过 rknn.export_rknn()接口将 RKNN 模型保存为一个文件（.rknn 后缀），以便后续模型的部署。rknn.export_rknn()接口参数如下：

- export_path 导出模型文件的路径。
- cpp_gen_cfg 可以选择是否生成 C++ 部署示例。

示例代码：

```
ret = rknn.export_rknn(export_path='./mobilenet_v1.rknn')
```

这些操作和接口涵盖了 RKNN-Toolkit2 模型转换步骤，根据不同的需求和应用场景，

用户可以选择不同的配置选项和量化算法进行自定义设置，方便后续进行部署。

3.1.6 模型转换工具 RKNN_Convert

rknn_convert 是 RKNN-Toolkit2 提供的一套常用模型转换工具，通过封装上述 API 接口，用户只需编辑模型对应的 yml 配置文件，就可以通过指令转换模型。以下是如何使用 rknn_convert 工具的示例命令以及支持的指令参数：

```
python -m rknn.api.rknn_convert -t rk3588 -i ./model_config.yml -o ./output_path
```

通过使用上述命令和参数，用户可以将模型转换为 RKNN 格式，并将转换后的模型保存到指定的输出路径。支持的指令参数说明如下：

- -i: 模型配置文件（.yaml）路径。
- -o: 转换后模型输出路径。
- -t: target_platform，目标平台可以选择 rv1103，rv1106，rk3568，rk3566，rk3562，rk3576 或 rk3588。
- -e: (选填) 评估连板运行时 model 的耗时和内存占用，若开启请输入-e。注：一定要连板和正确设置 target_platform，否则会报错，当有多设备时可开启-d device_id 选择设备 ID。
- -a: (选填) 评估生成的 rknn 模型精度，开启模拟器精度评估请输入-a "xx1.jpg xx2.jpg"，若要开启连板精度评估请配合-d 参数使用。
- -v: (选填) 指定是否要在屏幕上打印详细日志信息，若开启打印模式请输入-v。
- -d: (选填) 单个 adb 设备使用-d，多 adb 设备使用-d device_id，device_id 通过 adb devices 查询。

下面是一个示例的 yml 配置文件(object_detection.yml)：

```
models:
  # model output name
  name: object_detection
  # Original model framework
  platform: onnx
  # Model input file path
  model_file_path: ./object_detection.onnx
  # Describe information such as input and output shapes
  subgraphs:
    # model input tensor shape
    input_size_list:
      - 1,3,512,512
    # input tensor name
    inputs:
      - data
    # output tensor name
    outputs:
      - conv6-1
      - conv6-2
      - conv6-3
  # quantification flag
  quantize: true
  # Quantify dataset file path (relative yml path)
  dataset: ./dataset.txt
  configs:
    quantized_dtype: asymmetric_quantized-8
    # rknn.config mean_values
    mean_values: [127.5,127.5,127.5]
    # rknn.config std_values
    std_values: [128.0,128.0,128.0]
    # rknn.config quant_img_RGB2BGR
    quant_img_RGB2BGR: false
    # rknn.config quantized_algorithm
    quantized_algorithm: normal
```

这个配置文件包括了模型的名称、原始模型使用的框架、模型文件路径、输入输出信息、是否进行量化等详细信息。用户可以根据模型的特定需求编辑相应的配置文件。

yaml 模型转换配置文件附录表：

表 3-1 yaml 模型转换配置参数说明

参数名	填写内容
-name	模型输出名称
-platform	原始模型使用的框架，支持 tensorflow、tflite、caffe、onnx、pytorch、darknet
-model_file_path	原始模型文件路径，适用于单模型文件输入，例：tensorflow、tflite、onnx、pytorch
-quantize	是否开启量化
-dataset	量化 dataset 文件路径（相对 yml 配置文件路径），若要开启 accuracy_analysis 此项必填
-prototxt_file_path	platform 为 caffe 时，模型的 prototxt 文件
-caffemodel_file_path	platform 为 caffe 时，模型的 caffemodel 文件
-darknet_cfg_path	platform 为 darknet 时，模型的 cfg 文件
-darknet_weights_path	platform 为 darknet 时，模型的 weight 文件
-subgraphs	描述输入输出 shape 等信息。除特定框架外，一般情况下该参数及附带的子参数可不写，使用模型默认值
----input_size_list(子参数)	输入 tensor 的 shape
----inputs(子参数)	输入 tensor 的名称
----outputs(子参数)	输出 tensor 的名称
-configs	对应 rknn.config()配置
----quantized_dtype(子参数)	量化类型，RKNN_toolkit2: 可填写 [asymmetric_quantized-8], 不输入用默认值
----mean_values(子参数)	输入的均值归一数，模型为单输入 RGB 如 [123.675,116.28,103.53]，若为多输入如 [[123,116,103],[255,255,255]]

----std_values(子参数)	输入的方差归一数，模型为单输入 RGB 如 [58.395,58.295,58.391]，若为多输入如 [[127,127,127],[255,255,255]]
----quant_img_RGB2BGR(子参数)	用于控制量化时加载量化校正图像时是否需要先进行 RGB 到 BGR 的转换，默认值是 False
----quantized_algorithm(子参数)	量化算法，可选['normal', 'kl_divergence', 'mmse']，默认为 normal
----quantized_method(子参数)	量化方式，RKNN_toolkit2 可选['layer', 'channel']，默认为 channel
----optimization_level(子参数)	设置优化级别。默认为 3，表示使用所有默认优化选项
----model_pruning(子参数)	修剪模型以减小模型大小，默认为 false，开启为 true
----quantize_weight(子参数)	当 quantize 参数为 false 时，通过量化一些权重来减小 rknn 模型的大小。默认为 false，开启为 true
----single_core_mode(子参数)	是否仅生成单核模型，可以减小 RKNN 模型的大小和内存消耗。默认值为 False。目前对 RK3588/RK3576 生效。默认值为 False

3.1.7 RKNN-Toolkit2 模型量化功能

RKNN-Toolkit2 提供两种量化方式和三种量化算法，用户可以通过 rknn.config() 中的 quantized_algorithm 和 quantized_method 参数来选择。以下是这些量化算法和特征量化方式的介绍：

- 三种量化算法：

1. **Normal 量化算法：**通过计算模型中特征（feature）的浮点数最大值和最小值来确定量化范围的最大值和最小值。算法的特点是速度较快，但是遇到特征分布不均衡时效果较差，推荐量化数据量一般为 20-100 张左右。
2. **MMSE 量化算法：**通过最小化浮点数与量化反量化后浮点数的均方误差损失确定量化范围的最大值和最小值，能够一定程度的缓解大异常值带来的量化精度丢失问题。由于采用暴力迭代的方式，速度较慢，但通常会比 normal 具有更高的精度，推荐量化数据量一般

为 20-50 张左右，用户也可以根据量化时间长短对量化数据量进行适当增减。

3. **KL-Divergence 量化算法**：将模型中特征（feature）中浮点数和定点数抽象成两个分布，通过调整不同的阈值来更新浮点数和定点数的分布，并根据 KL 散度衡量两个分布的相似性来确定量化范围的最大值和最小值。所用时间会比 normal 多一些，但比 mmse 会少很多，在某些场景下（feature 分布不均匀时）可以得到较好的改善效果，推荐量化数据量一般为 20-100 张左右。

- 两种量化方式：

1. **Layer 量化方式**：Layer 量化方式将同一层网络的所有通道作为一个整体进行量化，所有通道共享相同的量化参数。
2. **Channel 量化方式**：Channel 量化方式将同一层网络的各个通道独立进行量化，每个通道有自己的量化参数。通常情况下，Channel 量化方式比 Layer 量化方式具有更高的精度。
3. 根据实际的模型量化效果和需求，用户可以选择合适的量化算法和特征量化方式，更详细的量化说明和原理请见第 6 章。

3.2 模型评估

3.2.1 模型推理

原始模型转换为 RKNN 模型后，可在模拟器或开发板上进行推理，对推理结果进行后处理检验其是否正确。若推理结果不正确，可以对量化模型进行精度分析和查看前后处理流程是否正确。

初始化 `rknn.init_runtime()` 接口参数如下：

- **target**：目标硬件平台。默认值为 None，推理在模拟器上进行。需要获取板端实际推理结果时需要填入相应的平台（RK3562 / RK3566 / RK3568 / RK3588 / RK3576 / RV1103 / RV1106）。
- **device_id**：设备编号。默认值为 None。若有设置 target 则选择唯一一台设备进行推理。如果电脑连接多台设备连板推理时，需要指定填入相应的设备 ID。若通过网络 adb 连接设备进行模型推理，则需要用户手动执行命令 `adb connect [IP]` 连接网络设备后再填入设备编号，通常为 [IP] 或 [IP:Port] 的形式。
- **perf_debug**：进行性能评估时是否开启 debug 模式。默认值为 False，调用 `rknn.val_perf()` 接口可以获取模型运行的总时间。设为 True 时，可以额外获取到每一层的运行时间，

RV1103 和 RV1106 平台不支持。

- **eval_mem**: 是否进入内存评估模式。默认值为 False。设为 True 进入内存评估模式后，可以调用 `rknn.eval_memory()` 接口获取模型运行时的内存使用情况。
- 推理 `rknn.inference()` 接口参数如下：
- **inputs**: 待推理的输入列表，格式为 ndarray。
- **data_format**: 输入数据的 layout 列表，只对 4 维的输入有效，格式为“NCHW”或“NHWC”。默认值为 None，表示所有输入的 layout 都为 NHWC。

示例代码：

```
ret = rknn.init_runtime(target=platform,
device_id='515e9b401c060c0b')

# Preprocess
image_src = cv2.imread(IMG_PATH)
img = preprocess(image_src)

# Inference
outputs = rknn.inference(inputs=[img])

# Postprocess
outputs = postprocess(outputs)
```

注意事项：

1. 在 RV1103 / RV1106 平台上推理时若遇到"E RKNN: failed to allocate fd, ret: -1, errno: 12" 报错，可以在开发板终端运行 `RkLunch-stop.sh`，关闭其他占用内存的应用后再连板推理。

3.2.2 模型精度分析

若量化模型推理结果有问题，可以使用 `rknn.accuracy_analysis()` 接口进行精度分析，查看每层算子的精度。

精度分析 `rknn.accuracy_analysis()` 接口参数如下：

- **inputs**: 输入文件路径列表（格式包括 jpg、png、bmp 和 npy）。
- **output_dir**: 结果保存目录，默认值为 './snapshot'。
- **target**: 目标硬件平台。默认值为 None，使用仿真器进行精度分析。如果设置了 **target**（RK3562 / RK3566 / RK3568 / RK3588 / RK3576 / RV1103 / RV1106），则会获取运行时每一层的结果，并进行精度分析。
- **device_id**: 设备编号。默认值为 None。若有设置 **target** 则选择唯一一台设备进行精度分析。如果电脑连接多台设备时，需要指定填入相应的设备 ID。若通过网络 adb 连接设备进行精度分析，则需要用户手动执行命令 `adb connect [IP]` 连接网络设备后再填入设备编号，

通常为 [IP] 或 [IP]:[Port] 的形式。

注意事项:

1. 精度分析时可能会因为模型太大且开发板上存储容量不够导致运行失败，可在开发板终端上使用 `df -h` 命令来确认存储容量，若空间不足请删除无用文件保证 `output_dir` 对应分区有足够的存储空间用于保存结果文件。
2. 通过 `rknn.load_rknn()` 加载 RKNN 模型后，因 RKNN 模型缺失原始模型信息，因此无法使用模型精度分析功能。

示例代码:

```
ret = rknn.accuracy_analysis(inputs=[img_path], target=platform,
device_id='515e9b401c060c0b')
```

精度分析结果如下图

```
# simulator_error: calculate the output error of each layer of the simulator (compared to the 'golden' value).
#               entire: output error of each layer between 'golden' and 'simulator', these errors will accumulate layer by layer.
#               single: single-layer output error between 'golden' and 'simulator', can better reflect the single-layer accuracy of the simulator.
# runtime_error: calculate the output error of each layer of the runtime.
#               entire: output error of each layer between 'golden' and 'runtime', these errors will accumulate layer by layer.
#               single_sim: single-layer output error between 'simulator' and 'runtime', can better reflect the single-layer accuracy of runtime.
```

layer_name	simulator_error				runtime_error			
	entire		single		entire		single_sim	
	cos	euc	cos	euc	cos	euc	cos	euc
[Input] images	1.00000	0.0	1.00000	0.0				
[exDataConvert] images_int8	1.00000	2.5780	1.00000	2.5780				
[Conv] 128	1.00000	2.5780	1.00000	2.5780	1.00000	2.5780	1.00000	0.0
[Conv] 286								
[Relu] 131	0.99977	37.674	0.99977	37.674	0.99977	37.682	1.00000	2.0193
[Conv] 132								
[Relu] 133	0.99949	53.789	0.99960	47.679	0.99949	53.790	1.00000	3.2487
....								
[Sigmoid] 283 int8	0.99903	3.6972	0.99995	0.8721				
[exDataConvert] 283	0.99903	3.6972	0.99996	0.7372	0.99908	3.5866	1.00000	0.1702
[Conv] 280								
[Sigmoid] output_int8	0.99934	6.3987	0.99995	1.7267				
[exDataConvert] output	0.99934	6.3987	0.99996	1.4713	0.99932	6.4315	1.00000	0.3788

图 3-2 精度分析结果

分为 4 列精度情况，说明如下:

simulator_error	entire	从头到当前层 simulator 结果与 golden 结果对比的余弦距离和欧氏距离。
	single	当前层 golden 输入时，simulator 结果与 golden 结果对比的余弦距离和欧氏距离。
runtime_error	entire	从头到当前层板端实际结果与 golden 结果对比的余弦距离和欧氏距离。
	single_sim	当前层 golden 输入时，板端当前层实际结果与 simulator 结果对比的余弦距离和欧氏距离。

3.2.3 模型性能评估

接口 `rknn.eval_perf()` 会输出当前的硬件频并获取模型的性能评估结果，`fix_freq` 参数表示是否需要尝试对硬件（包括 CPU/NPU/DDR）定频，如果要对硬件定频设置成 `True`，否则设置成 `False`，默认值为 `True`。若初始化时 `rknn.init_runtime()` 的 `perf_debug` 参数设置为 `True`，将输出每一层的耗时情况和总耗时情况，若为 `False` 则只输出总耗时情况。

注意：

1. 平台 RV1103 / RV1106 不支持 `perf_debug` 为 `True` 模式，只能输出模型总耗时情况。

示例代码：

```
ret = rknn.init_runtime(target=platform, perf_debug=True)
perf_detail = rknn.eval_perf()
```

以 RK3588 为例，执行 `eval_perf` 后输出的硬件频率如下（不同固件支持的频率可能会有所差异），其中，由于 RK3588 CPU 是大小核架构，CPU 频率栏会输出多个频率值，单位为 KHz，而 NPU 和 DDR 频率单位都是 Hz。

CPU Current Frequency List:

- 1800000
- 2256000
- 2304000

NPU Current Frequency List:

- 1000000000

DDR Current Frequency List:

- 2112000000

性能评估结果如下：

Network Layer Information Table													
ID	OpType	DataType	Target	InputShape	OutputShape	OPCycles	NPUCycles	MaxCycles	Time(us)	MaxAggr(%)	WorkLoad(0/1/2)-ImproveThermal	TaskNumber	FullName
1	InputOperator	UINT8	CPU	\	(1,3,224,224)	0	0	0	22	\	0.00/0.00/0.00-Up:0.0%	0/0	InputOperator:data
2	ConvRelu	UINT8	NPU	(1,3,224,224),(32,3,3,3),(32)	(1,32,112,112)	94156	112896	112896	428	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	3/0	Conv:conv1
3	ConvRelu	INT8	NPU	(1,32,112,112),(32,32,1,1),(32)	(1,32,112,112)	135965	50176	135965	351	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	2/0	Conv:conv2_1/expand
4	ConvRelu	INT8	NPU	(1,32,112,112),(1,32,3,3),(32)	(1,32,112,112)	135884	113096	135884	383	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	2/0	Conv:conv2_1/dwise
5	Conv	INT8	NPU	(1,32,112,112),(16,32,1,1),(16)	(1,16,112,112)	101942	50176	101942	219	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	2/0	Conv:conv2_1/linear
52	ConvRelu	INT8	NPU	(1,960,7,7),(1,960,3,3),(960)	(1,960,7,7)	18668	8640	18668	268	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	1/0	Conv:conv6_3/dwise
53	Conv	INT8	NPU	(1,960,7,7),(320,960,1,1),(320)	(1,320,7,7)	62983	19200	62983	295	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	1/0	Conv:conv6_3/linear
54	ConvRelu	INT8	NPU	(1,320,7,7),(1280,320,1,1),(1280)	(1,1280,7,7)	84248	25600	84248	322	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	1/0	Conv:conv6_4
55	Conv	INT8	NPU	(1,1280,7,7),(1,1280,7,7),(1280)	(1,1280,1,1)	23159	15680	23159	281	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	1/0	Conv:pool6
56	Conv	INT8	NPU	(1,1280,1,1),(1000,1280,1,1),(1000)	(1,1000,1,1)	218208	20480	218208	418	0.00/0.00/0.00	300.0%/0.0%/0.0%-Up:-200.0%	1/0	Conv:fc7
57	exDataConvert	INT8	NPU	(1,1000,1,1)	(1,1000,1,1)	509	0	509	257	\	300.0%/0.0%/0.0%-Up:-200.0%	2/0	exDataConvert:fc7_conv
58	exSoftmax13	FLOAT16	NPU	(1,1000,1,1),(1,1000,1,1)	(1,1000,1,1)	1012	0	1012	287	\	300.0%/0.0%/0.0%-Up:-200.0%	7/1	exSoftmax13:prob
59	OutputOperator	FLOAT16	CPU	(1,1000,1,1)	\	0	0	0	67	\	300.0%/0.0%/0.0%-Up:-200.0%	2/0	OutputOperator:prob
Total Operator Elapsed Per Frame Time(us): 14374													
Total Memory Read/Write Per Frame Size(KB): 12872.1													
Operator Time Consuming Ranking Table													
OpType	CallNumber	CPUTime(us)	GPTime(us)	NPUTime(us)	TotalTime(us)	TimeRatio(%)							
ConvRelu	36	0	0	9319	9319	64.83%							
Conv	9	0	0	2238	2238	15.57%							
ConvAdd	18	0	0	2184	2184	15.19%							
exSoftmax13	1	0	0	287	287	2.00%							
exDataConvert	1	0	0	257	257	1.79%							
OutputOperator	1	67	0	0	67	0.47%							
InputOperator	1	22	0	0	22	0.15%							

图 3-3 性能评估结果

部分参数说明如下：

参数	描述
ID	算子编号
OpType	算子类型
DataType	输入的数据类型
Target	算子运行的硬件（CPU/NPU/GPU）
InputShape	输入形状
OutputShape	输出形状
DDRCycles	DDR 读写时钟周期数
NPUCycles	NPU 计算时钟周期数
MaxCycles	DDR Cycles 和 NPU Cycles 的最大值
Time(us)	算子计算耗时（us）
MacUsage(%)	MAC 使用率
WorkLoad(0/1/2)	0/1/2 核负载情况（仅 RK3588 平台）
WorkLoad(0/1)	0/1 核负载情况（仅 RK3576 平台）
TaskNumber	NPU 任务数
RW(KB)	读写的数据总量（KB）
FullName	算子全名
Total Operator Elapsed Per Frame Time(us)	模型推理的单帧总耗时
Total Memory Read/Write Per Frame Size(KB)	模型推理的单帧总带宽消耗
CallNumber	单帧内该算子运行次数
CPUTime(us)	单帧内该算子在 CPU 上的总耗时
GPUPTime(us)	单帧内该算子在 GPU 上的总耗时
NPUTime(us)	单帧内该算子在 NPU 上的总耗时

TotalTime(us)	单帧内该算子的总耗时
TimeRatio(%)	单帧内该算子的总耗时与单帧总耗时的比值

3.2.4 模型内存评估

接口 `rknn.eval_memory()` 可以获取模型的内存消耗评估结果。初始化时 `rknn.init_runtime()` 的 `eval_mem` 参数设置为 `True`，将输出各部分内存消耗情况。

示例代码：

```
ret = rknn.init_runtime(target=platform, eval_mem=True)
memory_detail = rknn.eval_memory()
```

内存评估结果如下：

```
=====
Memory Profile Info Dump
=====
NPU model memory detail(bytes):
Weight Memory: 8.67 MiB
Internal Tensor Memory: 7.42 MiB
Other Memory: 3.03 MiB
Total Memory: 19.12 MiB

INFO: When evaluating memory usage, we need consider
the size of model, current model size is: 11.86 MiB
=====
```

部分参数说明如下：

Total Weight Memory	模型中权重的内存占用（MB）
Total Internal Tensor Memory	模型中间 tensor 内存占用（MB）
Other Memory	其他内存占用（例如寄存器配置、输入输出 tensor） （MB）
Total Memory	模型的内存总占用（MB）

3.3 板端 C 推理

此章节介绍通用 API 接口的调用流程。零拷贝调用流程请参考[章节 5.2](#)。

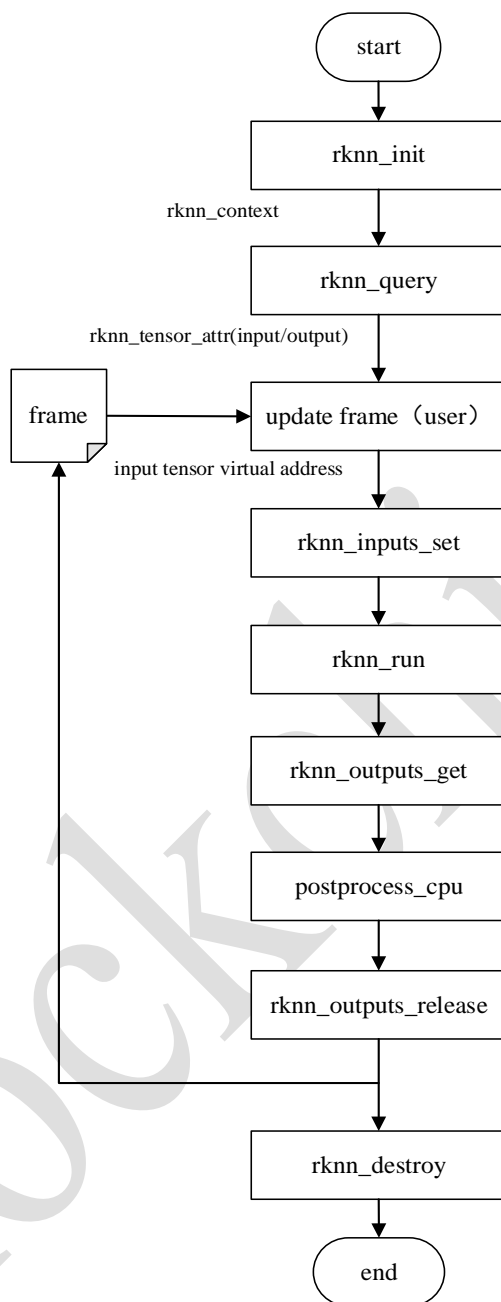


图 3-4 通用 API 调用流程

RKNN 通用 API 接口调用流程：

1. rknn_init()初始化模型；
2. rknn_query()查询模型的输入输出属性；
3. 对输入进行前处理；
4. rknn_inputs_set()设置输入数据；
5. rknn_run()进行模型推理；
6. rknn_outputs_get()获取推理结果数据；

7. 对输出进行后处理;
8. rknn_outputs_release()释放输出数据内存;
9. rknn_destroy()销毁 RKNN;
10. 通用 API 调用流程如图 3-4 所示, 黄色字体流程表示用户行为可循环输入数据。

通用 API 调用流程示例代码:

```
// Init RKNN model
ret = rknn_init(&ctx, model, model_len, 0, NULL);

// Get Model Input Output Number
rknn_input_output_num io_num;
ret = rknn_query(ctx, RKNN_QUERY_IN_OUT_NUM, &io_num,
sizeof(io_num));

// Get Model Input Info
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++)
{
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_ATTR,
&(input_attrs[i]), sizeof(rknn_tensor_attr));
}

// Get Model Output Info
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++)
{
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_OUTPUT_ATTR,
&(output_attrs[i]), sizeof(rknn_tensor_attr));
}

rknn_input inputs[io_num.n_input];
rknn_output outputs[io_num.n_output];
memset(inputs, 0, sizeof(inputs));
memset(outputs, 0, sizeof(outputs));

// Pre-process
// Read Image
image_buffer_t src_image;
memset(&src_image, 0, sizeof(image_buffer_t));
ret = read_image(image_path, &src_image);

// Set Input Data
inputs[0].index = 0;
inputs[0].type = RKNN_TENSOR_UINT8;
inputs[0].fmt = RKNN_TENSOR_NHWC;
inputs[0].size = src_image.size;
inputs[0].buf = src_image.virt_addr;

ret = rknn_inputs_set(rknn_ctx, io_num.n_input, inputs);
```

```
// Run
ret = rknn_run(rknn_ctx, nullptr);

// Get Output Data
ret = rknn_outputs_get(rknn_ctx, io_num.n_output, outputs, NULL);

// Post-process
post_process(outputs, results);

// Release RKNN Output
rknn_outputs_release(rknn_ctx, io_num.n_output, outputs);

if (rknn_ctx != 0)
{
    rknn_destroy(rknn_ctx);
}
```

3.4 板端 Python 推理

RKNN-Toolkit Lite2 为用户提供板端模型推理的 Python 接口，方便用户使用 Python 语言进行 AI 应用开发。

注：在使用 RKNN-Toolkit Lite2 开发 AI 应用前，需要通过 RKNN-Toolkit2 将各深度学习框架导出的模型转成 RKNN 模型。模型转换详细方法请参考 [3.1](#) 章节。

3.4.1 系统依赖说明

使用 RKNN-Toolkit Lite2 需满足以下运行环境要求：

表 3-2 RKNN-Toolkit Lite2 运行环境

操作系统版本	Debian 10 / 11 (aarch64)
Python 版本	3.7 / 3.8 / 3.9 / 3.10 / 3.11
Python 依赖库	'numpy'、'ruamel.yaml'、'psutils'

3.4.2 工具安装

请通过 `pip3 install` 命令安装 RKNN-Toolkit Lite2。安装步骤如下：

- 如果系统中没有安装 `python3/pip3` 等程序，请先通过 `apt-get` 方式安装，参考命令如下：

```
sudo apt-get update
sudo apt-get install -y python3 python3-dev python3-pip gcc
```

注：安装部分依赖模块时，需要编译源码，此时将用到 `python3-dev` 和 `gcc`，因此该步骤将这两个包也一起安装，避免后面安装依赖模块时编译失败。

- 安装依赖模块：opencv-python 和 numpy，参考命令如下：

```
sudo apt-get install -y python3-opencv  
sudo apt-get install -y python3-numpy
```

注：

1. RKNN-Toolkit Lite2 本身并不依赖 opencv-python，但是在示例中需要使用该模块对图像进行处理。
2. 在 Debian10 固件上通过 pip3 安装 numpy 可能失败，建议用上述方法安装。

- 安装 RKNN-Toolkit Lite2

各平台的安装包都放在 SDK 的 rknn-toolkit-lite2/packages 文件夹下。进入 packages 文件夹，执行以下命令安装 RKNN-Toolkit Lite2：

```
# Python 3.7  
pip3 install rknn_toolkit_lite2-x.y.z-cp37-cp37m-  
linux_aarch64.whl  
# Python 3.8  
pip3 install rknn_toolkit_lite2-x.y.z-cp38-cp38-linux_aarch64.whl  
# Python 3.9  
pip3 install rknn_toolkit_lite2-x.y.z-cp39-cp39-linux_aarch64.whl  
# Python 3.10  
pip3 install rknn_toolkit_lite2-x.y.z-cp310-cp310-  
linux_aarch64.whl  
# Python 3.11  
pip3 install rknn_toolkit_lite2-x.y.z-cp311-cp311-  
linux_aarch64.whl
```

3.4.3 基本使用流程

使用 RKNN-Toolkit Lite2 部署 RKNN 模型的基本流程如下图所示：

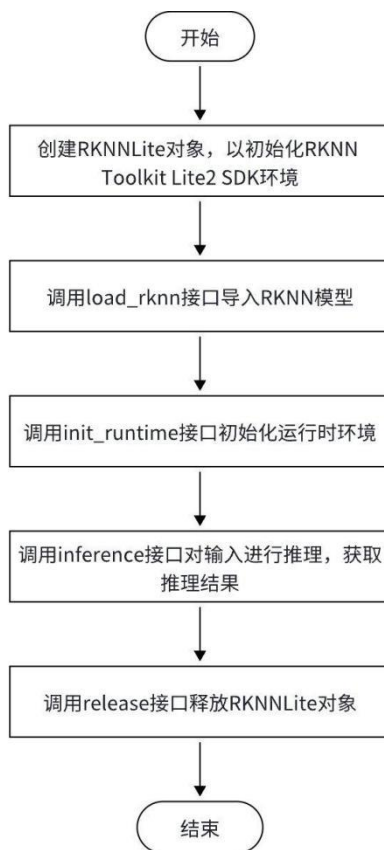


图 3-5 RKNN-Toolkit Lite2 基本使用流程

注：

1. 在调用 inference 接口进行推理之前，需要获取输入数据，并做相应的预处理，然后根据输入信息设置 inference 接口中的各项参数。
2. 在调用 inference 接口后，通常需要对推理结果进行相应的处理，以完成上层应用相关功能。

3.4.4 运行参考示例

在 SDK/rknn-toolkit-lite2/examples 目录，提供了一个基于 RKNN-Toolkit Lite2 开发的图像分类应用，该应用调用 RKNN-Toolkit Lite2 的接口加载 Resnet18 RKNN 模型，对输入图片进行预测，打印 top5 分类结果。

运行该示例的方法：

1. 准备一块安装有 RKNN-Toolkit Lite2 的开发板；
2. 将 SDK/rknn-toolkit-lite2/examples 目录推到开发板上；

3. 在开发板上进入 `examples/resnet18` 目录，执行如下命令运行该示例：

```
python test.py
```

参考运行结果如下所示：

```
model: resnet18
-----TOP 5-----
[812]: 0.999676 [class: space shuttle]
[404]: 0.000249 [class: airliner]
[657]: 0.000014 [class: missile]
[833]: 0.000009 [class: bullet train, bullet | submarine,
pigboat, sub, U-boat]
[466]: 0.000009 [class: bullet train, bullet | submarine,
pigboat, sub, U-boat]
```

3.4.5 RKNN-Toolkit Lite2 API 详细说明

本章节将详细说明 RKNN-Toolkit Lite2 提供的所有 API 的用法。

3.4.5.1 RKNNLite 初始化及对象释放

在使用 RKNN-Toolkit Lite2 时，需要先调用 `RKNNLite()` 方法初始化一个 `RKNNLite` 对象，并在用完后调用该对象的 `rknn_lite.release()` 方法将资源释放掉。

初始化 `RKNNLite` 对象时，可以设置 `verbose` 和 `verbose_file` 参数，以打印详细的日志信息。其中 `verbose` 参数指定是否要在屏幕上打印详细日志信息；如果设置了 `verbose_file` 参数，且 `verbose` 参数值为 `True`，日志信息还将写到这个参数指定的文件中。

举例如下：

```
# 将详细的日志信息输出到屏幕，并写到 inference.log 文件中
rknn_lite = RKNNLite(verbose=True,
verbose_file='./inference.log')
# 只在屏幕打印详细的日志信息
rknn_lite = RKNNLite(verbose=True)
...
rknn_lite.release()
```

3.4.5.2 加载 RKNN 模型

API	load_rknn
描述	加载 RKNN 模型
参数	Path: RKNN 模型路径
返回值	0: 加载成功；-1: 加载失败。

举例如下：

```
# 从当前目录加载 resnet_18.rknn 模型
ret = rknn_lite.load_rknn('./resnet_18.rknn')
```

3.4.5.3 初始化运行时环境

在模型推理之前，必须先初始化运行时环境。

API	init_runtime
描述	初始化运行时环境。
参数	<p>core_mask: NPU 工作核心配置模式。可选值如下：</p> <p>RKNNLite.NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的 NPU 核上。</p> <p>RKNNLite.NPU_CORE_0: 模型运行在 NPU Core0 上。</p> <p>RKNNLite.NPU_CORE_1: 模型运行在 NPU Core1 上。</p> <p>RKNNLite.NPU_CORE_2: 模型运行在 NPU Core2 上。</p> <p>RKNNLite.NPU_CORE_0_1: 模型同时运行在 NPU Core0 和 NPU Core1 上。</p> <p>RKNNLite.NPU_CORE_0_1_2: 模型同时运行在 NPU Core0, NPU Core1 和 NPU Core2 上。</p> <p>RKNNLite.NPU_CORE_ALL: 模型同时运行在所有 NPU 核上。</p> <p>默认值为 NPU_CORE_AUTO，即默认使用的是自动调度模式。</p> <p>注：该参数对 RK3588/RK3576 有效。</p>
返回值	0: 初始化运行时环境成功；-1: 初始化运行时环境失败。

举例如下：

```
# 初始化运行时环境
ret = rknn_lite.init_runtime(core_mask=RKNNLite.NPU_CORE_AUTO)
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
```

3.4.5.4 模型推理

API	inference
描述	对指定输入进行推理，返回推理结果。
参数	<p>inputs: 输入数据，如 OpenCV 读取的图片（如果输入是四维的，需要手动扩成 4 维）。类型是 list，列表成员是 ndarray。</p> <p>data_format: 数据排列方式，类型是 list，对于每个输入可选值"nhwc", "nchw", 只对四维输入有效。默认值为 None。如果不填写该参数，对于 4 维输入，数据 buffer 应按照 NHWC 排列，对于非 4 维输入，数据 buffer 应按照模型输入要求的格式排列。如果要填写该参数，对于非 4 维输入，数据 buffer 应按照模型输入要求的格式排列（不管填"nhwc"还是"nchw"）；对于 4 维输入，数据 buffer 应按照该参数设置的格式排列；对于多输入模型，填写该参数时要包括所有输入。</p>
返回值	results: 推理结果，类型是 list，列表成员是 ndarray。

以分类模型为例，模型推理代码参考如下：

```
# Get input data
img = cv2.imread('./space_shuttle_224.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = np.expand_dims(img, 0)
# Inference
outputs = rknn_lite.inference(inputs=[img])
# Show the classification results
show_top5(outputs)
```

3.4.5.5 查询 SDK 版本

API	get_sdk_version
描述	<p>获取 Runtime，驱动和 RKNN 模型版本信息。</p> <p>注：使用该接口前必须完成模型加载和初始化运行环境。</p>
参数	无
返回值	sdk_version: runtime，驱动版本信息。类型为字符串。

举例如下：

```
# 获取 SDK 版本信息
sdk_version = rknn_lite.get_sdk_version()
```

返回的 SDK 信息参考如下：

```
=====
RKNN VERSION:
  API: 1.5.2 (71720f3fc@2023-08-21T09:29:52)
  DRV: 0.7.2
=====
```

3.4.5.6 查询模型可运行平台

API	list_support_target_platform
描述	查询给定 RKNN 模型可运行的芯片平台。
参数	rknn_model: RKNN 模型路径。如果不指定模型路径，则按类别打印 RKNN-Toolkit Lite2 当前支持的芯片平台。
返回值	support_target_platform: 返回模型可运行的芯片平台。如果 RKNN 模型路径为空或不存在，返回 None。

参考代码如下：

```
rknn_lite.list_support_target_platform(rknn_model='mobilenet_v1.rknn')
```

参考结果如下：

```
*****
Target platforms filled in RKNN model:      ['rk3566']
Target platforms supported by this RKNN model: ['RK3566',
'RK3568']
*****
```

3.5 矩阵乘法接口

Matmul API 是 Runtime 提供的一套单独的 C API，用于在 NPU 上运行矩阵乘法运算，**RV1103/RV1106 暂不支持**。矩阵乘法是线性代数中的一种重要操作，该操作定义如下： $C=AB$ 。其中，A 是一个 $M \times K$ 矩阵，B 是一个 $K \times N$ 矩阵，C 是一个 $M \times N$ 矩阵。

3.5.1 主要用途和特点

Matmul API 多用于深度学习中的参数计算任务，例如，在当前被广泛应用的 Transformer 模型结构的主要模块（自注意力机制和前馈神经网络层）中都大量使用了矩阵

乘法进行计算，因此矩阵乘法的运算效率对 Transformer 模型的整体性能至关重要。Matmul API 具有以下特点：

- **高效：**底层使用 RKNPU 实现，具有高性能低功耗的特点。
- **灵活：**无需加载 RKNN 模型，支持 int8 和 float16 两种边缘端计算常用的数据类型，提供单独的内存分配接口或使用外部内存的机制，用户可管理和复用矩阵的输入输出内存。

3.5.2 Matmul API 使用流程

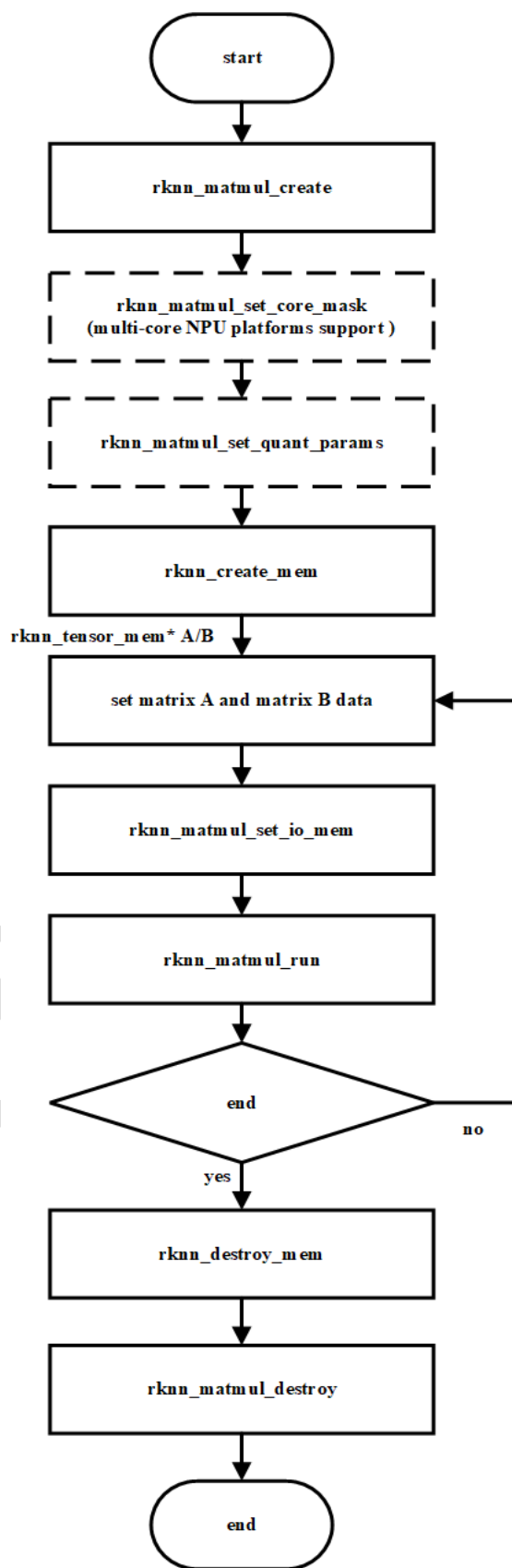


图 3-6 Matmul API 基础调用流程

首先，Matmul API 的结构体和接口位于 `rknn_matmul_api.h` 头文件，开发者程序编译时需要包含该头文件。Matmul API 基础用法的流程如上图所示。

使用 Matmul API 通常包括以下步骤：

1. 创建上下文：设置 `rknn_matmul_info` 结构体，包含 M、K、N、输入和输出矩阵的数据类型、输入和输出矩阵使用的数据排列方式等信息，然后，调用 `rknn_matmul_create` 接口初始化上下文。在初始化后，获取以 `rknn_matmul_io_attr` 结构体指针，它包含了输入和输出矩阵 `tensor` 信息。
2. 指定运行 NPU 核（仅多核 NPU 的芯片平台有效）：调用 `rknn_matmul_set_core_mask`，设置掩码来指定某一个 NPU 核做运算，多核 NPU 的芯片平台不调用该接口的默认行为是自动选择空闲的核心。
3. 设置矩阵 A、B 和 C 的量化参数：调用 `rknn_matmul_set_quant_params` 传入包含对应矩阵量化参数值的 `rknn_quant_params` 结构体，不调用该接口的默认行为是各个矩阵的 `scale=1.0, zero_point=0`。
4. 创建输入和输出内存：调用 `rknn_create_mem` 接口，根据输入和输出矩阵 `Tensor` 信息中的大小创建内存。
5. 填充输入数据：根据形状和数据类型填充输入矩阵 A 和 B 的数据。
6. 设置输入和输出内存：调用 `rknn_matmul_set_io_mem` 将填充好数据的输入矩阵记录到上下文中，输出内存也同样记录到上下文中。除了记录内存地址外，该接口还会涉及对数据做重新排列，必须在填充或更新输入数据后调用，与零拷贝 API 中的 `rknn_set_io_mem` 接口行为有区别。
7. 执行矩阵乘法运算：设置好输入和输出内存后，调用 `rknn_matmul_run` 执行矩阵乘法运算。
8. 处理输出：执行矩阵乘法运算后，从输出内存中读取结果。
9. 销毁资源：执行结束后，调用 `rknn_destroy_mem` 和 `rknn_matmul_destroy` 分别销毁内存和上下文资源。

3.5.3 矩阵乘法高级用法

在创建上下文时，要求用户设置 `rknn_matmul_info` 结构体，`rknn_matmul_info` 表示用于执行矩阵乘法的规格信息，它包含了矩阵乘法的规模、输入和输出矩阵的数据类型和数据排列。其中，`B_layout` 和 `AC_layout` 用于设置高性能的数据排列方式。具体的结构体定义

如下表所示：

表 3-3 rknn_matmul_info 结构体定义

成员变量	数据类型	含义
M	int32_t	A 矩阵的行数
K	int32_t	A 矩阵的列数
N	int32_t	B 矩阵的列数
type	rknn_matmul_type	<p>输入输出矩阵的数据类型：</p> <p>RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT32： 表示矩阵 A 和 B 是 float16 类型，矩阵 C 是 float32 类型；</p> <p>RKNN_INT8_MM_INT8_TO_INT32：表示矩阵 A 和 B 是 int8 类型，矩阵 C 是 int32 类型；</p> <p>RKNN_INT8_MM_INT8_TO_INT8：表示矩阵 A、B 和 C 是 int8 类型；</p> <p>RKNN_FLOAT16_MM_FLOAT16_TO_FLOAT16： 表示矩阵 A、B 和 C 是 float16 类型；</p> <p>RKNN_FLOAT16_MM_INT8_TO_FLOAT32：表示矩阵 A 是 float16 类型，矩阵 B 是 int8 类型，矩阵 C 是 float32 类型；</p> <p>RKNN_FLOAT16_MM_INT8_TO_FLOAT16：表示矩阵 A 是 float16 类型，矩阵 B 是 int8 类型，矩阵 C 是 float16 类型；</p> <p>RKNN_FLOAT16_MM_INT4_TO_FLOAT32：表示矩阵 A 是 float16 类型，矩阵 B 是 int4 类型，矩阵 C 是 float32 类型；</p> <p>RKNN_FLOAT16_MM_INT4_TO_FLOAT16：表示矩阵 A 是 float16 类型，矩阵 B 是 int4 类型，矩阵 C 是 float16 类型；</p> <p>RKNN_INT4_MM_INT4_TO_INT16：表示矩阵 A</p>

		和 B 是 int4 类型，矩阵 C 是 int16 类型。 RKNN_INT8_MM_INT4_TO_INT32 ：表示矩阵 A 和 B 是 int4 类型，矩阵 C 是 int32 类型。
B_layout	int16_t	指定矩阵 B 的数据排列方式。 0：表示矩阵 B 按照原始形状排列 1：表示矩阵 B 按照高性能形状排列
B_quant_type	int16_t	指定矩阵 B 的量化方式类型。 0：表示矩阵 B 按照 Per-Layer 方式量化 1：表示矩阵 B 按照 Per-Channel 方式量化
AC_layout	int16_t	指定矩阵 A 和矩阵 C 的数据排列方式。 0：表示矩阵 A 和 C 按照原始形状排列 1：表示矩阵 A 和 C 按照高性能形状排列
AC_quant_type	int16_t	指定矩阵 A 和 C 的量化方式类型。 0：表示矩阵 A 和 C 按照 Per-Layer 方式量化 1：表示矩阵 A 和 C 按照 Per-Channel 方式量化
iommu_domain_id	int32_t	矩阵上下文所在的 IOMMU 地址空间域的索引。 IOMMU 地址空间与上下文一一对应，每个 IOMMU 地址空间大小为 4GB。该参数主要用于矩阵 A、B 和 C 的参数规格较大，某个域内 NPU 分配的内存超过 4GB 以后需切换另一个域时使用。
reserved	int8_t[]	预留字段

其中，矩阵 A 的原始形状是 MxK，矩阵 B 的原始形状是 KxN，矩阵 C 的原始形状是 MxN。

3.5.3.1 指定量化参数的矩阵乘法

如果矩阵乘积的高比特数据需要根据量化参数量化成低比特数据，例如，矩阵 A 和 B 是 int8 类型，矩阵 C 是 int8 类型，需要设置矩阵的量化参数，量化参数由 scale 和 zero_point 组成，用 rknn_quant_params 结构体表示，通过 rknn_matmul_set_quant_params 接口设置矩阵 A 和 C 的量化参数，量化参数设置成功后，Per-Channel 量化方式下的矩阵 B 的量化参数可以通过 rknn_matmul_get_quant_params 接口读取。rknn_quant_params 数据结构定义如下表所示：

表 3-4 rknn_quant_params 结构体定义

成员变量	数据类型	含义
name	char[]	矩阵的名称
scale	float*	量化参数 scale 数组的首地址
scale_len	int32_t	量化参数 scale 数组的长度
zp	int32_t*	量化参数 zero_point 数组的首地址
zp_len	int32_t	量化参数 zero_point 数组的长度

3.5.3.2 动态 shape 输入的矩阵乘法

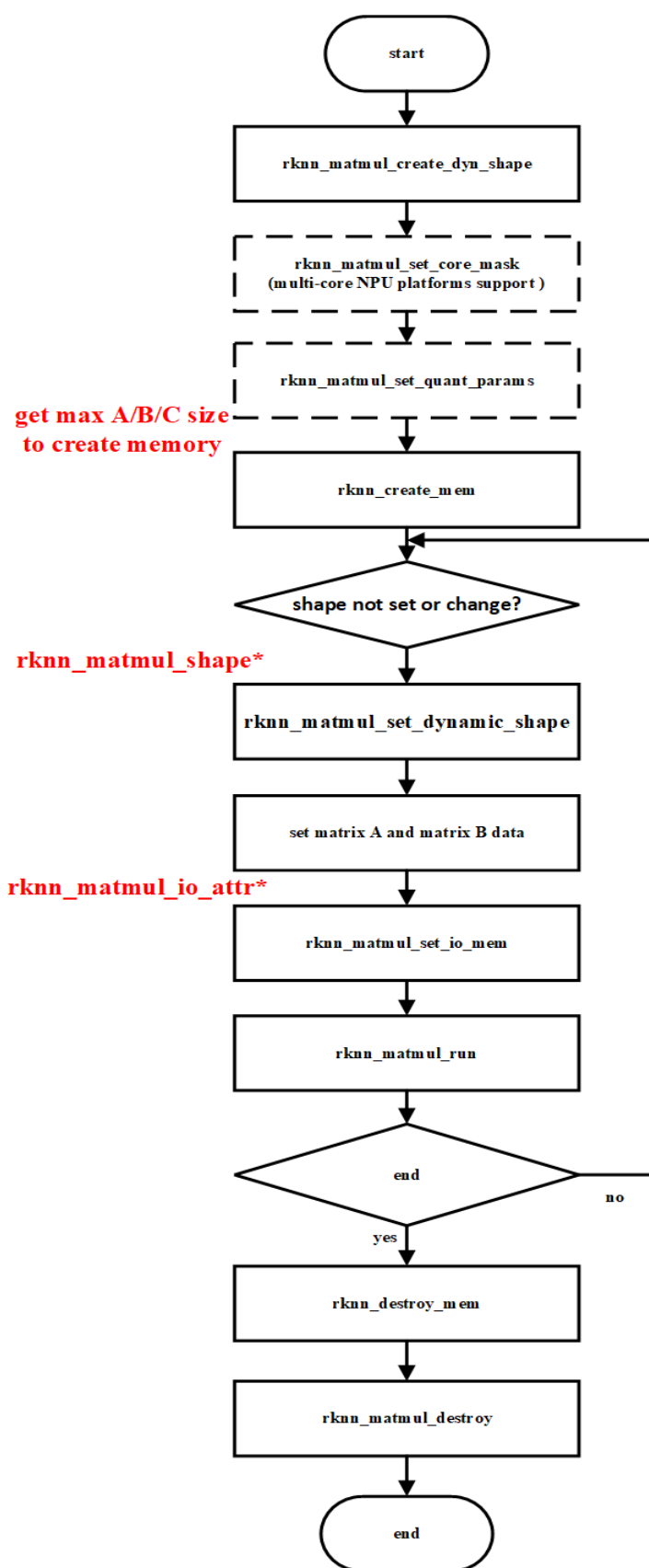


图 3-7 动态 shape 输入的 Matmul API 调用流程

动态 shape 输入的 Matmul API 用法的流程如上图所示。使用动态 shape Matmul API 与基础的 Matmul API 的流程差异如下：

1. 创建上下文：配置 shape 数量和所需的 shape，设置 rknn_matmul_info 结构体，包含输入和输出矩阵的数据类型、输入和输出矩阵使用的数据排列方式等信息。**注意，动态 shape Matmul 接口参数 rknn_matmul_info 中的 M、K、N 不需要设置。**然后，调用 rknn_matmul_set_dynamic_shape 接口初始化上下文。在初始化后，获取 rknn_matmul_io_attr 结构体数组，它包含了所有 shape 配置下的输入和输出矩阵 Tensor 信息。
2. 创建输入和输出内存：首先从 rknn_matmul_io_attr 结构体数组获取最大的 Tensor 大小，调用 rknn_create_mem 接口，使用最大的输入和输出矩阵 Tensor 大小创建内存。
3. 设置输入 shape：通过调用 rknn_matmul_set_dynamic_shape 接口，传入 rknn_matmul_shape 指针，设置当前推理过程使用的 shape。
4. 设置输入和输出内存：调用 rknn_matmul_set_io_mem 将填充好数据的输入矩阵记录到上下文中，输出内存也同样记录到上下文中。传入的 rknn_tensor_mem* 参数是前面创建的最大 size 的输入和输出内存，rknn_matmul_tensor_attr* 参数是当前 shape 对应的矩阵属性信息。

3.5.4 高性能的数据排列方式

由于 NPU 是专用的硬件架构，读取 $M \times K$ 和 $K \times N$ 这种原始形状的数据不是最高效的，同样的，写入 $M \times N$ 形状的 C 矩阵也不是最高效的，用户使用特殊的数据排列方式可以实现更高的性能。**AC_layout 参数控制矩阵 A 和 C 是否使用高性能数据排列，B_layout 参数控制矩阵 B 是否使用高性能数据排列。**

假设矩阵 A 的原始形状是 $M \times K$ ，矩阵 B 的原始形状是 $K \times N$ ，矩阵 C 的原始形状是 $M \times N$ ，要求的数据排列方式如下：

1. 若 AC_layout=0 且 B_layout=0，矩阵 A 的形状为 $[M, K]$ ，矩阵 B 的形状为 $[K, N]$ ，矩阵 C 的形状为 $[M, N]$ 。
2. 若 AC_layout=1 且 B_layout=1，不同芯片平台和数据类型下矩阵 A、B 和 C 的高性能数据排列如下表所示（表中除法结果都是上取整，多出部分用 0 补齐）：

表 3-5 各个芯片平台矩阵 A、B 和 C 的高性能形状

	RK3562	RK3566/RK3568	RK3576	RK3588
A 形状(int4)	暂不支持	暂不支持	暂不支持	[K/32,M,32]
B 形状(int4)	暂不支持	暂不支持	暂不支持	[N/64,K/32,64,32]
C 形状 (int16)	暂不支持	暂不支持	暂不支持	[N/8,M,8]
A 形状(int8)	[K/16,M,16]	[K/8,M,8]	[K/16,M,16]	[K/16,M,16]
B 形状(int8)	[N/16,K/32,16,32]	[N/16,K/32,16,32]	[N/32,K/32,32,32]	[N/32,K/32,32,32]
C 形状 (int32)	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]
A 形状 (float16)	[K/8,M,8]	[K/4,M,4]	[K/8,M,8]	[K/8,M,8]
B 形状 (float16)	[N/8,K/32,8,32]	[N/8,K/16,8,16]	[N/16,K/32,16,32]	[N/16,K/32,16,32]
C 形状 (float32)	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]	[N/4,M,4]

以 RK3566/RK3568 平台，int8 数据类型为例，矩阵 A 从[M,K]变换到[K/8,M,8]的元素对应关系如下图所示：

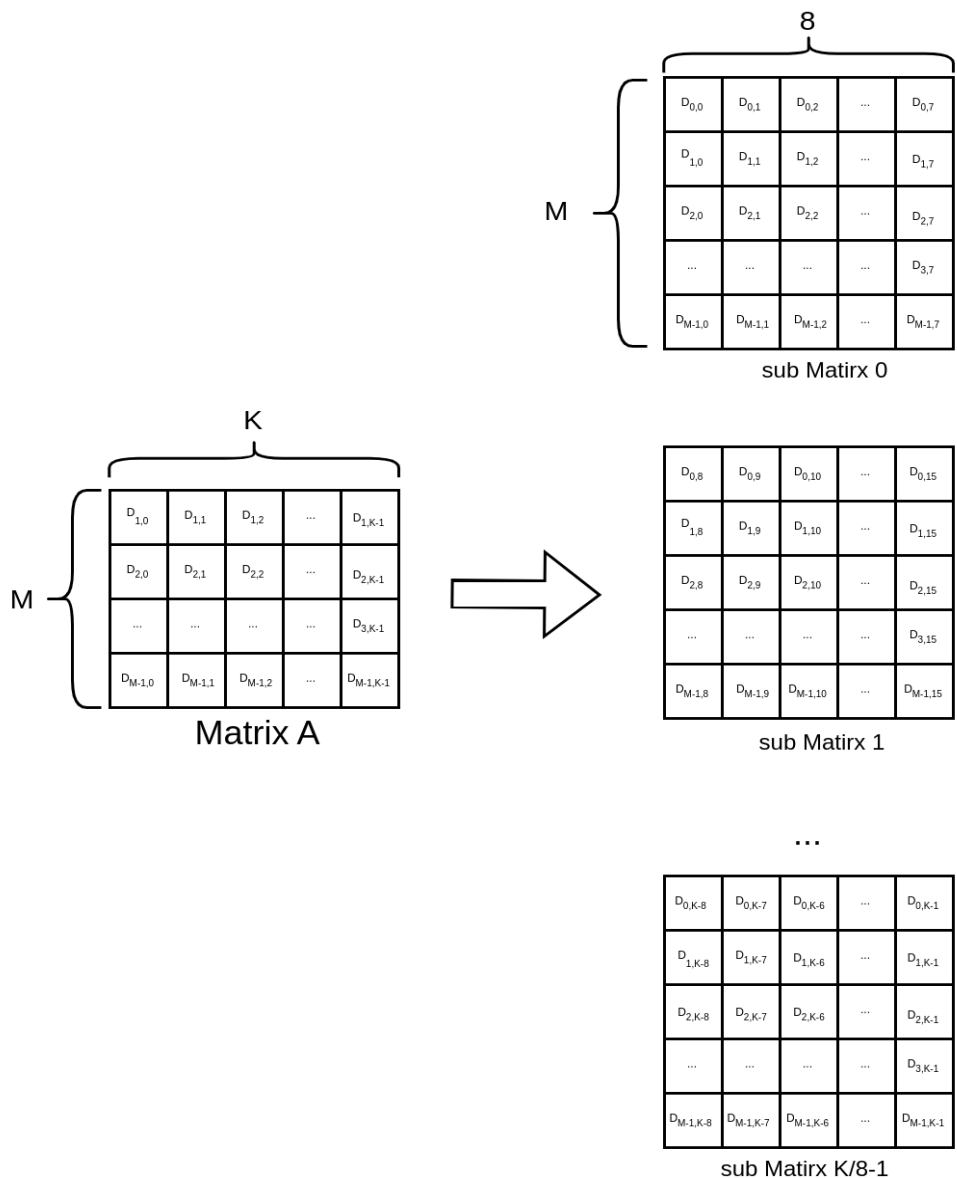


图 3-8 int8 类型矩阵 A 从[M,K]变换到[K/8,M,8]的元素对应关系图

其中，矩阵 A 第(i, j)元素对应的数据为 $D_{i,j}$ ，左图是 $\text{shape}=[M,K]$ 的矩阵，右图每个小矩阵 $\text{shape}=[M,8]$ ，从上到下一共 K/8 个。

矩阵 A 或 C 从原始形状转换成高性能形状的示例代码如下：

```
template <typename Ti, typename To>
void norm_layout_to_perf_layout(Ti *src, To *dst, int32_t M,
int32_t K, int32_t subK){
    int outter_size = (int)std::ceil(K * 1.0f / subK);
    for (int i = 0; i < outter_size; i++)
    {
        for (int m = 0; m < M; m++)
        {
            for (int j = 0; j < subK; j++)
            {
                int ki = i * subK + j;
                if (ki >= K)

```



```

    {
        dst[i * M * subK + m * subK + j] = 0;
    }
    else
    {
        dst[i * M * subK + m * subK + j] = src[m * K +
ki];
    }
}
}
}
}
}

```

以 RK3566/RK3568 平台，int8 数据类型的矩阵 B 从[K,N]变换到[N/16,K/32,16,32]的元素对应关系如下图所示：

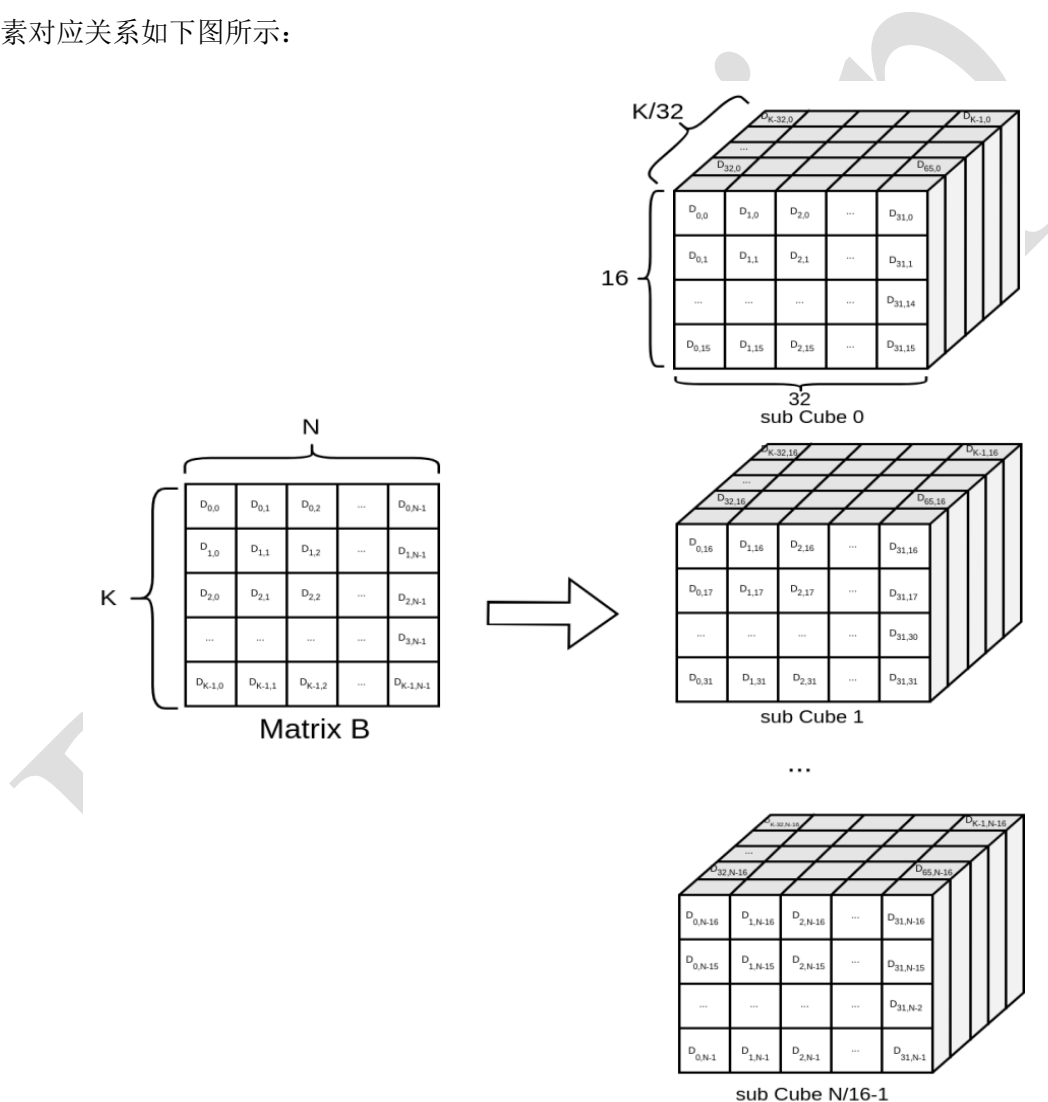


图 3-9 int8 类型矩阵 B 从[K,N]变换到[N/16,K/32,16,32]的元素对应关系图

其中，矩阵 B 第(i, j)元素对应的数据为 $D_{i,j}$ ，左图是 $\text{shape}=[K,N]$ 的矩阵，右图每个小立方体 $\text{shape}=[K/32,16,32]$ ，从上到下一共 N/16 个。

矩阵 B 从原始形状转换成高性能形状的示例代码如下：

```
template <typename Ti, typename To>
void norm_layout_to_native_layout(Ti *src, To *dst, int32_t K,
int32_t N, int32_t subN, int32_t subK)
{
    int N_remain = (int)std::ceil(N * 1.0f / subN);
    int K_remain = (int)std::ceil(K * 1.0f / subK);
    for (int i = 0; i < N_remain; i++)
    {
        for (int j = 0; j < K_remain; j++)
        {
            for (int n = 0; n < subN; n++)
            {
                int ni = i * subN + n;
                for (int k = 0; k < subK; k++)
                {
                    int ki = j * subK + k;
                    if (ki < K && ni < N)
                    {
                        dst[((i * K_remain + j) * subN + n) * subK +
k] = src[ki * N + ni];
                    }
                    else
                    {
                        dst[((i * K_remain + j) * subN + n) * subK +
k] = 0;
                    }
                }
            }
        }
    }
}
```

注意:

- AC_layout 和 B_layout 可分别设置为 0 或 1。若设置为 0，则参数控制的矩阵按照原始形状排列；若设置为 1，则参数控制的矩阵按照高性能形状排列。
- Matmul 接口输入和输出的数据位宽不同，目前仅支持的三种矩阵数据类型（int4 仅支持 RK3588）如下表：

表 3-6 Matmul 接口支持的矩阵 A、B 和 C 的数据类型

	矩阵 A	矩阵 B	矩阵 C
类型 1	int4	int4	int16
类型 2	int8	int8	int32
类型 3	float16	float16	float32

其中，float16 浮点格式遵循 IEEE-754 标准，具体格式请参考 [IEEE-754 half] (https://en.wikipedia.org/wiki/Half-precision_floating-point_format)。

- Matmul API 完整示例代码见 https://github.com/airockchip/rknn-toolkit2/tree/master/rknpu2/examples/rknn_matmul_api_demo。

3.5.4.1 矩阵规格限制

Matmul API 是基于 NPU 的硬件架构实现，受硬件规格限制。K 和 N 大小限制如下：

表 3-7 各个芯片平台 K 和 N 的大小限制

	RK3562	RK3566/RK3568	RK3576	RK3588
K 大小限制(int4)	暂不支持	暂不支持	暂不支持	32 对齐
K 大小限制(int8)	≤ 10240 且 32 对齐	≤ 10240 且 32 对齐	32 对齐	32 对齐
K 大小限制(float16)	≤ 10240 且 32 对齐	≤ 10240 且 32 对齐	32 对齐	32 对齐
N 大小限制(int4)	暂不支持	暂不支持	暂不支持	≤ 8192 且 64 对齐
N 大小限制(int8)	≤ 4096 且 16 对齐	≤ 4096 且 16 对齐	≤ 4096 且 32 对齐	≤ 4096 且 32 对齐
N 大小限制(float16)	≤ 4096 且 16 对齐	≤ 4096 且 16 对齐	≤ 4096 且 32 对齐	≤ 4096 且 32 对齐

对于 RK3588:

当 $K > 8192$, B 会被分成 T 段.

$\text{int } T = \text{std::ceil}(K / 8192);$

例如: normal layout -> native layout

$K = 20488, N = 4096, T = 3$, 数据会被分成 3 段.

$\text{subN} = \text{rknn_matmul_io_attr.B.dims}[2];$

$\text{subK} = \text{rknn_matmul_io_attr.B.dims}[3];$

	(8196, 4096)	(4096 / subN, 8196 / subK, subN, subK)
(K, N) = (20488, 4096)	-> (8196, 4096)	-> (4096 / subN, 8196 / subK, subN, subK)
normal layout	(4096, 4096)	(4096 / subN, 4096 / subK, subN, subK)
	T normal layout	T native layout

推荐使用 rknn_B_normal_layout_to_native_layout 接口进行直接数据转换。

对于 RK3576:

当 $K > 4096$, B 会被分成 T 段.

$\text{int } T = \text{std::ceil}(K / 4096);$

例如: normal layout -> native layout

$K = 10240, N = 2048, T = 3$, 数据会被分成 3 段.

$\text{subN} = \text{rknn_matmul_io_attr.B.dims}[2];$

$\text{subK} = \text{rknn_matmul_io_attr.B.dims}[3];$

	(4096, 2048)	(2048 / subN, 4096 / subK, subN, subK)
(K, N) = (10240, 2048)	-> (4096, 2048)	-> (2048 / subN, 4096 / subK, subN, subK)

normal layout (2048, 2048) (2048 / subN, 2048 / subK, subN, subK)

T normal layout T native layout

推荐使用 rknn_B_normal_layout_to_native_layout 接口进行直接数据转换。

Rockchip

4 示例

RKNN 提供了不同模型的参考示例，包括 MobileNet 图像分类、YOLOv5 目标检测等，代码工程位于 https://github.com/airockchip/rknn_model_zoo/tree/main/examples 目录下。

本章节以 PC 端 Ubuntu22.04，Conda 环境的 Python3.8，开发板为 RK3588 Linux 平台为例。有关开发环境的安装可参考第二章，其他平台的部署流程可参考 Rockchip_RKNPU_Quick_Start_RKNN_SDK。

4.1 MobileNet 模型部署示例

本章节以 MobileNet 模型部署为例，介绍如何快速上手模型转换、模型连板运行、模型评估和模型板端部署。

4.1.1 模型转换

1. 进入 rknn_model_zoo/examples/mobilenet/python 目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2. 执行模型转换并进行图片推理

```
python mobilenet.py --model ../model/mobilenetv2-12.onnx --target rk3588
```

执行该命令后模型是在电脑模拟器上进行推理，转换后的模型默认保存路径为 rknn_model_zoo/examples/mobilenet/model/mobilenet_v2.rknn。

4.1.2 模型连板运行

1. 进入 rknn_model_zoo/examples/mobilenet/python 目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2. 执行模型连板运行

```
python mobilenet.py --target rk3588 --npu_device_test
```

执行该命令后模型通过连板的方式在板端上进行推理。输出结果如下：

```
-----TOP 5-----  
[494] score=0.99 class="n03017168 chime, bell, gong"  
[469] score=0.00 class="n02939185 caldron, cauldron"  
[653] score=0.00 class="n03764736 milk can"  
[747] score=0.00 class="n04023962 punching bag, punch bag,  
punching ball, punchball"  
[505] score=0.00 class="n03063689 coffeepot"
```

4.1.3 模型评估

RKNN 提供(模拟器和板端)精度评估、耗时评估和内存评估的功能，辅助 RKNN 模型的优化和部署。

4.1.3.1 精度评估

1.进入 rknn_model_zoo/examples/mobilenet/python 目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型连板精度分析

```
python mobilenet.py --target rk3588 --accuracy_analysis --npu_device_test
```

模型连板精度分析的输出结果如下：

```
# simulator_error: calculate the output error of each layer of the
# simulator (compared to the 'golden' value).
#     entire: output error of each layer between 'golden' and
# 'simulator', these errors will accumulate layer by layer.
#     single: single-layer output error between 'golden' and
# 'simulator', can better reflect the single-layer accuracy of the
# simulator.
layer_name          simulator_error
                  entire          single
                  cos    euc      cos    euc
-----
...
[Conv] 464           0.99202 | 4.1079    0.99998 | 0.1981
[Conv] output_conv   0.99308 | 13.235    0.99992 | 1.4133
[Reshape] output_int8 0.99308 | 13.235    0.99993 | 1.3043
[exDataConvert] output 0.99308 | 13.235    0.99993 | 1.3043

# runtime_error: calculate the output error of each layer of the
# runtime.
#     entire: output error of each layer between 'golden' and
# 'runtime', these errors will accumulate layer by layer.
#     single_sim: single-layer output error between 'simulator' and
# 'runtime', can better reflect the single-layer accuracy of runtime.
layer_name          runtime_error
                  entire          single_sim
                  cos    euc      cos    euc
-----
...
[Conv] 464           0.99210 | 4.2718    1.00000 | 0.0
[Conv] output_conv   0.99203 | 14.847    1.00000 | 0.2007
[Reshape] output_int8 0.99203 | 14.847    1.00000 | 0.0
[exDataConvert] output 0.99203 | 14.847    1.00000 | 0.0
```

4.1.3.2 耗时评估

1.进入 rknn_model_zoo/examples/mobilenet/python 目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

2.执行模型耗时评估

```
python mobilenet.py --target rk3588 --eval_perf
```

模型耗时评估的输出结果如下：

Network Layer Information Table					
ID	OpType	DataType	Target	Time(us)	...
1	InputOperator	UINT8	CPU	17	
2	ConvClip	UINT8	NPU	331	
3	ConvClip	INT8	NPU	429	
4	Conv	INT8	NPU	292	
....					
55	Conv	INT8	NPU	374	
56	Reshape	INT8	CPU	61	
57	OutputOperator	INT8	CPU	11	

Total Operator Elapsed Per Frame Time(us): 12631
Total Memory Read/Write Per Frame Size(KB): 10563

Operator Time Consuming Ranking Table					
OpType	CallNumber	...	NPUTime(us)	TotalTime(us)	TimeRatio(%)
ConvClip	35		8436	8436	66.79%
Conv	9		2093	2093	16.57%
ConvAdd	10		2013	2013	15.94%
Reshape	1		0	61	0.48%
InputOperator	1		0	17	0.13%
OutputOperator	1		0	11	0.09%

4.1.3.3 内存评估

- 1.进入 rknn_model_zoo/examples/mobilenet/python 目录

```
cd rknn_model_zoo/examples/mobilenet/python
```

- 2.执行模型内存评估

```
python mobilenet.py --target rk3588 --eval_memory
```

模型内存评估的输出结果如下:

```
=====
Memory Profile Info Dump
=====
NPU model memory detail(bytes):
  Weight Memory: 3.53 MiB
  Internal Tensor Memory: 1.53 MiB
  Other Memory: 377.19 KiB
  Total Memory: 5.43 MiB

INFO: When evaluating memory usage, we need consider
the size of model, current model size is: 3.98 MiB
=====
```

4.1.4 板端部署

- 1.在 rknn_model_zoo 工程下的 build-linsx.sh 脚本中指定 gcc 交叉编译器路径

```
GCC_COMPILER=~/.opts/gcc-linaro-6.3.1-2017.05-
x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu
```

有关 gcc 交叉编译器的下载和安装方法可参考 Rockchip_RKNPU_Quick_Start_RKNN_SDK。

2. 编译模型相关文件

```
./build-linux.sh -t rk3588 -a aarch64 -d mobilenet
```

3. 推送可执行文件到板端

```
adb root
adb remount
adb push install/rk3588_linux_aarch64/rknn_mobilenet_demo/
/userdata/
```

4. 板端执行

```
adb shell

cd /userdata/rknn_mobilenet_demo/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/mobilenet_v2.rknn
model/bell.jpg
```

输出结果如下：

```
-----TOP 5-----
[494] score=0.99 class="n03017168 chime, bell, gong"
[469] score=0.00 class="n02939185 caldron, cauldron"
[653] score=0.00 class="n03764736 milk can"
[747] score=0.00 class="n04023962 punching bag, punch bag"
[505] score=0.00 class="n03063689 coffeepot"
```

4.2 YOLOv5 模型部署示例

4.2.1 模型转换

1. 下载模型

```
cd rknn_model_zoo/examples/yolov5/model
./download_model.sh
```

2. 执行模型转换

```
cd rknn_model_zoo/examples/yolov5/python
python convert.py ../model/yolov5s_relu.onnx rk3588
i8 ../model/yolov5s_relu.rknn
```

转换后的模型保存路径为 rknn_model_zoo/examples/yolov5/model/yolov5s_relu.rknn。

4.2.2 模型连板运行

1. 进入 rknn_model_zoo/examples/yolov5/python 目录

```
cd rknn_model_zoo/examples/yolov5/python
```

2. 执行模型连板运行

```
python yolov5.py --model_path ../model/yolov5s_relu.rknn --
target rk3588 --img_show
```

默认输入图片是 model/bus.jpg，结果图片如下所示：

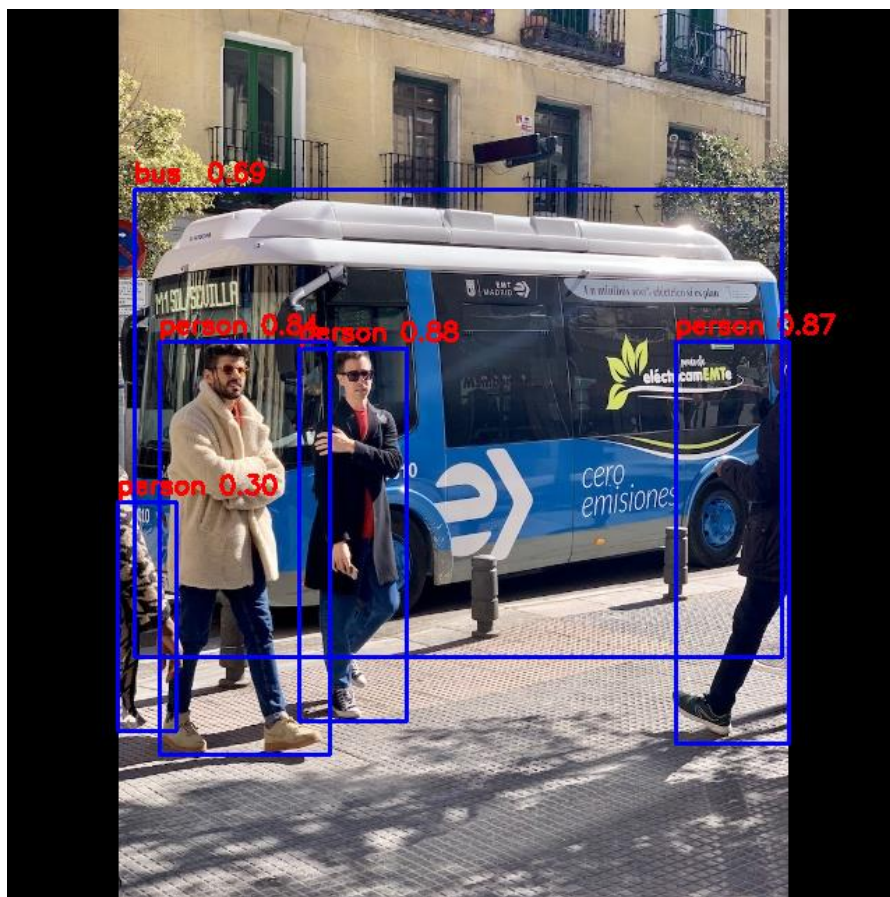


图 4-1 RKNN Python 可视化结果

4.2.3 板端部署运行

1. 在 rknn_model_zoo 工程下的 build-linux.sh 脚本中指定 gcc 交叉编译器路径

```
GCC_COMPILER=~/.opts/gcc-linaro-6.3.1-2017.05-  
x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu
```

有关 gcc 交叉编译器的下载和安装方法可参考 Rockchip_RKNPU_Quick_Start_RKNN_SDK。

2. 编译模型相关文件

```
cd rknn_model_zoo  
./build-linux.sh -t rk3588 -a aarch64 -d yolov5
```

3. 推送可执行文件到板端

```
adb root  
adb remount  
  
adb push install/rk3588_linux_aarch64/rknn_yolov5_demo/  
/userdata/  
adb push examples/yolov5/model/yolov5s_relu.rknn  
/userdata/rknn_yolov5_demo/model/
```

4. 板端运行

```
adb shell

cd userdata/rknn_yolov5_demo/
export LD_LIBRARY_PATH=./lib
./rknn_yolov5_demo model/yolov5s_relu.rknn model/bus.jpg
```

5. 从板端拉取到本地查看，在本地电脑的终端中，执行以下命令：

```
adb pull /userdata/rknn_yolov5_demo/out.png .
```

输出结果图片如下所示：

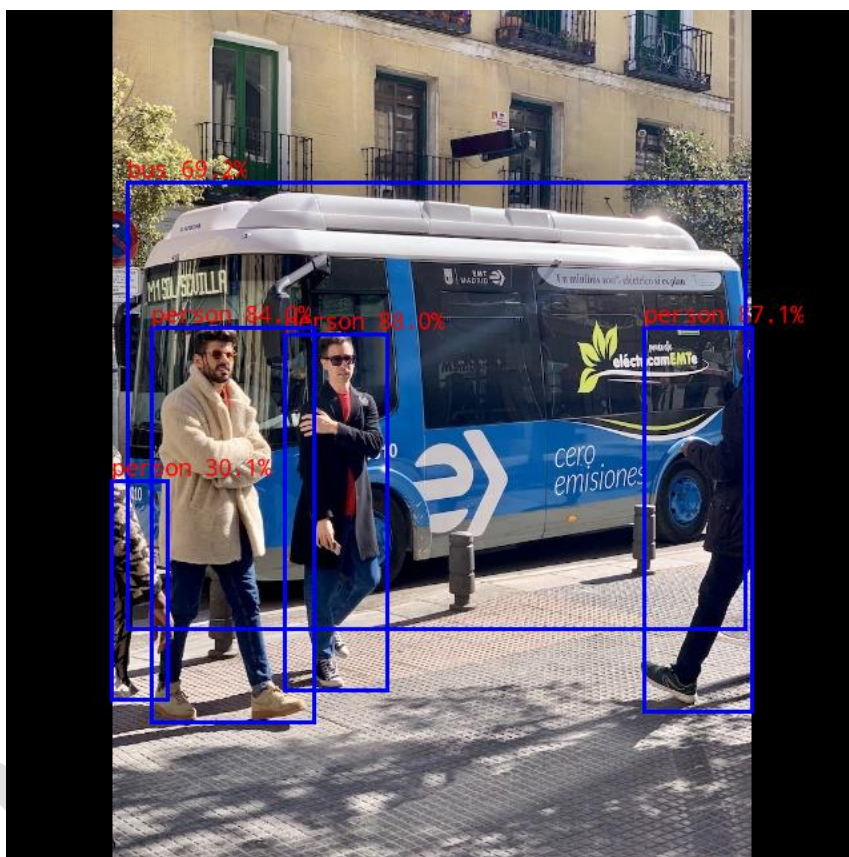


图 4-2 RKNN C demo 可视化结果

5 RKNN 进阶使用说明

5.1 数据排列格式

目前 RKNN 的数据排列格式主要有以下四种，NHWC、NCHW、NC1HWC2、UNDEFINE。

其中 NHWC 和 NCHW 的数据排布为深度学习常见数据排列方式，本章节不做额外说明，重点讲述 RKNPU 硬件专用的 NC1HWC2 数据格式的存储以及转换。

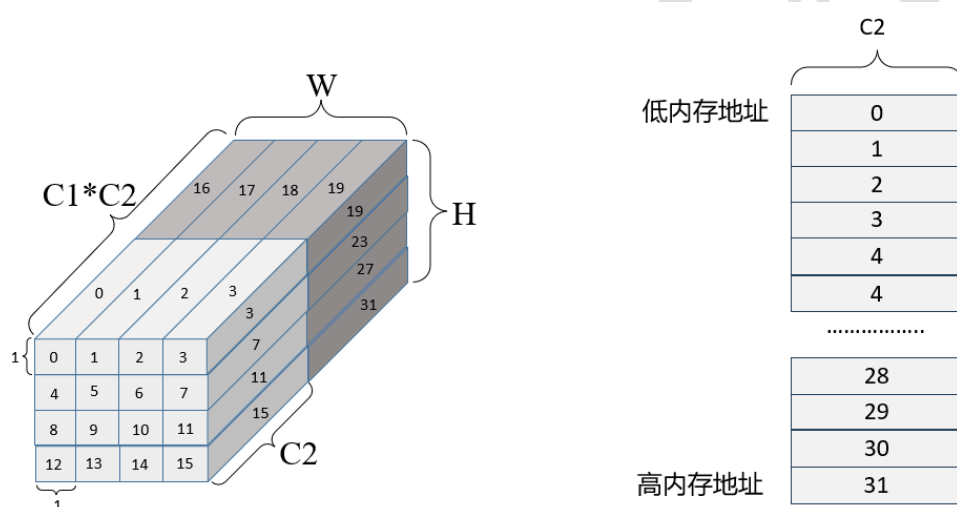


图 5-1 RKNPU NC1HWC2 数据排布与存储

如图 5-1 所示，数字 0 代表一笔数据，即一次存放 $C2$ 个数据，其中 $C2$ 是由平台决定的，不同硬件平台的 $C2$ 的规则约束由表 5-1 所示， $C1$ 为 $C/C2$ 的上取整值。NC1HWC2 数据存放的顺序与图中数值增长的顺序一致，先存放 0-15 的数据，再存放 16-31 的数据。以 RK3568 平台为例当 feature 为(1,13,4,4)的 int8 数据，对应的 NC1HWC2 为(1,2,4,4,8),此时 $C2$ 为 8， $C1$ 为 2，feature 在内存中在 16-31 排放的数据中，对应的每个 $C2$ 数据块只有前 5 个数据有效，剩下的 3 个数据是额外补的对齐数据。

表 5-1 不同硬件平台对应的 $C2$ 值

	RK3566/RK3568	RK3588/RK3576	RV1103/RV1106	RK3562
int8	8	16	16	16
float16	4	8	8	8

接下来重点介绍 NC1HWC2 数据排列转 NCHW 和 NHWC 数据在内存中的变化过程。

以 feature (1, 13, 2, 2) RK3568 为例，数据在内存排布中的转换，根据前文的对齐要求可知 feature(1, 13, 2, 2) 对应的 NC1HWC2 为(1, 2, 2, 2, 8)，NC1HWC2 的存储如下图所示，红色部分为额外对齐的无效数据。

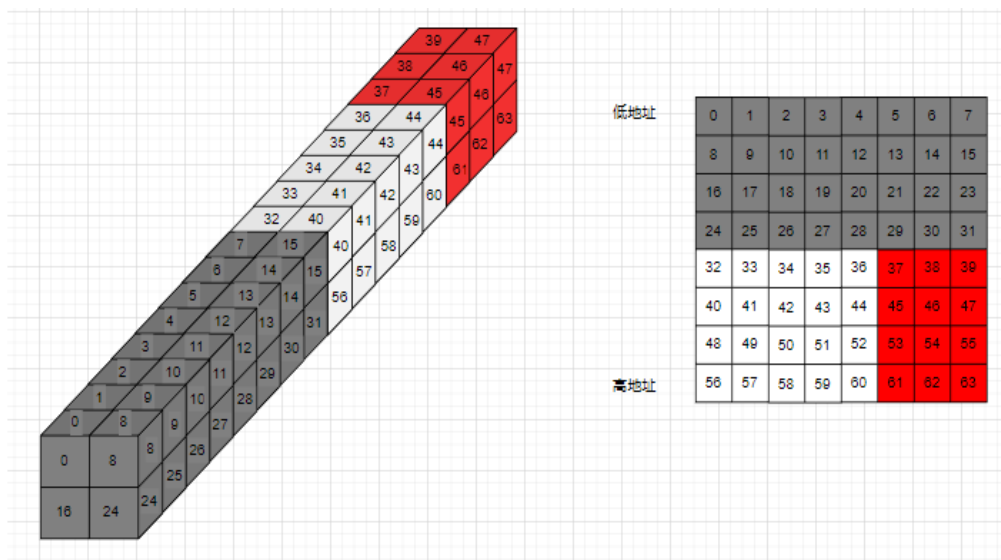


图 5-2 NC1HWC2 数据排布展开

移除无效数据转成 NCHW 即 (1, 13, 2, 2)数据，在内存中的排布如下：

低地址	0	8	16	24
	1	9	17	25
	2	10	18	26
	3	11	19	27
	4	12	20	28
	5	13	21	29
	6	14	22	30
	7	15	23	31
	32	40	48	56
	33	41	49	57
	34	42	50	58
	35	43	51	59
高地址	36	44	52	60

图 5-3 NCHW 数据排布

移除无效数据转成 NHWC 即 (1, 2, 2, 13)数据，在内存中的排布如下：

低地址	0	1	2	3	4	5	6	7	32	33	34	35	36
	8	9	10	11	12	13	14	15	40	41	42	43	44
	16	17	18	19	20	21	22	23	48	49	50	51	52
高地址	24	25	26	27	28	29	30	31	56	57	58	59	60

图 5-4 NHWC 数据排布

转换示例代码：

NC1HWC2 转 NCHW：以 int8 数据排列的 NC1HWC2 转成 int8 数据排列的 NCHW 如下所示：

```

/*
 *src:      表示 NC1HWC2 输入 tensor 的地址
 *dst:      表示 NCHW 输出 tensor 的地址
 *dims:     表示 NC1HWC2 的 shape 信息
 *channel:  表示 NCHW 输入的 c 的值
 * h :      表示 NCHW 的 h 的值
 * w :      表示 NCHW 的 w 的值
 */
int NC1HWC2_to_NCHW(const int8_t* src, int8_t* dst, int* dims,
int channel, int h, int w)
{
    int batch = dims[0];
    int C1 = dims[1];
    int C2 = dims[4];
    int hw_src = dims[2] * dims[3];
    int hw_dst = h * w;
    for (int i = 0; i < batch; i++) {
        src = src + i * C1 * hw_src * C2;
        dst = dst + i * channel * hw_dst;
        for (int c = 0; c < channel; ++c) {
            int plane = c / C2;
            const int8_t* src_c = plane * hw_src * C2 + src;
            int offset = c % C2;
            for (int cur_h = 0; cur_h < h; ++cur_h)
                for (int cur_w = 0; cur_w < w; ++cur_w) {
                    int cur_hw = cur_h * w + cur_w;
                    dst[c * hw_dst + cur_h * w + cur_w] = src_c[C2 * cur_hw +
offset];
                }
        }
    }
    return 0;
}

```

NC1HWC2 转 NHWC：以 int8 数据排列的 NC1HWC2 转成 int8 数据排列的 NHWC 如下所示：

```
/*
 *src:      表示 NC1HWC2 输入 tensor 的地址
 *dst:      表示 NCHW 输出 tensor 的地址
 *dims:     表示 NC1HWC2 的 shape 信息
 *channel:   表示 NHWC 输入的 c 的值
 * h :      表示 NCHW 的 h 的值
 * w:       表示 NCHW 的 w 的值
 */
int NC1HWC2_to_NHWC(const int8_t* src, int8_t* dst, int* dims,
int channel, int h, int w)
{
    int batch = dims[0];
    int C1     = dims[1];
    int C2     = dims[4];
    int hw_src = dims[2] * dims[3];
    int hw_dst = h * w;
    for (int i = 0; i < batch; i++) {
        src = src + i * C1 * hw_src * C2;
        dst = dst + i * channel * hw_dst;
        for (int cur_h = 0; cur_h < h; ++cur_h) {
            for (int cur_w = 0; cur_w < w; ++cur_w) {
                int cur_hw = cur_h * dims[3] + cur_w;
                for (int c = 0; c < channel; ++c) {
                    int plane = c / C2;
                    const auto* src_c = plane * hw_src * C2 + src;
                    int offset = c % C2;
                    dst[cur_h * w * channel + cur_w * channel + c] = src_c[C2
* cur_hw + offset];
                }
            }
        }
    }
    return 0;
}
```

5.2 RKNN Runtime 零拷贝调用

5.2.1 零拷贝介绍

目前在 RK3562/RK3566/RK3568/RK3588/RK3576 上有两组 API 可以使用，分别是通用 API 接口和零拷贝流程的 API 接口，RV1106/RV1103 只支持零拷贝流程的 API 接口。

在推理 RKNN 模型时，原始数据要经过输入处理、NPU 运行模型、输出处理三大流程。目前根据不同模型输入格式和量化方式，接口内部会存在通用 API 和零拷贝 API 两种处理流程，如图 5-5 和图 5-6 所示，两组 API 的主要区别在于，通用接口每次更新帧数据，需要将外部模块分配的数据拷贝到 NPU 运行时的输入内存，而零拷贝流程的接口会直接使用预

先分配的内存（包括 NPU 运行时创建的或外部其他框架创建的，比如 DRM 框架），减少了内存拷贝的花销，性能更优，带宽更少。当用户输入数据只有虚拟地址时，只能使用通用 API 接口；当用户输入数据有物理地址或 fd 时，两组接口都可以使用。**通用 API 和零拷贝 API 不能混合调用。**

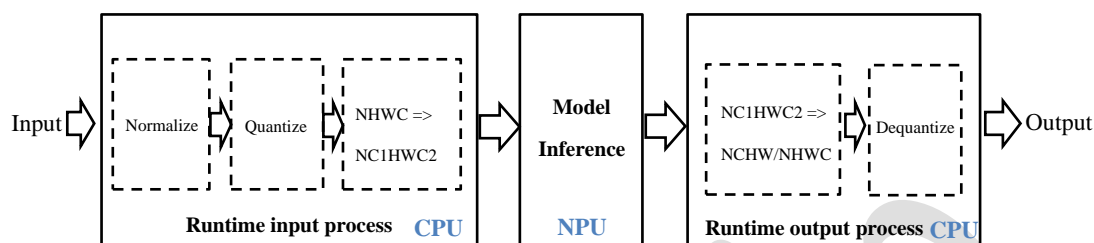


图 5-5 通用 API 的数据处理流程

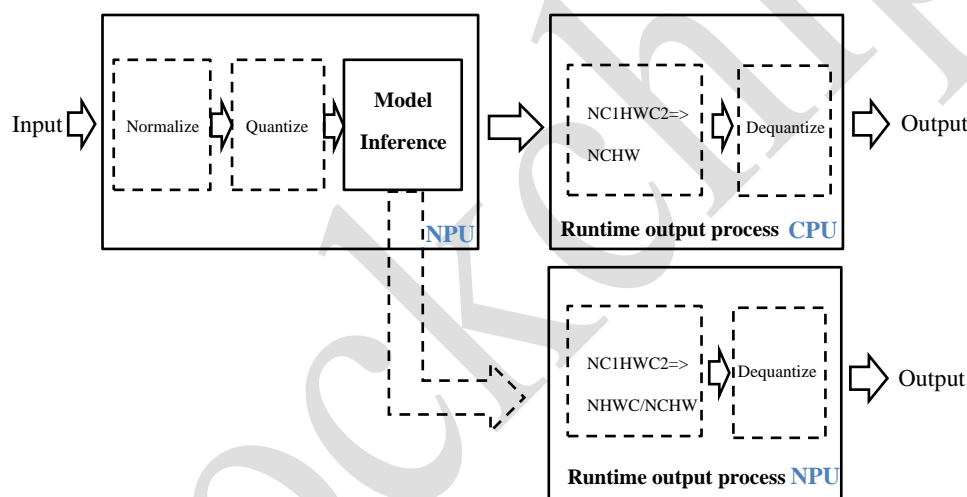


图 5-6 零拷贝 API 数据处理流程

1. 通用 API

通用 API 的流程如图 5-5 所示。对于数据的归一化、量化、数据排布格式转换、反量化均在 CPU 上运行，模型本身的推理在 NPU 上运行。

2. 零拷贝 API

零拷贝 API 的流程如图 5-6 所示。优化了通用 API 的数据处理流程，零拷贝 API 归一化、量化和模型推理都会在 NPU 上运行，NPU 输出的数据排布格式和反量化过程在 CPU 或者 NPU 上运行。零拷贝 API 对于输入数据流程的处理效率会比通用 API 高。

零拷贝场景的条件如下表所示：

表 5-2 零拷贝输入要求

输入维度	输入对齐要求	
	RK3566/RK3568	RK3576/RK3562/RK3588/RV1106/RV1103
4 维，通道数是 1、3、4	宽 8 字节对齐	宽 16 字节对齐
非 4 维	总大小 8 字节对齐	总大小 16 字节对齐

5.2.2 C API 零拷贝整体流程

零拷贝 API 接口使用 `rknn_tensor_memory` 结构体，需要在推理前创建并设置该结构体，并在推理后读取该结构体中的内存信息。根据用户是否需要自行分配模型的模块内存（输入/输出/权重/中间结果）和内存表示方式（文件描述符/物理地址等）差异，有下列三种典型的零拷贝调用流程，如图 5-7 至图 5-9 所示，红色部分表示专为零拷贝加入的接口和数据结构，斜体表示接口调用之间传递的数据结构。

- 输入/输出内存由运行时分配

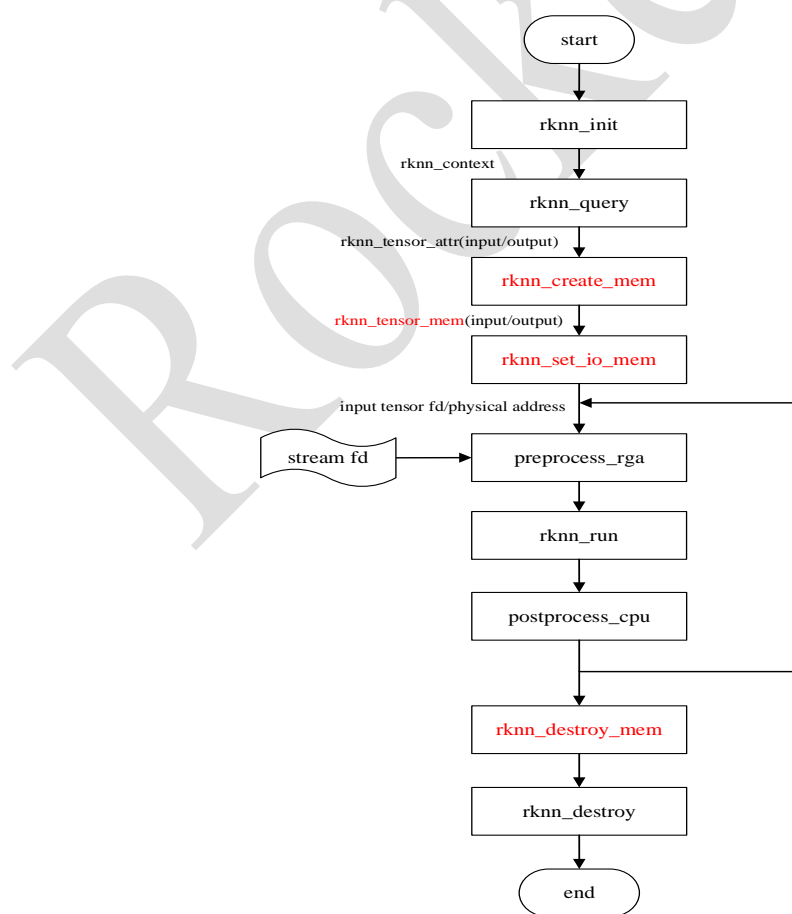


图 5-7 零拷贝 API 接口调用流程（输入/输出内部分配）

如图 5-7 所示，输入/输出内存由运行时分配调用的是 rknn_create_mem()接口创建 rknn_tensor_memory 结构体，rknn_set_io_mem()设置输入输出 rknn_tensor_memory 结构体。

rknn_create_mem()接口创建的输入/输出内存信息结构体包含了文件描述符成员和物理地址，RGA 的接口使用到 NPU 分配的内存信息，preprocess_rga()表示 RGA 的接口，stream_fd 表示 RGA 的接口输入源的内存数据，postprocess_cpu()表示后处理的 CPU 实现。

- 输入/输出内存由外部分配

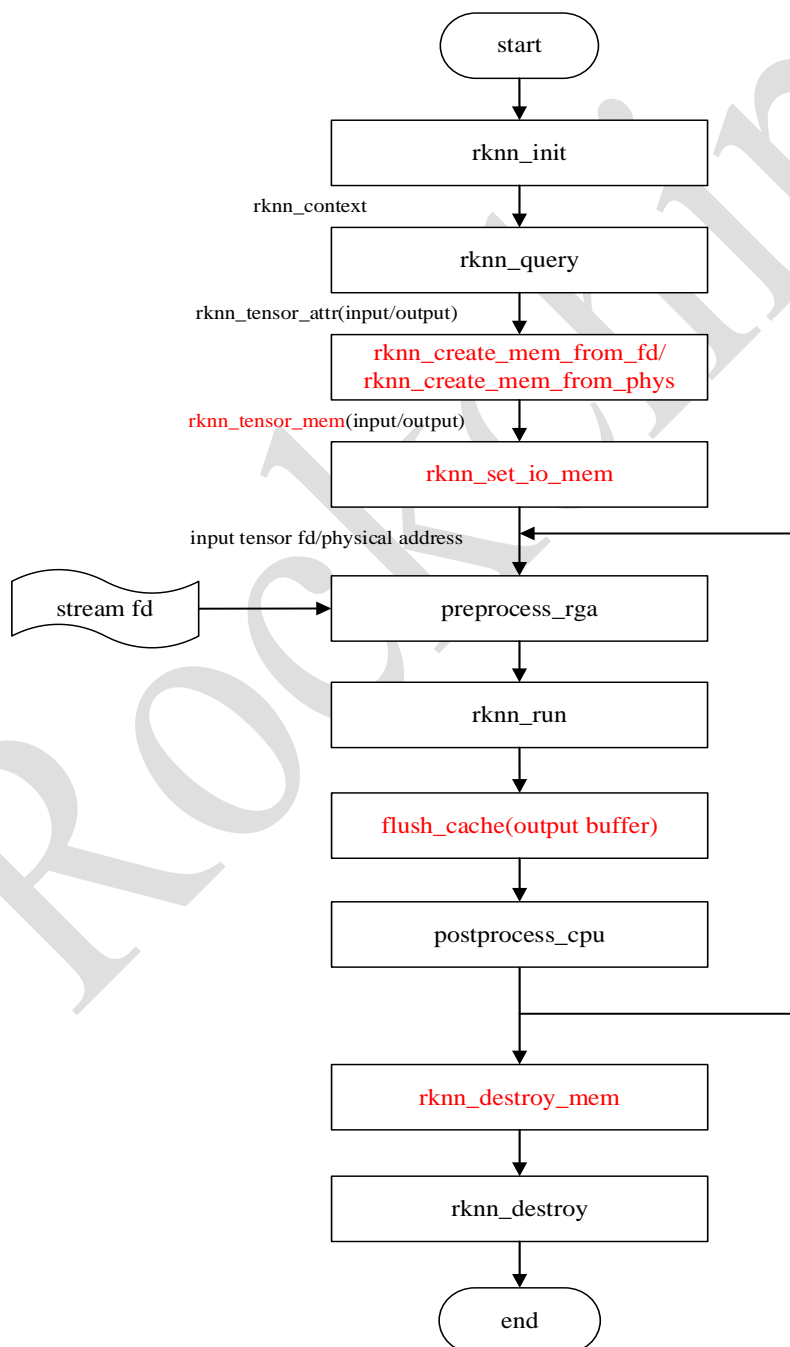


图 5-8 零拷贝 API 接口调用流程（输入/输出外部分配）

如图 5-8 所示，输入 / 输出内存由外部分配调用的是 rknn_create_mem_from_fd()/rknn_create_mem_from_phys()接口创建 rknn_tensor_memory 结构体，rknn_set_io_mem()设置输入输出 rknn_tensor_memory 结构体。flush_cache 表示用户需要调用与分配的内存类型关联的接口来刷新输出缓存。

- 输入/输出/权重/中间结果内存由外部分配

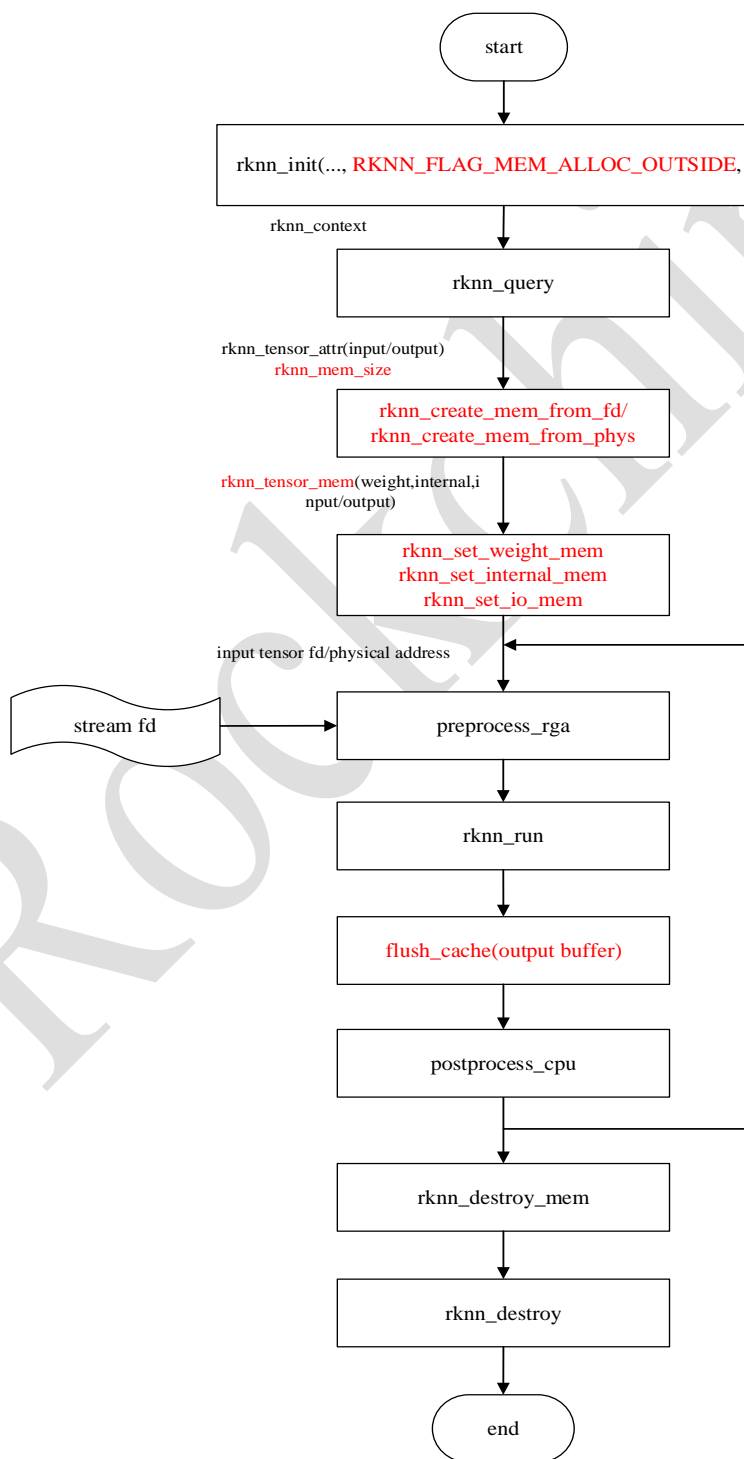


图 5-9 零拷贝 API 接口调用流程（输入/输出/权重/中间结果外部分配）

如图 5-9 所示，输入/输出/权重/中间结果内存由外部分配调用的是 `rknn_create_mem_from_fd()/rknn_create_mem_from_phys()` 接口创建 `rknn_tensor_memory` 结构体，`rknn_set_io_mem()` 设置输入输出 `rknn_tensor_memory` 结构体，`rknn_set_weight_mem()/rknn_set_internal_mem()` 设置权重/中间结果 `rknn_tensor_memory` 结构体。

5.2.3 C API 零拷贝的用法

以图 5-7 零拷贝 API 接口调用流程（输入/输出内部分配）为例，用法如下：

- `rknn_query()`

输入：

用 `RKNN_QUERY_NATIVE_INPUT_ATTR` 查询相关的属性（注意，不是 `RKNN_QUERY_INPUT_ATTR`）。当查询出来的 `fmt`（或者称为 `layout`）不同时，需要提前处理的方式也不一样。该方式查询出来的是输入硬件效率最优的 `layout` 和 `type`。

`rknn_query()` 输入的情况如下：

- 当 `layout` 为 `RKNN_TENSOR_NCHW` 时，这种情况一般输入是 4 维，并且数据类型为 `bool` 或者 `int64`，当传数据给 NPU 时，也需要按照 `NCHW` 格式排列给 NPU。
- 当 `layout` 为 `RKNN_TENSOR_NHWC` 时，这种情况一般输入是 4 维，并且数据类型为 `float32/float16/int8/uint8`，同时，输入通道数是 1、3、4。当传数据给 NPU 时，也需要按照 `NHWC` 格式排列给 NPU。需要注意的是当 `pass_through=1` 时，`width` 可能需要做 `stride` 对齐，具体取决于查询出来的 `w_stride` 的值。
- 当 `layout` 为 `RKNN_TENSOR_NC1HWC2` 时，这种情况一般输入是 4 维，并且数据类型为 `float16/int8`，同时，输入通道数不是 1、3、4。当 `pass_through=0` 时，输入数据按照 `NHWC` 格式排列，接口内部会进行 `NHWC` 到 `NC1HWC2` 的 `cpu` 转换；当 `pass_through=1` 时，输入数据按照 `NC1HWC2` 格式排列，用户外部需转换好。
- 当 `layout` 为 `RKNN_TENSOR_UNDEFINED` 时，这种情况一般输入不是 4 维，当传数据给 NPU 时，需要按照 `ONNX` 模型输入格式传给 NPU。NPU 不做任何的 `mean/std` 处理以及 `layout` 转换。

如果用户需要的输入配置不同于查询接口获取的 `rknn_tensor_attr` 结构体，可以对 `rknn_tensor_attr` 结构体进行对应修改，目前支持的可修改的输入数据类型如表 5-3 所示。特别注意：如果查询的数据类型是 `uint8`，用户想传入 `float32` 类型，则 `rknn_tensor_attr` 结构体

的 size 要修改成原 size 的四倍，同时其中的数据类型要修改成 RKNN_TENSOR_FLOAT32。用该方式修改后硬件效率就不是最优了，接口内部会调用 cpu 进行数据类型转换。

表 5-3 输入可修改的输入数据类型表

		rknn_query 查询得到的模型数据类型					
		bool	int8	float16	int16	int32	Int64
用户接口修改的数据类型	bool	Y					
	int8		Y				
	uint8		Y	Y			
	float32		Y	Y			
	float16			Y			
	int16				Y		
	int32					Y	
	Int64						Y

输出:

用 **RKNN_QUERY_NATIVE_OUTPUT_ATTR** 查询相关的属性（注意，不是 **RKNN_QUERY_OUTPUT_ATTR**）。当查询出来的 fmt（或者称为 layout）不同时，需要后处理的方式也不一样。该方式查询出来的是输出硬件效率最优的 layout 和 type。

当输出是 4 维且用户需要 NHWC layout 的四维输出时，可以用 **RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR** 查询相关的属性。该方式可以直接获得 NPU 输出 nhwc layout。

rknn_query()输出的情况如下:

- 当 layout 为 RKNN_TENSOR_NC1HWC2 时，这种情况一般输出是 4 维，并且数据类型为 float16/int8。当用户需要 NCHW layout 时，外部需进行 NC1HWC2 到 NCHW 的 layout 转换。
- 当 layout 为 RKNN_TENSOR_UNDEFINED 时，这种情况一般输出非 4 维，并且数据类型为 float16/int8。用户外部无需进行 layout 转换。
- 当 layout 为 RKNN_TENSOR_NCHW 时，这种情况一般输出是 4 维，并且数据类型为

float16/int8。用户外部无需进行 layout 转换。

- d. 当 layout 为 RKNN_TENSOR_NHWC 时，这种情况一般输出是 4 维，并且数据类型为 float16/int8。这种情况一般是用户调用 RKNN_QUERY_NATIVE_NHWC_OUTPUT_ATTR 接口查询出来的 layout。

如果用户需要的输出配置不同于查询接口获取的 rknn_tensor_attr 结构体，可以对 rknn_tensor_attr 结构体进行对应修改，可修改的配置信息如表 5-4，表 5-5 所示，特别注意：如果查询输出的数据类型是 int8，用户想获取成 float32 类型输出，则 rknn_tensor_attr 结构体的 size 要修改成原 size 的四倍，同时其中的数据类型要修改成 RKNN_TENSOR_FLOAT32。用该方式修改后硬件效率就不是最优了，接口内部会调用 cpu 进行数据类型转换。

表 5-4 输出可修改的输入数据类型表

		rknn_query 查询得到的模型数据类型					
		bool	int8	float16	int16	int32	Int64
用户接口修改的数据类型	bool	Y					
	int8		Y				
	uint8						
	float32		Y	Y			
	float16			Y			
	int16				Y		
	int32					Y	
	Int64						Y

表 5-5 输出可修改的 layout 类型表

		rknn_query 查询得到的模型 layout 类型			
		NC1HWC2	NCHW	NHWC	UNDEFINE
用户接口设置的 layout 类型	NC1HWC2	Y			
	NCHW	Y	Y		
	NHWC			Y	
	UNDEFINE				Y

RK3562/RK3566/RK3568/RK3588/RK3576 支持的零拷贝接口 NPU 输出配置如表 5-6 所示，RV1106/RV1103 支持零拷贝接口 NPU 输出配置如表 5-7 所示。

表 5-6 RK3562/RK3566/RK3568/RK3588/RK3576 零拷贝接口 NPU 支持的输出配置

模型类型	输出数据类型	输出维度	可支持 output layout
int8 模型	int8 float16 float32	4 维	NCHW NC1HWC2 NHWC
		非 4 维	UNDEFINE
float16 模型	float16 float32	4 维	NCHW NC1HWC2 NHWC
		非 4 维	UNDEFINE

表 5-7 RV1106/RV1103 零拷贝接口 NPU 支持的输出配置

模型类型	输出数据类型	输出维度	可支持 output layout
int8 模型	int8 float16	4 维	NCHW NC1HWC2 NHWC
		非 4 维	UNDEFINE

- rknn_create_mem

零拷贝 API 接口使用 `rknn_tensor_memory` 结构体，需要在推理前创建并设置该结构体，并在推理后读取该结构体中的内存信息。当无需对 `RKNN_QUERY_NATIVE_INPUT_ATTR`, `RKNN_QUERY_NATIVE_OUTPUT_ATTR` 出来的 `layout` 和 `type` 进行修改时，直接采用默认配置的 `size_with_stride` 创建内存大小。若修改了相应的 `layout` 和 `type`，则需按照相应的 `size` 创建内存大小。(例如输出的数据类型是 `int8`，用户想获取成 `float32` 类型输出，`size` 要修改成原 `size` 的四倍)。

- rknn_set_io_mem

`rknn_set_io_mem()` 用于设置包含模型输入/输出内存信息的 `rknn_tensor_mem` 结构体，和 `rknn_init()` 类似，只要在最开始调用一次，后面反复执行 `rknn_run()` 即可。

5.3 NPU 多核配置

RK3588 通过 3 核 NPU，RK3576 通过 2 核 NPU 提供更强的算力。本章节将详细介绍多核 NPU 的配置方法，以提高模型的推理效率。

注：多核运行适用于网络层计算量较大的网络，对小网络提升幅度较小，甚至可能因为单核多核的切换（该切换需 CPU 介入）而导致性能下降。

5.3.1 多核运行配置方法

如果使用 Python 作为应用程序开发语言，可以通过 `RKNN-Toolkit2` 或 `RKNN-Toolkit Lite2` `init_runtime()` 接口中的 “`core_mask`” 参数设置模型运行的 NPU 核心。该参数的详细说明如下表：

表 5-8 init_runtime 接口 core_mask 参数说明

参数	详细说明
core_mask	<p>该参数用于设置模型运行的 NPU 核心。可选值和相应说明如下：</p> <p>NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的 NPU 核上。</p> <p>NPU_CORE_0: 模型运行在 NPU Core0 上。</p> <p>NPU_CORE_1: 模型运行在 NPU Core1 上。</p> <p>NPU_CORE_2: 模型运行在 NPU Core2 上。</p> <p>NPU_CORE_0_1: 模型同时运行在 NPU Core0 和 NPU Core1 上。</p> <p>NPU_CORE_0_1_2: 模型同时运行在 NPU Core0, Core1 和 Core2 上。</p> <p>NPU_CORE_ALL: 根据平台自定配置 NPU 核心数量。</p> <p>默认值为 NPU_CORE_AUTO。</p> <p>注：在 RKNN-Toolkit Lite2 上设置该参数时，值的前面要加上 RKNNLite，例如 RKNNLite.NPU_CORE_AUTO；如果在 RKNN-Toolkit2 上设置该参数时，值的前面要加上 RKNN，例如 RKNN.NPU_CORE_AUTO。</p>

RKNN-Toolkit2 设置 NPU 核心，参考代码如下：

```
#Python
.....
# Init runtime environment
print('--> Init runtime environment')
ret = rknn.init_runtime(target='rk3588',
core_mask=RKNN.NPU_CORE_0)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)
```

RKNN-Toolkit Lite2 设置 NPU 核心，参考代码如下：

```
#Python
.....
# Init runtime environment
print('--> Init runtime environment')
ret = rknn_lite.init_runtime(core_mask=RKNNLite.NPU_CORE_0)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)
print('done')
```

如果使用 C/C++ 作为应用程序开发语言，可以调用 rknn_set_core_mask()接口设置模

型运行的 NPU 核心。该接口 `core_mask` 的详细说明如下表：

Rockchip

表 5-9 rknn_set_core_mask 接口 core_mask 参数说明

参数	详细说明
core_mask	<p>该参数用于设置模型运行的 NPU 核心。可选值和相应说明如下：</p> <p>RKNN_NPU_CORE_AUTO: 自动调度模式，模型将以单核模式自动运行在当前空闲的 NPU 核上。</p> <p>RKNN_NPU_CORE_0: 模型运行在 NPU Core0 上。</p> <p>RKNN_NPU_CORE_1: 模型运行在 NPU Core1 上。</p> <p>RKNN_NPU_CORE_2: 模型运行在 NPU Core2 上。</p> <p>RKNN_NPU_CORE_0_1: 模型同时运行在 NPU Core0 和 NPU Core1 上。</p> <p>RKNN_NPU_CORE_0_1_2: 模型同时运行在 NPU Core0, Core1 和 Core2 上。</p>

使用 C/C++ API 设置模型运行 NPU 核心，参考代码如下：

```
// C++
// rknn_context ctx;
rknn_core_mask core_mask = RKNN_NPU_CORE_0;
int ret = rknn_set_core_mask(ctx, core_mask);
```

5.3.2 查看多核运行效果

本章节将详细说明 RKNN 模型以多核模式运行时的效果。

如果使用 RKNN-Toolkit2 连接开发板进行模型推理，需要在调用 rknn.init_runtime()接口时将 perf_debug 参数设置成 True，接着调用 rknn.eval_perf()接口，即可打印每层的运行信息。参考代码如下：

```
#Python
# Init runtime environment
ret = rknn.init_runtime(target='rk3588',
device_id='29d5dd97766a5c27', perf_debug=True)
if ret != 0:
    print('Init runtime environment failed!')
    exit(ret)

# Eval performance
rknn.eval_perf()
```

如果是直接在板端进行模型推理，需要在运行应用前将 RKNN_LOG_LEVEL 设成 4 或以上，此时将打印模型每层的运行信息。设置方法如下：

```
# Python
# 使用 RKNN-Toolkit Lite2 提供的 Python 接口，只需在创建 RKNNLite 对象时将
verbose 设成 True 即可
rknnlite = RKNNLite(verbose=True)

# 使用 C/C++接口，则需要在运行二进制程序前设置如下环境变量
export RKNN_LOG_LEVEL=4
```

以 lenet 模型为例，通过上述设置后，终端将打印类似如下日志（为方便展示，删除了 InputShape, OutputShape, DDR Cycles, NPU Cycles, Total Cycles, Time(us), MacUsage(%), Task Number, Lut Number, RW(kb), FullName 等字段）：

ID	OpType	DataType	Target	WorkLoad(0/1/2)-ImproveTherical
1	InputOperator	UINT8	CPU	100.0%/0.0%/0.0% - Up:0.0%
2	Conv	UINT8	NPU	50.0%/50.0%/0.0% - Up:50.0%
3	MaxPool	INT8	NPU	100.0%/0.0%/0.0% - Up:0.0%
4	Conv	INT8	NPU	50.0%/50.0%/0.0% - Up:50.0%
5	MaxPool	INT8	NPU	100.0%/0.0%/0.0% - Up:0.0%
6	ConvRelu	INT8	NPU	48.1%/51.9%/0.0% - Up:48.1%
7	Conv	INT8	NPU	100.0%/0.0%/0.0% - Up:0.0%
8	Softmax	INT8	CPU	0.0%/0.0%/0.0% - Up:0.0%
9	OutputOperator	FLOAT16	CPU	0.0%/0.0%/0.0% - Up:0.0%

Total Operator Elapsed Time(us): 591
Total Memory RW Amount(MB): 0

模型每层运行信息中的"WorkLoad(0/1/2)-ImproveTherical"一列只在多核 NPU 上会打印，记录了模型每一层的任务在 NPU 核心上是如何分配以及其理论性能提升情况。例如 "50.0%/50.0%/0.0% - Up:50.0%"代表该层的计算量以 Core0 负责 50%，Core1 负责 50%进行分配，该层的性能相比单核运行，理论能提升 50%。如果某一层的性能没有提升，例如 "100.0%/0.0%/0.0% - Up:0.0%"，可能存在以下几种情况：

该层的负载太小，小于 NPU 多核任务分配的粒度，因此该层运行在单核上；

该类算子在 NPU 驱动中未实现多核任务切分，待后续版本支持。现有已支持多核任务切分的算子有：**Conv, DepthwiseConvolution, Add, Concat, Relu, Clip, Relu6, ThresholdedRelu, PRelu, LeakyRelu**。

5.3.3 多核性能提升技巧

可以尝试如下方法，以得到较高的多核运行性能：

- 将 CPU/DDR/NPU 频率定到最高
- 将应用绑定至 CPU 大核
- 将 NPU 中断绑定至上面一点相应的 CPU 大核

不同固件对应的定频命令有所区别，请参考 [8.1.1](#) 章节。

以将应用绑定到 CPU4 大核心为例，上面提到的后两点可以参考如下脚本：

```
interrupts=$(cat /proc/interrupts | grep npu)
interrupts_array=($interrupts)

irq1=$(echo ${interrupts_array[0]} | awk -F ':' '{print $1}')
irq2=$(echo ${interrupts_array[14]} | awk -F ':' '{print $1}')
irq3=$(echo ${interrupts_array[28]} | awk -F ':' '{print $1}')

for irq in $irq1 $irq2 $irq3; do
    echo 4 > /proc/irq/$irq/smp_affinity_list
done

taskset 10 ./rknn_benchmark lenet.rknn "" 10 3 # CPU4 对应的
taskset 掩码值为 0x10
```

上述脚本会执行如下操作：

- 执行 `cat /proc/interrupts | grep npu` 命令并解析出三个中断号（去除冒号）
- 使用循环将每个中断号的 `smp_affinity_list` 设置为 4（CPU4 对应的 ID 为 4）
- 最后执行 `taskset 10 ./rknn_benchmark lenet.rknn "" 10 3` 命令，CPU4 对应的 `taskset` 参数为 10（有关 `taskset` 的具体用法，请参考：<https://man7.org/linux/man-pages/man1/taskset.1.html>）

通过上述操作，NPU 中断以及应用程序“rknn_benchmark”都将在 CPU4 上运行，这样可以消除 NPU 中断处理的核心切换开销。

5.4 动态 Shape

5.4.1 动态 Shape 功能介绍

动态 shape 是指模型输入数据的形状在运行时可以改变。它可以帮助处理输入数据大小不固定的情况，增加模型的灵活性。在之前仅支持静态 shape 的 RKNN 模型情况下，如果用户需要使用多个输入 shape，传统的做法是生成多个 RKNN 模型，在模型部署时初始化多个上下文分别执行推理，而在引入动态 shape 后，用户可以只保留一份与静态 shape RKNN 模型大小接近的动态 shape RKNN 模型，并使用一个上下文进行推理，从而节省 Flash 占用和 DDR 占用，动态 shape 在图像处理 and 序列模型推理中具有重要的作用，它的典型应用场景包括：

- 序列长度改变的模型，常见于 NLP 模型，例如 BERT, GPT
- 空间维度变化的模型，例如分割和风格迁移
- 带 Batch 模型，Batch 维度上变化
- 可变输出数量的目标检测模型

5.4.2 RKNN SDK 版本和平台要求

- RKNN-Toolkit2 版本 $\geq 1.5.0$
- RKNNPU Runtime 库(librknnrt.so)版本 $\geq 1.5.0$
- RK3566/RK3568/RK3588/RK3588S/RK3562/RK3576 平台的 NPU 支持该功能

5.4.3 生成动态 Shape 的 RKNN 模型

本节介绍使用 RKNN-Toolkit2 的 Python 接口生成动态 shape 的 RKNN 模型的步骤：

1. 确认模型支持动态 shape

如果模型文件本身不是动态 shape, RKNN-Toolkit2 支持扩展成动态 shape 的 RKNN 模型。首先, 用户要确认模型本身不存在限制动态 shape 的算子或子图结构, 例如, 常量的形状无法改变, RKNN-Toolkit2 工具在转换过程会报错, 如果遇到不支持动态 shape 扩展的情况, 用户要根据报错信息, 修改模型结构, 重新训练模型以支持动态 shape。建议使用原始模型本身就是动态 shape 的模型。

2. 设置需要使用的输入形状

由于 NPU 硬件特性, 动态 shape RKNN 模型不支持输入形状任意改变, 要求用户设置有限个输入形状。对于多输入的模型, 每个输入的 shape 个数要相同。例如, 在使用 RKNN-Toolkit2 转换 Caffe 模型时, Python 代码示例如下:

```
# Python
dynamic_input = [
    [[1,3,224,224]], # set the first shape for all inputs
    [[1,3,192,192]], # set the second shape for all inputs
    [[1,3,160,160]], # set the third shape for all inputs
]

# Pre-process config
rknn.config(mean_values=[103.94, 116.78, 123.68],
std_values=[58.82, 58.82, 58.82], quant_img_RGB2BGR=True,
dynamic_input=dynamic_shapes)
```

上述接口配置会生成支持 3 个 shape 分别是[1,3,224,224]、[1,3,192,192]和[1,3,160,160]的动态 shape RKNN 模型。

dynamic_input 中的 shape 与原始模型框架的 layout 一致。例如, 对于相同的 224x224 大小的 RGB 图片做分类, TensorFlow/TFLite 模型输入是[1,224,224,3], 而 ONNX 模型输入是[1,3,224,224]。

3. 量化

在设置好输入 shape 后, 如果要做量化, 则需要设置量化校正集数据。工具会读取用

户设置的最大分辨率输入做量化（是所有输入尺寸之和的最大的一组 shape）。例如，模型有两个输入，一个输入 shape 分别是[1,224]和[1,112]，另一个输入 shape 分别[1,40]和[1,80]，第一组 shape 所有输入尺寸之和是 $1*224+1*40=264$ ，第二组 shape 所有输入尺寸之和是 $1*112+1*80=192$ ，第一组 shape 所有输入尺寸之和更大，因此使用两个输入分别以[1,224]和[1,40]的 shape 做量化。

- 如果量化矫正集是 jpg/png 图片格式，用户可以使用不同的分辨率的图片做量化，因为工具会对图片使用 opencv 的 resize 方法缩放至最大分辨率后做量化。
- 如果量化矫正集是 npy 格式，则用户必须使用最大分辨率输入的 shape。量化后，模型内所有 shape 在运行时使用同一套量化参数进行推理。

另外，输入的最大分辨率 shape 在调用 rknn.config 时也会打印出来，如下：

```
W config: The 'dynamic_input' function has been enabled, the
MaxShape is dynamic_input[0] = [[1,224],[1,40]]!
    The following functions are subject to the MaxShape:
        1. The quantified dataset needs to be configured
according to MaxShape
        2. The eval_perf or eval_memory return the results of
MaxShape
```

4. 推理评估或精度分析

动态 shape RKNN 模型做推理或做精度分析时，用户必须提供第 2 步中设置的其中一组 shape 的输入。接口使用上与静态 shape RKNN 模型场景一致，此处不做赘述。

完整的创建动态 shape RKNN 模型示例，请参考(https://github.com/airockchip/rknn-toolkit2/tree/master/rknn-toolkit2/examples/functions/dynamic_shape)

5.4.4 C API 部署

得到动态 shape RKNN 模型后，接着使用 RKNPU2 C API 进行部署。按照接口形式，分为通用 API 和零拷贝 API 部署流程。

5.4.4.1 通用 API

使用通用 API 部署动态 shape RKNN 模型的流程如下图所示：

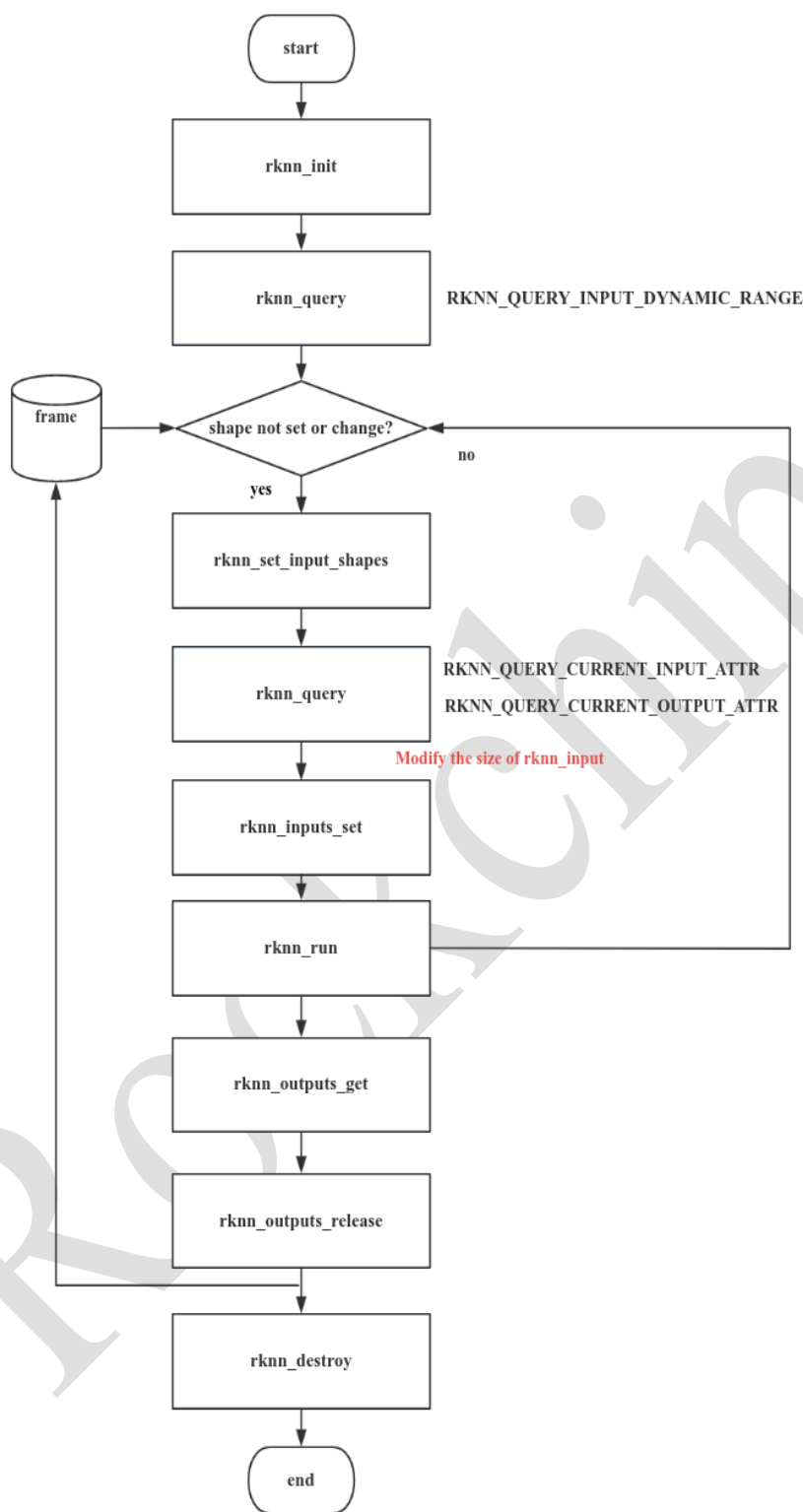


图 5-10 动态 shape 输入接口的通用 API 调用流程

加载动态 shape RKNN 模型后，可以在运行时动态修改输入的 shape。首先，通过 rknn_query() 可以查询 RKNN 模型支持的输入 shape 列表，每个输入支持的 shape 列表信息以 rknn_input_range 结构体形式返回，它包含了每个输入的名称、数据布局信息、shape 个数以及具体 shape。接着，通过调用 rknn_set_input_shapes() 接口，传入包含每个输入 shape 信

息的 `rknn_tensor_attr` 数组指针可以设置当前推理使用的 `shape`。在设置输入 `shape` 后，可以再次调用 `rknn_query()` 查询当前设置成功后的输入和输出 `shape`。

最后，按照通用 API 流程完成推理。每次切换输入 `shape` 时，需要再设置一次新的 `shape`，准备新 `shape` 大小的数据并再次调用 `rknn_inputs_set()` 接口。如果推理前不需要切换输入 `shape`，无需重复调用 `rknn_set_input_shapes()` 接口。

1. 初始化

调用 `rknn_init()` 接口初始化动态 `shape` RKNN 模型，

对于动态 `shape` RKNN 模型，在初始化上下文时有如下限制：

- 不支持权重共享功能（带 `RKNN_FLAG_SHARE_WEIGHT_MEM` 标志的初始化）。
- 不支持上下文复用功能（具体说明见 `rknn_dup_context` 接口）。

2. 查询 RKNN 模型支持的输入 shape 组合

初始化成功后，通过 `rknn_query()` 可以查询到 RKNN 模型支持的输入 `shape` 列表，每个输入支持的 `shape` 列表信息以 `rknn_input_range` 结构体形式返回，它包含了每个输入的名称，`layout` 信息，支持的 `shape` 个数以及具体 `shape`。C 代码示例如下：

```
// 查询模型支持的输入 shape
rknn_input_range dyn_range[io_num.n_input];
memset(dyn_range, 0, io_num.n_input *
sizeof(rknn_input_range));
for (uint32_t i = 0; i < io_num.n_input; i++)
{
    dyn_range[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_DYNAMIC_RANGE,
&dyn_range[i], sizeof(rknn_input_range));
    if (ret != RKNN_SUCC)
    {
        fprintf(stderr, "rknn_query error! ret=%d\n", ret);
        return -1;
    }
    dump_input_dynamic_range(&dyn_range[i]);
}
```

注意：对于多输入的模型，所有输入的 `shape` 按顺序一一对应，例如，有两个输入、多种 `shape` 的 RKNN 模型，第一个输入的第一个 `shape` 与第二个输入的第一个 `shape` 组合有效，不存在交叉的 `shape` 组合。例如，模型有两个输入 A 和 B，A 的 `shape` 分别是 [1,224] 和 [1,112]，B 的 `shape` 分别 [1,40] 和 [1,80]，此时，只支持以下两组输入 `shape` 的情况：

- A shape = [1,224], B shape=[1,40]
- A shape = [1,112], B shape=[1,80]

3.设置输入 shape

在首次设置输入数据或者输入数据 shape 发生改变时，需要调用 rknn_set_input_shapes() 接口动态修改输入 shape。加载动态 shape RKNN 模型后，可以在运行时动态修改输入的 shape。通过调用 rknn_set_input_shapes() 接口，传入所有输入的 rknn_tensor_attr 数组，每个 rknn_tensor_attr 中的 dims, n_dims 和 fmt 三个成员信息表示了当前次推理的 shape。C 代码示例如下：

```
/**
 * dynamic inputs shape range:
 * index=0, name=data, shape_number=2, range=[[1, 224, 224,
 * 3], [1, 112, 224, 3]], fmt = NHWC
 */
input_attrs[0].dims[0] = 1;
input_attrs[0].dims[1] = 224;
input_attrs[0].dims[2] = 224;
input_attrs[0].dims[3] = 3;
input_attrs[0].fmt=RKNN_TENSOR_NHWC;
ret = rknn_set_input_shapes(ctx, io_num.n_input, input_attrs);
if (ret < 0)
{
    fprintf(stderr, "rknn_set_input_shapes error! ret=%d\n",
ret);
    return -1;
}
```

其中，io_num.n_input 是输入数量,input_attrs 是模型输入的 rknn_tensor_attr 结构体数组。

注：这里设置的 shape 必须包含在第 2 步查询到的 shape 列表中。

在设置输入 shape 后，可以再次调用 rknn_query 查询当前设置成功后的输入和输出 shape，C 代码示例如下：

```
// 获取当前次推理的输入和输出 shape
rknn_tensor_attr cur_input_attrs[io_num.n_input];
memset(cur_input_attrs, 0, io_num.n_input *
sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_input; i++)
{
    cur_input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_CURRENT_INPUT_ATTR,
&(cur_input_attrs[i]), sizeof(rknn_tensor_attr));
    if (ret < 0)
    {
        printf("rknn_init error! ret=%d\n", ret);
        return -1;
    }
    dump_tensor_attr(&cur_input_attrs[i]);
}
rknn_tensor_attr cur_output_attrs[io_num.n_output];
memset(cur_output_attrs, 0, io_num.n_output *
sizeof(rknn_tensor_attr));
for (uint32_t i = 0; i < io_num.n_output; i++)
{
```

```

        cur_output_attrs[i].index = i;
        ret = rknn_query(ctx, RKNN_QUERY_CURRENT_OUTPUT_ATTR,
&(cur_output_attrs[i]), sizeof(rknn_tensor_attr));
        if (ret != RKNN_SUCC)
        {
            printf("rknn_query fail! ret=%d\n", ret);
            return -1;
        }
        dump_tensor_attr(&cur_output_attrs[i]);
    }

```

注意事项:

- rknn_set_input_shapes 接口要求输入 tensor 的 shape 为 4 维时, fmt 使用 NHWC, 非 4 维时使用 UNDEFINED。
- 在 rknn_set_input_shapes 尚未调用前, 使用带 RKNN_QUERY_CURRENT 前缀的命令查询的 shape 信息是无效的。

4.推理

在设置好当前输入 shape 后, 假设输入 Tensor 的 shape 信息保存在 cur_input_attrs 数组中, 以通用 API 接口为例, C 代码示例如下:

```

// 设置输入信息
rknn_input inputs[io_num.n_input];
memset(inputs, 0, io_num.n_input * sizeof(rknn_input));
for (int i = 0; i < io_num.n_input; i++)
{
    int height = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ?
cur_input_attrs[i].dims[1] : cur_input_attrs[i].dims[2];
    int width = cur_input_attrs[i].fmt == RKNN_TENSOR_NHWC ?
cur_input_attrs[i].dims[2] : cur_input_attrs[i].dims[3];
    cv::resize(imgs[i], imgs[i], cv::Size(width, height));
    inputs[i].index = i;
    inputs[i].pass_through = 0;
    inputs[i].type = RKNN_TENSOR_UINT8;
    inputs[i].fmt = RKNN_TENSOR_NHWC;
    inputs[i].buf = imgs[i].data;
    inputs[i].size = imgs[i].total() * imgs[i].channels();
}

// 将输入数据转换成正确的格式后, 放到输入缓冲区
ret = rknn_inputs_set(ctx, io_num.n_input, inputs);
if (ret < 0)
{
    printf("rknn_input_set fail! ret=%d\n", ret);
    return -1;
}

// 进行推理
printf("Begin perf ...\n");
double total_time = 0;
for (int i = 0; i < loop_count; ++i)
{

```

```
int64_t start_us = getCurrentTimeUs();
ret = rknn_run(ctx, NULL);
int64_t elapse_us = getCurrentTimeUs() - start_us;
if (ret < 0)
{
    printf("rknn run error %d\n", ret);
    return -1;
}
total_time += elapse_us / 1000.f;
printf("%4d: Elapse Time = %.2fms, FPS = %.2f\n", i,
elapse_us / 1000.f, 1000.f * 1000.f / elapse_us);
}
printf("Avg FPS = %.3f\n", loop_count * 1000.f / total_time);

// 获取输出结果
rknn_output outputs[io_num.n_output];
memset(outputs, 0, io_num.n_output * sizeof(rknn_output));
for (uint32_t i = 0; i < io_num.n_output; ++i)
{
    outputs[i].want_float = 1;
    outputs[i].index = i;
    outputs[i].is_prealloc = 0;
}

ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
if (ret < 0)
{
    printf("rknn_outputs_get fail! ret=%d\n", ret);
    return ret;
}

//释放输出缓冲区 buffer
ret = rknn_outputs_release(ctx, io_num.n_output, outputs);
```

5.4.4.2 零拷贝 API

对于零拷贝 API 而言，初始化成功后，通过 `rknn_query()` 可以查询 RKNN 模型支持的输入 shape 列表，调用 `rknn_create_mem()` 接口分配的输入和输出内存。接着，通过调用 `rknn_set_input_shapes()` 接口，传入包含每个输入 shape 信息的 `rknn_tensor_attr` 数组指针可以设置当前推理使用的 shape。在设置输入 shape 后，可以再次调用 `rknn_query()` 查询设置成功后的输入和输出 shape。最后，调用 `rknn_set_io_mem()` 接口设置需要的输入输出内存。每次切换输入 shape 时，需要再设置一次新的 shape，准备新 shape 大小的数据并再次调用 `rknn_set_io_mem()` 接口，如果推理前不需要切换输入 shape，无需重复调用 `rknn_set_input_shapes()` 接口。典型用法流程如下图所示：

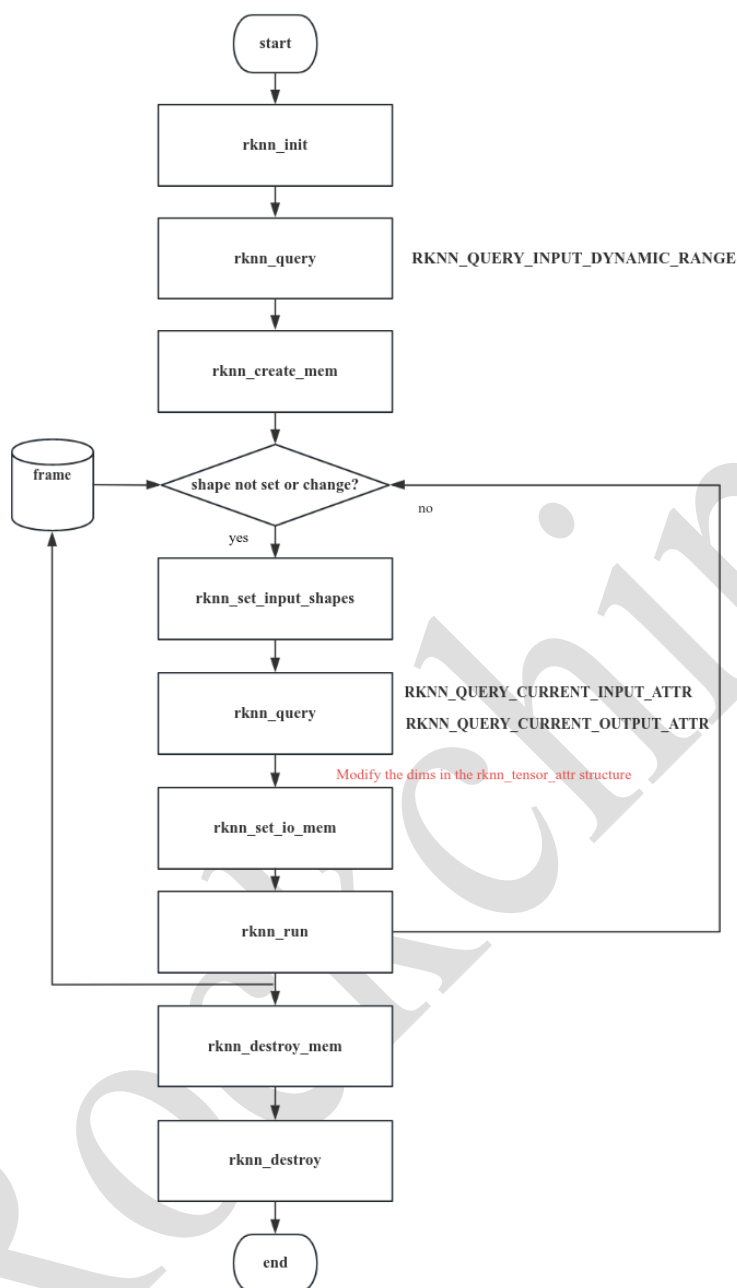


图 5-11 动态 shape 输入接口的零拷贝 API 调用流程

初始化、查询 RKNN 模型支持的输入 shape 组合、设置输入 shape 使用与上述通用 API 相同，此处不做赘述。不同之处在于，在设置输入 shape 后，使用的接口不同。零拷贝推理 C 代码示例如下：

```

// 创建最大的输入 tensor 内存
rknn_tensor_mem *input_mems[io_num.n_input];
for (int i = 0; i < io_num.n_input; i++)
{
    // default input type is int8 (normalize and quantize need
    compute in outside)
    // if set uint8, will fuse normalize and quantize to npu
    input_attrs[i].type = RKNN_TENSOR_UINT8;
    // default fmt is NHWC, npu only support NHWC in zero copy

```

```

mode
    input_attrs[i].fmt = RKNN_TENSOR_NHWC;

    input_mems[i] = rknn_create_mem(ctx,
input_attrs[i].size_with_stride);
}

// 创建最大的输出 tensor 内存
rknn_tensor_mem *output_mems[io_num.n_output];
for (uint32_t i = 0; i < io_num.n_output; ++i)
{
    // default output type is depend on model, this require
float32 to compute top5
    // allocate float32 output tensor
    int output_size = output_attrs[i].size * sizeof(float);
    output_mems[i] = rknn_create_mem(ctx, output_size);
}

// 加载输入并设置模型输入 shape, 每次切换输入 shape 要调用一次
for (int s = 0; s < shape_num; ++s)
{
    for (int i = 0; i < io_num.n_input; i++)
    {
        for (int j = 0; j < input_attrs[i].n_dims; ++j)
        {
            input_attrs[i].dims[j] =
shape_range[i].dyn_range[s][j];
        }
    }
    ret = rknn_set_input_shapes(ctx, io_num.n_input,
input_attrs);
    if (ret < 0)
    {
        fprintf(stderr, "rknn_set_input_shape error! ret=%d\n",
ret);
        return -1;
    }

    // 获取当前次推理的输入和输出 shape
    printf("current input tensors:\n");
    rknn_tensor_attr cur_input_attrs[io_num.n_input];
    memset(cur_input_attrs, 0, io_num.n_input *
sizeof(rknn_tensor_attr));
    for (uint32_t i = 0; i < io_num.n_input; i++)
    {
        cur_input_attrs[i].index = i;
        // query info
        ret = rknn_query(ctx, RKNN_QUERY_CURRENT_INPUT_ATTR,
&(cur_input_attrs[i]), sizeof(rknn_tensor_attr));
        if (ret < 0)
        {
            printf("rknn_init error! ret=%d\n", ret);
            return -1;
        }
        dump_tensor_attr(&cur_input_attrs[i]);
    }

    printf("current output tensors:\n");

```

```

    rknn_tensor_attr cur_output_attrs[io_num.n_output];
    memset(cur_output_attrs, 0, io_num.n_output *
sizeof(rknn_tensor_attr));
    for (uint32_t i = 0; i < io_num.n_output; i++)
    {
        cur_output_attrs[i].index = i;
        // query info
        ret = rknn_query(ctx, RKNN_QUERY_CURRENT_OUTPUT_ATTR,
&(cur_output_attrs[i]), sizeof(rknn_tensor_attr));
        if (ret != RKNN_SUCC)
        {
            printf("rknn_query fail! ret=%d\n", ret);
            return -1;
        }
        dump_tensor_attr(&cur_output_attrs[i]);
    }

    // 指定NPU 核心数量, 仅 3588/3576 支持
    rknn_set_core_mask(ctx, (rknn_core_mask)core_mask);

    // 设置输入信息
    rknn_input inputs[io_num.n_input];
    memset(inputs, 0, io_num.n_input * sizeof(rknn_input));
    std::vector<cv::Mat> resize_imgs;
    resize_imgs.resize(io_num.n_input);
    for (int i = 0; i < io_num.n_input; i++)
    {
        int height = cur_input_attrs[i].fmt ==
RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[1] :
cur_input_attrs[i].dims[2];
        int width = cur_input_attrs[i].fmt ==
RKNN_TENSOR_NHWC ? cur_input_attrs[i].dims[2] :
cur_input_attrs[i].dims[3];
        int stride = cur_input_attrs[i].w_stride;
        cv::resize(imgs[i], resize_imgs[i], cv::Size(width,
height));
        int input_size = resize_imgs[i].total() *
resize_imgs[i].channels();
        // 拷贝外部数据到零拷贝输入缓冲区
        if (width == stride)
        {
            memcpy(input_mems[i]->virt_addr,
resize_imgs[i].data, input_size);
        }
        else
        {
            int height = cur_input_attrs[i].dims[1];
            int channel = cur_input_attrs[i].dims[3];
            // copy from src to dst with stride
            uint8_t *src_ptr = resize_imgs[i].data;
            uint8_t *dst_ptr = (uint8_t
*)input_mems[i]->virt_addr;
            // width-channel elements
            int src_wc_elems = width * channel;
            int dst_wc_elems = stride * channel;
            for (int b = 0; b < cur_input_attrs[i].dims[0]; b++)
            {
                for (int h = 0; h < height; ++h)

```

```

        {
            memcpy(dst_ptr, src_ptr, src_wc_elems);
            src_ptr += src_wc_elems;
            dst_ptr += dst_wc_elems;
        }
    }
}

// 更新输入零拷贝缓冲区内存
for (int i = 0; i < io_num.n_input; i++)
{
    cur_input_attrs[i].type = RKNN_TENSOR_UINT8;
    ret = rknn_set_io_mem(ctx, input_mems[i],
&cur_input_attrs[i]);
    if (ret < 0)
    {
        printf("rknn_set_io_mem fail! ret=%d\n", ret);
        return -1;
    }
}

// 更新输出零拷贝缓冲区内存
for (uint32_t i = 0; i < io_num.n_output; ++i)
{
    // default output type is depend on model, this require
float32 to compute top5
    cur_output_attrs[i].type = RKNN_TENSOR_FLOAT32;
    cur_output_attrs[i].fmt = RKNN_TENSOR_NCHW;
    // set output memory and attribute
    ret = rknn_set_io_mem(ctx, output_mems[i],
&cur_output_attrs[i]);
    if (ret < 0)
    {
        printf("rknn_set_io_mem fail! ret=%d\n", ret);
        return -1;
    }
}

// 推理
printf("Begin perf ...\n");
double total_time = 0;
for (int i = 0; i < loop_count; ++i)
{
    int64_t start_us = getCurrentTimeUs();
    ret = rknn_run(ctx, NULL);
    int64_t elapse_us = getCurrentTimeUs() - start_us;
    if (ret < 0)
    {
        printf("rknn run error %d\n", ret);
        return -1;
    }
    total_time += elapse_us / 1000.f;
    printf("%4d: Elapse Time = %.2fms, FPS = %.2f\n", i,
elapse_us / 1000.f, 1000.f * 1000.f / elapse_us);
}
printf("Avg FPS = %.3f\n", loop_count * 1000.f /
total_time);

```

}

Rockchip

注意事项:

1. `rknn_set_io_mem()`接口在动态 shape 情况下, 输入 buffer 的 shape 和大小说明:
 - 初始化完成后和调用 `rknn_set_input_shapes()` 接口前, `rknn_query()` 接口使用 `RKNN_QUERY_INPUT_ATTR` 和 `RKNN_QUERY_OUTPUT_ATTR` 查询输入和输出 Tensor 的 shape 通常是最大的, 用户可以使用这两个命令获取的大小来分配输入和输出内存。若遇到多输入模型, 部分输入的 shape 可能不是最大的, 此时需要搜索支持的 shape 中最大的规格, 并分配最大的输入和输出内存。
 - 如果输入是非 4 维度, 使用 `fmt=UNDEFINED`, 传递原始模型输入 shape 的 buffer, 大小则根据输入 shape 和 type 计算得到。
 - 如果输入是 4 维度, 支持使用 `fmt=NHWC` 或者 `NC1HWC2`, 传递 `NHWC` 或者 `NC1HWC2` shape 和对应 size 的 buffer(通过 `rknn_query` 查询相应字段获取 shape 和 size)。
 - `rknn_query()` 接口中, 标志位为 `RKNN_QUERY_CURRENT_INPUT_ATTR` 和 `RKNN_QUERY_CURRENT_OUTPUT_ATTR` 时获取原始模型输入/输出的 shape, 其格式为 `NHWC` 或者 `UNDEFINED`; 标志位为 `RKNN_QUERY_CURRENT_NATIVE_INPUT_ATTR` 和 `RKNN_QUERY_CURRENT_NATIVE_OUTPUT_ATTR` 时获取 NPU 以最优性能读取数据时模型输入/输出的 shape, 其格式为 `NHWC` 或者 `NC1HWC2`。
2. `rknn_set_io_mem()`接口中使用的 buffer 排列格式为 `NHWC` 时, `rknn_tensor_attr` 中的 shape 和 `fmt` 需按照 `RKNN_QUERY_CURRENT_INPUT_ATTR` 查询到的信息进行设置; 如果使用的 buffer 排列格式为 `NC1HWC2` 时, 需要按照 `RKNN_QUERY_CURRENT_NATIVE_INPUT_ATTR` 查询到的信息进行设置。
完整的动态 shape C API Demo 请参考 (https://github.com/airockchip/rknn-toolkit2/tree/master/rknpu2/examples/rknn_dynamic_shape_input_demo)

5.5 自定义算子

5.5.1 自定义算子介绍

RKNN SDK 提供了一种自定义算子的机制, 它允许开发者在 RKNN 模型的推理阶段定义和执行自定义的算子。通过实现自定义算子, 开发者可以扩展模型功能, 并且针对特定硬件 (CPU 或者 GPU) 进行优化, 以充分利用硬件资源并提高推理速度。同时, 开发自定义算子需要深刻的理解深度学习计算原理和目标硬件平台的特性, 以确保正确性和性能。

目前只支持 ONNX 模型自定义算子。

RKNN 自定义算子主要包括两大步骤：

- 使用 RKNN-Toolkit2 注册自定义算子并导出 RKNN 模型。
- 编写自定义算子的 C 代码实现，通过 RKNN API 加载注册并执行。

整体流程如下图所示：

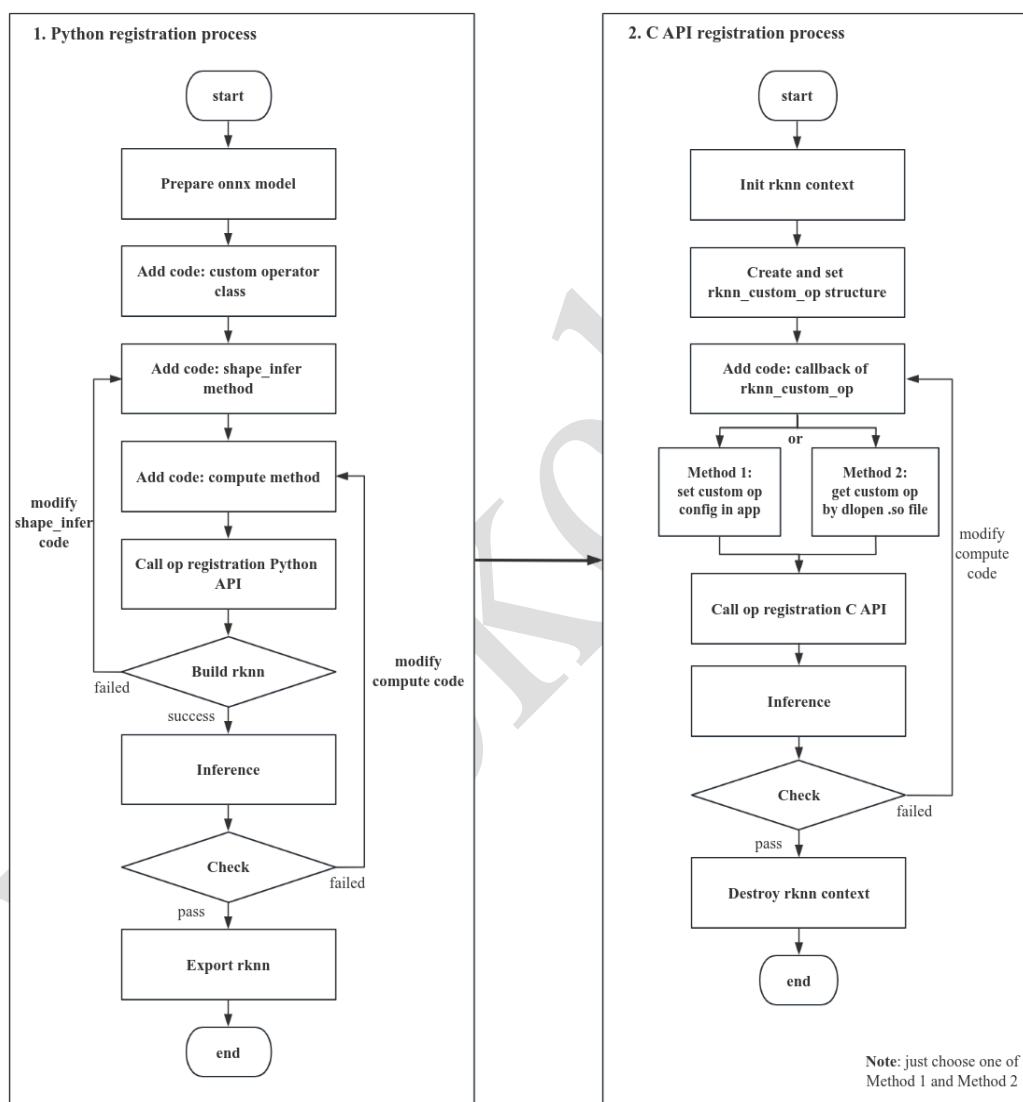


图 5-12 注册自定义算子的完整流程

5.5.2 整体流程介绍

5.5.2.1 使用 RKNN-Toolkit2 注册自定义算子并导出 RKNN 模型

1. 准备 ONNX 模型: 按照 ONNX 模型标准规范, 用户设计自定义算子的 op 类型,名字, op 属性,输入/输出数量,并将该算子插入到 ONNX 中的拓扑图位置, 用户使用 ONNX 包提供的 api 设计和导出 ONNX 模型。

2. 实现自定义算子 Python 类, 类里主要包括 `shape_infer()` 和 `compute()` 两个函数接口, 并调用 `rknn.reg_custom_op()` 注册该算子。
3. 如果 `rknn.build()` 执行成功, 可以进行推理, 否则需要检查步骤 2 的 `shape_infer()` 代码实现。然后运行仿真, 如果仿真结果正确, 可以调用 `rknn.export_rknn()` 接口导出 RKNN 模型, 否则需要检查步骤 2 的 `compute()` 代码。

5.5.2.2 编写自定义算子的 C 代码实现, 通过 RKNN API 加载注册并执行

1. 根据 `rknn_custom_op.h` 的 `rknn_custom_op` 类, 编写自定义算子的 C 代码实现, 编写完成后, 填写 `rknn_custom_op` 类的信息。
2. 调用 `rknn_register_custom_ops()` 注册 `rknn_custom_op` 类的信息。
3. 参考通用 API 或零拷贝 API 的流程, 正常构建、推理模型, 可以开启模型详细日志和 Dump 功能确认自定义算子实现的正确性。

5.5.2.3 使用 RKNN-Toolkit2 连板推理或精度分析

若用户需要对包含自定义算子的模型做 Python 连板精度分析, 需要将自定义算子的回调函数实现代码编译成 so 后, 放在指定的路径, 并重启 RKNN Server。具体参考 [5.5.4.4](#) 章节。

5.5.3 Python 端处理

目前只有 ONNX 模型支持自定义算子, 支持用户添加非 ONNX 标准的算子。

添加非 ONNX 标准的自定义算子用于新增一个不存在于 ONNX 算子列表内的新算子, 该算子除了要满足 ONNX spec 规范以外, 还要满足以下规则:

- 算子的 `op_type` 不能与 ONNX 标准算子相同, 推荐以 "cst" 字符开头。
- 算子与其他算子必须要有连接关系, 包含各个输入/输出的 `shape`, 数据类型等。
- 算子输入属性, 支持 `bool`、`int32`、`float32`、`int64` 类型的单值或者数组。
- 算子常量输入, 支持 `bool`、`int32`、`float32`、`int64` 类型, 该类型指未量化 ONNX 模型的数据类型。

因为非 ONNX 标准算子并不是 ONNX SPEC 内的标准算子, 所以用户需要自行通过 ONNX 的 API 或其他框架的 API 来构建并导出一个包含非 ONNX 标准算子的 ONNX 模型。

这边为方便起见, 以一个简单的修改 Softmax 的定义为示例, 来构建一个包含非

ONNX 标准算子 `cstSoftmax` 的 ONNX 模型，修改方法如下：

```
import onnx

path="test_softmax.onnx"
model=onnx.load(path)

for node in model.graph.node:
    if node.op_type == "Softmax":
        node.op_type = "cstSoftmax"
        ... # 修改 cstSoftmax 的属性定义等
onnx.save(model, "./test_softmax_custom.onnx")
```

在构建完包含自定义算子的 ONNX 模型后，可使用 `Netron` 打开该 ONNX 模型，并检查该自定义算子是否符合自定义算子的规范，满足规范之后，就可以实现该自定义算子的算子类，具体实现如下（以自定义 `Softmax` 为例）：

```
import numpy as np
from rknn.api.custom_op import get_node_attr
class cstSoftmax:
    op_type = 'cstSoftmax'
    def shape_infer(self, node, in_shapes, in_dtypes):
        out_shapes = in_shapes.copy()
        out_dtypes = in_dtypes.copy()
        return out_shapes, out_dtypes
    def compute(self, node, inputs):
        x = inputs[0]
        axis = get_node_attr(node, 'axis')
        x_max = np.max(x, axis=axis, keepdims=True)
        tmp = np.exp(x - x_max)
        s = np.sum(tmp, axis=axis, keepdims=True)
        outputs = [tmp / s]
        return outputs
```

- 自定义算子必须为一个 Python 类。
- 自定义算子类必须包含一个名为 `op_type` 的字符串变量，与构建的 ONNX 中自定义算子类型名一致。
- 自定义算子类必须包含成员函数 `shape_infer(self, node, in_shapes, in_dtypes)`，函数名、参数名都必须一致，否则报错。该函数用于自定义算子的 `shape` 推理，其中，`node` 为 ONNX 的算子节点对象，该对象里包含了自定义算子的属性和输入输出信息；`in_shapes` 为该算子所有输入的 `shape` 信息，格式为 `[shape_0, shape_1, ...]`，列表内的 `shape` 的类型为列表；`in_dtypes` 为该算子所有输入的 `dtype` 信息，格式为 `[dtype_0, dtype_1, ...]`，列表内的 `dtype` 的类型为 `numpy` 的 `dtype` 类型。另外该函数需要返回该算子所有输出的 `shape` 信息和 `dtype` 信息，格式与 `in_shapes` 和 `in_dtypes` 一致。
- 自定义算子类必须包含成员函数 `compute(self, node, inputs)`，函数名和参数名都必

须一致，否则报错。该函数用于自定义算子的推理。其中，`node` 为 ONNX 的算子节点对象，该对象里包含了自定义算子的属性和输入输出信息；`inputs` 为该算子的输入数据，格式为`[array_0, array_1, ...]`，列表内的 `array` 的类型为 `numpy` 的 `ndarray` 类型。另外该函数需要返回该算子所有输出的数据，格式与 `inputs` 一致。

- 如自定义算子含有自定义的属性，可通过 `from rknn.api.custom_op import get_node_attr` 来获取自定义算子的属性值。

在编写完自定义算子类后，可以通过 `rknn.reg_custom_op()` 进行算子类的注册，注册完后，就可以调用 `rknn.build()` 转换并生成 RKNN 模型。自定义算子类可以确保模型的转换和推理等功能的正常。具体实现如下（以自定义 Softmax 为例）：

```
from rknn.api import RKNN
# Create RKNN object
rknn = RKNN(verbose=True)

# Pre-process config
print('--> Config model')
rknn.config(mean_values=[103.94, 116.78, 123.68],
            std_values=[58.82, 58.82, 58.82],
            quant_img_RGB2BGR=True, target_platform='rk3566')
print('done')

print('--> Register cstSoftmax op')
ret = rknn.reg_custom_op(cstSoftmax())
if ret != 0:
    print('Register cstSoftmax op failed!')
    exit(ret)
print('done')

print('--> Loading model')
ret = rknn.load_onnx(model='mobilenet_v2.onnx')
if ret != 0:
    print('Load model failed!')
    exit(ret)
print('done')

# Build model
print('--> Building model')
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
if ret != 0:
    print('Build model failed!')
    exit(ret)
print('done')
```

`rknn.reg_custom_op()` 需要在 `rknn.config()` 和 `rknn.load_xxx()` 之间调用。

5.5.4 C API 部署

在得到带自定义算子的 RKNN 模型后，开始调用 C API 部署。首先，自定义算子的结

构体和接口位于 `rknn_custom_op.h` 头文件，开发者程序需要包含该头文件。注册使用自定义算子的流程如下图所示：

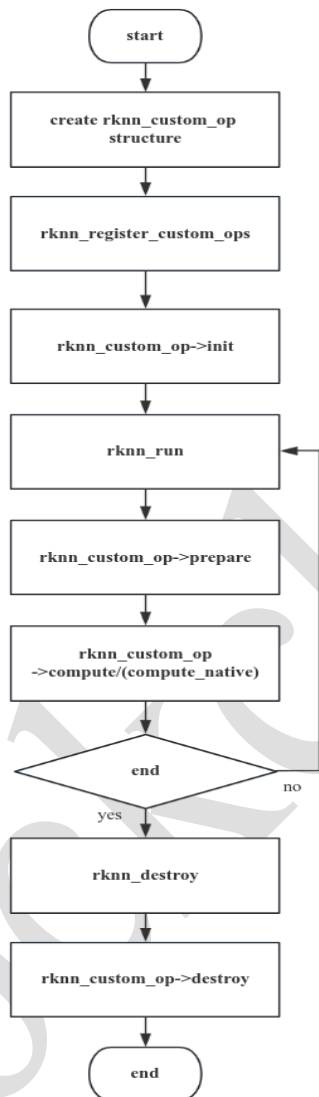


图 5-13 注册自定义算子的 C API 调用流程

5.5.4.1 初始化自定义算子结构体

创建 `rknn` 上下文后，开发者需要创建 `rknn_custom_op` 结构体，设置自定义算子信息。

算子信息包含以下内容：

- **version**: 算子的版本号。
- **target**: 算子的执行后端设备，目前支持 CPU 和 GPU。
- **op_type**: 算子的类型，与 ONNX 模型中的类型字段相同。
- **cl_kernel_name**: OpenCL 代码的 `cl_kernel` 函数名。注册 GPU 算子时必须配置。
- **cl_kernel_source**: 自定义算子的 `.cl` 文件全路径或者 OpenCL kernel 字符串。当

cl_source_size=0, 它表示.cl 文件全路径; 当 cl_source_size>0, 它表示 OpenCL kernel 代码字符串。注册 GPU 算子时必须配置。

- cl_source_size: cl_kernel_source 的大小。大小等于 0 是特殊情况, 它表示 cl_kernel_source 是路径。
- cl_build_options: OpenCL kernel 编译选项, 以字符串形式传入。注册 GPU 算子时必须配置。
- init: 可选, 在 rknn_register_custom_ops 被调用一次。
- prepare: 可选, 它是预处理回调函数, 每次 rknn_run 都会执行 prepare 和 compute/compute_native 回调, 执行顺序是 prepare 在前, compute/compute_native 在后。
- compute: 必须实现, 算子运算回调函数, 它的输入/输出都是 NCHW 的 float32 格式数据 (ONNX 模型如果指定输入/输出为 int64 的数据类型, 则 int64 格式数据)
- compute_native: 保留, 请设置成 NULL。
- destroy: 可选, rknn_destory 中执行一次。
- init/prepare/compute 回调函数参数定义规范如下:
 - rknn_custom_op_context* op_ctx: op 回调函数的上下文信息
 - rknn_custom_op_tensor* inputs: op 输入 tensor 数据和信息
 - uint32_t n_inputs: op 输入个数
 - rknn_custom_op_tensor* outputs: op 输出 tensor 数据和信息
 - uint32_t n_outputs: op 输出个数
- destroy 回调函数仅 rknn_custom_op_context* op_ctx 一个参数。

rknn_custom_op_context

包含 target (执行后端设备)、GPU 上下文、自定义算子私有上下文以及 priv_data, 其中 priv_data 的由开发者自行管理 (赋值, 读写, 销毁), GPU 上下文包含 cl_context、cl_command_queue、cl_kernel 指针, 可以通过强制类型转换得到对应的 OpenCL 对象。

priv_data 是一个用户可选是否配置的指针, 通常的用法是用户在 init()回调函数内创建资源, 并将 priv_data 指向该段内存地址, 在 prepare()/compute()回调函数中操作, 最终在 destroy()回调函数内销毁资源。

rknn_custom_op_tensor

表示输入/输出 tensor 的信息, 包含 tensor 的名称、形状、大小、量化参数、虚拟基地址、

fd、数据偏移等信息。

用户在回调 `compute()`回调函数内无需创建该算子的输入和输出 `tensor` 内存。虚拟地址对应的数据在进入 `compute()`回调函数时已经准备好。虚拟地址的计算公式是 **Tensor 的有效地址=虚拟基地址+数据偏移**，`mem` 成员的 `virt_addr` 表示虚拟基地址，`mem` 成员的 `offset` 表示数据偏移(以字节为单位)。用户在回调函数内可以读取输入 `tensor` 的有效地址，该指向前一层算子已经计算后的输出数据，输出 `tensor` 的有效地址指向即将送给下一层算子的输入。

rknn_custom_op_attr

开发者通过调用 `rknn_custom_op_get_op_attr()`接口传入属性字段获得属性信息，属性信息用 `rknn_custom_op_attr` 表示，`rknn_custom_op_attr` 中的 `void` 类型 `buffer`，`dtype` 以及元素数量表示一块内存段，开发者根据 `dtype` 使用 C/C++ 将 `buffer` 强制转换指针类型可以得到相应数值类型的数组。

5.5.4.1.1 init 回调函数

常用于解析算子信息或初始化临时缓冲区或者输入/输出缓冲区 `buffer`。分配临时 `buffer` 的 `init` 回调函数示例代码如下：

● CPU 算子

```
/**
 * cpu kernel init callback for custom op
 */
int custom_op_init_callback(rknn_custom_op_context* op_ctx,
rknn_custom_op_tensor* inputs, uint32_t n_inputs,
rknn_custom_op_tensor* outputs, uint32_t
n_outputs)
{
    printf("custom_op_init_callback\n");
    // create tmp buffer
    float* tmp_buffer = (float*)malloc(inputs[0].attr.n_elems *
sizeof(float));
    op_ctx->priv_data = tmp_buffer;
    return 0;
}
```

● GPU 算子


```
/**
 * opengl kernel init callback for custom op
 * */
int relu_init_callback_gpu(rknn_custom_op_context* op_ctx,
rknn_custom_op_tensor* inputs, uint32_t n_inputs,
rknn_custom_op_tensor* outputs, uint32_t n_outputs)
{
    printf("relu_init_callback_gpu\n");
    // 获取 opengl context
    cl_context cl_ctx = (cl_context)op_ctx->gpu_ctx.cl_context;

    // create tmp cl buffer
    cl_mem* memObject = (cl_mem*)malloc(sizeof(cl_mem) * 2);
    memObject[0] = clCreateBuffer(cl_ctx, CL_MEM_READ_WRITE,
inputs[0].attr.size, NULL, NULL);
    memObject[1] = clCreateBuffer(cl_ctx, CL_MEM_READ_WRITE,
outputs[0].attr.size, NULL, NULL);
    op_ctx->priv_data = memObject;
    return 0;
}
```

5.5.4.1.2 prepare 回调函数

该回调函数每帧推理都会调用，目前为预留实现。

5.5.4.1.3 compute 回调函数

它是自定义算子的计算函数, 开发者必须完成输入/输出是 NCHW 或 UNDEFINED 格式 float32 数据类型输入输出的核函数。

1. compute 回调 (CPU) 假设开发者想实现一个自定义层, 完成 softmax 功能, CPU 算子 compute 函数示例如下:

```

/**
 * float32 kernel implemetation sample for custom op
 * */
int compute_custom_softmax_float32(rknn_custom_op_context* op_ctx,
rknn_custom_op_tensor* inputs, uint32_t n_inputs,
rknn_custom_op_tensor* outputs, uint32_t
n_outputs)
{
    unsigned char*    in_ptr    = (unsigned char*)inputs[0].mem.virt_addr +
inputs[0].mem.offset;
    unsigned char*    out_ptr   = (unsigned char*)outputs[0].mem.virt_addr +
outputs[0].mem.offset;
    int                axis      = 0;
    const float*       in_data   = (const float*)in_ptr;
    float*             out_data  = (float*)out_ptr;
    std::string        name      = "";
    rknn_custom_op_attr op_attr;
    rknn_custom_op_get_op_attr(op_ctx, "name", &op_attr);
    if (op_attr.n_elems > 0 && op_attr.dtype == RKNN_TENSOR_UINT8) {
        name = (char*)op_attr.data;
    }

    rknn_custom_op_attr op_attr;
    rknn_custom_op_get_op_attr(op_ctx, "axis", &op_attr);
    if (op_attr.n_elems == 1 && op_attr.dtype == RKNN_TENSOR_INT64) {
        axis = ((int64_t*)op_attr.data)[0];
    }

    printf("op name = %s, axis = %d\n", name.c_str(), axis);
    float* tmp_buffer = (float*)op_ctx->priv_data;
    // kernel implemetation for custom op
    {
        int inside = 1;
        int outside = 1;
        int channel = 1;

        while (axis < 0) {
            axis += inputs[0].attr.n_dims;
        }

        for (int i = 0; i < axis; i++) {
            outside *= inputs[0].attr.dims[i];
        }
        channel = inputs[0].attr.dims[axis];
        for (int i = axis; i < inputs[0].attr.n_dims; i++) {
            inside *= inputs[0].attr.dims[i];
        }

        for (int y = 0; y < outside; y++) {
            const float* src_y = in_data + y * inside;
            float* dst_y = out_data + y * inside;
            float max_data = -FLT_MAX;
            float sum_data = 0.0f;

            for (int i = 0; i < inside; ++i) {
                max_data = fmaxf(max_data, src_y[i]);
            }
            for (int i = 0; i < inside; ++i) {
                tmp_buffer[i] = expf(src_y[i] - max_data);
                sum_data += tmp_buffer[i];
            }
            for (int i = 0; i < inside; ++i) {
                dst_y[i] = tmp_buffer[i] / sum_data;
            }
        }
    }
    return 0;
}

```

2. compute 回调函数（GPU）

对于 GPU 算子，开发者可以在回调函数中完成以下步骤：

- 开发者从 rknn_custom_op_context 里的 gpu_ctx 中获取 opengl 的 cl_context，cl_command_queue 以及 cl_kernel 对象，此过程需要开发者做数据类型转换。
- 如有必要，用户自行创建的 op 输入或输出的 cl_mem 对象缓冲区。
- 设置 cl_kernel 的函数参数。
- OpenCL kernel 的函数参数的输入 buffer 数据目前只能支持 float，其他类型暂时还不支持。
- 对于使用零拷贝的情况下，调用 clImportMemoryARM 可以自行协助用户把输入 tensor 的内存映射到 OpenCL 的 cl_mem 结构体中，输入 tensor 已包含输入数据，用户不需要自行再拷贝一次。该过程也可以在 init 回调函数中处理，然后将 cl_mem 结构体记录到 priv_data 成员，最后在 compute 回调中读取 priv_data 并使用它。
- 以阻塞的形式运行 cl_kernel。
- CL kernel 内的输入数据都是以 NCHW 形式排布给出。
- 如果在 GPU 运算完后，开发者需要 CPU 访问数据，需要通过调用 rknn_mem_sync 函数刷新输出 Tensor 的 cache 后再读取数据。

假设开发者想实现一个自定义层，完成 relu 功能，GPU 算子 compute 函数示例如下：

```
/**
 * opengl kernel init callback for custom op
 * */
int compute_custom_relu_float32(rknn_custom_op_context* op_ctx,
rknn_custom_op_tensor* inputs, uint32_t num_inputs,
rknn_custom_op_tensor* outputs,
uint32_t num_outputs)
{
    std::string name = "";
    rknn_custom_op_attr op_name;
    rknn_custom_op_get_op_attr(op_ctx, "name", &op_name);
    if (op_name.n_elems > 0 && op_name.dtype == RKNN_TENSOR_UINT8) {
        name = (char*)op_name.data;
    }

    // get context
    cl_context cl_ctx = (cl_context)op_ctx->gpu_ctx.cl_context;

    // get command queue
    cl_command_queue queue =
    (cl_command_queue)op_ctx->gpu_ctx.cl_command_queue;

    // get kernel
    cl_kernel kernel = (cl_kernel)op_ctx->gpu_ctx.cl_kernel;
```

```
// import input/output buffer
const cl_import_properties_arm props[3] = {
    CL_IMPORT_TYPE_ARM,
    CL_IMPORT_TYPE_DMA_BUF_ARM,
    0,
};

cl_int status;
cl_mem inObject = clImportMemoryARM(cl_ctx, CL_MEM_READ_WRITE,
props, &inputs[0].mem.fd,
                                inputs[0].mem.offset +
inputs[0].mem.size, &status);
if (status != CL_SUCCESS) {
    printf("Tensor: %s clImportMemoryARM failed\n",
inputs[0].attr.name);
}
cl_mem outObject = clImportMemoryARM(cl_ctx, CL_MEM_READ_WRITE,
props, &outputs[0].mem.fd,
                                outputs[0].mem.offset +
outputs[0].mem.size, &status);
if (status != CL_SUCCESS) {
    printf("Tensor: %s clImportMemoryARM failed\n",
outputs[0].attr.name);
}

int      in_type_bytes = get_type_bytes(inputs[0].attr.type);
int      out_type_bytes =
get_type_bytes(outputs[0].attr.type);
int      in_offset      = inputs[0].mem.offset /
in_type_bytes;
int      out_offset      = outputs[0].mem.offset /
out_type_bytes;
unsigned int elems      = inputs[0].attr.n_elems;

// set kernel args
int argIndex = 0;
clSetKernelArg(kernel, argIndex++, sizeof(cl_mem), &inObject);
clSetKernelArg(kernel, argIndex++, sizeof(cl_mem), &outObject);
clSetKernelArg(kernel, argIndex++, sizeof(int), &in_offset);
clSetKernelArg(kernel, argIndex++, sizeof(int), &out_offset);
clSetKernelArg(kernel, argIndex++, sizeof(unsigned int),
&elems);

// set global worksize
const size_t global_work_size[3] = {elems, 1, 1};

// enqueueNDRangeKernel
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
NULL, 0, NULL, NULL);

// finish command queue
clFinish(queue);

// //cpu access data after sync to device
// rknn_mem_sync(&outputs[0].mem, RKNN_MEMORY_SYNC_FROM_DEVICE);
// // save output npy
// char output_path[PATH_MAX];
// sprintf(output_path, "%s/cpu_output%d.npy", ".", 0);
```

```
// unsigned char* out_data = (unsigned
char*)outputs[0].mem.virt_addr+outputs[0].mem.offset;
// save_npy(output_path, (float*)out_data, &inputs[0].attr);
return 0;
}
```

5.5.4.1.4 destroy 回调函数

常用于销毁自定义算子的临时缓冲区或输入/输出 buffer。销毁临时 buffer 的示例代码如下：

- CPU 算子

```
/**
 * cpu kernel destroy callback for custom op
 */
int custom_op_destroy_callback(rknn_custom_op_context* op_ctx)
{
    printf("custom_op_destroy_callback\n");
    // clear tmp buffer
    free(op_ctx->priv_data);
    return 0;
}
```

- GPU 算子

```
/**
 * opengl kernel destroy callback for custom op
 */
int relu_destroy_callback_gpu(rknn_custom_op_context* op_ctx)
{
    // clear tmp buffer
    printf("relu_destroy_callback_gpu\n");
    cl_mem* memObject = (cl_mem*)op_ctx->priv_data;
    clReleaseMemObject(memObject[0]);
    clReleaseMemObject(memObject[1]);
    free(memObject);
    return 0;
}
```

5.5.4.2 注册自定义算子

在设置完 rknn_custom_op 结构体后，需要调用 rknn_register_custom_ops() 将其注册到 rknn_context 中，该接口支持同时注册多个自定义算子。

在完成 CPU 的 compute 回调函数后，注册一个名为“cstSoftmax”和“ArgMax”的 CPU 自定义算子的代码示例如下：

- CPU 算子

```
// CPU operators
rknn_custom_op user_op[2];
memset(user_op, 0, 2 * sizeof(rknn_custom_op));
strncpy(user_op[0].op_type, "cstSoftmax", RKNN_MAX_NAME_LEN - 1);
user_op[0].version = 1;
user_op[0].target = RKNN_TARGET_TYPE_CPU;
user_op[0].init = custom_op_init_callback;
user_op[0].compute = compute_custom_softmax_float32;
user_op[0].destroy = custom_op_destroy_callback;

strncpy(user_op[1].op_type, "ArgMax", RKNN_MAX_NAME_LEN - 1);
user_op[1].version = 1;
user_op[1].target = RKNN_TARGET_TYPE_CPU;
user_op[1].init = custom_op_init_callback;
user_op[1].compute = compute_custom_argmax_float32;
user_op[1].destroy = custom_op_destroy_callback;

ret = rknn_register_custom_ops(ctx, user_op, 2);
if (ret < 0) {
    printf("rknn_register_custom_ops fail! ret = %d\n", ret);
    return -1;
}
```

● GPU 算子

对于 GPU 算子而言，支持以常量字符串或者文件路径的两种方式注册 OpenCL kernel。当 rknn_custom_op 结构体中的 cl_source_size 等于 0 时，cl_kernel_source 表示 OpenCL kernel 的文件路径，当 cl_source_size 大于 0 时 cl_kernel_source 表示 OpenCL kernel 函数字符串。以字符串保存的 relu 功能的 OpenCL kernel 的示例代码如下：

```
char* cl_kernel_source = "#pragma OPENCL EXTENSION cl_arm_printf : enable\n"
    "#pragma OPENCL EXTENSION cl_khr_fp16 : enable\n"
    "__kernel void relu_float(__global const float* input, __global float* output, int\n"
    "in_offset, int out_offset, const unsigned int elems) {\n"
    "{\n"
    "    int gid = get_global_id(0);\n"
    "    if (gid < elems) {\n"
    "        float in_value = input[in_offset + gid];\n"
    "        output[out_offset + gid] = in_value >= 0.f ? in_value : 0.f;\n"
    "    }\n"
    "}\n"
    "__kernel void relu_half(__global const half* input, __global half* output, int in_offset, int\n"
    "out_offset, const unsigned int elems) {\n"
    "{\n"
    "    int gid = get_global_id(0);\n"
    "    if (gid < elems) {\n"
    "        half in_value = input[in_offset + gid];\n"
    "        output[out_offset + gid] = in_value >= 0.f ? in_value : 0.f;\n"
    "    }\n"
    "}\n";
```

在完成 OpenCL kernel 函数以及 GPU 的 compute 回调函数后，可以设置 rknn_custom_op 结构体数组并注册 GPU 算子，注册 GPU 算子示例代码如下：

```
// GPU operators
rknn_custom_op user_op[1];
memset(user_op, 0, sizeof(rknn_custom_op));
strncpy(user_op->op_type, "cstSoftmax", RKNN_MAX_NAME_LEN - 1);
user_op->version = 1;
user_op->target = RKNN_TARGET_TYPE_GPU;
user_op->init = relu_init_callback_gpu;
user_op->compute = compute_custom_relu_float32;
user_op->destroy = relu_destroy_callback_gpu;
#ifdef LOAD_FROM_PATH
user_op->cl_kernel_source = "./custom_op.cl";
user_op->cl_source_size = 0;
#else
user_op->cl_kernel_source = cl_kernel_source;
user_op->cl_source_size = strlen(cl_kernel_source);
#endif
strcpy(user_op->cl_kernel_name, "relu_float");

ret = rknn_register_custom_ops(ctx, user_op, 1);
if (ret < 0) {
    printf("rknn_register_custom_ops fail! ret = %d\n", ret);
    return -1;
}
```

注册调用该接口前要明确自定义算子的 op_type，准备好算子信息并配置 rknn_custom_op 数组。每个类型的自定义算子要调用一次注册接口，网络中同一类型的算子仅调用一次。

5.5.4.3 模型推理

在注册完所有算子后，可以使用通用 API 或零拷贝 API 流程完成推理。

5.5.4.4 连板精度分析

自定义算子的连板调试功能要求 rknn_server 版本 \geq 1.6.0。

连板调试时，RKNN Server 会采用 dlopen 的方式从特定目录打开用户编译好的自定义算子插件库来获取算子信息，对于插件库方式注册自定义算子，要求用户必须实现一个名为 get_rknn_custom_op 的函数。

若用户需要对包含自定义算子的模型做连板精度分析，具体步骤如下：

1. 实现一个 get_rknn_custom_op()函数和必须的回调函数，并编译成对应系统的库，编译的插件库名称必须以"librkst_"为前缀，例如库名是 librkst_relu.so。
2. 插件放到/vendor/lib64/（Android arm64-v8a）或/usr/lib/rknpu/op_plugins（Linux）
3. 主机端或者上位机使用 RKNN-Toolkit2 的 Python 接口执行连板精度分析。

get_rknn_custom_op 函数的示例代码如下：

```
/**
 * To obtain operator information to be registered, a plugin
 library must
 * have one and only one of this function.
 */
RKNN_CUSTOM_OP_EXPORT rknn_custom_op* get_rknn_custom_op()
{
    // register a custom op
    memset(&user_op, 0, sizeof(rknn_custom_op));
    strncpy(user_op.op_type, "cstSoftmax", RKNN_MAX_NAME_LEN - 1);
    user_op.version = 1;
    user_op.target = RKNN_TARGET_TYPE_CPU;
    user_op.init = custom_op_init_callback;
    user_op.compute = compute_custom_softmax_float32;
    user_op.destroy = custom_op_destroy_callback;
    return &user_op;
}
```

get_rknn_custom_op 函数必须增加 RKNN_CUSTOM_OP_EXPORT 属性,让插件导出 get_rknn_custom_op 函数符号。

RKNN Server 调用 dlopen 获取特定目录的所有 so 库文件句柄,再使用 dlsym 函数获取 get_rknn_custom_op 函数指针,调用该函数指针即可获取自定义算子结构体,示例代码如下:

```
std::vector<std::string> get_all_plugin_paths(std::string
plugin_dir)
{
    std::vector<std::string> plugin_paths;
    if (access(plugin_dir.c_str(), 0) != 0) {
        fprintf(stderr, "Can not access plugin directory: %s, please
check it!\n", plugin_dir.c_str());
    }

    DIR*          dir;
    struct dirent* ent;
    const char*    prefix = RKNN_CSTOP_PLUGIN_PREFIX; // 所有库文件名应
该以此前缀开头

    if ((dir = opendir(plugin_dir.c_str())) != NULL) {
        while ((ent = readdir(dir)) != NULL) {
            if (ent->d_type == DT_REG) {
                const char* filename = ent->d_name;
                size_t      len      = strlen(filename);

                if (len > 10 && strncmp(filename, prefix, strlen(prefix))
== 0) {
                    printf("Found plugin: %s file in %s\n", filename,
plugin_dir.c_str());
                    plugin_paths.push_back(plugin_dir + "/" + filename);
                }
            }
        }
        closedir(dir);
    } else {
        fprintf(stderr, "Unable to open directory");
    }
}
```



```

    }

    return plugin_paths;
}

// the default path of the custom operator plugin libraries
std::string plugin_dir =
#ifdef __ANDROID__
#   if defined(__aarch64__)
        "/vendor/lib64/";
#   else
        "/vendor/lib/";
#   endif // __aarch64__
#elif defined(__linux__)
        "/usr/lib/rknpu/op_plugins/";
#endif

std::vector<std::string> plugin_paths =
get_all_plugin_paths(plugin_dir);
std::vector<void*> so_handles;
for (auto path : plugin_paths) {
    printf("load plugin %s\n", path.c_str());
    void* plugin_lib = dlopen(path.c_str(), RTLD_NOW);
    char* error = dlerror();
    if (error != NULL) {
        fprintf(stderr, "dlopen %s fail: %s.\nPlease try to set
'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:%s'\n",
                path.c_str(), error, plugin_dir.c_str());
        dlclose(plugin_lib);
        return -1;
    }
    printf("dlopen %s successfully!\n", path.c_str());
    get_custom_op_func custom_op_func =
(get_custom_op_func)dlsym(plugin_lib, "get_rknn_custom_op");
    error = dlerror();
    if (error != NULL) {
        fprintf(stderr, "dlsym fail: %s\n", error);
        dlclose(plugin_lib);
        return -1;
    }

    rknn_custom_op* user_op = custom_op_func();
    ret = rknn_register_custom_ops(ctx, user_op,
1);
    if (ret < 0) {
        printf("rknn_register_custom_ops fail! ret = %d\n", ret);
        return -1;
    }
    so_handles.push_back(plugin_lib);
}

```

插件库有如下注意事项:

- 一个自定义算子插件库只能注册一个自定义算子, 如果需要注册多个自定义算子, 需要创建多个插件库。

- 插件库的名称必须以"librkcst_"开头，以.so 结尾。
- 如果是 C++代码实现的插件库，C 接口中为了正确打开函数符号，要在函数定义前后加入如下宏：

```
#ifndef __cplusplus
extern "C" {
#endif

//code

#endif
} // extern "C"
#endif
```

- 如果 dlopen 插件库失败,需要检查是否设置了 LD_LIBRARY_PATH 环境变量，并且检查插件库所在目录是否在环境变量指定的路径。

5.6 多 Batch 使用说明

5.6.1 多 Batch 原理

RK3588 NPU 内部有 3 个核心，RK3576 NPU 内部有 2 个核心，为了更高效得利用多核性能，提供了多 batch 推理功能。当开启多 batch 推理时，内部会调用 rknn_dup_context 将 context 进行拷贝（rknn_dup_context 只会对 context 的 Internal 进行拷贝，Weight 会复用）。当 rknn_batch_size=2 时，会拷贝 1 份，当 rknn_batch_size >=3 时，会拷贝 2 份（同一时刻最多只有 3 个核心工作，为了避免内存浪费只拷贝 2 份）。**每个 context core_mask 会设置成 0，让多核内部自动调度。**当执行 rknn_run()时，内部会起一个线程池，同一时刻调用 3 个线程同时对 3 个 context 进行推理。

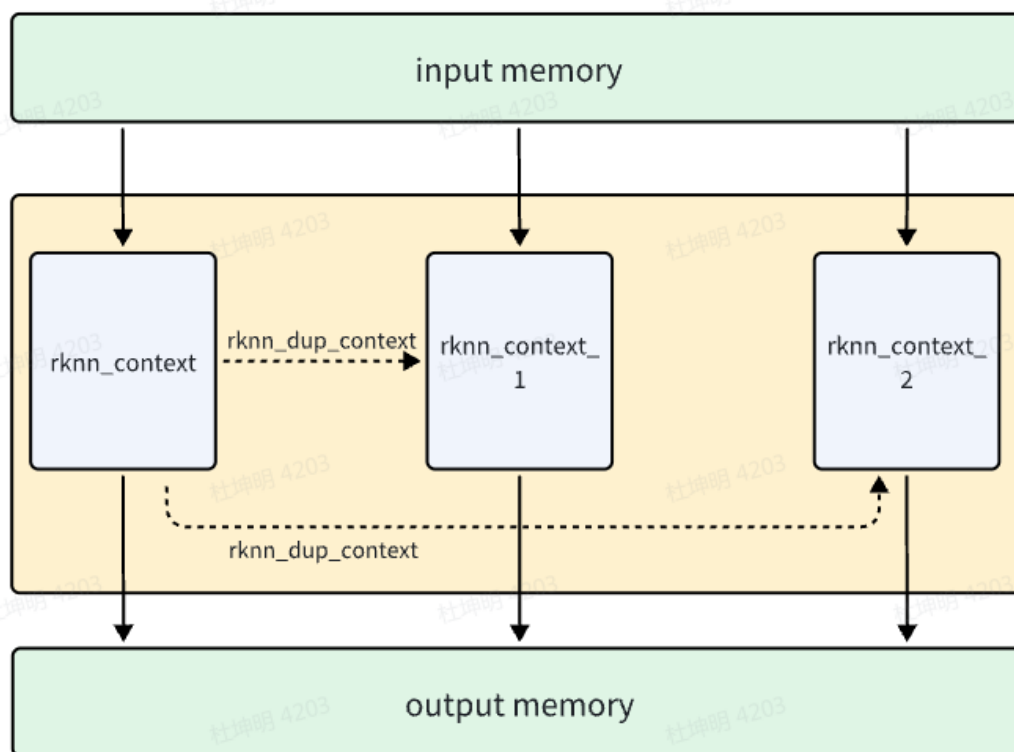


图 5-14 多 batch 内部原理图

5.6.2 多 Batch 使用方式

多 batch 使用方式如下：

1. Python 端开启多 batch 设置：

```
ret = rknn.build(do_quantization=True, dataset='./dataset.txt',
rknn_batch_size=3)
```

为了达到最优性能，RK3588 建议 rknn_batch_size 为 3 的倍数，RK3576 建议 rknn_batch_size 为 2 的倍数

2. 建议使用零拷贝接口

5.6.3 多 Batch 输入输出设置

当开启多 batch 功能时，查询出来的输入输出 size 是未开启时的 rknn_batch_size 倍。内部每个 context 会各自算自己的一个输入偏移量，按照这个输入偏移量取输入数据做推理，然后各自算自己的一个输出偏移量，按照这个输出偏移量写到各自的输出。以第二个 batch 为例，输入偏移量是查询出来的 input_size 除以 rknn_batch_size，输出偏移量是查询出来的 output_size 除以 rknn_batch_size。

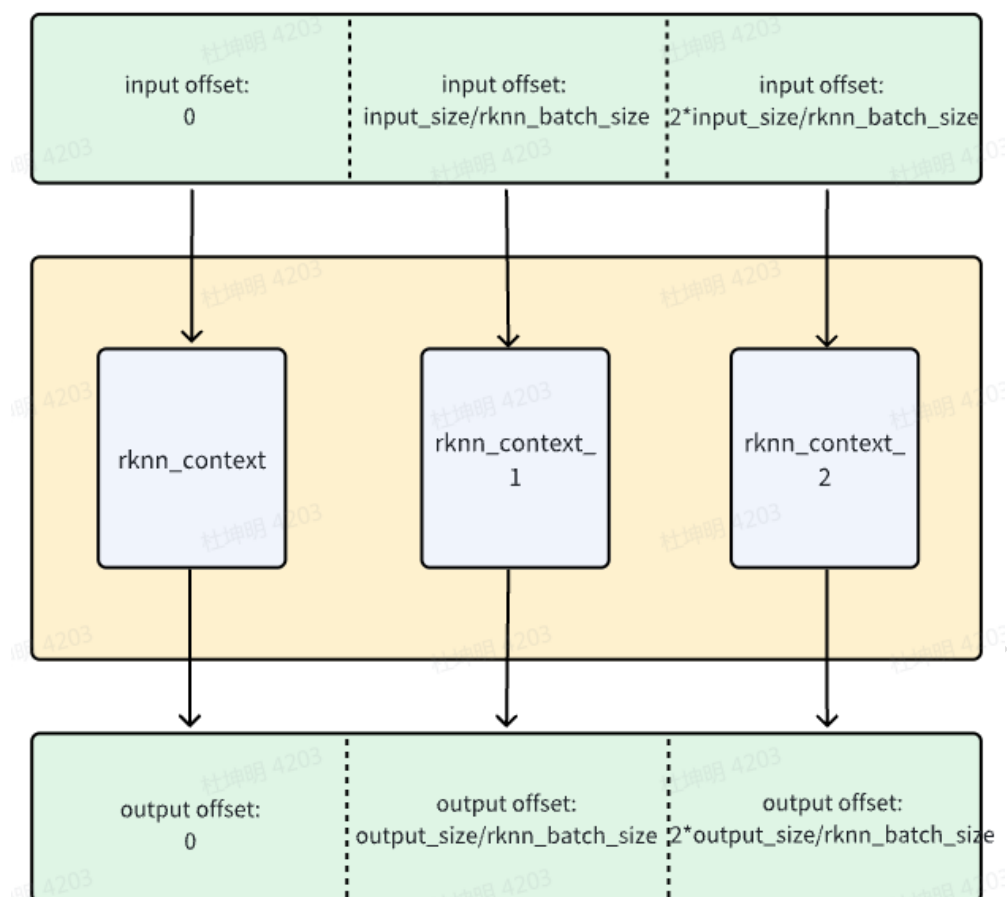


图 5-15 多 batch 内部输入输出地址偏移图

5.7 RK3588 NPU SRAM 使用说明

- RK3588 SOC 内部含有 1MB 的 SRAM，其中有 956KB 可供 SOC 上各个 IP 所使用。
- SRAM 可以帮助 RKNPU 应用减轻 DDR 带宽压力，但对推理耗时可能有一定影响。

5.7.1 板端环境要求

5.7.1.1 内核环境要求

- RKNPU 驱动版本 $\geq 0.9.2$
- 内核 config 需要开启 CONFIG_ROCKCHIP_RKNPU_SRAM=y

■ Android 系统 config 路径如下：

```
<path-to-your-kernel>/arch/arm64/configs/rockchip_defconfig
```

- Linux 系统 config 路径如下：

```
<path-to-your-kernel>/arch/arm64/configs/rockchip_linux_defconfig
```

- 内核相应 DTS 需要从系统 SRAM 中分配给 RKNPU 使用

- 从系统分配需求大小的 SRAM 给 RKNPU，最大可分配 956KB，且大小需要 4K 对齐。
- 注意：默认系统中可能已为其他 IP 分配 SRAM，比如编解码模块，各 IP 分配的 SRAM 区域不能重叠，否则会出现同时读写出现数据错乱现象。
- 如下为 956KB 全部分配给 RKNPU 的例子：

```
sysram: sram@ff001000 {
    compatible = "mmio-sram";
    reg = <0x0 0xff001000 0x0 0xef000>;

    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0x0 0xff001000 0xef000>;
    /* 分配 RKNPU SRAM /    / start address and size should be 4k
algin */
    rknpu_sram: rknpu_sram@0 {
        reg = <0x0 0xef000>; // 956KB
    };
};
```

- 把分配的 SRAM 挂到 RKNPU 节点，修改如下所示的 dtsi 文件：

```
<path-to-your-kernel>/arch/arm64/boot/dts/rockchip/rk3588s.dtsi
```

```
rknpu: npu@fdab0000 {
    compatible = "rockchip,rk3588-rknpu";
    /* ... /    / 增加 RKNPU sram 的引用 */
    rockchip,sram = <&rknpu_sram>;
    status = "disabled";
};
```

5.7.1.2 RKNN SDK 版本要求

- RKNPURuntime 库(librknnrt.so)版本>=1.6.0

5.7.2 使用方法

在 rknn_init()接口的 flags 参数指定 RKNN_FLAG_ENABLE_SRAM 即可在该 context 中开启 SRAM。

例如：

```
ret = rknn_init(&ctx, rknn_model, size, RKNN_FLAG_ENABLE_SRAM,
NULL);
```

当设置 RKNN_FLAG_ENABLE_SRAM 时，将从系统可用的 SRAM 中分配尽可能多的内存做为模型的 Internal Tensor 内存。

注意：

- 当 SRAM 被某一 rknn context 占用后，其他的 rknn context 不支持复用该段的 SRAM。
rknn_api.h 中的 RKNN_FLAG_SHARE_SRAM 功能暂未实现。
- 当某个 rknn context 未占用全部的 SRAM 时，剩余的 SRAM 可以给其他的 rknn context 使用。

5.7.3 调试方法

5.7.3.1 SRAM 是否启用查询

通过开机内核日志查看 SRAM 是否启用，包含为 RKNPU 指定 SRAM 的地址范围和大
小信息，如下所示：

```
rk3588_s:/ # dmesg | grep rknpu -i
RKNPU fdab0000.npu: RKNPU: sram region: [0x00000000ff001000,
0x00000000ff0f0000), sram size: 0xef000
```

5.7.3.2 SRAM 使用情况查询

- 可通过节点查询 SRAM 的使用情况
- 如下为未使用 SRAM 的位图表，每个点表示 4K 大小

```
rk3588_s:/ # cat /sys/kernel/debug/rknpu/mm
SRAM bitmap: "*" - used, "." - free (1bit = 4KB)
[000] [.....]
[001] [.....]
[002] [.....]
[003] [.....]
[004] [.....]
[005] [.....]
[006] [.....]
[007] [.....]
SRAM total size: 978944, used: 0, free: 978944
# 单位为 Byte
```

- 如下为分配使用 512KB 后的 SRAM 位图表

```
rk3588_s:/ # cat /sys/kernel/debug/rknpu/mm
SRAM bitmap: "*" - used, "." - free (1bit = 4KB)
[000] [*****]
[001] [*****]
[002] [*****]
[003] [*****]
[004] [.....]
[005] [.....]
[006] [.....]
[007] [.....]
SRAM total size: 978944, used: 524288, free: 454656
# 单位为 Byte
```

5.7.3.3 通过 RKNN API 查询 SRAM 大小

- 通过 rknn_query() 的 RKNN_QUERY_MEM_SIZE 接口查询 SRAM 大小信息

```
typedef struct _rknn_mem_size {
    uint32_t total_weight_size;
    uint32_t total_internal_size;
    uint64_t total_dma_allocated_size;
    uint32_t total_sram_size;
    uint32_t free_sram_size;
    uint32_t reserved[10];
} rknn_mem_size;
```

- 其中, total_sram_size 表示: 系统给 RKNPU 分配的 SRAM 总大小, 单位是 Byte。
- free_sram_size 表示: 剩余 RKNPU 能使用的 SRAM 大小, 单位是 Byte。

5.7.3.4 查看模型 SRAM 的占用情况

- 板端环境中, RKNN 应用运行前设置如下环境变量, 可打印 SRAM 使用预测情况:

```
export RKNN_LOG_LEVEL=3
```

- Internal 分配 SRAM 的逐层占用情况, 如下日志所示:

```
-----
Total allocated Internal SRAM Size: 524288, Addr: [0xff3e0000, 0xff460000)
-----
-----+-----
ID User      Tensor  DataType OrigShape  NativeShape  | [Start
End)      Size |      SramHit
-----+-----
1  ConvRelu   input0  INT8      (1,3,224,224) (1,1,224,224,3) |
0xff3b0000 0xff3d4c00 0x00024c00 | \
2  ConvRelu   output2 INT8      (1,32,112,112) (1,2,112,112,16) |
0xff404c00 0xff466c00 0x00062000 | 0x0005b400
3  ConvRelu   output4 INT8      (1,32,112,112) (1,4,112,112,16) |
0xff466c00 0xff52ac00 0x000c4000 | 0x00000000
4  ConvRelu   output6 INT8      (1,64,112,112) (1,4,112,112,16) |
0xff52ac00*0xff5eec00 0x000c4000 | 0x00000000
5  ConvRelu   output8 INT8      (1,64,56,56) (1,4,56,56,16) |
0xff3e0000 0xff411000 0x00031000 | 0x00031000
6  ConvRelu   output10 INT8     (1,128,56,56) (1,8,56,56,16) |
0xff411000 0xff473000 0x00062000 | 0x0004f000
7  ConvRelu   output12 INT8     (1,128,56,56) (1,8,56,56,16) |
0xff473000 0xff4d5000 0x00062000 | 0x00000000
8  ConvRelu   output14 INT8     (1,128,56,56) (1,8,56,56,16) |
0xff3e0000 0xff442000 0x00062000 | 0x00062000
9  ConvRelu   output16 INT8     (1,128,28,28) (1,8,28,28,16) |
0xff442000 0xff45a800 0x00018800 | 0x00018800
10 ConvRelu   output18 INT8     (1,256,28,28) (1,16,28,28,16) |
0xff3e0000 0xff411000 0x00031000 | 0x00031000
11 ConvRelu   output20 INT8     (1,256,28,28) (1,16,28,28,16) |
0xff411000 0xff442000 0x00031000 | 0x00031000
12 ConvRelu   output22 INT8     (1,256,28,28) (1,16,28,28,16) |
0xff3e0000 0xff411000 0x00031000 | 0x00031000
13 ConvRelu   output24 INT8     (1,256,14,14) (1,16,14,14,16) |
0xff411000 0xff41d400 0x0000c400 | 0x0000c400
14 ConvRelu   output26 INT8     (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
15 ConvRelu   output28 INT8     (1,512,14,14) (1,32,14,14,16) |
```

```

0xff3f8800 0xff411000 0x00018800 | 0x00018800
16 ConvRelu      output30 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
17 ConvRelu      output32 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3f8800 0xff411000 0x00018800 | 0x00018800
18 ConvRelu      output34 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
19 ConvRelu      output36 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3f8800 0xff411000 0x00018800 | 0x00018800
20 ConvRelu      output38 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
21 ConvRelu      output40 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3f8800 0xff411000 0x00018800 | 0x00018800
22 ConvRelu      output42 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
23 ConvRelu      output44 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3f8800 0xff411000 0x00018800 | 0x00018800
24 ConvRelu      output46 INT8   (1,512,14,14) (1,32,14,14,16) |
0xff3e0000 0xff3f8800 0x00018800 | 0x00018800
25 ConvRelu      output48 INT8   (1,512,7,7)   (1,33,7,7,16)   |
0xff3f8800 0xff3ff000 0x00006800 | 0x00006800
26 ConvRelu      output50 INT8   (1,1024,7,7)  (1,67,7,7,16)   |
0xff3e0000 0xff3ed000 0x0000d000 | 0x0000d000
27 ConvRelu      output52 INT8   (1,1024,7,7)  (1,67,7,7,16)   |
0xff3ed000 0xff3fa000 0x0000d000 | 0x0000d000
28 AveragePool   output54 INT8   (1,1024,7,7)  (1,67,7,7,16)   |
0xff3e0000 0xff3ed000 0x0000d000 | 0x0000d000
29 Conv          output55 INT8   (1,1024,1,1)  (1,64,1,1,16)   | 0xff3ed000
0xff3ed400 0x00000400 | 0x00000400
30 Softmax       output56 INT8   (1,1000,1,1)  (1,64,1,1,16)   |
0xff3e0000 0xff3e0400 0x00000400 | 0x00000400
31 OutputOperator output57 FLOAT  (1,1000,1,1)  (1,1000,1,1)    |
0xff3ae000 0xff3aefa0 0x00000fa0 | \
-----+-----
-----+-----
Total Weight Memory Size: 4260864
Total Internal Memory Size: 2157568
Predict Internal Memory RW Amount: 11068320
Predict Weight Memory RW Amount: 4260832
Predict SRAM Hit RW Amount: 6688768
-----+-----

```

- 其中上面文本图表中的 SRAM Hit 为当前层 Tensor 所占用的 SRAM 大小，表示与未开启 SRAM 对比，会节省对应大小的 DDR 读写数据量。
- Predict SRAM Hit RW Amount 为整个模型 SRAM 的读写预测情况，表示与未开启 SRAM 对比，每帧可节省的 DDR 读写数据量。
- 注意：Linux 环境日志重定向到终端，Android 环境日志重定向到 logcat。

5.8 模型剪枝

模型剪枝一般可分为有损剪枝和无损剪枝两种方式，RKNN-Toolkit2 的模型剪枝为无损剪枝，也就是可以在减小模型大小和计算量的前提下，不会降低模型的浮点精度，甚至可以提高模型的量化精度。

但并不是所有模型都可以进行无损剪枝，无损剪枝是根据模型的权重的稀疏化程度，

来去除一些对模型结果不造成影响的权重和 Feature 通道，以减小模型的大小和模型的计算量。在启用模型剪枝配置后（将 `rknn.config()` 的 `model_pruning` 参数设为 `True`），模型转换时会自动根据权重的稀疏化程度对模型进行剪枝。

如果模型剪枝成功，则会打印剪枝结果，如下：

```
I model_pruning results:
I   Weight: -1.12145 MB (-6.9%)
I   GFLOPs: -0.15563   (-13.4%)
```

其中，Weight 剪掉了 1.12145 MB（占比 6.9%），模型运算量减少 0.15563 GFLOPs（占比 13.4%）。

如果因模型稀疏化程度不够而剪枝失败，则不会做任何处理（也不会打印上述信息），因此该配置并不会影响正常的模型转换。

5.9 模型加密

模型加密指的是生成完 RKNN 模型后，再重新对其做进一步处理。使用 `rknn.export_rknn()` 生成的模型，可以通过 Netron 等第三方工具查看图结构。模型加密后，Netron 等第三方工具将无法查看相应的网络结构，也无法获取权重，起到对模型的保护作用。当前加密后的 RKNN 模型使用方法和未加密的模型一样，不需要在开发板推理时做任何修改。

使用方法如下：

```
# Create RKNN object
rknn = RKNN()
# Export encrypted RKNN model
crypt_level= 1
ret = rknn.export_encrypted_rknn_model("input.rknn",
"encrypt.rknn", crypt_level)
if ret != 0:
    print('Encrypt RKNN model failed!')
```

`crypt_level` 用来指定加密等级，有 1, 2 和 3 三个等级。默认值为 1。等级越高，安全性越高，解密越耗时；反之，安全性越低，解密越快。

- 支持平台：RK3562/RK3566/RK3568/RK3588/RK3576

5.10 Cacheable 内存一致性

Cacheable 内存一致性问题是指当 CPU 和 NPU 设备都会访问同一块带 cache 标志的内存时，CPU 会将数据缓存到 cache 中，如果 NPU 访问到的 DDR 数据与 CPU cache 不一致，

会导致读取数据错误。因此要调用刷新 cache 的接口保证 CPU 和 NPU 访问到的 DDR 内存数据是一致的。本章节介绍了同步数据的方向以及如何使用 rknn_mem_sync 接口刷新 cache。

5.10.1 Cacheable 内存同步的方向

当 CPU 写数据到 cacheable 的内存，之后 NPU 访问该内存时，要保证 CPU cache 的数据同步到 DDR 中，此时同步的方向是指从 CPU 到 NPU 设备；当 NPU 写完数据，CPU 开始访问该内存时，要保证 DDR 的数据与 CPU cache 中的一致，此时同步的方向是指从 NPU 设备到 CPU。RKNN C API 提供了 rknn_mem_sync_mode 枚举类型表示 cacheable 内存同步的方向，数据结构如下：

```
/*
 * The mode to sync cacheable rknn memory.
 */
typedef enum _rknn_mem_sync_mode {
    RKNN_MEMORY_SYNC_TO_DEVICE = 0x1, /* the mode used for
consistency of device access after CPU accesses data. */
    RKNN_MEMORY_SYNC_FROM_DEVICE = 0x2, /* the mode used for
consistency of CPU access after device accesses data. */
    RKNN_MEMORY_SYNC_SYNC_BIDIRECTIONAL =
RKNN_MEMORY_SYNC_TO_DEVICE | RKNN_MEMORY_SYNC_FROM_DEVICE, /* the
mode used for consistency of data access
between device and CPU in both directions. */
} rknn_mem_sync_mode;
```

- RKNN_MEMORY_SYNC_TO_DEVICE：表示数据同步方向是 CPU 到 NPU 设备
- RKNN_MEMORY_SYNC_FROM_DEVICE：表示数据同步方向是 NPU 设备到 CPU
- RKNN_MEMORY_SYNC_SYNC_BIDIRECTIONAL：表示数据在 NPU 和 CPU 之间双向同步在用户不确定同步数据的方向时可以使用该枚举

在明确数据同步方向时，建议使用单方向刷新 CPU cache 模式，能避免多余的刷新 CPU cache 动作导致性能损耗。

5.10.2 同步 Cacheable 内存

明确数据同步方向之后，就可以使用 rknn_mem_sync 接口对 cacheable 内存做同步。

接口形式如下：

```
int rknn_mem_sync(rknn_context context, rknn_tensor_mem* mem,
rknn_mem_sync_mode mode);
```

其中，context 是上下文(在非 RV1103/RV1106 平台上默认设置为 NULL)，mem 是 rknn_create_mem 接口返回的 rknn_tensor_mem* 指针类型，mode 是指同步数据的方向。

5.11 模型稀疏化推理

该功能用于 Torch 模型在训练阶段对模型权重自动稀疏化并使用 RKNN 进行稀疏化推理，提高模型推理速度。目前该功能仅支持 RK3576。

5.11.1 稀疏化原理

模型权重稀疏化在训练阶段根据用户自定义的方式，进行权重置零操作，具体方式有 4:2 输入方向稀疏化，4:2 输出方向稀疏化，16:4 输入输出稀疏化，16:4 输出输入稀疏化。其中 4:2 输入方向稀疏化实现原理如图 5-16 所示，将模型权重沿输入方向，在连续的 4 个数值中选择两个置零。需要注意的是，当模型权重指定方向非 4 对齐时，会进行补齐操作。

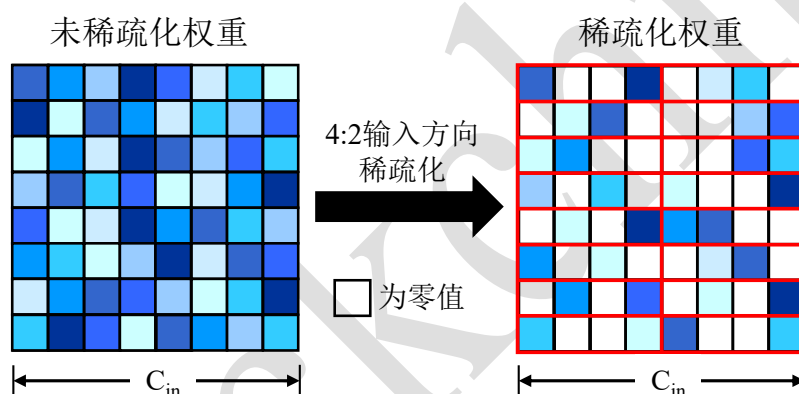


图 5-16 4:2 输入方向稀疏化原理图

有关四种权重稀疏化说明如表 5-10 所示，其中输入输出方向为权重的方向，例如 2 维卷积 shape 为 $C_{out} \times C_{in} \times K_h \times K_w$ ，输入方向即 C_{in} ，输出方向即 C_{out} 。在稀疏率为 75% 的方式中前置训练模型可以是未稀疏化模型也可以是稀疏模型，建议使用单方向稀疏模型作为前置预训练模型，可减少稀疏化后的精度损失。

表 5-10 模型权重稀疏化说明

稀疏化方式	稀疏率	说明
4:2 输入方向稀疏化	50%	沿输入方向对权重稀疏化
4:2 输出方向稀疏化	50%	沿输出方向对权重稀疏化
16:4 输入输出稀疏化	75%	先沿输入方向 4:2 稀疏，再沿输出方向 4:2 稀疏
16:4 输出输入稀疏化	75%	先沿输出方向 4:2 稀疏，再沿输入方向 4:2 稀疏

5.11.2 训练稀疏化模型

1. 首先确保 cuda 可使用并安装对应 python 版本的 autosparsity 包

```
pip install autosparsity-1.0-cp38-cp38m-linux_x86_64.whl
```

autosparsity 安装包路径为: [https://github.com/airockchip/rknn-](https://github.com/airockchip/rknn-toolkit2/tree/master/autosparsity/packages)

[toolkit2/tree/master/autosparsity/packages](https://github.com/airockchip/rknn-toolkit2/tree/master/autosparsity/packages)

2. 以 torchvision 中的 resnet50 的 4:2 输入方向稀疏化为例, 进行权重稀疏化训练

```
import torch
import torchvision.models as models
from autosparsity.sparsity import sparsity_model

if __name__ == "__main__":

    model = models.resnet50(pretrained=True).cuda()
    optimizer = None
    mode = 0
    sparsity_model(model, optimizer, mode)

    model.eval()
    x = torch.randn((1, 3, 224, 224)).cuda()
    torch.onnx.export(
        model, x, 'resnet50.onnx', input_names=['inputs'],
        output_names=['outputs']
    )
```

自定义模型的稀疏化在模型训练之前添加 sparsity_model 函数即可, 参考示例如下:

```
# insert model autosparsity code before training
import torch
import torchvision.models as models
from autosparsity.sparsity import sparsity_model

...

model = models.resnet34(pretrained=True).cuda()
mode = 0
sparsity_model(model, optimizer, mode)

# normal training
x, y = DataLoader(args)
for epoch in range(epochs):
    y_pred = model(x)
    loss = loss_func(y_pred, y)
    loss.backward()
    optimizer.step()
...
```

注: Attention 模型稀疏化训练依赖于一个已训练好的前置模型。所以在做稀疏化训练之前, 请先训练一个效果‘不错’的模型, 在该模型基础上进行稀疏化训练。

有关 sparsity_model 函数的参数说明如下:

表 5-11 sparsity_model 函数各参数说明

参数	详细说明
model	原训练模型
optimizer	原优化器，默认为 None
mode	稀疏化方式，可选值为 0, 1, 2, 3，默认为 0 0: 4:2 输入方向稀疏化(50%稀疏率) 1: 4:2 输出方向稀疏化(50%稀疏率) 2: 16:4 输入输出稀疏化(75%稀疏率) 3: 16:4 输出输入稀疏化(75%稀疏率)
verbose	log 等级，可选值为 0, 1, 2, 3，默认为 2 0: Errors 1: Errors and Warnings 2: Errors, warnings and info
whitelist	稀疏化支持的 module 列表，支持 1d conv, 2d conv, 3d conv, linear, MultiheadAttention，默认 [torch.nn.Linear, torch.nn.Conv2d]
allowed_layer_names	允许稀疏化的层名，用户配置时则只稀疏指定层，默认 None
disallowed_layer_names	不允许稀疏化的层名，用户配置时则会跳过该层，默认[]
fast	设为 True 代表使用快速方法计算 mask，默认为 False(默认的 mask 计算方法针对部分模型会失效，若稀疏化报错可尝试将该参数设为 True)

5.11.3 RKNN 稀疏化推理使用方法

通过 RKNN-Toolkit2 中 config()接口的“sparse_infer”参数设置模型稀疏化的开启和关闭，对应的参数为 True/False，默认值为 False。开启稀疏化推理的参考代码如下：

```
rknn.config(target_platform='rk3576', sparse_infer=True)
```

完整的稀疏化 Python 推理代码可参考：<https://github.com/airockchip/rknn-toolkit2/tree/master/autosparsity/examples>

使用 C API 进行部署时，首先使用 RKNN-Toolkit2 在 config()接口中设置 “sparse_infer” 参数为 True 生成带稀疏化推理的 RKNN 模型，之后正常调用通用 API 接口流程或零拷贝接口流程即可。

使用 Python 代码推理时可在构建 RKNN 对象时设置 verbose=True，开启日志打印各层稀疏化情况；使用 C API 推理时通过设置环境变量 RKNN_LOG_LEVEL=4，开启日志打印各层稀疏化情况。日志信息如下：

Network Layer Information Table									
ID	OpType	DataType	Target	InputShape	OutputShape	Cycles(DR/NPU/Total)	SparseRatio	Re(KB)	FullName
0	InputOperator	INT8	CPU	\	(1,3,224,224)	0/0/0	0%	0	InputOperator:inputs
1	Conv	INT8	NPU	(1,3,224,224),(64,3,7,7),(64)	(1,64,112,112)	129458/1229312/1229312	0%	159	Conv:conv1/conv
2	MaxPool	INT8	NPU	(1,64,112,112)	(1,64,56,56)	134540/0/134540	0%	159	MaxPool1:mapool1/maxpool
3	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	13807/25080/53807	50%(IC)	186	Conv:Layer1/Layer1.0/conv1/conv
4	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	5353/112896/112896	50%(IC)	194	Conv:Layer1/Layer1.0/conv2/conv
5	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	134560/100352/100352	50%(IC)	197	Conv:Layer1/Layer1.0/conv3/conv
6	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	24212/100352/24212	50%(IC)	981	Conv:Layer1/Layer1.0/downsample.0/conv
7	Conv	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	134362/50176/134362	50%(IC)	783	Conv:Layer1/Layer1.1/conv1/conv
8	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	5353/112896/112896	50%(IC)	194	Conv:Layer1/Layer1.1/conv2/conv
9	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	24212/100352/24212	50%(IC)	981	Conv:Layer1/Layer1.1/conv3/conv
10	Conv	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	134362/50176/134362	50%(IC)	783	Conv:Layer1/Layer1.2/conv1/conv
11	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	5353/112896/112896	50%(IC)	194	Conv:Layer1/Layer1.2/conv2/conv
12	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	24212/100352/24212	50%(IC)	981	Conv:Layer1/Layer1.2/conv3/conv
13	Conv	INT8	NPU	(1,256,56,56),(128,128,1,1),(128)	(1,128,56,56)	161175/100352/161179	50%(IC)	783	Conv:Layer2/Layer2.0/conv1/conv
14	Conv	INT8	NPU	(1,128,56,56),(128,64,3,3),(128)	(1,128,28,28)	66181/112896/112896	50%(IC)	384	Conv:Layer2/Layer2.0/conv2/conv
15	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	67215/50176/67215	50%(IC)	98	Conv:Layer2/Layer2.0/conv3/conv
16	Conv	INT8	NPU	(1,256,56,56),(512,128,1,1),(512)	(1,512,28,28)	214540/100352/214540	50%(IC)	1172	Conv:Layer2/Layer2.0/downsample.0/conv
17	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:Layer2/Layer2.1/conv1/conv
18	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:Layer2/Layer2.1/conv2/conv
19	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	126988/50176/126988	50%(IC)	490	Conv:Layer2/Layer2.1/conv3/conv
20	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:Layer2/Layer2.2/conv1/conv
21	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:Layer2/Layer2.2/conv2/conv
22	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	126988/50176/126988	50%(IC)	490	Conv:Layer2/Layer2.2/conv3/conv
23	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	66804/50176/66804	50%(IC)	389	Conv:Layer2/Layer2.3/conv1/conv
24	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	25789/112896/112896	50%(IC)	90	Conv:Layer2/Layer2.3/conv2/conv
25	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	126988/50176/126988	50%(IC)	490	Conv:Layer2/Layer2.3/conv3/conv
26	Conv	INT8	NPU	(1,512,28,28),(256,256,1,1),(256)	(1,256,28,28)	79835/100352/100352	50%(IC)	380	Conv:Layer3/Layer3.0/conv1/conv
27	Conv	INT8	NPU	(1,256,28,28),(256,256,3,3),(256)	(1,256,14,14)	112894/118080/118080	0%	774	Conv:Layer3/Layer3.0/conv2/conv

图 5-17 4:2 输入方向稀疏化 Python 打印日志

Network Layer Information Table												
ID	OpType	DataType	Target	InputShape	OutputShape	Cycles(DR/NPU/Total)	Time(us)	MemUsage(KB)	Workload(0/1)	SparseRatio	Re(KB)	FullName
0	InputOperator	UINT8	CPU	\	(1,3,224,224)	0/0/0	6	\	0.00/0.00	0%	0	InputOperator:inputs
1	Conv	INT8	NPU	(1,3,224,224),(64,3,7,7),(64)	(1,64,112,112)	70436/1229312/1229312	4281	9.06/0.00	100.00/0.00	0%	159	Conv:conv1/conv
2	MaxPool	INT8	NPU	(1,64,112,112)	(1,64,56,56)	47923/0/47923	715	\	100.00/0.00	0%	159	MaxPool1:mapool1/maxpool
3	Conv	INT8	NPU	(1,64,56,56),(64,32,1,1),(64)	(1,64,56,56)	24121/22880/25080	164	12.80/0.00	100.00/0.00	50%(IC)	186	Conv:Layer1/Layer1.0/conv1/conv
4	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24846/112896/112896	280	67.80/0.00	100.00/0.00	50%(IC)	413	Conv:Layer1/Layer1.0/conv2/conv
5	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	72375/100352/100352	426	19.83/0.00	100.00/0.00	50%(IC)	992	Conv:Layer1/Layer1.0/conv3/conv
6	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	184130/100352/100352	468	35.90/0.00	100.00/0.00	50%(IC)	1776	Conv:Layer1/Layer1.0/downsample.0/conv
7	Conv	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	48484/50176/50176	300	28.10/0.00	100.00/0.00	50%(IC)	994	Conv:Layer1/Layer1.1/conv1/conv
8	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24083/112896/112896	279	68.12/0.00	100.00/0.00	50%(IC)	415	Conv:Layer1/Layer1.1/conv2/conv
9	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	184130/100352/100352	468	35.90/0.00	100.00/0.00	50%(IC)	1776	Conv:Layer1/Layer1.1/conv3/conv
10	Conv	INT8	NPU	(1,256,56,56),(64,128,1,1),(64)	(1,64,56,56)	48522/50176/50176	297	28.44/0.00	100.00/0.00	50%(IC)	995	Conv:Layer1/Layer1.2/conv1/conv
11	Conv	INT8	NPU	(1,64,56,56),(64,32,3,3),(64)	(1,64,56,56)	24083/112896/112896	279	68.12/0.00	100.00/0.00	50%(IC)	415	Conv:Layer1/Layer1.2/conv2/conv
12	Conv	INT8	NPU	(1,64,56,56),(256,32,1,1),(256)	(1,256,56,56)	184130/100352/100352	468	35.90/0.00	100.00/0.00	50%(IC)	1776	Conv:Layer1/Layer1.2/conv3/conv
13	Conv	INT8	NPU	(1,256,56,56),(128,128,1,1),(128)	(1,128,56,56)	64840/100352/100352	391	43.21/0.00	100.00/0.00	50%(IC)	1282	Conv:Layer2/Layer2.0/conv1/conv
14	Conv	INT8	NPU	(1,128,56,56),(128,64,3,3),(128)	(1,128,28,28)	11751/112896/112896	354	53.60/0.00	100.00/0.00	50%(IC)	571	Conv:Layer2/Layer2.0/conv2/conv
15	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	37642/50176/50176	236	35.79/0.00	100.00/0.00	50%(IC)	531	Conv:Layer2/Layer2.0/conv3/conv
16	Conv	INT8	NPU	(1,256,56,56),(512,128,1,1),(512)	(1,512,28,28)	81157/100352/100352	417	40.51/0.00	100.00/0.00	50%(IC)	1608	Conv:Layer2/Layer2.0/downsample.0/conv
17	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25680/50176/50176	220	35.40/0.00	100.00/0.00	50%(IC)	532	Conv:Layer2/Layer2.1/conv1/conv
18	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	51341/112896/112896	267	71.10/0.00	100.00/0.00	50%(IC)	280	Conv:Layer2/Layer2.1/conv2/conv
19	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	53704/50176/50176	264	31.52/0.00	100.00/0.00	50%(IC)	920	Conv:Layer2/Layer2.1/conv3/conv
20	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25680/50176/50176	218	35.70/0.00	100.00/0.00	50%(IC)	532	Conv:Layer2/Layer2.2/conv1/conv
21	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	51341/112896/112896	267	71.10/0.00	100.00/0.00	50%(IC)	282	Conv:Layer2/Layer2.2/conv2/conv
22	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	53704/50176/50176	264	31.52/0.00	100.00/0.00	50%(IC)	922	Conv:Layer2/Layer2.2/conv3/conv
23	Conv	INT8	NPU	(1,512,28,28),(128,256,1,1),(128)	(1,128,28,28)	25720/50176/50176	217	36.93/0.00	100.00/0.00	50%(IC)	533	Conv:Layer2/Layer2.3/conv1/conv
24	Conv	INT8	NPU	(1,128,28,28),(128,64,3,3),(128)	(1,128,28,28)	51341/112896/112896	267	71.10/0.00	100.00/0.00	50%(IC)	280	Conv:Layer2/Layer2.3/conv2/conv
25	Conv	INT8	NPU	(1,128,28,28),(512,64,1,1),(512)	(1,512,28,28)	53722/50176/50176	271	36.90/0.00	100.00/0.00	50%(IC)	927	Conv:Layer2/Layer2.3/conv3/conv
26	Conv	INT8	NPU	(1,512,28,28),(256,256,1,1),(256)	(1,256,28,28)	93580/100352/100352	316	53.40/0.00	100.00/0.00	50%(IC)	671	Conv:Layer3/Layer3.0/conv1/conv
27	Conv	INT8	NPU	(1,256,28,28),(256,256,3,3),(256)	(1,256,14,14)	35572/118080/118080	531	71.50/0.00	100.00/0.00	0%	824	Conv:Layer3/Layer3.0/conv2/conv

图 5-18 4:2 输入方向稀疏化 C API 打印日志

其中 SparseRatio 代表稀疏率，4:2 输入方向稀疏化对应 50%(IC)，4:2 输出方向稀疏化对应 50%(OC)，16:4 输入输出稀疏化和 16:4 输出输入稀疏化对应 75%。0%则代表未做稀疏化。

5.11.4 RKNN 稀疏化推理限制

稀疏化推理是基于 NPU 的硬件架构实现，受硬件规格限制，RK3576 的稀疏化推理目前只支持 2d Conv 并且 group 参数为 1，其余限制如下：

表 5-12 RK3576 稀疏化推理限制

	通道数量	数据类型
4:2 输入方向稀疏化	输入 32 对齐, 输出 32 对齐	Int8 Float16 不支持
4:2 输出方向稀疏化	输入 32 对齐, 输出 32 对齐	Int8 Float16 不支持
16:4 输入输出稀疏化	输入 32 对齐, 输出 32 对齐	Int8 Float16 不支持
16:4 输出输入稀疏化	输入 32 对齐, 输出 32 对齐	Int8 Float16 不支持

5.12 生成部署 C 代码

RKNN-Toolkit2 2.0.0 版本新增 `Codegen` 接口, 用于生成模型部署代码, 简化开发者的上手难度。`Codegen` 基于 `CAPI` 零拷贝接口进行二次封装, 接口风格与 `RKNN_Model_Zoo` 中的 `demo` 一致。生成的部署代码可用于直接测试性能、验证精度, 开发者也可以基于生成的代码进行二次开发。

使用示例:

```
ret = rknn.codegen(output_path='./rknn_app_demo',  
inputs=['../caffe/mobilenet_v2/dog_224x224.jpg'],  
overwrite=True)
```

- 调用 `codegen` 接口前, 必须先调用 `rknn.export` 接口保存 RKNN 模型
- `output_path` 为输出文件夹目录, 用户可配置目录名称
- `inputs` 填写模型输入的路径列表, 允许不填。有效文件格式为 `jpg/png/npz`, 以 `npz` 文件为输入时, `npz` 数据的维度信息应与模型输入的维度信息保持一致
- `overwrite` 设为 `True` 时, 会覆盖 `output_path` 指定目录下的文件。默认值为 `False`
- 生成部署代码后, 请参考生成目录下的 `README.md` 文档说明进行编译、测试
- 若 `inputs` 填入有效值, 部署代码示例在推理后, 会评估 `CAPI` 接口与 RKNN-Toolkit2 模拟器之间的推理结果差异, 评估方式为对比每一个输出的余弦相似度
- 无 NPU 硬件平台限制, 要求板端系统为 Linux 或 Android

- 支持量化、非量化模型

5.13 ONNX 模型编辑

部分模型在转 RKNN 模型后，可能存在冗余的 op，常见于模型的输入输出节点处存在冗余的 reshape、transpose op，影响了 RKNN 模型的推理性能。RKNN-Toolkit2 提供了 onnx_edit 接口，用于修改 ONNX 模型的输入输出的维度定义，使调整后的 onnx 模型能转出性能更好的 RKNN 模型，减少冗余的 reshape、transpose op。

5.13.1 onnx_edit 接口说明

使用示例：

```
from rknn.utils import onnx_edit
ret = onnx_edit(model = './concat_block.onnx',
                export_path = './concat_block_edited.onnx',
                inputs_transform = {'k_in':
'a,b,c,d->1,ad,b,c'},
                outputs_transform = {'k_cache':
'a,b,c,d->1,ab,c,d'},
                dataset = './dataset.txt'
)
```

- model 填入待修改模型，为必填参数
- export_path 填入新模型的生成路径，为必填参数
- inputs_transform 填入输入节点的变换公式字典，key 为节点名称，value 为变换公式。为可选参数，默认为空字典
- outputs_transform 填入输出节点的变换公式字典。为可选参数，默认为空字典
- dataset 填入输入数据的路径集文件，文件格式要求与 rknn.build 接口对 dataset 的要求一致。填入后，onnx_edit 接口除了对模型的输入输出定义进行调整，也会将 dataset 中对应的数据进行调整，在 export_path 同级目录下生成新的 dataset 数据，可用于新模型的验证、量化。为可选参数，默认为空。

5.13.2 onnx_edit 变换公式说明

onnx_edit 接口中的 inputs_transform、outputs_transform 需要填入变换公式。变换公式的定义与 einsum 算子的定义类似，例如'a,b,c,d->1,ad,b,c'公式，指将原始的维度为'a,b,c,d'，变换后为'1,ad,b,c'。变换公式的填写规则如下：

- 必须有两部分字符组成，并用 ‘->’ 隔开，左边为原始字符 shape，右边是变换后字符 shape
- 字符 shape 里面的符号只允许是 [a-z] 或 ‘,’ ‘1’，除了‘1’以外，不支持其他数字 shape 字符，原因是相同数字字符无法判断 transpose 的前后关系
- 字符[a-z]认为是独立的，没有顺序关系，即 ‘a,b->b,a’ 和 ‘c,a->a,c’ 表示的是一样的变换
- 等式左边的原始字符 shape，被 ‘,’ 分隔成 n 份，n 的个数必须和模型的维度匹配，例如模型输入定义是 [32,4,1,64]，输入字符可以是 ‘a,b,c,d’ 或 ‘a,b,1,d’
- 原始字符 shape 的每一个符号，除了‘1’，都必须存在于变换后字符 shape，例如 ‘a,1,c,d->ac,d,1’ 是有效的，‘a,1,c,d->ac,1’ 是无效的，缺了 ‘d’
- 变换后字符 shape，允许插入任意个数的 ‘1’，达成扩维效果，例如 ‘a,b,c->a,1,cb,1,1’
- 原始字符 shape，允许用多个字母以及赋值公式来表示对 shape 进行拆分，例如原始输入定义是 [32,4,1,64]，‘ab,c,d,qk[a=2,k=8]->aq,cd,1,kb’，表示将 32 拆分成 2x16，将 64 拆分成 8x8，再进行 transpose, reshape 操作。其中 ‘[]’ 的部分称为赋值公式，多个公式用 ‘,’ 符号分隔。此外，允许拆分中的某个字符没有赋值，此时会自动推断对应的 shape，例如赋值公式只给了 a=2，已知在模型中 ab=32，则自动推断出 b=16；若推断出的 shape 异常会直接报错，比如 ab=32，若赋值 a=5，则 b=6.4，又维度必须是整数，此时会抛出异常错误。

5.13.3 变换公式示例

- 将 3 维输入修改为 4 维输入：‘a,b,c->a,b,1,c’
- 将 5 维输入修改为 4 维输入：‘a,b,c,d,e->ab,c,d,e’
- 进行 transpose(0,3,1,2)操作：‘a,b,c,d->a,d,b,c’
- Transpose 并合并部分维度：‘a,b,c,d->d,acb,1’
- 拆分维度、transpose、合并维度：‘a,bc,de,f[b=2,d=4]->ab,fe,dc,1’

6 量化说明

6.1 量化介绍

6.1.1 量化定义

模型量化是指将深度学习模型中的浮点参数和操作转换为定点表示，如 FLOAT32 转换为 INT8 等。量化能够降低内存占用，实现模型压缩和推理加速，但会造成一定程度的精度损失。

6.1.2 量化计算原理

以线性非对称量化为例，浮点数量化为有符号定点数的计算原理如下：

$$x_{\text{int}} = \text{clamp}(\lfloor \frac{x}{s} \rceil + z; -2^{b-1}, 2^{b-1} - 1) \quad (6-1)$$

其中 x 为浮点数， x_{int} 为量化定点数， $\lfloor \cdot \rceil$ 为四舍五入运算， s 为量化比例因子， z 为量化零点， b 为量化位宽，如 INT8 数据类型中 b 为 8； clamp 为截断运算，具体定义如下：

$$\text{clamp}(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c, \end{cases} \quad (6-2)$$

从定点数转换为浮点数称为反量化过程，具体定义如下：

$$x \approx \hat{x} = s(x_{\text{int}} - z) \quad (6-3)$$

设量化范围为 (q_{\min}, q_{\max}) ，截断范围为 (c_{\min}, c_{\max}) ，量化参数 s 和 z 的计算公式如下：

$$s = \frac{q_{\max} - q_{\min}}{c_{\max} - c_{\min}} = \frac{q_{\max} - q_{\min}}{2^{b-1}} \quad (6-4)$$

$$z = c_{\max} - \lfloor \frac{q_{\max}}{s} \rfloor \text{ 或 } z = c_{\min} - \lfloor \frac{q_{\min}}{s} \rfloor \quad (6-5)$$

其中截断范围是根据量化的数据类型决定，例如 INT8 的截断范围为 $(-128, 127)$ ；量化范围根据不同的量化算法确定，具体可参考 [6.1.6 量化算法](#) 章节。

6.1.3 量化误差

量化会造成模型一定程度的精度丢失。根据公式(6-1)可知，量化误差来源于舍入误差和截断误差，即 $\lfloor \cdot \rceil$ 和 clamp 运算。四舍五入的计算方式会产生舍入误差，误差范围为 $(-\frac{1}{2}s, \frac{1}{2}s)$ 。当浮点数 x 过大，比例因子 s 过小时，容易导致量化定点数超出截断范围，产生截断误差。理论上，比例因子 s 的增大可以减小截断误差，但会造成舍入误差的增大。因此

为了权衡两种误差，需要设计合适的比例因子和零点，来减小量化误差。

6.1.4 线性对称量化和线性非对称量化

线性量化中定点数之间的间隔是均匀的，例如 INT8 线性量化将量化范围均匀等分为 256 个数。线性对称量化中零点是根据量化数据类型确定并且零点 z 位于量化定点数范围上的中心对称点，例如 INT8 中零点为 0。线性非对称量化中零点根据公式(6-5)计算确定并且零点 z 一般不在量化定点数范围上的中心对称点。

对称量化是非对称量化的简化版本，理论上非对称量化能够更好的处理数据分布不均匀的情况，因此实践中大多采用非对称量化方案。

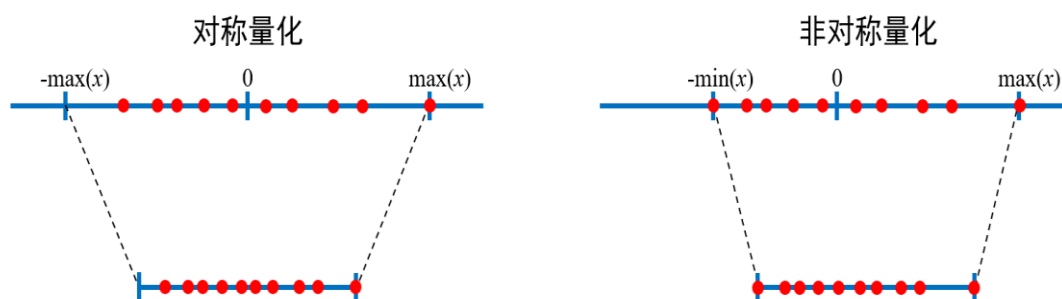


图 6-1 线性对称量化和线性非对称量化

6.1.5 Per-Layer 量化和 Per-Channel 量化

Per-Layer 量化将网络层的所有通道作为一个整体进行量化，所有通道共享相同的量化参数。Per-Channel 量化将网络层的各个通道独立进行量化，每个通道有自己的量化参数。Per-Channel 量化更好的保留各通道的信息，能够更好的适应不同通道之间的差异，提供更好的量化效果。

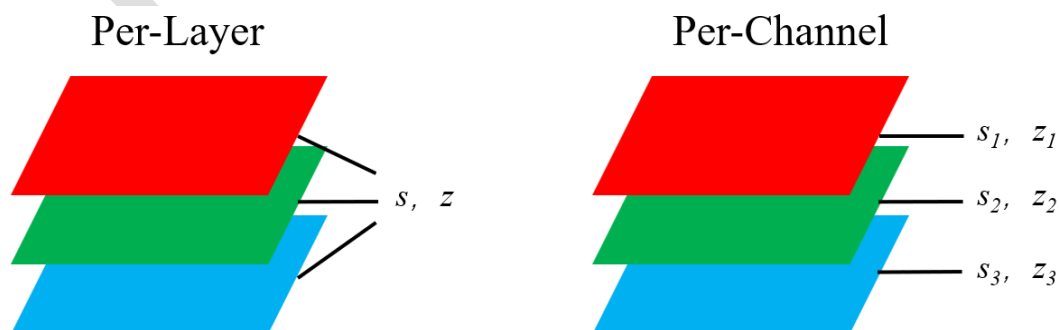


图 6-2 Per-Layer 量化和 Per-Channel 量化

注：RKNN-Toolkit2 中的 Per-Channel 量化中只针对权重进行 Per-Channel 量化，激活值和中间值仍为 Per-Layer 量化。

6.1.6 量化算法

量化比例因子s和零点z是影响量化误差的关键参数，而量化范围的求解对量化参数起到决定性作用。本章节介绍三种关于量化范围求解的算法，Normal，KL-Divergence 和 MMSE。

Normal 量化算法是通过计算浮点数中的最大值和最小值直接确定量化范围的最大值和最小值。从 6.1.2 量化计算原理可知，Normal 量化算法不会产生截断误差，但对异常值很敏感，因为大异常值可能会导致舍入误差过大。

$$q_{min} = \min V \quad (6-6)$$

$$q_{max} = \max V \quad (6-7)$$

其中V为浮点数 Tensor。

KL-Divergence 量化算法计算浮点数和定点数的分布，通过调整不同的阈值来更新浮点数和定点数的分布，并根据 KL 散度最小化两个分布的相似性来确定量化范围的最大值和最小值。KL-Divergence 量化算法通过最小化浮点数和定点数之间的分布差异，能够更好地适应非均匀的数据分布并缓解少数异常值的影响。

$$\underset{q_{min}, q_{max}}{\operatorname{argmin}} H(\Psi(V), \Psi(V_{int})) \quad (6-8)$$

其中H(…)为KL散度计算公式， $\Psi(\cdot)$ 为分布函数，将对应数据计算为离散分布， V_{int} 为量化定点数 Tensor。

MMSE 量化算法通过最小化浮点数与量化反量化后浮点数的均方误差损失，确定量化范围的最大值和最小值，在一定程度上缓解大异常值带来的量化精度丢失问题。由于 MMSE 量化算法的具体实现是采用暴力迭代搜索近似解，速度较慢，内存开销较大，但通常会比 Normal 量化算法具有更高的量化精度。

$$\underset{q_{min}, q_{max}}{\operatorname{argmin}} \|V - \hat{V}(q_{min}, q_{max})\|_F^2 \quad (6-9)$$

其中 $\hat{V}(q_{min}, q_{max})$ 为V的量化、反量化形式， $\|\cdot\|_F$ 为F范数。

6.2 量化配置

6.2.1 量化数据类型

RKNN-Toolkit2 支持的量化数据类型为 INT8。

6.2.2 量化算法建议

Normal 量化算法运行速度快，适用于一般场景。

KL-Divergence 量化算法运行速度略慢于 Normal 量化算法，对于存在非均匀分布的部分模型能够改善量化精度，部分场景下能够缓解少数异常值造成的量化精度丢失问题。

MMSE 量化算法运行速度较慢，内存消耗大，相比 KL_Divergence 量化算法能够更好的缓解异常值造成的量化精度丢失问题。对于量化友好的模型可尝试使用 MMSE 量化算法来提高量化精度，因为在多数场景下 MMSE 量化精度要高于 Normal 和 KL-Divergence 量化算法。

默认情况下使用 Normal 量化算法，当遇到量化精度问题时可尝试使用 KL-Divergence 和 MMSE 量化算法。

6.2.3 量化校正集建议

量化校正集用于计算激活值的量化范围，在选择量化校正集时应覆盖模型实际应用场景的不同数据分布，例如对于分类模型，量化校正集应包含实际应用场景中不同类别的图片。一般推荐量化校正集数量为 20-200 张，可根据量化算法的运行时间适当增减。需要注意的是，增加量化校正集数量会增加量化算法的运行时间但不一定能提高量化精度。

6.2.4 量化配置方法

RKNN-Toolkit2 中量化的配置方法在 `rknn.config()`和 `rknn.build()`接口实现。其中量化方法配置由 `rknn.config()`接口实现，量化开关和校正集路径的选择由 `rknn.build()`接口实现。

`rknn.config()`接口包含以下相关量化配置项：

1. `quantized_dtype`: 选择量化类型，目前仅支持线性非对称的 INT8 量化，默认为 `asymmetric_quantized-8`。
2. `quantized_algorithm`: 选择量化算法，包括 Normal, KL-Divergence 和 MMSE 量化算法。可选值为 `normal`, `kl_divergence` 和 `mmse`，默认为 `normal`。
3. `quantized_method`: 选择 Per-Layer 和 Per-Channel 量化。可选值为 `layer` 和 `channel`，默认为 `channel`。

`rknn.build()`接口包含以下相关量化配置项：

1. `do_quantization`: 是否开启量化，默认为 `False`。
2. `dataset`: 量化校正集的路径，默认为空。

目前支持文本文件格式，用户可以把用于校正的图片(jpg 或 png 格式)或 npy 文件路径放到一个.txt 文件中。文本文件里每一行为一条路径信息，如：

```
a.jpg  
b.jpg
```

如有多个输入，则每个输入对应的文件用空格隔开，如：

```
a0.jpg a1.jpg  
b0.jpg b1.jpg
```

6.3 混合量化

混合量化对模型不同层采用不同的量化数据类型，将不适合量化的层使用较高精度的数据类型表达，以此缓解模型量化精度损失的问题，但混合精度量化会增加额外开销，并且需要用户确定不同层的量化数据类型。

6.3.1 混合量化用法

为了在性能和精度之间做更好的平衡，RKNN-Toolkit2 提供了混合量化功能，用户可以通过精度分析的输出结果来手动指定各层是否进行量化。

目前混合量化功能支持如下用法：

1. 将指定的量化层改成非量化层，如用 `FLOAT16` 进行计算。(因 NPU 上非量化算力较低，所以推理速度会有一定降低)。
2. 每一层的量化参数也可以进行修改。(量化参数不建议修改)

6.3.2 混合量化使用流程

使用混合量化功能时，具体分四步进行。

1. 加载原始模型，生成量化配置文件、临时模型文件和数据文件。具体的接口调用流程如下：

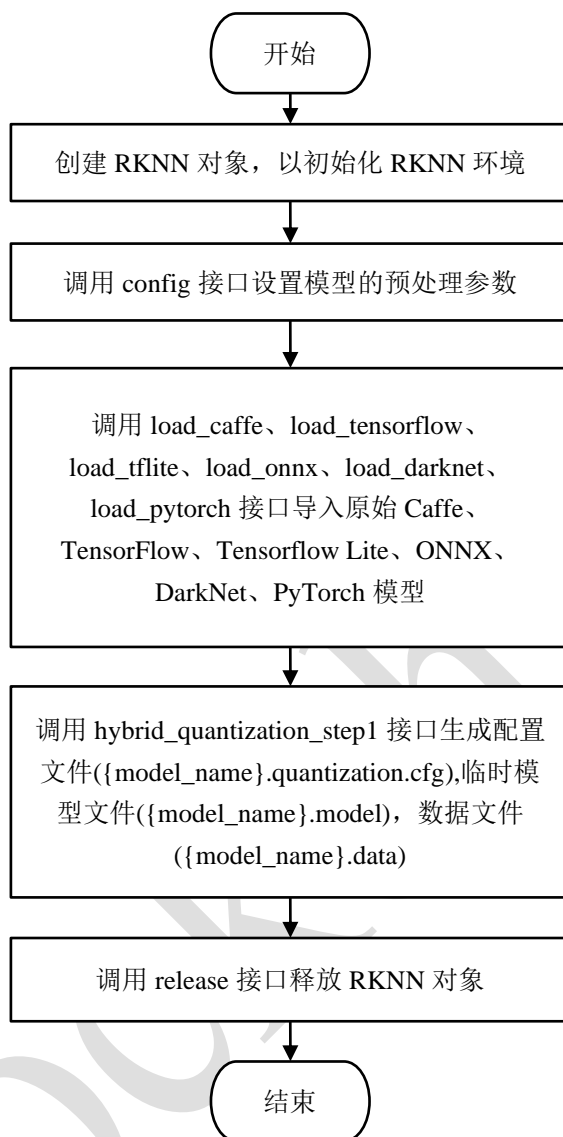


图 6-3 混合量化第一步

2. 修改第一步生成的量化配置文件。

第一步调用混合量化接口 `hybrid_quantization_step1` 完成后会在当前目录下生成名为 `{model_name}.quantization.cfg` 的配置文件。配置文件格式如下：

```

custom_quantize_layers:
  Conv__350:0: float16
  Conv__358:0: float16
  ....
quantize_parameters:
  ....
  FeatureExtractor/MobilenetV2/Conv/Relu6:0:
    qtype: asymmetric_quantized
    qmethod: layer
    dtype: int8
  
```

```
min:
- 0.0
max:
- 6.0
scale:
- 0.023529411764705882
zero_point:
- -128

....
```

`custom_quantize_layers` 下每一行可按照 `tensor` 名: 量化类型的格式添加自定义量化层, 该 `tensor` 对应层的运算类型即改为指定运算类型。目前量化数据类型可选择 `float16`。

`quantize_parameters` 下是模型中每个 `tensor` 的量化参数。每个 `tensor` 的量化参数按照 `tensor` 名: 量化属性和参数的格式呈现。其中 `min/max` 代表量化范围的最小最大值, `tensor` 名可根据精度分析输出结果查看或使用 Netron 打开临时模型文件 `{model_name}.model` 查看对应输出 `tensor` 名。

3. 生成 RKNN 模型。具体的接口调用流程如下:

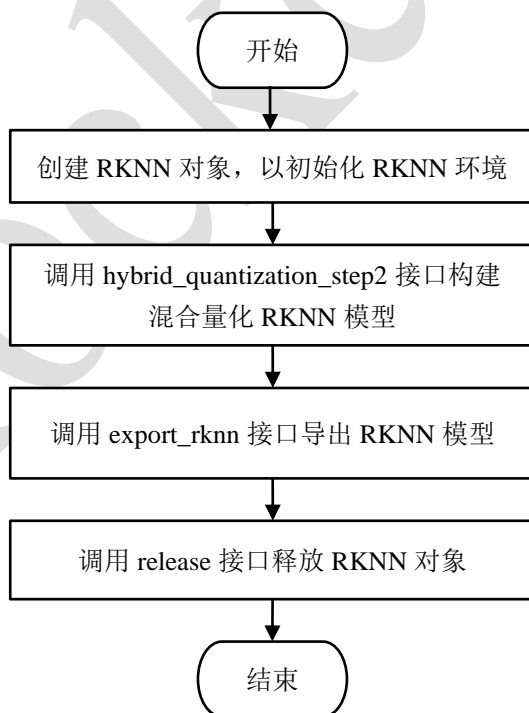


图 6-4 混合量化第三步

4. 使用第三步生成的 RKNN 模型进行推理。

注: RKNN-Toolkit2 工程中 `examples/functions/hybrid_quant` 目录下提供了一个混合量化

的例子，具体可以参考该例子对模型进行混合量化。

6.4 量化感知训练

6.4.1 QAT 简介

量化感知训练（英文名称 Quantization-aware Training，简称 QAT）是一种量化训练方式，该方式旨在解决低比特量化的精度损失问题。低比特量化会掉精度，是因为值域从浮点数调整到定点数时会有精度损失，QAT 训练时会将量化误差计入训练的损失函数，训练出一个带量化参数的模型。

与 RKNN-Toolkit2 工具提供的后训练量化（英文名称 Post Training Quantization，简称 PTQ）对比，两种量化方法的特点如下：

量化方法	基于原始框架二次训练	数据集	权重参数是否被调整	损失函数	性能
后训练量化（PTQ）	不需要	少量未标注数据	否	无关	最优
量化感知训练（QAT）	需要	完整的训练数据集	是	量化损失计入训练损失函数	存在算子不支持 QAT 时，性能略弱于 PTQ

6.4.2 QAT 原理

量化感知训练时，所有权重的存放格式、算子的计算单元都是按照浮点数进行的，这是为了保证反向传播功能可以正常生效、模型的训练可以正常进行。与训练浮点模型不同，在模型可被量化的位置上，量化感知训练会插入 FakeQuantize 模块进行伪量化操作，模拟浮点数调整到定点数的精度损失，使其能被损失函数识别、优化，最终使模型转为定点模型时仍可以保持准确的推理结果。

量化感知训练目前已被广泛使用，各主流推理框架皆有实现，可参考下文所列链接获取更详细的使用说明：

Pytorch - <https://pytorch.org/blog/quantization-in-practice/#quantization-aware-training-qat>

Paddle - https://paddleslim.readthedocs.io/zh-cn/develop/api_cn/dygraph/quantizer/qat.html

Tensorflow - https://www.tensorflow.org/model_optimization/guide/quantization/training

6.4.3 QAT 使用依据

由于 QAT 需要增加额外的训练代码，且部分开源仓库的代码可能与 QAT 功能存在冲

突，推荐在同时满足以下两种情况时，考虑使用 QAT 训练进行模型量化：

1. 参考[章节 7](#)进行量化精度排查，确认 RKNN 的 PTQ 功能不满足精度要求。
2. 尝试[章节 6.3](#)进行混合量化，确认 RKNN 的混合量化功能不满足精度、性能要求。

6.4.4 QAT 实现简例及配置说明

以下是各框架 QAT 功能的说明文档，实际使用请以官方文档为准：

Pytorch: <https://pytorch.org/docs/stable/quantization.html> (Pytorch 目前存在多套量化接口，

RKNN 目前支持 FX 接口生成的量化模型，即 `prepare_qat_fx` 接口及其配套接口)

Paddle: <https://www.paddlepaddle.org.cn/tutorials/projectdetail/3949129#anchor-14>

Tensorflow: https://www.tensorflow.org/model_optimization/guide/quantization/training

这里我们以 Pytorch 为例，说明实现 QAT 的流程及一些需要注意的地方。

```
# for 1.10 <= torch <= 1.13
import torch
class M(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv(3, 8, 3, 1)

    def forward(self, x):
        x = self.conv(x)
        return x

# initialize a floating point model
float_model = M().train()

from torch.quantization import quantize_fx, QConfig, FakeQuantize,
MovingAverageMinMaxObserver, MovingAveragePerChannelMinMaxObserver

qconfig = QConfig(activation=FakeQuantize.with_args(observer=
MovingAverageMinMaxObserver,
quant_min=0,
quant_max=255,
reduce_range=False),
#reudece_range 默认是 True
weight=FakeQuantize.with_args(observer=
MovingAveragePerChannelMinMaxObserver,
quant_min=-128,
quant_max=127,
dtype=torch.qint8,
```

```

                                qscheme=torch.per_channel_affine,
#参数 qscheme 默认是 torch.per_channel_symmetric
                                reduce_range=False))
qconfig_dict = {"": qconfig}
model_qat = quantize_fx.prepare_qat_fx(float_model, qconfig_dict)

# define the training loop for quantization aware training
def train_loop(model, train_data):
    model.train()
    for image, target in data_loader:
        ...
# Run training
train_loop(model_qat, train_loop)

model_qat = quantize_fx.convert_fx(model_qat)

```

以上流程代码中，除了 `qconfig` 的配置针对 RKNN 硬件做了部分调整，其余操作均按照官方代码的指引实现。`qconfig` 中主要有以下两处改动：

activation 量化配置指定 `reduce_range` 为 `False`。`reduce_range` 为 `False` 时，有效量化数值范围为-128~127；`reduce_range` 为 `True` 时，有效量化数值范围为-64~63，量化效果较差。RKNN 硬件支持 `reduce_range` 为 `False`。

weight 量化配置指定 `qscheme` 为 `torch.per_channel_affine`。默认的 `torch.per_channel_symmetric` 会限制 `zero_point` 固定为 0，RKNN 硬件支持 `zero_point` 非 0，故选择 `torch.per_channel_affine`。

6.4.5 QAT 支持的算子

以 Pytorch 为例，使用 QAT 量化的过程中，先对带有权重参数的 Conv, Linear 采取 QAT 规则量化，再检验其他算子，若符合常规量化规则，则对其采用常规量化。在算子不符合 QAT 或常规量化规则的情况下，算子会维持 FP32 的计算规则。

不同框架的支持情况存在差异，用户可以参考以下链接，根据使用的框架及版本，查阅更详细的量化算子支持情况。

Pytorch:

https://github.com/pytorch/pytorch/blob/main/torch/ao/quantization/quantization_mappings.py

Paddle:

<https://github.com/PaddlePaddle/Paddle/blob/86df789a567f1285101c57b6e3ada4b952c58f48/python/paddle/quantization/config.py>

Tensorflow:

https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/quantization/keras/QuantizeConfig

6.4.6 QAT 模型中浮点算子的处理

QAT 模型中，当算子无法量化，会采用浮点进行计算。这类算子在转为 RKNN 模型时，分为两种情况讨论。

1. 前后为可量化算子：

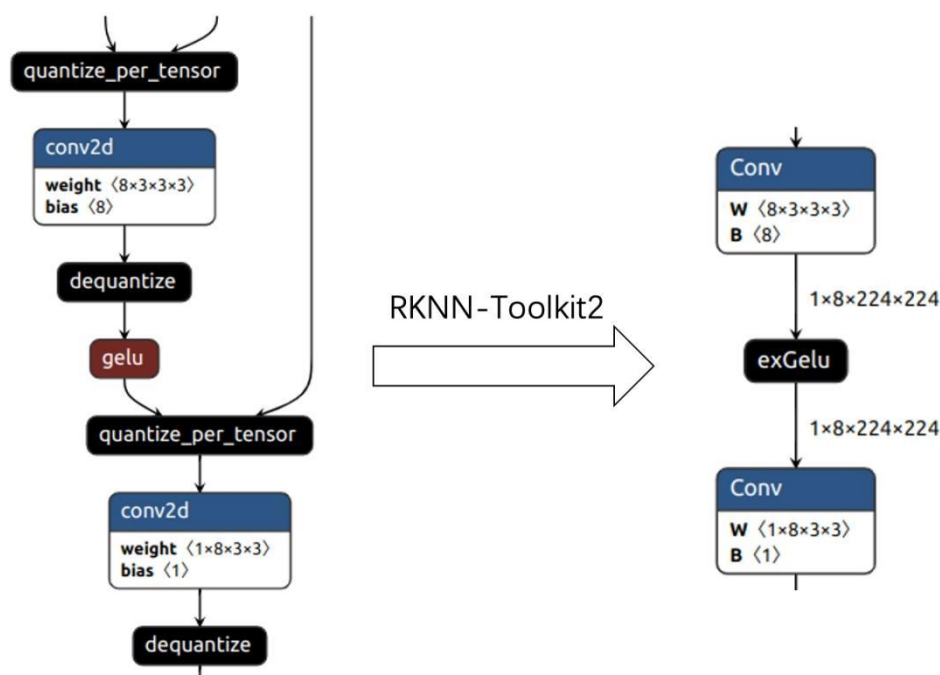


图 6-5 QAT OP 前后为可量化算子状态

如上图左边所示，图中的 gelu 算子在原模型中为浮点算子，其前后的 conv 为量化算子，RKNN-Toolkit2 在加载模型时，会将两个已量化算子中间的浮点算子转为量化算子，提升推理性能，这个操作不会影响精度。转为 RKNN 模型后，结构如上图右边所示。

2. 前后存在非量化算子：

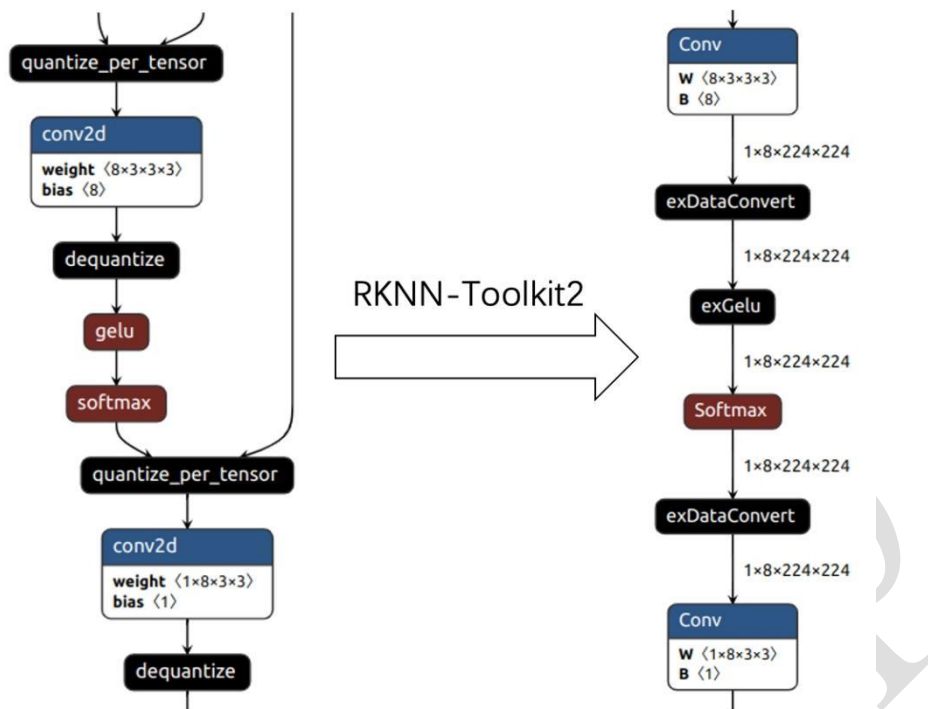


图 6-6 QAT OP 前后为非量化算子状态

如上图左边所示，图中的 `gelu`、`softmax` 算子在原模型中为浮点算子，其前后的 `conv` 为量化算子，RKNN-Toolkit2 在加载模型时，由于 `gelu`、`softmax` 中间的量化参数缺失，`gelu`、`softmax` 仍保持浮点类型。转为 RKNN 模型后结构如上图右边所示，图中 RKNN 模型在 `gelu` 的前面插入反量化算子，`softmax` 后面插入量化算子，这些插入的量化、反量化算子都会增加额外的耗时。

6.4.7 QAT 经验总结

1. QAT 配置

根据不同硬件的特性，QAT 训练往往需要调整配置才能达到更好的效果。对于 RKNPU 而言，建议参考 [6.4.4](#) 的代码说明配置 `qconfig` 参数。

2. 模型中保存的量化参数可能需要二次调整

以 `sigmoid` 为例子，模型中 `sigmoid` 算子记录的量化参数可能不是实际推理时使用的量化参数。在官方的代码 (<https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/quantized/cpu/qsigmoid.cpp>) 中，`sigmoid` 的量化参数在推理时会进行调整，将 `min` 置为 0，`max` 置为 1。

针对这类算子，RKNN-Toolkit2 在转换 QAT 模型时，会在模型转换阶段调整其量化参数，使推理结果和 Pytorch 原始推理结果更接近。

7 精度排查

模型精度问题排查一般从两个方面进行，一是模拟器精度排查，二是板端 Runtime 精度排查。模拟器推理结果正确是板端 Runtime 推理正确的前提，所以需优先保证模拟器推理结果正确，再进行板端 Runtime 精度问题的排查。

因此本章节将针对**模拟器精度排查**以及**Runtime 精度排查**两个方面给出排查建议以及处理方案。下图为具体排查步骤：

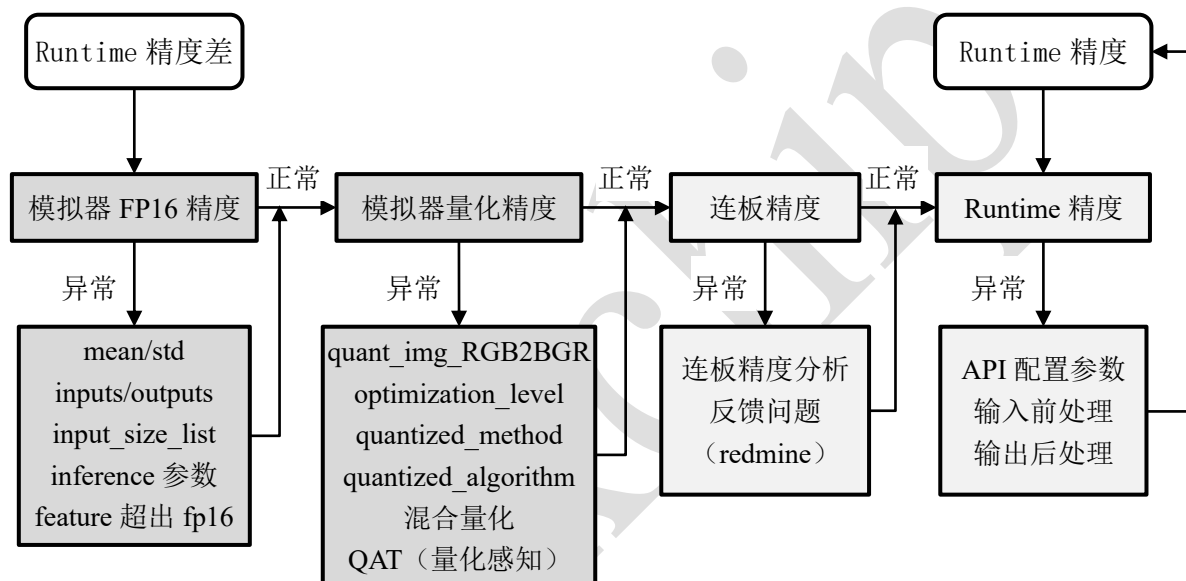


图 7-1 精度排查步骤

- 模拟器精度排查，主要分为**模拟器 FP16 精度**和**模拟器量化精度**排查两个子步骤（上图深色框部分）。
- Runtime 精度排查，主要分为**连板精度**和**Runtime 精度**排查两个子步骤（上图浅色框部分）。

判断精度可以使用余弦距离作为基本的判断依据，但最终量化对精度的影响仍需要在数据集上验证。

7.1 模拟器精度排查

模拟器推理结果正确是板端 Runtime 推理正确的前提，所以需优先保证模拟器推理结果正确。

RKNN-Toolkit2 上的模拟器推理根据模型是否量化分为**FP16 推理**和**量化推理**。FP16 推

理结果正确是量化推理的结果正确的前提，因此当存在量化推理精度问题时，优先验证 FP16 推理的正确性，再排查量化推理的精度问题。

7.1.1 模拟器 FP16 精度

RKNPU 目前不支持 FP32 的计算方式，因此模拟器在不开启量化的情况下，默认是 FP16 的运算类型，所以只需要在使用 rknn.build()接口时，将 do_quantization 参数设置为 False，即将原始模型转换为 FP16 的 RKNN 模型，接着调用 rknn.init_runtime(target=None)和 rknn.inference()接口进行 FP16 模拟推理并获取输出结果。

如果 FP16 推理输出结果错误，则可以进行以下排查：

- 配置错误

模型的配置信息主要集中在 rknn.config()接口里，同时其他的 API 里也有少数的配置信息可能影响 FP16 的精度，主要参数如下：

mean_values / std_values: 模型的归一化参数，一般原始模型的输入归一化操作是放在模型的输入预处理里实现的，但 RKNN 模型在推理时可以包含该归一化的操作（在开启量化后，对量化校正集也会先进行归一化操作），因此在原始模型有归一化步骤时，要确保该参数和原始模型使用的归一化参数一致。

input_size_list: rknn.load_tensorflow()、rknn.load_pytorch()和 rknn.load_onnx()接口的输入节点 shape 信息，如果配置错误会导致错误的推理结果。

inputs / outputs: rknn.load_tensorflow()和 rknn.load_onnx()接口的输入和输出节点名称，如果配置错误会导致错误的推理结果。

inference 接口参数: rknn.inference()接口的输入参数，主要包括 inputs 和 data_format。

- 一般在 Python 环境下，图像数据都是通过 cv2.imread()读取的，此时需要注意 cv2.imread()读取的图像格式为 BGR，如果原始模型的输入为 BGR（如大部分的 caffe 模型），则不需要做 RGB 顺序的调整；而如果原始模型的输入为 RGB，则需要调用 cv2.cvtColor(img, cv2.COLOR_BGR2RGB)将图像数据转为 RGB；另外，通过 cv2.imread()读取的图像的 shape 维度为 3 维，但是一般模型的输入 shape 为 4 维，因此还需要调用 np.expand_dims(img, 0)将输入 shape 扩为 4 维；之后才可以传给 rknn.inference()接口进行推理。通过 cv2.imread()读取的图像的 layout 为 NHWC，

data_format 的默认值也是 NHWC，因此不需要设置 data_format 参数。

- 如果模型的输入数据不是通过 cv2.imread() 读取，此时必须清楚知道输入数据的 layout 并设置正确的 data_format 参数，同时也要确保输入数据的 shape 和原始模型一致。如果输入数据是图像数据，也要确保其 RGB 顺序与原始模型一致。

参数配置的检查是很重要的环节，是很多用户出现 FP16 推理输出结果错误的主要原因。

具体排查步骤如下：

- a. 使用原始模型在原始推理框架下进行推理，并将推理结果保存下来。
- b. 使用 RKNN-Toolkit2 对原始模型进行转换并推理，此时需要使用与前一步骤里同样的输入数据，并设置 FP16 的推理方式（rknn.build() 的 do_quantization 设为 False），同时 rknn.init_runtime() 的 target 参数设为 None，以调用 RKNN-Toolkit2 的模拟器进行推理，同样将推理的结果保存下来。
- c. 对比两次推理的结果，如果结果较为一致（可以用余弦距离来判断一致性），说明上述的配置都没有问题。
- d. 如果结果不一致，检查上述参数是否正确。

如果确认上述参数配置无误，结果仍然不一致，则有可能是模型中间的 Tensor 超出 FP16 表达范围导致。

- 超出 FP16 表达范围

模型的中间 Tensor 在由 FP32 转为 FP16 后，可能会出现运算溢出的问题。因为一般模型的推理数据类型是 FP32，如果推理时模型中的 Tensor 有数值超过 FP16 表达范围（-65504~65504），则该 Tensor 就会溢出，导致模型推理结果异常。

对于溢出问题，可以通过调用 rknn.accuracy_analysis(..., target=None) 接口（参考 [3.2.2](#) 章节）进行模拟器 FP16 精度分析，如果分析结果中的 simulator_error 的 entire 列或 single 列出现异常值（出现 ‘inf’ 等的字样），则可能出现了 FP16 溢出。此时可以尝试修改模型结构来保证模型中的所有 Tensor 不会出现 FP16 溢出（如添加一些 BN 层等）。

如果确认上述参数配置无误，并且也不是 FP16 溢出，但结果仍然不一致，则可能是模拟器内部实现问题，请将该模型的复现文件提供给瑞芯微 NPU 团队进行分析解决。**模拟器量化精度**

在排除 FP16 精度问题后，就可以对模型进行量化（使用 rknn.build() 接口时，将 do_quantization 参数设置为 True），然后通过调用 rknn.accuracy_analysis(..., target=None) 接

口（参考 3.2.2 章节）进行模拟器量化精度分析。

如果在分析结果中，发现 `simulator_error` 的 `entire` 列精度下降的比较厉害，并且 `simulator_error` 的 `single` 没有发现有哪层精度下降特别多的情况，则主要从以下几个方面进行排查：

- 配置错误

与 FP16 推理的配置问题类似，错误的配置也会导致量化推理精度问题，因此在保证 FP16 推理正确的配置基础上，仍然要对以下量化配置参数进行检查。

`quant_img_RGB2BGR`：表示在加载量化图像时是否需要先做 RGB2BGR 的操作，一般用于 caffe 模型，更多详细信息见 `quant_img_RGB2BGR` 参数说明，该参数务必和训练时的图像通道顺序保持一致，在配置错误时也会导致量化精度下降比较多。

`optimization_level`：优化等级的选择，默认为 3，表示速度优先，这种情况下会开启一些对提升性能有益，但却会略微影响到精度的优化规则，将该配置调小（如改为 0），则会禁用这些优化规则。

`dataset`：`rknn.build()`接口的量化校正集配置，用于在量化过程中，计算每个 Tensor 合适的量化参数（`scale / zero_point`）。如果选择了和实际部署场景差异较大的校正集，则可能会出现精度下降的问题，此外校正集的数量过多或过少都会影响精度（一般选择 20~200 张）。

具体检查量化参数配置问题，一般可按如下步骤进行：

1. 直接进行量化推理，然后检查推理的结果与原始模型在原始推理框架下推理的结果进行比较，如果结果差异不是很大，则可以认为 `quant_img_RGB2BGR` 和 `dataset` 参数基本无误。
2. 如果结果差异还是很明显：
 - a. 如原始模型输入的图像格式是 BGR（多见于 caffe 的模型），此时可以修改 `quant_img_RGB2BGR` 为 True，关于模型输入的 RGB 顺序，其实从前面 FP16 推理精度验证步骤的输入数据的处理代码中可以得知输入数据的 RGB 顺序。
 - b. 可以先使用一张图像进行量化（`dataset.txt` 中只留一行），推理时也使用这张图像进行推理，如果此时单张图像的精度提升较多，则说明先前使用的量化校正集选择不佳，可以重新选择与部署场景较吻合的图片（如果提升并不明显，则可能不是 `dataset` 的问题）。
 - c. 如原先只使用一张图像进行量化（`dataset.txt` 中只有一行），此时可以尝试使

用更多的图像进行量化，可以提高到 20~200 张左右。

经过上述配置排查之后，应该不会出现量化结果完全错误的情况，如果出现完全错误的情况，请重新检查上述的配置。在确认配置无误的情况下，如果模型的精度还是不够，可以尝试修改量化方法和算法等相关配置。

● 量化方法和量化算法

有些模型本身对量化并不友好，此时可以尝试切换不同的量化方法和量化算法。目前量化方法主要有两种，分别是 `layer` 和 `channel`，可通过 `rknn.config()` 接口里的 `quantized_method` 参数进行设置（默认是 `channel`）。量化算法主要分为三种，分别是 `normal`，`kl_divergence` 和 `mmse`，可通过 `rknn.config()` 接口里的 `quantized_algorithm` 参数进行设置（默认是 `normal`）。步骤如下：

1. 如原先使用的是 `layer` 的量化方法，可以改为 `channel` 的量化方法，一般情况下，`channel` 的量化方法精度比 `layer` 的量化方法精度会高许多。
2. 如量化方法已经是 `channel`，但精度还是无法满足要求，此时可以将量化算法由 `normal` 改为 `kl_divergence` 或 `mmse`，这种方式会导致量化的时间大幅增加，但会带来比 `normal` 更好的精度表现，同时运行时的性能并不会受到影响。

如果使用上述方法后，从分析结果中仍然发现 `simulator_error` 的 `entire` 列精度还是不好，并且 `simulator_error` 的 `single` 列有部分层精度掉的比较多，这可能是这些层的权重数值分布不好，导致量化后会出现精度下降较多的情况。如：`Conv` 的 `weight` 的分布很不均匀的情况下，此时可以考虑使用**混合量化**来进一步提高模型精度。步骤如下：

1. 先使用精度分析接口对精度进行分析，找出造成精度下降比较多的层，记录对应层的输出 `Tensor name`。（这边需要注意的是，因为误差是会逐层累积的，所以越前面的层对最终的精度影响也会越大，因此不仅要考虑 `simulator_error` 的 `single` 列的精度损失情况，也要考虑层在模型中的位置）
2. 使用混合量化的方法，将上个步骤获取的 `Tensor name` 写入混合量化的配置文件中（参考 [6.3](#) 章节）。
3. 完成混合量化的步骤，并测试精度情况（可以继续使用精度分析接口来看精度变化情况）。

一般经过混合量化之后，模型的精度是可以提高的，如果提高不明显或不够，则可以尝试将更多的层进行混合量化，但是同时也会造成推理性能下降，因此混合量化需要用户自行权衡精度和速度。还有一种特殊方式是，当精度下降的 `Op` 处于最后一层时，也可以选

择将该层 Op 的运行放在后处理中进行，同样会有效避免该层的精度问题。

- QAT 量化感知训练

如果混合量化后精度还是不够，或者精度够但因混合量化而导致性能达不到要求，此时可以尝试使用量化感知训练重新训练模型，并导出带有量化参数的模型（如 onnx/pt/pb/tflite 格式），有关量化感知训练更多内容，请参考 [6.4](#) 章节。

7.2 Runtime 精度排查

在模拟器精度正常的情况下，仍可能在板端 C API 部署时出现推理结果异常。出现这种问题的原因一般有三种，第一种是板端的 Runtime 的 bug 导致；第二种是调用 RKNPU2 的 C API 编程时接口没有正确使用导致；第三种是模型的前后处理不正确导致。

当遇到这种问题时，可以先通过连板功能快速排查是否是板端 Runtime 的 bug 导致，如果连板没有问题，再排查 C API 部署的问题。

7.2.1 连板精度

1. 在配置好连板调试环境的情况下（连板调试环境配置方法参考 [2.2](#) 章节），将开发板通过 USB 连接到电脑上，然后使用 RKNN-Toolkit2 进行连板推理（设置 `rknn.init_runtime()` 的 `target` 参数，如 `target='rk3566'`），并检查推理结果是否大致正确（因为模拟器并没有严格模拟 NPU 硬件，所以结果可能与模拟器并没有完全一致）。
2. 如果上述步骤里的推理结果与模拟器推理结果差异较大，则可以初步确定板端的 Runtime 存在 bug，此时可以使用精度分析的接口（参考 [3.2.2](#) 章节）进行连板精度分析（调用 `rknn.accuracy_analysis()` 接口，并设置 `target` 参数即可，如 `target='rk3566'`），精度分析完后会输出每层的分析结果。
3. 检查分析结果中的 `runtime_error` 的 `single_sim` 列，如其 `cos` 余弦距离偏低或 `euc` 欧氏距离偏高（显示黄色或红色），从而导致 `runtime_error` 的 `entire` 列与 `simulator_error` 的 `entire` 列差异越来越大，则可能 Runtime 在实现该层时有出现精度丢失或异常的问题，此时可以将该分析结果以及复现的模型反馈给瑞芯微 NPU 团队进行修复。

7.2.2 Runtime 精度

如果连板精度没有问题，但精度仍然有问题，则问题可能出在用户调用 RKNN 的 C API 进行编程的 C/C++代码本身，这时用户需要仔细检验下 RKNN 的 C API 的接口配置等是否配置正确，以及模型的前处理和后处理流程是否正确（需要与模拟器端的流程完全一致）。可以按照以下步骤查看：

1. 检查输入配置和数据

查看 C API 的输入是否配置正确。例如，RKNN-Toolkit2 在转换 RKNN 模型时已经配置均值和方差，则在 C/C++代码中不需要做归一化。对于 3 通道的输入，通道顺序与模型训练时设置的输入通道一致；对于四维输入形状，fmt=NHWC；对于非四维输入，fmt=UNDEFINED。若使用通用 API，输入 buffer 的 size 等于输入 Tensor 元素个数*每个元素的字节数，若使用零拷贝 API，rknn_create_mem 接口创建的内存大小以及输入数据格式参考《RKNN Runtime 零拷贝调用》章节。

在确认配置正确后，需查看输入层的数据，可以在应用运行前设置 RKNN_LOG_LEVEL=5，然后再运行应用，推理时逐层的结果会以 numpy 格式文件保存在 /data/dumps（Android 系统）或 /userdata/dumps（Linux 系统）目录下。查看包含 InputOperator 字段的 numpy 文件是否符合预期，如果使用通用 API，它是输入归一化的结果；如果使用零拷贝 API，它是未归一化前的数据。

2. 检查输出配置和数据

在确保输入正确后，查看代码中输出是否配置正确。例如，如果使用通用 API，当设置 want_float=1 后，输出是 float32 类型结果，当设置 want_float=0 后，输出是量化数据类型或者 float16 类型(非量化数据类型)。如果使用零拷贝 API，rknn_create_mem()接口创建的内存大小以及输入数据格式参考《RKNN Runtime 零拷贝调用》章节。

查看输出层的数据，同样在上述逐层 numpy 文件生成后，打开包含 OutputOperator 字段的 numpy 文件，查看数据是否正确。如果确认输入结果正确，输出仍然错误，可能 Runtime 在特定的输入/输出类型处理上有问题，此时可以将该分析结果以及复现的模型反馈给瑞芯微 NPU 团队进行修复。

8 性能优化

模型部署时，用户在跑通结果后，会有进一步的性能优化需求，此节将从完整的模型性能优化流程来介绍如何调优。并展开介绍用户最常用的操作：模型性能分析，图级别优化，算子级别优化。完整优化流程如下图所示：

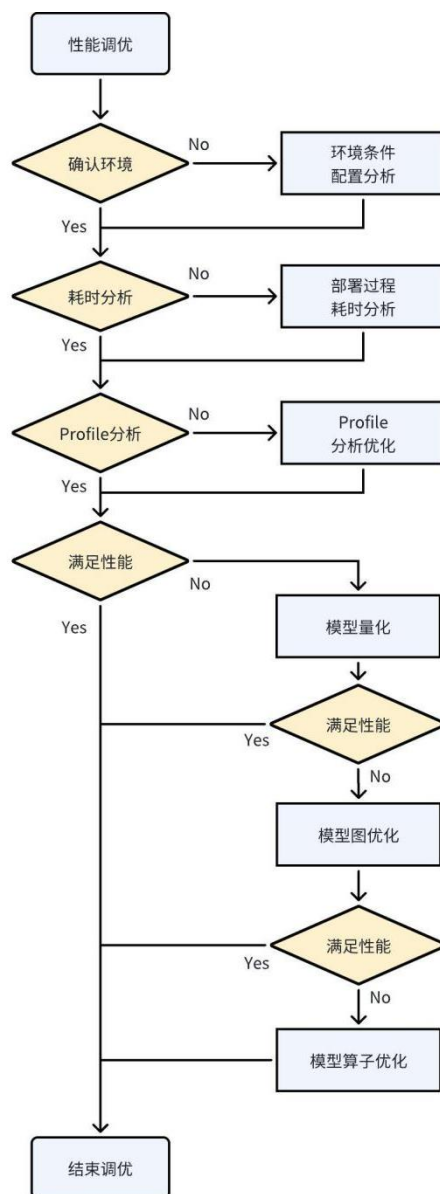


图 8-1 模型性能优化流程

8.1 模型性能优化前期分析流程

8.1.1 环境条件与配置检查

在所有性能分析及优化之初，应该优先确定测试环境的基准，只有在基准相同时，所测性能数据才是有意义的。例如非定频下测试同一模型的推理性能波动幅度是比较大的，无法以某一批测试耗时作为平均的单帧耗时。

查询和设置测试环境的条件和配置有如下几个方面：

- 查询和设置 CPU、DDR、NPU 频率

定频频率直接影响运行速率，频率越高性能相对越好。频率变化与性能提升不是线性关系。频率增加相对来说功耗也会增加。定频命令参考如下：

https://github.com/airockchip/rknn_model_zoo/blob/main/scaling_frequency.sh

也可以简单的使用如下命令设置为性能模式：

```
echo performance | tee $(find /sys/ -name *governor)
/dev/null || true
```

- 检查 NPU 内核 Driver 版本

有些功能或者算子的性能优化项与内核驱动版本有关，较新的内核驱动版本能应用到最新的底层优化实现。所以请检查是否使用到较新的内核驱动版本，目前建议更新到 0.9.2 之后的版本。该环节非强制更新。

检查内核驱动版本命令如下：

```
cat /sys/kernel/debug/rknpu/version # for
RK3566/RK3568/RK3588/RK3562/RK3576
cat /proc/rknpu/version # for RV1106/RV1103
```

- 检查 NPU 的负载

NPU 的负载为单位时间内 NPU 执行任务的时间占比。负载能够反应 NPU 的繁忙程度，如果查询到的负载较低，则表明 NPU 等待任务提交的时间较长，需要检查数据输入输出拷贝用时、应用程序前后处理优化等等。或者在应用程序中使用多线程处理方式来提升 NPU 负载。

另外需要注意的是，NPU 的负载不代表实际的 MAC 利用率，MAC 利用率反应的是执行中任务在 NPU 硬件单元中的执行效率。

查询 NPU 负载的命令如下：


```
cat /sys/kernel/debug/rknpu/load  
# or  
cat /proc/debug/rknpu/load
```

8.1.2 部署过程耗时分析

整个部署程序的推理部分耗时的占用有如下三个方面：用户应用程序耗时、输入输出数据拷贝耗时、模型推理耗时。分析各环节耗时占比能够直观的确定优化重点。测定这些步骤的耗时可以在应用程序上通过打时间戳的方式获得。

- 用户应用程序耗时

用户应用程序耗时主要指推理过程中非 NPU 相关的耗时，一般来说主要是数据的前处理、后处理和逻辑代码的耗时。这部分耗时由用户全权控制。用户在发现应用程序的耗时占总体耗时非常高时，除精简优化代码以外，可以尝试将一部分操作通过专用硬件加速。

例如将一些矩阵乘加操作，采用 Matmul API 接口来调用 NPU 辅助执行计算。又如部分图像的缩放类操作可以调用 RGA 的接口来实现加速。

- 输入输出拷贝耗时

当采用通用 API 时，用户设置的输入输出内存与 NPU 的内存是存在拷贝耗时的，这个耗时可以在调用通用 API 时打印出来。拷贝耗时取决于 DDR 与 CPU 的性能，在输入输出数据量较小的时候 Normal API 的拷贝耗时较低，但数据量较大时，Normal API 的耗时就不可忽略。因此更多推荐采用零拷贝 API。

当采用零拷贝 API 时，用户设置的输入输出内存被 NPU 直接访问，所以输入输出拷贝耗时为 0。零拷贝的接口的使用详细参考 [5.2](#) 章节。

- 推理耗时

NPU 执行推理的耗时，该部分耗时直接体现部署模型的耗时。受推理模型规模、编译优化版本影响。RKNN 的 LOG 打印的推理耗时在不同 RKNN_LOG_LEVEL 等级时不一样，因为 LOG 打印存在一定的耗时。一般在查看单帧推理耗时时，设置的 LOG 等级为 1，并且跑多次取平均为准。

8.2 模型性能分析

8.2.1 获取 Profile 信息

当需要了解模型推理逐层耗时情况时，可以在运行程序前输入以下指令打印详细信息：

```
export RKNN_LOG_LEVEL=4
./run_rknn_test ./test.rknn ./input.jpg
```

如果是 Android 平台，运行后可以通过 logcat 命令获取详细日志。

如果是使用 rknn-toolkit2，你可以使用如下方式来获取每层的耗时：

```
rknn.init_runtime(target=platform, perf_debug=True)
rknn.eval_perf()
```

性能分析报告信息如下（仅截出性能相关部分）

Operator Time Consuming Ranking Table						
OpType	CallNumber	CPUTime(us)	GPTime(us)	NPUTime(us)	TotalTime(us)	TimeRatio(%)
ConvLeakyRelu	7	0	0	4584	4584	53.77%
MaxPool	6	0	0	2273	2273	26.66%
Conv	2	0	0	846	846	9.92%
ConvAdd	1	0	0	511	511	5.99%
Split	1	0	0	152	152	1.78%
LeakyRelu	1	0	0	68	68	0.80%
OutputOperator	1	62	0	0	62	0.73%
InputOperator	1	29	0	0	29	0.34%

图 8-2 性能分析报告

ID	OpType	DataType	Target	InputShape	OutputShape	DDRCycles	NPUcycles	MaxCycles	Time(us)
0	InputOperator	UINT8	CPU	\	(1,3,480,640)	0	0	0	9
1	ConvExSwish	UINT8	NPU	(1,3,480,640),(32,3,3,3),(32)	(1,32,240,320)	794117	1382400	1382400	2805
2	ConvExSwish	INT8	NPU	(1,32,240,320),(64,32,3,3),(64)	(1,64,120,160)	665207	691200	691200	1605
3	ConvExSwish	INT8	NPU	(1,64,120,160),(32,64,1,1),(32)	(1,32,120,160)	332036	153600	332036	832
4	ConvExSwish	INT8	NPU	(1,64,120,160),(32,64,1,1),(32)	(1,32,120,160)	332036	153600	332036	788
5	ConvExSwish	INT8	NPU	(1,32,120,160),(32,32,1,1),(32)	(1,32,120,160)	248988	153600	248988	722
6	ConvExSwish	INT8	NPU	(1,32,120,160),(32,32,3,3),(32)	(1,32,120,160)	250087	345600	345600	768
7	Add	INT8	NPU	(1,32,120,160),(1,32,120,160)	(1,32,120,160)	329574	0	329574	298
8	Concat	INT8	NPU	(1,32,120,160),(1,32,120,160)	(1,64,120,160)	494407	0	494407	453
9	ConvExSwish	INT8	NPU	(1,64,120,160),(64,64,1,1),(64)	(1,64,120,160)	497258	307200	497258	1427
10	ConvExSwish	INT8	NPU	(1,64,120,160),(128,64,3,3),(128)	(1,128,60,80)	401854	691200	691200	962
11	ConvExSwish	INT8	NPU	(1,128,60,80),(64,128,1,1),(64)	(1,64,60,80)	167682	76800	167682	405

图 8-3 性能分析报告

可以针对总体性能 Profile 和逐层性能 Profile 快速定位想要的信息。并根据数据来制定后续不同侧重的优化策略。获得 Profile 后可以进行分析：分析逐层耗时，找出高耗时算子；分析非 NPU 算子影响；分析 NPU 算子性能瓶颈。下文将详细讨论。

8.2.2 分析逐层耗时

如下图中，可以从 Time 一栏找出耗时高的算子，优先优化高耗时算子。也可以兼顾观察 Op Type 一栏找出高耗时的算子是不是属于同类 OpType，以便统一优化。

ID	OpType	DataType	Target	InputShape	OutputShape	DDRCycles	NPUCycles	MaxCycles	Time(us)
117	Sigmoid	INT8	NPU	(1,83,60,80)	(1,83,60,80)	186625	0	186625	541
118	ConvRelu	INT8	NPU	(1,128,60,80),(128,128,3,3),(128)	(1,128,60,80)	327096	1382400	1382400	1595
119	ConvRelu	INT8	NPU	(1,128,60,80),(128,128,3,3),(128)	(1,128,60,80)	327096	1382400	1382400	1596
120	Conv	INT8	NPU	(1,128,60,80),(4,128,1,1),(4)	(1,4,60,80)	103405	19200	103405	150
121	Conv	INT8	NPU	(1,128,60,80),(1,128,1,1),(1)	(1,1,60,80)	103345	19200	103345	150
122	Sigmoid	INT8	NPU	(1,1,60,80)	(1,1,60,80)	32181	0	32181	142
...									
144	Mul	INT8	NPU	(1,83,60,80),(1,1,60,80)	(1,83,60,80)	319655	0	319655	414
145	ReduceMax	INT8	CPU	(1,83,60,80)	(1,1,60,80)	0	0	0	7441
146	ArgMax	INT8	CPU	(1,83,60,80)	(1,1,60,80)	0	0	0	7389
147	Reshape	INT64	CPU	(1,1,60,80),(3)	(1,60,80)	0	0	0	69
148	OutputOperator	INT64	CPU	(1,60,80)	\	0	0	0	13

图 8-4 高耗时算子性能分析

值得说明的是，高耗时算子不一定是低效算子，某些算子是高算力消耗的，高耗时的算子如果 Mac 利用率很高时，应该考虑能否降低此算子的尺寸规模以减少耗时。但当利用率很低的高耗时算子出现时，这类算子就是优化重点。

8.2.3 分析 CPU 算子影响

如下图中，可以看到某些高耗时的算子是运行在 CPU 上的，将这些 CPU 算子 NPU 化将可以极大改善高耗时影响。一般来说，用户的大部分性能优化问题都会在将 CPU 算子 NPU 化后得到解决。因此要重点关注 CPU 算子的耗时情况。

Operator Time Consuming Ranking Table						
OpType	CallNumber	CPUTime(us)	GPTime(us)	NPTime(us)	TotalTime(us)	TimeRatio(%)
ConvRelu	125	0	0	54613	54613	42.09%
ConvExSwish	35	0	0	20047	20047	15.45%
ReduceMax	7	16104	0	0	16104	12.41%
ArgMax	7	14833	0	0	14833	11.43%
Concat	36	0	0	10464	10464	8.06%
Conv	25	0	0	3685	3685	2.84%
exSoftmax13	1	0	0	1916	1916	1.48%
Resize	11	0	0	1893	1893	1.46%
Sigmoid	12	0	0	1595	1595	1.23%
Reshape	14	388	0	699	1087	0.84%
Mul	7	0	0	1065	1065	0.82%
Add	7	0	0	1046	1046	0.81%
MaxPool	9	0	0	784	784	0.60%
OutputOperator	32	569	0	0	569	0.44%
ConvSigmoid	1	0	0	40	40	0.03%
InputOperator	1	9	0	0	9	0.01%

图 8-5 CPU 算子性能分析

一般来说算子运行在非 NPU 上的原因有如下几种：

- 算子尺寸超限（查询 OpList 文档的算子尺寸限制）
- 算子尚未支持在 NPU 上运算（查询 OpList 是否支持该算子，可以在 Github 工程

上提 Issue)

- NPU 硬件限制无法支持（是否可以算法等效成其他 NPU 可支持的其他实现）

8.2.4 分析 NPU 算子性能瓶颈

考虑 NPU 算子的高耗时问题时，可以根据 DDR Cycles/NPU Cycles/Total Cycles 这三栏来判断该算子耗时的理论瓶颈是带宽瓶颈还是算力瓶颈。这里的 DDR Cycles 是根据 NPU 频率换算过后的数据，指该层算子读写数据换算成 NPU 频率下所需的 Cycle 数，因此可以直接与 NPU Cycle 比较。

如下图所示：

例如第三层 DDR Cycles 远大于 NPU Cycles 时，说明该层读写数据花费 Cycle 数量远大于运算所需 Cycle 数量，所以该 Conv 瓶颈来自带宽。

例如第十二层 DDR Cycles 远小于 NPU Cycles 时，说明该层读写数据花费 Cycle 数量远小于运算所需 Cycle 数量，所以该 Conv 瓶颈来自算力。

ID	OpType	DataType	Target	InputShape	OutputShape	DDR Cycles	NPU Cycles	Total Cycles	Time(us)	MacUsage(%)	Task Number	Regcmd Size	RV
0	InputOperatc	UINT8	CPU	\	(1,3,300,300)	0	0	0	12	\	0	0	0
1	Conv	UINT8	NPU	(1,3,300,300)	(1,3,300,300)	261543	1582	261543	585	0.3	0	0	1
2	Split	INT8	CPU	(1,3,300,300)	(1,1,300,300)	0	0	0	6677	\	0	0	0
3	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1556	1.23	0	0	1
4	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1505	1.27	0	0	1
5	Conv	INT8	NPU	(1,1,300,300)	(1,8,150,150)	275553	17226	275553	1573	1.22	0	0	1
6	Concat	INT8	CPU	(1,8,150,150)	(1,24,150,150)	0	0	0	4684	\	0	0	1
7	Conv	INT8	NPU	(1,24,150,150)	(1,64,150,150)	329723	67500	329723	684	10.96	0	0	2
8	Clip	INT8	NPU	(1,64,150,150)	(1,64,150,150)	438387	0	438387	729	\	0	0	2
9	MaxPool	INT8	NPU	(1,64,150,150)	(1,64,75,75)	275490	0	275490	725	\	0	0	1
10	Conv	INT8	NPU	(1,64,75,75)	((1,64,75,75)	110676	45000	110676	281	17.79	0	0	0
11	Clip	INT8	NPU	(1,64,75,75)	((1,64,75,75)	109676	0	109676	275	\	0	0	0
12	Conv	INT8	NPU	(1,64,75,75)	((1,192,75,75)	253595	1215000	1215000	1456	92.72	0	0	1
13	Clip	INT8	NPU	(1,192,75,75)	(1,192,75,75)	328817	0	328817	599	\	0	0	2
14	MaxPool	INT8	NPU	(1,192,75,75)	(1,192,38,38)	206599	0	206599	535	\	0	0	1
15	AveragePool	INT8	CPU	(1,192,38,38)	(1,192,38,38)	0	0	0	16159	\	0	0	0
16	Conv	INT8	NPU	(1,192,38,38)	(1,32,38,38)	50564	17328	50564	192	10.03	0	0	0
17	Clip	INT8	NPU	(1,32,38,38)	((1,32,38,38)	14169	0	14169	121	\	0	0	0
18	Conv	INT8	NPU	(1,192,38,38)	(1,64,38,38)	58609	34656	58609	193	19.95	0	0	0
19	Clip	INT8	NPU	(1,64,38,38)	((1,64,38,38)	28233	0	28233	132	\	0	0	0
20	Conv	INT8	NPU	(1,64,38,38)	((1,96,38,38)	43855	155952	155952	304	57	0	0	0
21	Clip	INT8	NPU	(1,96,38,38)	((1,96,38,38)	42297	0	42297	152	\	0	0	0
22	Conv	INT8	NPU	(1,96,38,38)	((1,96,38,38)	55095	233928	233928	399	65.14	0	0	0
23	Clip	INT8	NPU	(1,96,38,38)	((1,96,38,38)	42297	0	42297	153	\	0	0	0
24	Conv	INT8	NPU	(1,192,38,38)	(1,64,38,38)	58609	34656	58609	144	26.74	0	0	0

图 8-6 NPU 算子性能瓶颈分析

目前 NPU Cycles 一栏主要显示 Conv 所需的 Cycles，其他算子类型后续会逐步补充。

8.3 量化加速

模型量化能大幅降低模型的运算规模，节约带宽消耗。由于采用了 INT8 的量化的卷积采用的是 INT8 的运算单元计算。同时算力上，相比于 Float16 的运算单元，INT8 的运算单元规模更大，理论算力更高。模型量化的具体使用方式详见[第六章](#)。

8.4 图级别优化

模型的图级别优化是最容易从整体角度去统筹优化模型的方法。在分析出耗时占比较高的算子或图区域后，我们可以有多种不同的方式去改造图进而达成优化的目的。图优化主要以节省多余算子、非 NPU OP 的 NPU 化、面向硬件高效率算子改造等为目标。这些目标有可能有些时候是存在矛盾的，例如为了非 NPU OP 的 NPU 化，可能需要额外多出几个算子，看似违背了节省多余算子的目标，但总体推理性能提升，便是有意义的。

在 RKNN-Toolkit2 工具链中，软件栈在转换模型的过程中已经会进行一定程度上的图优化。但这一过程不是万能和尽善尽美的，有些未被考虑的场景仍然会出现冗余的操作，用户可以根据本节介绍的一些思路来进行预先性的图优化。以下仅作为每一种优化方法的介绍，不是强制固定，实际场景需要灵活运用。

8.4.1 非 NPU OP 通过图变换实现 NPU 化

对于非 NPU op，可以做一些等效的图变换来，替换成 NPU 可支持的算子，以达成 NPU 化的优化目的。

例如下图，以 shufflenetv2_0.5 模型为例，将其中 channel shuffle 操作改为卷积近似替换。weight 数值为 0/1，可以达成重排数据的效果，假如无法在 NPU 实现该 Transpose、Reshape 操作，可以将这些算子整合成 Conv 算子实现数据重排。

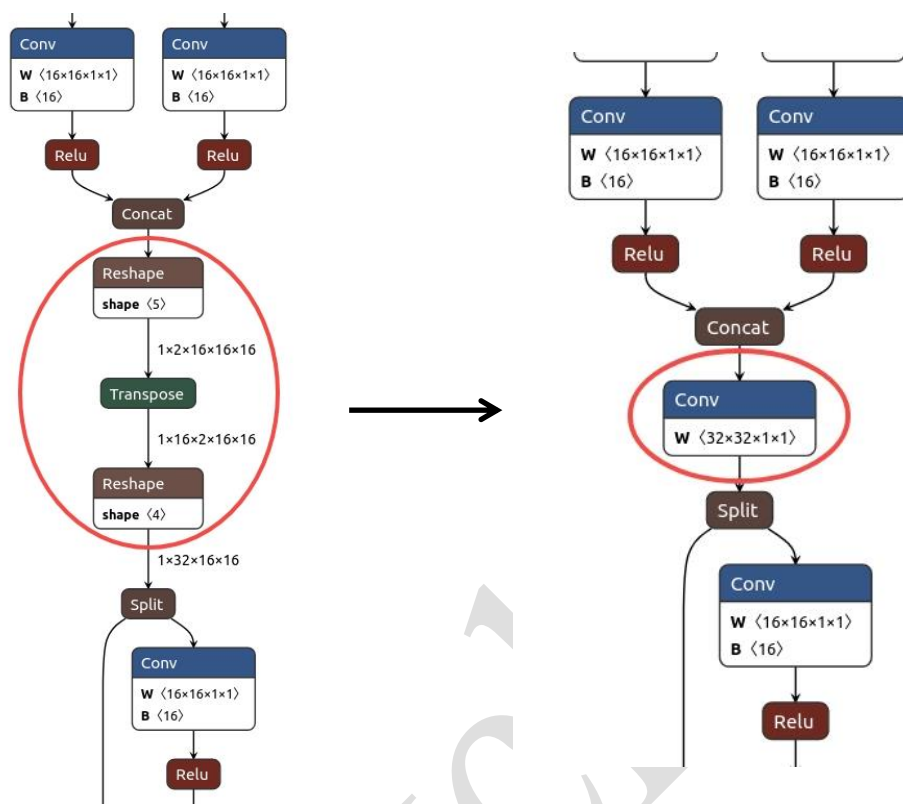


图 8-7 卷积重排

8.4.2 利用硬件 Fuse 特性设计网络或图优化

NPU 支持一些算子组合进行融合，可以适当调整部分算子的运算流程，以适应 NPU 的融合规则实现算子融合优化。

尽管 RKNN 软件栈会有一定程度的图优化，但无法做到全面覆盖到所有情况。某些特殊情况下出现了理论上可融合简化，但最终图优化未能融合的图结构，用户可以算法上手动调整以快速解决该优化问题。

例如下图，在不改变计算正确性的情况下，通过调整 Transpose 与 Clip 算子的顺序，使得 Conv 与 Clip 融合运算，提高了性能。

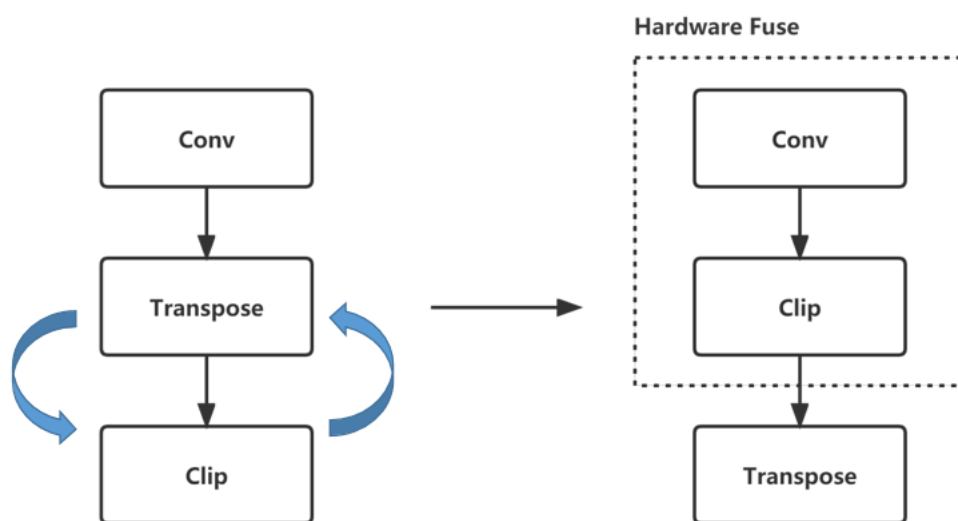


图 8-8 算子图优化/融合

利用融合规则设计，融合规则如下：

已支持的融合规则	未来计划支持的融合规则
Conv+Relu	Activation+Add(Mul)
Conv+PRelu(LeakyRelu)	Add(Mul)+Activation
Conv+Clip	Conv+Mul
Conv+Sigmoid(Tanh/Elu/Silu...)	Conv+Activation+Mul
Conv+Add	Conv+Activation+Pooling
Conv+Activation+Add	Conv+Activation+Add(Mul)+Pooling

8.4.3 算法等效变换或者子图单 OP 化

在分析某个图区域时，有时算法上要实现某个行为功能可能设计上追求表达的直白性不会考究具体部署上的性能影响，容易产生出复杂冗余的图结构，可以通过算法等效的方式，将某个区域的子图单 op 化，减少算子计算步骤，达成优化目的。

例如下图为 YOLOv5-nano 等效图变换，将若干复杂的 Slice 取数融合到 Conv 中，形成一个新的 Conv，极大简化了图结构。

方案来源：<https://github.com/ultralytics/yolov5/issues/4825>

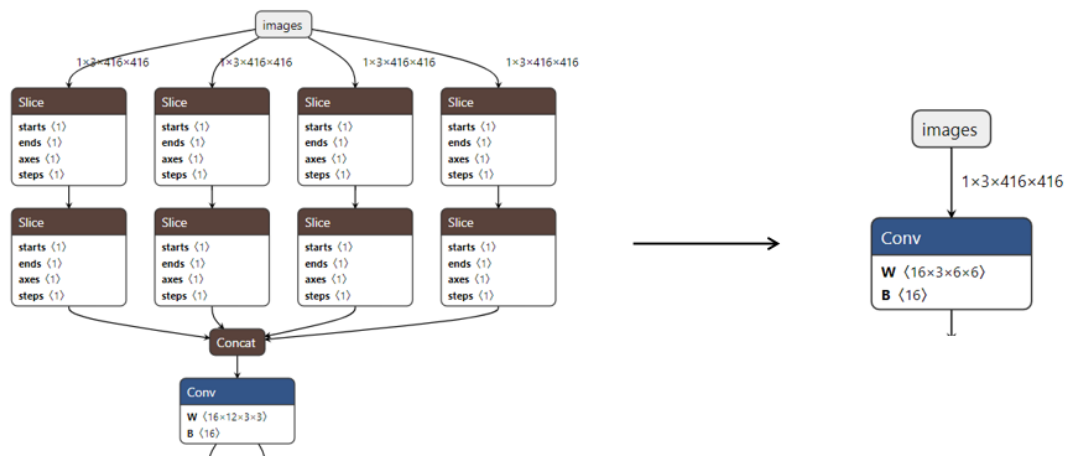


图 8-9 算子等效图变换

8.4.4 算子等效进行“同类项合并”、“提取公因式”

某些算子连续多次运算时，可以简省合并为同一个算子，如 Reshape、Transpose、Slice、部分 Add/Mul/Sub/Div 等。

例如下图可以通过简单调整图顺序以达成同类算子合并目的。

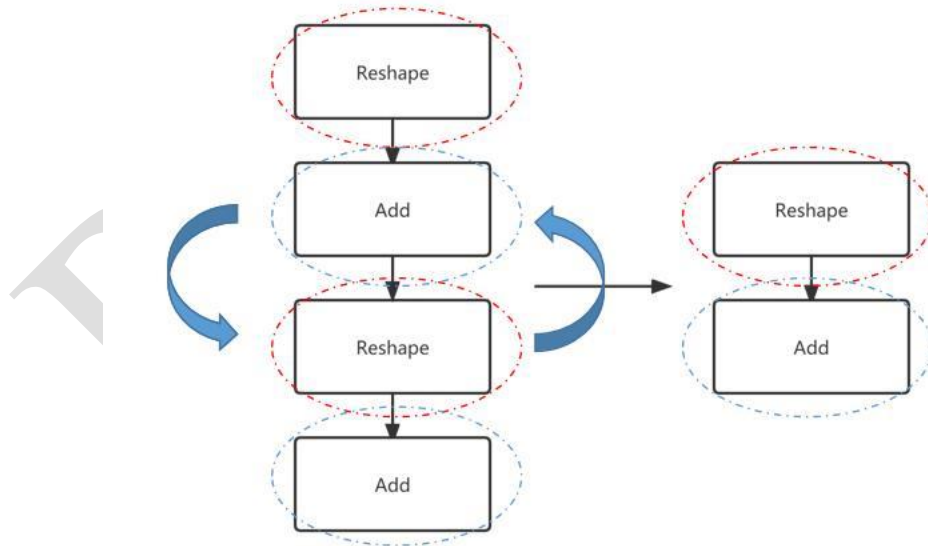


图 8-10 同类项算子合并

某些图结构有一些共有部分的可类型操作可以调整顺序以提取成单一操作。

例如下图可以通过调整算子顺序将重复性的同类算子单独提取出来只执行一次操作。

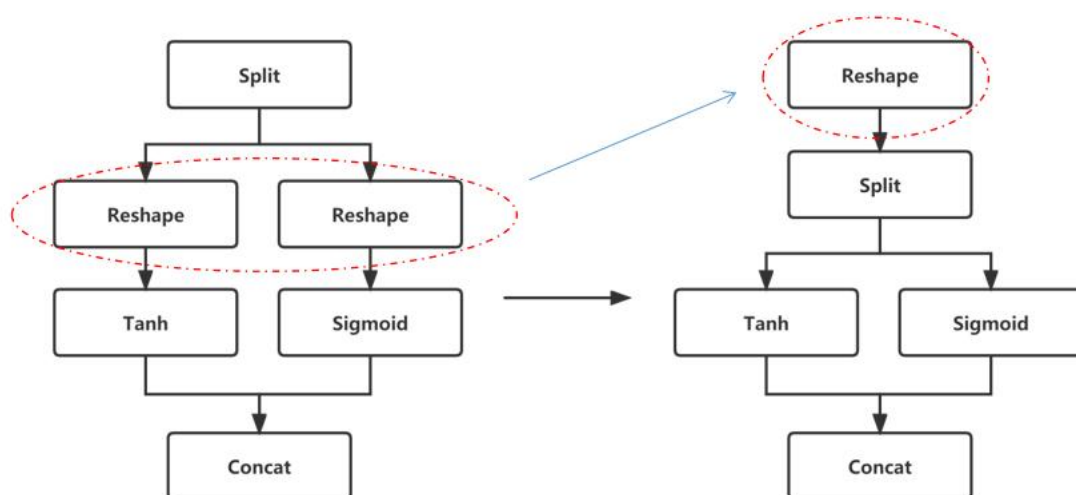


图 8-11 重复性算子合并

8.5 算子级别优化

模型的算子级别优化是针对性比较强的细节优化，对于某些特定算子的具体改造设计，以期进一步提升性能。算子的优化更多是针对性进行算子尺寸设计，以硬件实现效率最高的尺寸规格运行，例如某些算子尺寸规模相似，对齐与非对齐的运行耗时可能差别巨大，差别的原因在于硬件对于部分非对齐尺寸的算子会需要额外的冗余操作来保证正确性，因此算子的尺寸设计对于模型性能也能起到很大的影响，用户可以根据如下一些思路来进行预先性的算子优化。

8.5.1 面向 DDR 性能优化的 OP 尺寸设计（非强制）

在一些对齐尺寸下，除 NPU 运算效率更高外，对于 DDR 的读写也更友好，同等带宽条件下，更友好的读写会提升 DDR 的带宽效率，从而达到更好的性能。以下列出一些对于 DDR 读写更友好的尺寸规则，这些规则不强制。

- Channel 按对齐量对齐

对齐表格如下所示

表 8-1 RK3566/RK3568

	Conv		Depthwise Conv	Others
Dtype	InputChannel	OutputChannel	Channel	Channel
Int8	32	16	32	8
Int16	16	8	16	4
Float16	16	8	16	4
BFloat16	16	8	16	4

表 8-2 RK3588/RK3576

	Conv		Depthwise Conv	Others
Dtype	InputChannel	OutputChannel	Channel	Channel
Int8	32	32	64	16
Int16	32	16	32	8
Float16	32	16	32	8
BFloat16	32	16	32	8
TFloat32	16	16	16	4

表 8-3 RV1106/RV1103

	Conv		Depthwise Conv	Others
Dtype	InputChannel	OutputChannel	Channel	Channel
Int8	32	16	32	16
Int16	16	16	16	8

表 8-4 RK3562

	Conv		Depthwise Conv	Others
Dtype	InputChannel	OutputChannel	Channel	Channel
Int8	32	16	32	16
Int16	32	8	16	8
Float16	32	8	16	8
BFloat16	32	8	16	8
TFloat32	16	8	8	4

- Height * Width > 1 时 4 对齐
- 同等规模的算子，Width 大 Height 小的尺寸，面向 DDR 读写更友好。如下图：右图卷积效率高于左图卷积

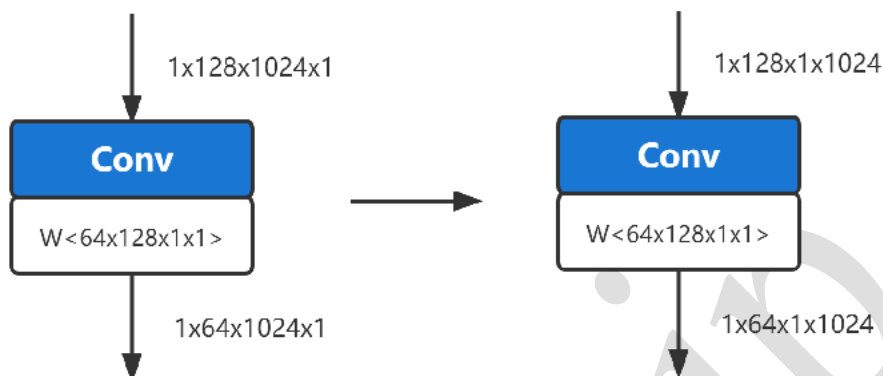


图 8-12 同等规格卷积对比

8.5.2 高利用率模型算子的设计

要在模型设计上整体提高算力的利用率，一般要尽量避免低效算子，以及选择容易跑出更高利用的算子尺寸设计。这里主要列出一些可以尽可能避免的低效算子、讨论卷积尺寸与利用率的关系。

- 规避低效算子原则设计

模型中尽量减少以下三类算子

表 8-5 三类低效算子

数据搬运类	尺寸变换类	非 Relu 类激活函数
Transpose	Resize	Sigmoid
Reshape	Tile	Tanh
Split	Pooling	Softplus
Concat	Pad	Hardswish

- 卷积尺寸与利用率关系的讨论

由于卷积的性能会受到算力和带宽的双重影响，在评估性能时常采用 MAC 利用率来说明硬件算力发挥程度。

卷积尺寸与硬件算力和带宽读写相关，因此这里讨论卷积尺寸与利用率关系，作为用户设计模型的性能参考辅助。以下根据经验数据来作为一个大致的参考：

以下名词注释：KH（kernel height），KW（kernel width），KC（kernel channel），type_bytes（权重位宽除以 8），Ksize（KW 或 KH），Kstride（KW 或 KH 方向上的 stride）

- 卷积的输入输出 Tensor 的 Channel 符合对齐要求时（见 8.4.1 中对齐表格数据），利用率更高。
- 输入 Tensor 的 Channel < 256 时利用率相对较高，当 Channel > 512 以后，随着 Channel 增大，利用率会逐渐下降。
- 权重尺寸上， $KH * KW * KC * \text{type_bytes} < 6K \text{ Bytes}$ 时利用率相对较高。当超过一定大小后利用率会明显下降。
- Ksize / Kstride 的比值越大，利用率相对更高。例如(Ksize=3, Kstride=1 优于 Ksize=2, Kstride=1)
- 输出 Tensor 的 Height * Width < 16 时利用率下降。
- 输出 Tensor 的 Channel 越大，利用率越高。

以上讨论仅是考察独立尺寸影响利用率的因素，实际部署模型里的卷积所呈现出的实测利用率则是诸多因素综合后的结果，开发者如果对某一卷积性能不够满意，希望通过提升利用率以优化其性能，可以参考上述尺寸与利用率关系的讨论进行针对性调整。

8.5.3 子图融合的匹配

RKNN 软件栈会将某些特定的图关系匹配成自定义算子，如下图所示，如果没有被融合成对应的算子，可以考察一下是否连接关系不同没有匹配上。

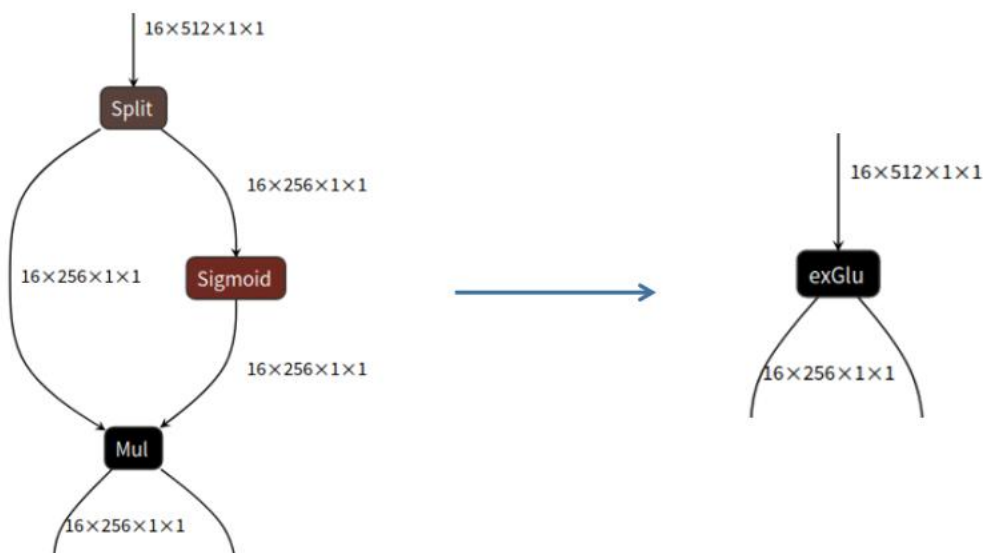


图 8-1 Glu 子图融合

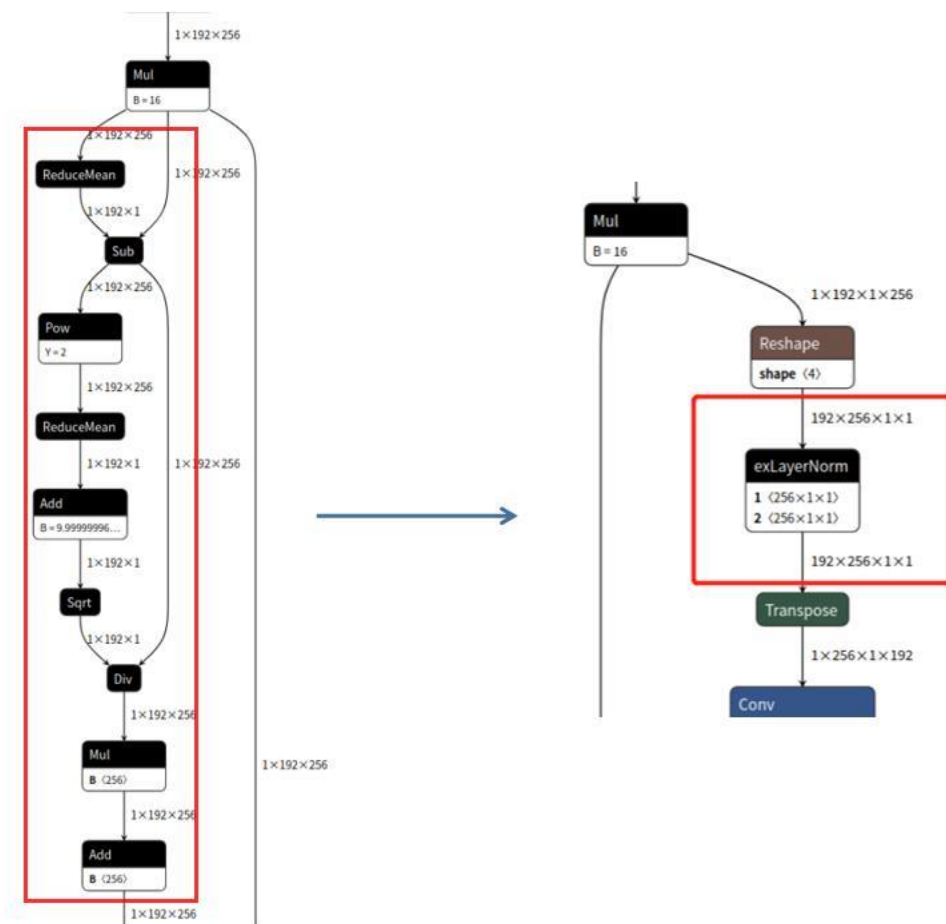


图 8-2 LayerNorm 子图融合

目前已经支持的子图融合规则有:

- Split + Sigmoid + Mul -> GLU
- ReduceMean + Sub + Pow + ReduceMean + Add + Sqrt + Div (+ Mul + Add) -> LayerNorm

9 内存使用优化

9.1 模型运行时内存组成及分析方法介绍

9.1.1 RKNN 模型运行时内存组成

RKNN 模型运行时内存主要由权重和 internal tensor、寄存器配置、输入输出 tensor 四种组成。运行时的内存通常是在 rknn_init 的时候创建完成。

9.1.2 模型内存分析方法

在 rknn_init()接口调用完毕后，当用户需要查看模型分配的内存或者需要外部分配模型权重的时候，调用 rknn_query 接口，传入 RKNN_QUERY_MEM_SIZE 即可查询模型的权重、internal 的内存(不包括输入和输出)、模型推理所用的所有 DMA 内存以及 SRAM 内存（如果 SRAM 没开或者没有此项功能则为 0）的占用情况。

以下是示例代码：

```
rknn_context ctx = 0;

// Load RKNN Model
int ret = rknn_init(&ctx, model_path, 0, NULL, NULL);
if (ret < 0) {
    printf("rknn_init fail! ret=%d\n", ret);
    return -1;
}

// Get weight and internal mem size
rknn_mem_size mem_size;
ret = rknn_query(ctx, RKNN_QUERY_MEM_SIZE, &mem_size,
sizeof(mem_size));
if (ret != RKNN_SUCC) {
    printf("rknn_query fail! ret=%d\n", ret);
    return -1;
}
printf("total weight size: %d, total internal size: %d\n",
mem_size.total_weight_size, mem_size.total_internal_size);
```

9.2 如何使用外部分配内存

9.2.1 输入输出内存外部分配

根据章节 [5.2.2](#) 里提到的，如果用户使用零拷贝 API，可以在外部分配内存给输入输出 tensor，然后配置给 NPU 使用，具体流程可以参照 [5.2.2](#) 里的流程图。注意，要用外部内存

分配的方式，只能使用零拷贝 API。该方法主要适用场景是用户需要手动分配内存给 NPU 使用，而不是通过 `rknn_create_mem()` 接口来让 NPU 自己分配内存。

外部内存可以用物理地址和 `fd` 记录，主要通过下面 2 个接口创建：

- `rknn_create_mem_from_phys()`：通过物理地址来创建 `rknn_tensor_mem` 的结构体
- `rknn_create_mem_from_fd()`：通过 `fd` 来创建 `rknn_tensor_mem` 的结构体

这里提供了一个用 `mpi mmz` 创建内存的例子。该例子通过 `rknn_create_mem_from_phys()` 接口，引入外部内存的物理地址，创建一个 `rknn tensor mem` 的结构体带物理内存的信息。

以下是示例代码：

```
// Create input tensor memory
rknn_tensor_mem* input_mems[1];
// default input type is int8 (normalize and quantize need
compute in outside)
// if set uint8, will fuse normalize and quantize to npu
input_attrs[0].type = input_type;
// default fmt is NHWC, npu only support NHWC in zero copy mode
input_attrs[0].fmt = input_layout;

input_mems[0] = rknn_create_mem_from_phys(ctx, input_phys,
input_virt, input_attrs[0].size_with_stride);
...
// Create output tensor memory
rknn_tensor_mem* output_mems[io_num.n_output];
for (uint32_t i = 0; i < io_num.n_output; ++i) {
    output_mems[i] = rknn_create_mem_from_phys(ctx,
output_physs[i], output_virts[i], output_attrs[i].size);
}

// Set input tensor memory
ret = rknn_set_io_mem(ctx, input_mems[0], &input_attrs[0]);
if (ret < 0) {
    printf("rknn_set_io_mem fail! ret=%d\n", ret);
    return -1;
}

// Set output tensor memory
for (uint32_t i = 0; i < io_num.n_output; ++i) {
    // set output memory and attribute
    ret = rknn_set_io_mem(ctx, output_mems[i], &output_attrs[i]);
    if (ret < 0) {
        printf("rknn_set_io_mem fail! ret=%d\n", ret);
        return -1;
    }
}
```

除了引用外部内存的物理地址外，还可以通过引用 `fd` 的方式来使用外部分配内存，

示例代码如下：

```

int mb_flags = RK_MMZ_ALLOC_TYPE_CMA | RK_MMZ_ALLOC_UNCACHED;

// Allocate weight memory in outside
MB_BLK      weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb, mem_size.total_weight_size,
mb_flags);
if (ret < 0) {
    printf("RK_MPI_MMZ_Alloc failed, ret: %d\n", ret);
    return ret;
}
void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
if (weight_virt == NULL) {
    printf("RK_MPI_MMZ_Handle2VirAddr failed!\n");
    return -1;
}
int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);
if (weight_fd < 0) {
    printf("RK_MPI_MMZ_Handle2Fd failed!\n");
    return -1;
}
weight_mem = rknn_create_mem_from_fd(ctx, weight_fd,
weight_virt, mem_size.total_weight_size, 0);
printf("weight mb info: virt = %p, fd = %d, size: %d\n",
weight_virt, weight_fd, mem_size.total_weight_size);

int mb_flags = RK_MMZ_ALLOC_TYPE_CMA | RK_MMZ_ALLOC_UNCACHED;

// Allocate weight memory in outside
MB_BLK      weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb, mem_size.total_weight_size,
mb_flags);
if (ret < 0) {
    printf("RK_MPI_MMZ_Alloc failed, ret: %d\n", ret);
    return ret;
}
void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
if (weight_virt == NULL) {
    printf("RK_MPI_MMZ_Handle2VirAddr failed!\n");
    return -1;
}
int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);
if (weight_fd < 0) {
    printf("RK_MPI_MMZ_Handle2Fd failed!\n");
    return -1;
}
weight_mem = rknn_create_mem_from_fd(ctx, weight_fd,
weight_virt, mem_size.total_weight_size, 0);
printf("weight mb info: virt = %p, fd = %d, size: %d\n",
weight_virt, weight_fd, mem_size.total_weight_size);

```

9.2.2 模型内存的外部分配

在 9.1 的章节提到模型内存占用有分两部分，一部分是 `internal` 内存，另外一部分是 `weight` 内存。应用如果需要使用外部分配的模型内存，可以通过接口 `rknn_set_weight_mem()`, `rknn_set_internal_mem()` 接口设置模型 `weight` 和 `internal` 使用的内存。参考示例如下：

```
// Load RKNN Model
ret = rknn_init(&ctx, model_virt, model_size,
RKNN_FLAG_MEM_ALLOC_OUTSIDE, NULL);
TIME_END(rknn_init);
if (ret < 0) {
    printf("rknn_init fail! ret=%d\n", ret);
    return -1;
}
//query and inset input / output tensor
...
// Allocate weight memory in outside
MB_BLK          weight_mb;
rknn_tensor_mem* weight_mem;
ret = RK_MPI_MMZ_Alloc(&weight_mb,
SIZE_ALIGN_128(mem_size.total_weight_size), mb_flags);
void* weight_virt = RK_MPI_MMZ_Handle2VirAddr(weight_mb);
int weight_fd = RK_MPI_MMZ_Handle2Fd(weight_mb);

weight_mem = rknn_create_mem_from_fd(ctx, weight_fd,
weight_virt, mem_size.total_weight_size, 0);

ret = rknn_set_weight_mem(ctx, weight_mem);
if (ret < 0) {
    printf("rknn_set_weight_mem fail! ret=%d\n", ret);
    return -1;
}
printf("weight mb info: virt = %p, fd = %d, size: %d\n",
weight_virt, weight_fd, mem_size.total_weight_size);

// Allocate internal memory in outside
MB_BLK          internal_mb;
rknn_tensor_mem* internal_mem;
ret = RK_MPI_MMZ_Alloc(&internal_mb,
SIZE_ALIGN_128(mem_size.total_internal_size), mb_flags);
void* internal_virt = RK_MPI_MMZ_Handle2VirAddr(internal_mb);
int internal_fd = RK_MPI_MMZ_Handle2Fd(internal_mb);

internal_mem = rknn_create_mem_from_fd(ctx, internal_fd,
internal_virt, mem_size.total_internal_size, 0);
ret = rknn_set_internal_mem(ctx, internal_mem);
if (ret < 0) {
    printf("rknn_set_internal_mem fail! ret=%d\n", ret);
    return -1;
}
printf("internal mb info: virt = %p, fd = %d, size: %d\n",
internal_virt, internal_fd, mem_size.total_internal_size);
```

9.3 Internal 内存复用

RKNN API 提供了外部管理 NPU 内存的机制，通过 RKNN_FLAG_MEM_ALLOC_OUTSIDE 参数，用户可以指定模型中间的 feature 内存由外部分配。该功能的典型应用场景如下：

- 部署时，所有 NPU 内存均是用户自行分配，便于对整个系统内存进行统筹安排。
- 用于多个模型串行运行场景，中间 feature 内存存在不同上下文复用，特别是针对 RV1103/RV1106 这种内存极为紧张的情况。

例如，下图中有两个模型，模型 1 的 Internal Tensor 占用大于模型 2，如果模型 1 和模型 2 顺序地运行，可以只开辟 0x00000000~0x000c4000 地址的一块内存给模型 1 和 2 共用，模型 1 推理结束后，这块内存可以被模型 2 用来读写 Internal Tensor 数据，从而节省内存。

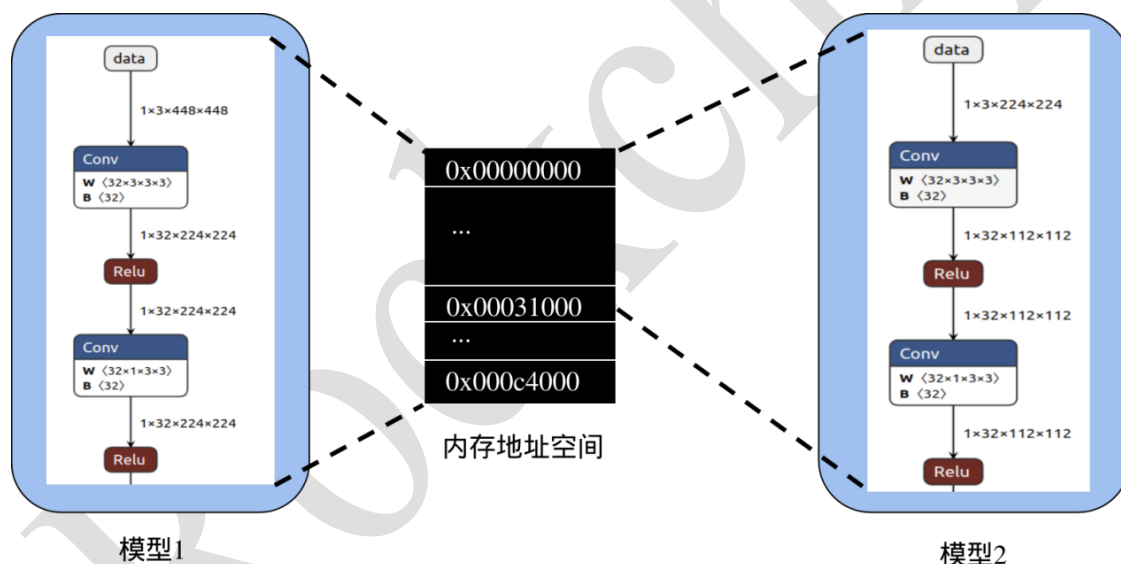


图 9-1 两个模型 Internal Tensor 共享同一块内存地址空间的示例图

假设模型 1 的路径是 model_path_a，模型 2 路径是 model_path_b，示例代码如下：

```
rknn_init(&ctx_a, model_path_a, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE,
NULL);
rknn_query(ctx_a, RKNN_QUERY_MEM_SIZE, &mem_size_a,
sizeof(mem_size_a));

rknn_init(&ctx_b, model_path_b, 0, RKNN_FLAG_MEM_ALLOC_OUTSIDE,
NULL);
rknn_query(ctx_b, RKNN_QUERY_MEM_SIZE, &mem_size_b,
sizeof(mem_size_b));

// 获取两个模型最大的 internal size
max_internal_size = MAX(mem_size_a.total_internal_size,
mem_size_b.total_internal_size);
```



```
internal_mem_max = rknn_create_mem(ctx_a, max_internal_size);
// 设置 a 模型 internal memory
internal_mem_a = rknn_create_mem_from_fd(ctx_a,
internal_mem_max->fd,
internal_mem_max->virt_addr,
mem_size_a.total_internal_size, 0);
rknn_set_internal_mem(ctx_a, internal_mem_a);
// 设置 b 模型 internal memory
internal_mem_b = rknn_create_mem_from_fd(ctx_b,
internal_mem_max->fd,
internal_mem_max->virt_addr,
mem_size_b.total_internal_size, 0);
rknn_set_internal_mem(ctx_b, internal_mem_b);
```

9.4 多线程复用上下文

在多线程场景中，一个模型可能会被多个线程同时执行，如果每个线程都单独初始化一个上下文，那么内存消耗会很大，因此可以考虑共享一个上下文，避免数据结构重复构造，减少运行时内存占用。RKNN API 提供了复用上下文的接口，接口定义如下：

```
int rknn_dup_context(rknn_context* context_in, rknn_context*
context_out)
```

其中，`context_in` 是已初始化的上下文，而 `context_out` 是复用 `context_in` 的上下文。如下图所示，两个 `context` 的模型结构相同，因此可以复用上下文。

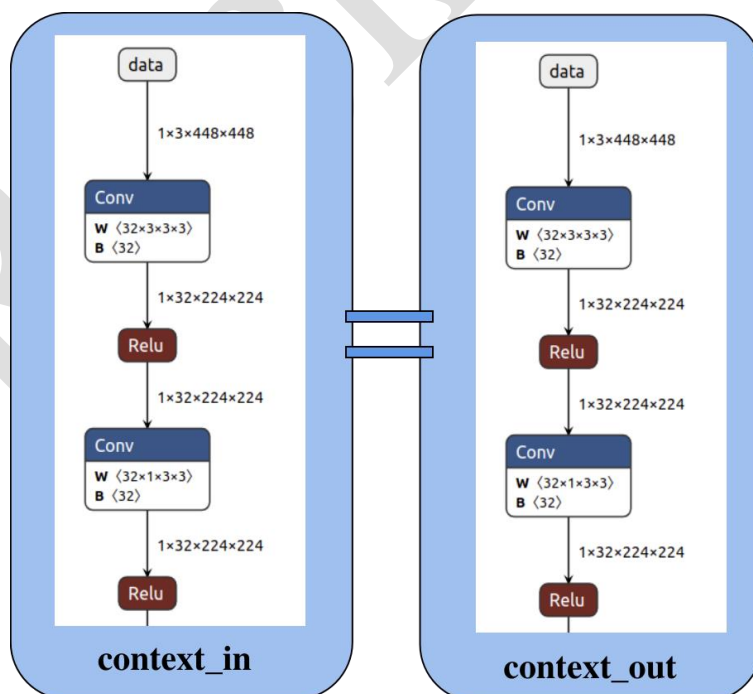


图 9-2 两个相同的模型复用上下文的示例图

9.5 多种分辨率模型共享相同权重

当多个不同分辨率模型有相同的权重时，可以共享相同的权重，以减少内存占用。在 RKNPU SDK \leq 1.5.0 版本时，此功能可以以较小的内存占用实现不同分辨率模型间的动态切换。在 1.5.0 版本后，该功能被动态 shape 功能替代。

如下图所示，模型 A 和模型 B 的权重完全相同。

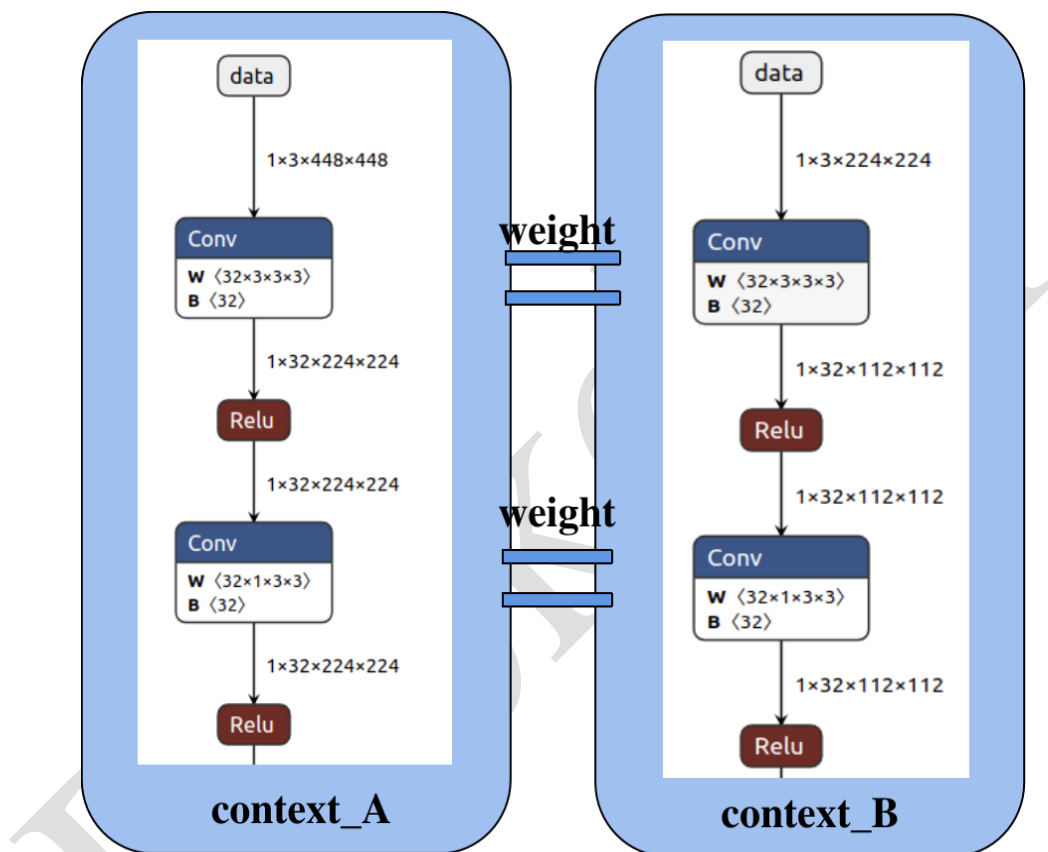


图 9-3 两个不同分辨率模型共享权重的示例图

可按照以下步骤实现多分辨率模型共享相同权重：

- 1) 在转换 RKNN 模型时，其中一个模型设置为主模型，rknn.config 接口设置参数 remove_weight=False，另一个模型设置为从模型，设置参数 remove_weight=True。主 RKNN 模型包含权重，从 RKNN 模型不包含卷积类权重。
- 2) 部署时，先初始化主 RKNN 模型，再初始化从 RKNN 模型。初始化从模型时，使用 RKNN_FLAG_SHARE_WEIGHT_MEM 标志，并新增 rknn_init_extend 参数，该参数值为主模型的上下文。假设主模型路径是 model_A，从模型路径是 model_B，示例代码如下：

```
rknn_context context_A;  
rknn_context context_B;  
ret = rknn_init(&context_A,model_A,0,0,NULL);  
...  
rknn_init_extend extend;  
extend.ctx = context_A;  
ret = rknn_init(&context_B, model_B,0,  
RKNN_FLAG_SHARE_WEIGHT_MEM,&extend);
```

10 常见问题

10.1 NPU 环境准备问题

- 版本兼容性

- NPU 内核驱动和 Runtime 版本兼容

建议将 NPU 内核驱动升级到 0.9.2 或之后的版本。在遇到问题时，先更新到最新版本的 NPU 内核驱动。

- RKNN-Toolkit2 导出的模型和 Runtime 版本之间的兼容关系如下表所示：

表 10-1 RKNN 模型和 Runtime 版本对应关系

RKNN 模型版本	Runtime 版本
1.2.0	$\geq 1.2.0$ and $\leq 1.5.0$
1.3.0	$\geq 1.3.0$ and $\leq 1.5.0$
1.4.0	$\geq 1.4.0$ and $\leq 1.5.0$
1.5.0	1.5.0
1.5.2	$\geq 1.5.2$
1.6.0	$\geq 1.5.2$

- 如何更新 NPU 内核驱动

建议升级完整固件以更新 NPU 驱动，对应固件可以找厂商提供。

- 板端 docker 环境中如何使用 NPU

在板端使用 docker 部署应用时，如果要使用 NPU 资源，需要在启动容器时，映射 NPU 相关资源，参考命令如下：

```
docker run -t -i --privileged -v
/dev/dri/renderD129:/dev/dri/renderD129 -v /proc/device-
tree/compatible:/proc/device-tree/compatible -v
/usr/lib/librknnrt.so:/usr/lib/librknnrt.so
ai_application:v1.0.0 /bin/bash
```

该命令中重点关注以下参数：

- **/dev/dri/renderD129:** RK3588 NPU 设备节点，Runtime 依赖该节点以使能 NPU。
- **/proc/device-tree/compatible:** 该文件记录 SOC 型号，RKNN-Toolkit Lite2 等组件依赖该文件获取当前 SOC 信息。
- **/usr/lib/librknnrt.so:** Runtime 库存放位置，RKNN-Toolkit Lite2 和 RKNPU2 C API

依赖该文件以使用 NPU 资源。

- **ai_application:v1.0.0:** 待启动容器所使用的镜像名和版本。

10.2 工具安装问题

- **RKNN-Toolkit2 依赖的环境限制太严格，导致无法成功安装**

在所有依赖库都已安装、但部分库的版本和要求不匹配时，可以尝试在安装指令后面加上"no-deps"参数，取消安装 Python 库时的环境检查。如：

```
pip install rknn-toolkit2*.whl --no-deps
```

- **PyTorch 依赖说明**

RKNN-Toolkit2 的 PyTorch 模型加载功能，依赖于 PyTorch。PyTorch 的模型分为浮点模型和已量化模型（包含 QAT 及 PTQ 量化模型）。

对于 PyTorch 1.6.0 导出的模型，建议将 RKNN-Toolkit2 依赖的 PyTorch 版本降级至 1.6.0 以免出现加载失败的问题。

对于已量化模型（QAT、PTQ），我们推荐使用 PyTorch 1.10~1.13.1 导出模型，并将 RKNN-Toolkit2 依赖的 PyTorch 版本升级至 1.10~1.13.1。

另外在加载 PyTorch 模型时，建议导出原模型的 PyTorch 版本，要与 RKNN-Toolkit2 依赖的 PyTorch 版本尽量一致。

推荐使用的 PyTorch 版本为 1.6.0、1.9.0、1.10 或 1.13.1 版本。

- **TensorFlow 依赖说明**

RKNN-Toolkit2 的 TensorFlow 模型加载功能依赖于 TensorFlow。由于 TensorFlow 各版本之间的兼容性一般，其他版本可能会造成 RKNN-Toolkit2 模型加载异常，所以在加载 TensorFlow 模型时，建议导出原模型的 TensorFlow 版本，要与 RKNN-Toolkit2 依赖的 TensorFlow 版本一致。

对于 TensorFlow 版本引发的问题，通常会体现在"rknn.load_tensorflow()"阶段，且出错信息会指向依赖的 TensorFlow 路径。

推荐使用的 TensorFlow 版本为 2.6.2 或 2.8.0。

- **RKNN-Toolkit2 安装包命名规则**

以 1.5.2 版本的发布件为例，RKNN-Toolkit2 wheel 包命名规则如下：

```
rknn_toolkit2-1.5.2+b642f30c-cp38-cp38-linux_x86_64.whl
```

- rknn_toolkit2: 工具名称。

- 1.5.2: 版本号。
- b642f30c: 提交号。
- cp<xx>-cp<xx>: 适用的 Python 版本, 例如 cp38-cp38 表示适用的 Python 版本是 3.8。
- linux_x86_64: 系统类型和 CPU 架构。

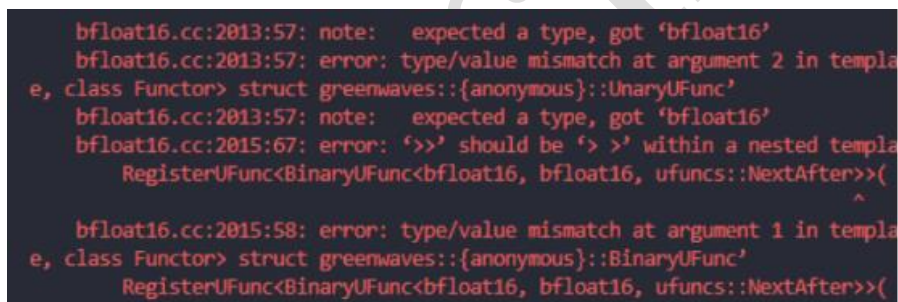
请按照自己所用的操作系统、CPU 架构和 Python 版本安装对应的工具包, 否则安装会失败。

● RKNN-Toolkit2 是否有 ARM Linux 版本

RKNN-Toolkit2 没有 ARM Linux 版, 如果需要在 ARM Linux 上使用 Python 接口进行推理, 可以安装 RKNN-Toolkit-lite2, 该工具可以在 ARM Linux 上用 Python 运行推理。

● bfloat16 依赖库安装不上

bfloat16 的依赖库安装出错, 如下:



```
bfloat16.cc:2013:57: note: expected a type, got 'bfloat16'
bfloat16.cc:2013:57: error: type/value mismatch at argument 2 in template
e, class Functor> struct greenwaves::{anonymous}::UnaryUFunc'
bfloat16.cc:2013:57: note: expected a type, got 'bfloat16'
bfloat16.cc:2015:67: error: '>' should be '>' within a nested template
RegisterUFunc<BinaryUFunc<bfloat16, bfloat16, ufuncs::NextAfter>>>(^
bfloat16.cc:2015:58: error: type/value mismatch at argument 1 in template
e, class Functor> struct greenwaves::{anonymous}::BinaryUFunc'
RegisterUFunc<BinaryUFunc<bfloat16, bfloat16, ufuncs::NextAfter>>>(^
```

图 10-1 bfloat16 依赖库安装失败日志

更换 pip 源为阿里源, 或者更新 RKNN-Toolkit2 至 1.5.0 以及之后的版本 (1.5.0 以及之后的版本已经去除 bfloat16 库的依赖)。

10.3 模型转换常用参数说明

本章节主要覆盖模型转换阶段常用参数的使用说明。

● 根据模型确定参数

模型转换时, "rknn.config()" 和 "rknn.build()" 接口会影响模型转换结果。
"rknn.load_onnx()", "rknn.load_tensorflow()" 指定输入输出节点, 会影响模型转换结果。
"rknn.load_pytorch()", "rknn.load_tensorflow()" 指定输入的尺寸大小会影响模型转换结果。

可以参考以下基本步骤进行模型转换：

1. 准备量化数据，提供 dataset.txt 文件。
2. 确定模型要使用的 NPU 平台，如 RK3566、RV1106 等，并填写 rknn.config()接口中的 target_platform 参数。
3. 当输入是 3 通道的图像，且量化数据采用的是图片格式（如 jpg、png 格式）时，需要确认模型的输入是 RGB 还是 BGR，以决定 rknn.config() 接口中 quant_img_RGB2BGR 参数的值。
4. 确认模型训练时候的归一化参数，以决定 rknn.config 接口中的 mean_values 和 std_values 参数的值。
5. 确认模型输入的尺寸信息，填入 load 接口相应参数中，如 rknn.load_pytorch()接口中的 input_size_list 参数。
6. 确认模型要量化比特数，以决定 rknn.config()接口中的 quantized_dtype 参数的值。
不对模型进行量化或加载的是已量化模型时可以忽略此步骤。
7. 确认模型量化时使用的量化算法，以决定 rknn.config()接口中 quantized_algorithm 参数的值。
不对模型进行量化或加载已量化模型时可以忽略此步骤。
8. 确认是否对模型进行量化，以决定 rknn.build()接口中 do_quantization 参数的值。
选择对模型进行量化时，需要额外填写 rknn.build()接口中的 dataset 参数，指定量化正数据。

● RKNN 模型的跨平台兼容性

对于 rknn.config()的 target_platform 设置的平台参数，兼容性关系如下：

- RK3566、RK3568 平台使用的模型是相互兼容的。
- RK3588、RK3588S 平台使用的模型是相互兼容的。
- RV1103、RV1106 平台使用的模型是相互兼容的。

● 量化校正数据的格式及要求

量化校正数据的格式有两种选择，一种是图片格式（jpg，png），RKNN-Toolkit2 会调用 OpenCV 接口进行读取；另一种是 npy 格式，RKNN-Toolkit2 会调用 numpy 接口进行读取。

对于非 RGB/BGR 图片输入的模型，建议使用 numpy 的 npy 格式提供量化数据。

● 多输入模型 dataset.txt 文件的填写方式

模型量化需要用 dataset.txt 文件指定量化数据的路径。规则为一行作为一组输入，模型

存在多输入时，多个输入写在同一行，并用空格隔开。

如单输入模型，使用两组量化数据：

```
sampleA.npy  
sampleB.npy
```

如三个输入的模型，两组量化数据按如下方式填写：

```
sampleA_in0.npy sampleA_in1.npy sampleA_in2.npy  
sampleB_in0.npy sampleB_in1.npy sampleB_in2.npy
```

- **确认 rknn.config() 的 quant_img_RGB2BGR 参数**

采用图片（jpg，png）作为量化数据时，需要考虑设置 quant_img_RGB2BGR 参数。

模型采用 RGB 图片进行训练时，则 quant_img_RGB2BGR 参数设为 False 或不设置。

且在使用 Python inference 接口或 RKNPU2 C API 进行推理时，输入 RGB 图片。

模型采用 BGR 图片进行训练时，则 quant_img_RGB2BGR 参数设为 True。但在使用 Python inference 接口或 RKNPU2 C API 进行推理时，同样需要输入 BGR 图片（quant_img_RGB2BGR 只会影响从量化校正集读入的图像）。

若量化数据采用 numpy 的 npy 格式，则建议不要使用 quant_img_RGB2BGR 参数，避免产生使用混乱的问题。

- **rknn.config() 的 mean、std 和 quant_img_RGB2BGR 的计算顺序问题**

因为 quant_img_RGB2BGR 只控制在量化过程中读取校正集图像时是否要进行转换通道，并不会影响其他的步骤。因此对于 RKNN-Toolkit2 的 inference 接口及 RKNPU2 C API，对输入数据都只先进行减均值（mean）、再除标准差（std）的操作，并没有通道转换的操作。

- **模型是非 3 通道输入或多输入时，rknn.config() 的 mean_values 和 std_values 的设置问题**

mean_values 和 std_values 的设置格式是一致的。以 mean_values 为例子。

假设输入有 N 个通道，则 mean_values 的值为[[channel_1, channel_2, channel_3, channel_4, ..., channel_n]]。

存在多输入时，则 mean_values 的值为[[channel_1, channel_2, channel_3, channel_4, ..., channel_n], [channel_1, channel_2, channel_3, channel_4, ..., channel_n]]。

- **量化参数矫正算法和量化图片数量的选取**

RKNN-Toolkit2 中量化算法 (rknn.config() 的 quantized_algorithm) 参数提供三种算法进行参数矫正, 分别为 normal、mmse 和 kl_divergence, 默认使用 normal。normal 为常规的量化参数矫正算法; 而 mmse 会迭代中间层的计算结果, 对权重数值进行一定范围的裁剪, 以获得更高的推理精度。使用 mmse 不一定能提升量化精度, 但相比 normal 方式, 量化时会占用更多的内存、耗费更长的模型转换时间; 使用 kl_divergence 量化算法所用时间会比 normal 多一些, 但比 mmse 会少很多, 在某些场景下 (feature 分布不均匀时) 可以得到较好的改善效果。

建议先使用 normal 算法, 如果量化效果不佳, 可尝试使用 mmse 或 kl_divergence 算法。

使用 normal 或 kl_divergence 算法时, 推荐给出 20-200 组数据进行量化。使用 mmse 量化时, 推荐使用 20-50 组数据进行量化。

- **量化模型与非量化模型, 推理时输入输出的差异**

调用通用 RKNPU2 C API 时 (指不使用 pass_through、zero_copy 的方式调用 C API), 输入数据的数据类型 (如 uint8 数据, float 数据) 与模型的量化与否没有关系。输出数据的数据类型可以选择自动处理成 float32 格式, 也可以选择直接输出模型推理结果, 此时数据类型与输出节点的数据类型一致。使用 Python 推理接口会有点差异, 具体关系如下表:

表 10-2 Python 推理接口和通用 C API 接口区别

模型量化后	Python 推理 (rknn.inference())	C API 推理 (rknn.run()) (非 pass_through、zero_copy)
输入类型是否 有限制	无限制。 rknn.inference()的输入为 numpy 数组，本身带有 data type 属性，该输入会自动转 成 RKNN 模型需要的数据格 式。	无限制。 rknn_inputs 的 rknn_tensor_type 参数可 以根据实际输入，指定 RKNN_TENSOR_FLOAT32、 RKNN_TENSOR_FLOAT16、 RKNN_TENSOR_INT8、 RKNN_TENSOR_UINT8、 RKNN_TENSOR_INT16。指定后，会 将输入自动转成 RKNN 模型需要的数 据格式。
输出类型是否 变化	无变化。 无论模型量化与否，Python 的 rknn.inference()接口总是 返回 float 类型输出。无法选 择其他数据类型。	有变化。 RKNNPU2 C API 的 rknn outputs attr，可 以设置 want_float=1，得到 float 类型 的输出。而量化后，可以设置 want_float=0，此时可以输出最后一个 节点的原始输出数据，如 i8 量化时， 输出 int8 数据。
输入 format 是 否有变化 (NCHW， NHWC)	无变化。 无论模型量化与否， rknn.inference()接口的 data_format 参数，可以根据 需要设置为 nchw 或 nhwc。	无变化。 无论模型量化与否，rknn inputs 结构体 的 rknn_tensor_format 参数，可以根据 需要设置为 NCHW 或 NHWC。

● 是否存在在线预编译的模式

RKNN-Toolkit2 只支持导出离线预编译的模型，不支持导出在线预编译的模型（RKNN-Toolkit1 支持），因此并不存在离线预编译和在线预编译的模式选择。

● RKNN-Toolkit 转出来的 RKNN 模型可以在 RK3566 平台上使用吗

不可以。

RKNN-Toolkit 转出来的 RKNN 模型适用于 RK1806 / RK1808 / RK3399Pro / RV1109 / RV1126 等平台；RK3566 平台需要用 RKNN-Toolkit2 转出来的 RKNN 模型。RKNN-Toolkit2 转出来的 RKNN 模型适用于 RK3566 / RK3568 / RK3588 / RK3588S / RV1103 / RV1106 / RK3562 / RK3576 等平台。

RKNN-Toolkit 工具的使用说明请参考以下工程：

<https://github.com/airockchip/rknn-toolkit>

RKNN-Toolkit2 工具的使用说明请参考以下工程：

<https://github.com/airockchip/rknn-toolkit2>

10.4 模型加载问题

10.4.1 RKNN-Toolkit2 支持的深度学习框架和对应版本

请参考 [3.1](#) 章节。

10.4.2 各框架的 OP 支持列表

RKNN-Toolkit2 对不同框架的支持程度有差异，详细信息可以参考以下目录中的 RKNNToolkit2_OP_Support 文档：

<https://github.com/airockchip/rknn-toolkit2/blob/master/doc/>

10.4.3 ONNX 模型转换常见问题

- 加载模型时出现“Error parsing message”报错

转换 examples/onnx/resnet50v2 模型时，提示加载失败：

```
E load_onnx: Catch exception when loading onnx model:
/rknn_resnet_demo/resnet50v2.onnx!
E load_onnx: Traceback (most recent call last):
E load_onnx: File "rknn/api/rknn_base.py", line 1094, in
rknn.api.rknn_base.RKNNBase.load_onnx
E load_onnx: File "/usr/local/lib/python3.6/dist-
packages/onnx/__init__.py", line 115, in load_model
.....
E load_onnx: google.protobuf.message.DecoderError: Error
parsing message
```

原因可能是 resnet50v2.onnx 模型损坏导致（如没下载全），需要重新下载该模型，并确保其 MD5 值正确，如：

```
22ed6e6a8fb9192f0980acca0c941414 resnet50v2.onnx
```

- 是否支持动态的输入 shape

1.5.2 之前的 RKNN-Toolkit2 不支持动态的输入 shape，比如 onnx 输入维度为[-1, 3, -1, -1]，表示 batch、height 和 width 维度是不固定的。

1.5.2 以及之后的版本可以通过 rknn.config()的 dynamic_input 参数进行动态输入 shape 的仿真，详见 5.4 章节。

- 自定义输出节点时报错

rknn.load_onnx()时传入 outputs 参数进行模型的裁剪，但报如下错误：

```
E load_onnx: the '378' in outputs=['378', '439', '500'] is
invalid!
```

日志提示输出节点 378 是无效的，因此 outputs 参数需设置正确的输出节点名称。

10.4.4 Pytorch 模型转换常见问题

- 加载 Pytorch 模型时出现 torch._C 没有 _jit_pass_inline 属性的错误

错误日志如下：

```
'torch._C' has no attribute '_jit_pass_inline'
```

请将 PyTorch 升级到 1.6.0 或之后的版本。

- Pytorch 模型的保存格式

目前只支持'torch.jit.trace()'导出的模型。'torch.save()'接口仅保存权重参数字典，缺乏网络结构信息，无法被正常导入并转成 RKNN 模型。

- 转换时遇到 PytorchStreamReader 失败的错误

详细错误如下：

```
E Catch exception when loading pytorch
model: ./mobilenet0.25_Final.pth!
E Traceback (most recent call last):
.....
E   cpp_module = torch._C.import_ir_module(cu, f,
map_location, extra_files)
E RuntimeError: [enforce fail at inline
container.cc:137]. PytorchStreamReader failed reading zip
archive: failed finding central directory frame #0 .....
```

出错原因是输入的 PyTorch 模型没有网络结构信息。

通常.pth 只有权重，并没有网络结构信息。对于已保存的模型权重文件，可以通过初始化对应的网络结构，再使用 net.load_state_dict()加载.pth 权重文件。最后通过 torch.jit.trace()接口将网络结构和权重参数固化成一个.pt 文件。得到 torch.jit.trace()处理过以后的.pt 文件，就可以用 rknn.load_pytorch()接口将其转为 RKNN 模型。

- 转换时遇到 KeyError 的错误

错误日志如下:

```
E Traceback (most recent call last):
.....
E KeyError: 'aten::softmax'
```

出现形如 `KeyError: 'aten::xxx'` 的错误信息时, 表示该算子当前版本还不支持。RKNN-Toolkit2 在每次版本升级时都会修复此类 bug, 请使用最新版本的 RKNN-Toolkit2 试试。

● 转换时遇到 "Syntax error in input! LexToken(xxx)" 的错误

错误日志如下:

```
WARNING: Token 'COMMENT' defined, but not used
WARNING: There is 1 unused token
!!!! Illegal character '"'
Syntax error in input! LexToken(NAMED_IDENTIFIER, 'fc',
1, 27)
!!!! Illegal character '"'
```

该错误的原因有很多种, 请按照以下顺序排查:

- 1) 未继承 `torch.nn.module` 创建网络。请继承 `torch.nn.module` 基类来创建网络, 然后再用 `torch.jit.trace()` 生成 pt 文件。
- 2) 更新 RKNN-Toolkit2 1.4.0 或之后的版本, `torch` 建议使用 1.6.0, 1.9.0, 1.10.0 或 1.13.1 版本。

10.4.5 TensorFlow 模型转换常见问题

● Tensorflow1.x 模型报错

使用 `rknn.load_tensorflow()` 接口加载 tensorflow1.x 模型如出现报错提示:

```
E load_tensorflow: Catch exception when loading
tensorflow model: ./yolov3_mobilenetv2.pb!
E load_tensorflow: Traceback (most recent call last):
.....
E load_tensorflow:
tensorflow.python.framework.errors_impl.InvalidArgumentError:
Node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1'
expects to be colocated with unknown node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
E load_tensorflow: During handling of the above
exception, another exception occurred:
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "rknn/api/rknn_base.py", line
990, in rknn.api.rknn_base.RKNNBase.load_tensorflow
.....
E load_tensorflow: return func(*args, **kwargs)
E load_tensorflow: File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/importer.py", line 431,
in import_graph_def
E load_tensorflow: raise ValueError(str(e))
E load_tensorflow: ValueError: Node
```

```
'MobilenetV2/expanded_conv/depthwise/BatchNorm/cond/Switch_1'
expects to be colocated with unknown node
'MobilenetV2/expanded_conv/depthwise/BatchNorm/moving_mean'
```

建议：

- 如当前安装的是 1.x 的 TensorFlow，请安装 2.x 的 TensorFlow。
- 更新 RKNN-Toolkit2 / RKNPU2 至最新版本。

● TransformGraph 类似的报错

TensorFlow 的模型转成 RKNN 时报错：

```
Traceback (most recent call last):
  File "test.py", line 80, in <module>
    input_size_list=[[1, 368, 368, 3]])
  File "/usr/local/lib/python3.6/site-
packages/rknn/api/rknn.py", line 68, in load_tensorflow
    input_size_list=input_size_list, outputs=outputs)
  File "rknn/api/rknn_base.py", line 940, in
rknn.api.rknn_base.RKNNBase.load_tensorflow
  File "/usr/local/lib/python3.6/dist-
packages/tensorflow/tools/graph_transforms/__init__", line 51,
in TransformGraph.transforms_string, status)
  File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/errors_impl.py". ;ome
548, in __exit__
    C_api.TF_GetCode(self.status.status)
  Tensorflow.python.framework.error_impl.InvalidArgumentError
: Beta input to batch norm has bad shape: [24]
```

原因：

- 1) 该模型直接调用 TensorFlow 原生的 TransformGraph 类进行优化时，也会报上面的错误（RKNN-Toolkit2 里同样会调用 TransformGraph 进行优化，因此也会报同样的错误）。
- 2) 可能是模型生成时的 TensorFlow 版本与目前安装的版本已经不兼容了。

建议：

使用 1.14.0 的 TensorFlow 版本重新生成该模型，或者寻找其他框架的同类型模型。

● "Shape must be rank 4 but is rank 0"报错

加载 pb 模型时：

```
rknn.load_tensorflow(tf_pb='./model.pb',
    inputs=["X", "Y"],
    outputs=['generator/xs'],
    input_size_list=1, INPUT_SIZE, INPUT_SIZE, 3)
```

会产生报错：

```
E load_tensorflow: Catch exception when loading
tensorflow model: ./model.pb!
E load_tensorflow: Traceback (most recent call last):
E load_tensorflow: File "/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/importer.py", line 427,
in import_graph_def
E load_tensorflow: graph._c_graph, serialized, options)
# pylint: disable=protected-access
E load_tensorflow:
tensorflow.python.framework.errors_impl.InvalidArgumentError:
Shape must be rank 4 but is rank 0 for
'generator/conv2d_3/Conv2D' (op: 'Conv2D') with input shapes:
[], [7,7,3,32].
```

原因可能是该模型是多输入模型，rknn.load_tensorflow()的 input_size_list 没按规范填写，可以参考 examples/functions/multi_input_test 里的以下用法：

```
rknn.load_tensorflow(tf_pb='./conv_128.pb',
    inputs=['input1', 'input2', 'input3', 'input4'],
    outputs=['output'],
    input_size_list=[[1, 128, 128, 3], [1, 128, 128, 3],
        [1, 128, 128, 3], [1, 128, 128, 1]])
```

● 加载模型出错时的排查步骤

首先确认原始深度学习框架是否可以加载该模型并进行正确的推理，检查原始模型是否有问题。

其次请将 RKNN-Toolkit2 升级到最新版本。如果模型有 RKNN-Toolkit2 不支持的层（或 OP），通过打开调试日志开关，在日志中可以看到哪一个算子是 RKNN-Toolkit2 不支持的。

如果仍无法解决，请将使用的 RKNN-Toolkit2 版本和详细的错误日志反馈给瑞芯微 NPU 开发团队。

10.5 模型量化问题

- 量化对模型体积的影响

分两种情况，当导入的模型是量化的模型时，`rknn.build()`接口的 `do_quantization=False` 会使用该模型里面的量化参数。当导入的模型是浮点的模型时，`do_quantization=False` 不会做量化的操作，但是会把权重从 `float32` 转成 `float16`，这块几乎不会有精度损失。这两种情况都减少了模型权重的体积，从而使得整个模型占用空间变小。

- 模型量化时，图片是否需要和模型输入的尺寸一致

不需要。RKNN-Toolkit2 会自动对这些图片进行缩放处理。但是缩放操作也可能会使图片信息发生改变，对量化精度产生一定影响，所以最好使用尺寸相近的图片。

如果是非图像格式的校正数据，如 `numpy` 格式，则需要与模型输入的 `shape` 一致。

- 量化校正集是否需要根据 `rknn_batch_size` 参数进行修改

不需要。

`rknn.build()`的 `rknn_batch_size` 参数只会修改最后导出的 RKNN 模型的 `batch` 维（由 1 改为 `rknn_batch_size`），并不会影响量化阶段的流程，因此量化校正集还是按照 `batch` 为 1 的方式来设置即可。

- 模型量化时，程序运行一段时间后被 `kill` 掉或程序卡住

在模型量化过程中，RKNN-Toolkit2 会申请较多的系统内存，有可能造成程序被 `kill` 掉或卡住。

解决方法：增加电脑内存或增大虚拟内存（交换分区）。

10.6 模型转换问题

- 常见转换 `bug` 报错的问题

如遇到如下类似转换报错，很可能是由于当前版本存在 `bug`，可尝试将 RKNN-Toolkit2 更新至最新版本。

- `infer_shapes` 类似错误


```
(op_type:Mul, name:Where_2466_mul): Inferred elem type
differs from existing elem type: (FLOAT) vs (INT64)
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1555, in
rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 5409, in
rknn.api.graph_optimizer.GraphOptimizer.run
E build: File "rknn/api/graph_optimizer.py", line 5123, in
rknn.api.graph_optimizer.GraphOptimizer._fuse_ops
E build: File "rknn/api/ir_graph.py", line 180, in
rknn.api.ir_graph.IRGraph.rebuild
E build: File "rknn/api/ir_graph.py", line 140, in
rknn.api.ir_graph.IRGraph._clean_model
E build: File "rknn/api/ir_graph.py", line 56, in
rknn.api.ir_graph.IRGraph.infer_shapes
E build: File "/home/anaconda3/envs/rk2/lib/python3.6/site-
packages/onnx/shape_inference.py", line 35, in infer_shapes
E build: inferred_model_str = C.infer_shapes(model_str,
check_type)
E build: RuntimeError: Inferred elem type differs from
existing elem type: (FLOAT) vs (INT64)
```

或:

```
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1643, in
rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 6256, in
rknn.api.graph_optimizer.GraphOptimizer.fuse_ops
E build: File "rknn/api/ir_graph.py", line 285, in
rknn.api.ir_graph.IRGraph.rebuild
E build: File "rknn/api/ir_graph.py", line 149, in
rknn.api.ir_graph.IRGraph._clean_model
E build: File "rknn/api/ir_graph.py", line 62, in
rknn.api.ir_graph.IRGraph.infer_shapes
E build: File "/usr/local/lib/python3.6/dist-
packages/onnx/shape_inference.py", line 35, in infer_shapes
E build: inferred_model_str = C.infer_shapes(model_str,
check_type)
E build: RuntimeError: Inferred shape and existing shape
differ in rank: (0) vs (3)
```

或:

```
(op_type:ReduceMax, name:ReduceMax_18): Interred shape and
existing shape differ in rank: (3) vs (0)
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
.....
E build: RuntimeError: Interred shape and existing shape differ
in rank: (3) vs (0)
```

■ p_fuse_two_mul 类似错误

```
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1643, in
rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 6197, in
rknn.api.graph_optimizer.GraphOptimizer.fuse_ops
E build: File "rknn/api/graph_optimizer.py", line 204, in
rknn.api.graph_optimizer._p_fuse_two_mul
E build: ValueError: non-broadcastable output operand with
shape () doesn't match the broadcast shape (3,2)
```

■ "Segmentation fault"类似错误

如 picodet 模型转换报错:

```
I _fold_constant remove nodes = ['Shape_0', 'Gather_4',
'Shape_1', 'Gather_6', 'Unsqueeze_0', 'Concat_8', 'Cast_3']
Segmentation fault (Core dumped)
```

■ p_fuse_mul_into_conv 类似错误

```
E build: Catch exception when building RKNN model:
.....
E build: ValueError: non broadcastable output operand with
shape (1,258,1,256) doesn't match the broadcast shape
(80256,258,1,256)
```

● 怎么判断算子 RKNN 是否支持

直接进行模型的转换，如果不支持会有相关提示。

也可参考以下两个算子支持文档：

- 1) RKNN-Toolkit2 发布包的 doc/RKNNToolKit2_OP_Support-x.x.x.md 文档，该文档为各个框架的粗略支持列表
- 2) RKNPU2 发布包的 doc/RKNN_Compiler_Support_Operator_List_vx.x.x.pdf 文档，该文档包含详细的 RKNN 算子规格支持情况。

● 转换时提示 Expand 算子不支持

建议：

- 1) 新版本已经支持 CPU 的 Expand，可尝试更新 RKNN-Toolkit2 / RKNPU2 至最新版本。
- 2) 修改模型，采用 repeat 算子来替代 expand 算子。

● 转换时提示 "Meet unsupported dims in reducesum"

模型转换出现 Meet unsupported dims in reducesum, dims: 6，具体如下：

```
D RKNN: [14:54:19.434] >>>>> start:
N4rknn17RKNNInitCastConstE
D RKNN: [14:54:19.434] <<<<<<< end:
N4rknn17RKNNInitCastConstE
D RKNN: [14:54:19.434] >>>>> start:
N4rknn20RKNNMultiSurfacePassE
D RKNN: [14:54:19.434] <<<<<<< end:
N4rknn20RKNNMultiSurfacePassE
D RKNN: [14:54:19.434] >>>>> start: N4rknn14RKNNtilingPassE
D RKNN: [14:54:19.434] <<<<<<< end: N4rknn14RKNNtilingPassE
D RKNN: [14:54:19.434] >>>>> start:
N4rknn23RKNNProfileAnalysisPassE
D RKNN: [14:54:19.434] <<<<<<< end:
N4rknn23RKNNProfileAnalysisPassE
D RKNN: [14:54:19.434] >>>>> start: OpEmit
E RKNN: [14:54:19.438] Meet unsupported dims in reducesum,
dims: 6
Aborted (core dumped)
```

目前 RKNN 不支持 6 维的 OP，大多数情况下只支持 4 维。

- **因 NonMaxSuppression 或 TopK 等后处理 Op 导致转换报错**

- NonMaxSuppression 或 TopK 等后处理 Op，RKNN 目前不支持。

- 可以将图的后处理子图部分移除，如：

```
rknn.load_onnx(model='picodet_XXX.onnx',
outputs=['concat_4.tmp_0', 'tmp_16'])
```

- 移除的子图在 cpu 端另行进行处理。

- **"invalid expand shape"类似报错**

例如 rvm_mobilenetv3_fp32.onnx 转换时出现以下报错：

```
[E:onnxruntime:, sequential_executor.cc:333 Execute] Non-zero
status code returned while running Expand node. Name:'Expand_294'
Status Message: invalid expand shape
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build: File "rknn/api/rknn_base.py", line 1638, in
rknn.api.rknn_base.RKNNBase.build
E build: File "rknn/api/graph_optimizer.py", line 5529, in
rknn.api.graph_optimizer.GraphOptimizer.fold_constant
E build: File "rknn/api/session.py", line 69, in
rknn.api.session.Session.run
E build: File
"/home/cx/work/tools/Anaconda3/envs/rknn/lib/python3.8/site-
packages/onnxruntime/capi/onnxruntime_inference_collection.py",
line 124, in run
E build: return self._sess.run(output_names, input_feed,
run_options)
E build:
onnxruntime.capi.onnxruntime_pybind11_state.InvalidArgument:
[ONNXRuntimeError] : 2 : INVALID_ARGUMENT : Non-zero status code
returned while running Expand node. Name:'Expand_294' Status
Message: invalid expand shape
```

因为 `downsample_ratio` 的输入值会改变模型中间 feature 的 size，所以说这种图本质上是动态图。建议修改模型 `downsample_ratio` 的逻辑，不要用输入的数值来控制中间 feature 的 shape。如需使用动态图功能，可在更新 1.5.2 的 RKNN-Toolkit2 后，使用动态 shape 的功能来模拟动态图（同样需要修改模型 `downsample_ratio` 的逻辑，不要用输入的数值来控制中间 feature 的 shape，目前动态 shape 功能只支持输入的 shape 是可变的情况）。

● rknn.config()的 mean_values 报错提示

设置 mean/std 为：

```
rknn.config(mean_values=[128, 128, 128], std_values=[128, 128, 128])
```

时转换模型报错：

```
--> Loading model
transpose_input for input_1: shape must be rank 4, ignored
E load_tflite: The len of mean_values ([128, 128, 128]) for
input 0 is wrong, expect 32!
```

原因可能是模型的输入不是 3 通道图像数据（例如输入 shape 是 1x32，非图像数据），此时：

- 需要根据输入通道个数来设置 `mean_values / std_values`。
- 如果模型不需要指定 `mean/std`，`rknn.config()` 可以不设置 `mean_values / std_values`（`mean/std` 一般只对图像输入有效）。

● 模型存在 4 维以上 Op 时报错（如 5 维或 6 维）

当模型存在 4 维以上 Op 时（如 5 维或 6 维），会有如下报错：

```
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build:   File "rknn/api/rknn_base.py", line 1580, in
rknn.api.rknn_base.RKNNBase.build
E build:   File "rknn/api/rknn_base.py", line 341, in
rknn.api.rknn_base.RKNNBase._generate_rknn
E build:   File "rknn/api/rknn_base.py", line 307, in
rknn.api.rknn_base.RKNNBase._build_rknn
E build: IndexError: vector::_M_range_check: __n (which is
4) >= this->size() (which is 4)
```

RKNN 目前暂不支持 4 维以上的 OP，可以手工将这些节点去掉。

● RKNN 是否支持动态卷积

目前 RK3588/RK3576 平台支持 `group` 参数为 1 的动态卷积。其他平台暂不支持。

● "Not support input data type 'float16'"报错

pytorch 训练的权重类型为 float16 的模型，在转换 RKNN 时出现以下报错：

```
--> Building model
E build: Not support input data type 'float16'
W build: ===== WARN(3) =====
E rknn-toolkit2 version: 1.3.0-11912b58
E build: Catch exception when building RKNN model!
E build: Traceback (most recent call last):
E build:   File "rknn/api/rknn_base.py", line 1638, in
rknn.api.rknn_base.RKNNBase.build
E build:   File "rknn/api/graph_optimizer.py", line 5524, in
rknn.api.graph_optimizer.GraphOptimizer.fold_constant
E build:   File "rknn/api/load_checker.py", line 63, in
rknn.api.load_checker.create_random_data
E build:   File "rknn/api/rknn_log.py", line 113, in
rknn.api.rknn_log.RKNNLog.e
E build: ValueError: Not support input data type 'float16'!
```

目前 RKNN-Toolkit2 还还不支持 float16 的权重类型的 Pytorch 模型，需改为 float32。

- 动态图相关报错

转换模型时，如果出现以下类似报错：

```
E build: ValueError: The Op of 'NonZero' is not support! it
will cause the graph to be a dynamic graph!
```

说明包含该 OP 的模型为动态图，需要手动修改模型，用其他 OP 替换或将其移除。

- RKNN 模型大小问题

模型转换结束后，可能存在转换出来的 RKNN 模型比原始模型大的现象，甚至跟模型的输入 shape 也有关系，这种现象是正常的。因为 RKNN 模型里不仅仅包含权重和图结构信息，还会有很多 NPU 的寄存器配置信息，并且为了提高运行效率，可能也会做 OP 的拆解等操作，这些都会导致 RKNN 模型变大。

10.7 模拟器推理及连板推理的说明

- 术语说明

模拟器推理：RKNN-Toolkit2 在 Linux x86_64 平台提供模拟器功能，可以在没有开发板的情况下进行模型推理，获取推理结果。（该功能输出结果未必与连板或板端一致，更推荐使用连板推理或板端推理）。

连板推理：指在开发板已连接电脑的情况下，调用 RKNN-Toolkit2 的 Python API 推理模型，获取推理结果。

板端推理：指在开发板上调用 RKNN-Toolkit2 的 C API 接口推理模型，获取推理结果。

- **模拟器推理结果与连板推理结果不一致**

发生此情况时，可能意味着板端的结果不正确。

由于硬件和驱动的差异，模拟器不保证可以和板端获取一模一样的结果。但如果差异实在太，则大概率是板端驱动 Bug 导致，可以将问题反馈给 RK 的 NPU 团队进行分析。

- **连板推理的工作原理**

使用连板推理时，RKNN-Toolkit2 会与板端的 RKNN Server 进行通信，通信时会将模型、模型的输入由 PC 端传至板端，随后调用 RKNPU2 C API 进行模型推理，板端推理完成后将结果回传至 PC 端。

- **连板推理与板端推理结果有差异**

连板推理是基于 RKNPU2 C API 实现的，理论上连板推理结果会与 RKNPU2 C API 推理结果一致。当这两者出现较大差异时，请确认输入的预处理、数据类型、数据的排布方式（NCHW，NHWC）是否有差异。

需指出，如差异很小且发生在小数点后 3 位及之后的数值上，则属于正常现象。差异可能产生在使用不同的库读取图片、转换数据类型等步骤上。

- **板端推理的速度比连板推理更快**

由于连板推理存在额外的数据拷贝、传输过程，会导致连板推理的性能不如板端的 RKNPU2 C API 推理性能。因此，NPU 实际推理性能以 RKNPU2 C API 的推理性能为准。

- **涉及连板调试、连板推理功能时，获取详细的错误日志**

连板推理时，模型的初始化、推理等操作主要在开发板上完成，此时日志信息主要产生在板端上。

为了获取具体的板端调试信息，可以通过串口进入开发板操作系统。然后执行以下两条命令设置获取日志的环境变量。保持串口窗口不要关闭，再进行连板调试，此时板端的错误信息就会显示在串口窗口上：

```
export RKNN_LOG_LEVEL=5
restart_rknn.sh
```

10.8 模型评估常见问题

- 量化模型精度不及预期

参考本文档的[第7章节](#)。

- 支持哪些框架的已量化模型

RKNN-Toolkit2 1.4 及之后的版本支持 TensorFlow、TensorFlow Lite 和 PyTorch 框架的已量化模型。

- 连板调试时，连接设备失败

连板精度分析（rknn.accuracy_analysis()）时出现如下报错：

```
E accuracy_analysis: Connect to Device Failure (-1)
E accuracy_analysis: Catch exception when init runtime!
E accuracy_analysis: Traceback (most recent call last):
E accuracy_analysis: File "rknn/api/rknn_base.py", line 2001,
in rknn.api.rknn_base.RKNNBase.init_runtime
E accuracy_analysis: File "rknn/api/rknn_runtime.py", line
194, in rknn.api.rknn_runtime.RKNNRuntime.__init__
E accuracy_analysis: File "rknn/api/rknn_platform.py", line
331, in rknn.api.rknn_platform.start_ntp_or_adb
```

或连板推理（rknn.inference）时出现如下报错：

```
I target set by user is: rk3568
I Starting ntp or adb, target is RK3568
I Device [0c6a9900ef4871e1] not found in ntb device list.
I Start adb...
I Connect to Device success!
I NPUTransfer: Starting NPU Transfer Client, Transfer version
2.1.0 (b5861e7@2020-11-23T11:50:36)
D NPUTransfer: Transfer spec = local:transfer_proxy
D NPUTransfer: ERROR: socket read fd = 3, n = -1: Connection
reset by peer
D NPUTransfer: Transfer client closed fd = 3
E RKNNAPI: rknn_init, server connect fail! ret = -
9(ERROR_PIPE)!
E init_runtime: Catch exception when init_runtime!
E init_runtime: Traceback (most recent call last):
E init_runtime: File "rknn/api/rknn_base.py", line 2001, in
rknn.api.rknn_base.RKNNBase.init_runtime
E init_runtime: File "rknn/api/rknn_runtime.py", line 361, in
rknn.api.rknn_runtime.RKNNRuntime.build_graph
E init_runtime: Exception: RKNN init failed. error code:
RKNN_ERR_DEVICE_UNAVAILABLE
```

原因可能是板端未开启 RKNN Server 服务，请根据 [2.2 章节](#) 相关说明运行板端 RKNN Server 服务。

- 连板调试时，rknn_init 失败，返回-6 或模型非法的错误

错误信息如下：

```
E RKNNAPI: rknn_init, msg_load_ack fail, ack = 1(ACK_FAIL),  
expect 0(ACK_SUCC)!  
D NPUTransfer: Transfer client closed, fd = 4  
E init_runtime: Catch exception when init runtime!  
E init_runtime: Traceback (most recent call last):  
E init_runtime:   File "rknn/api/rknn_base.py", line 2011, in  
rknn.api.rknn_base.RKNNBase.init_runtime  
E init_runtime:   File "rknn/api/rknn_runtime.py", line 361,  
in rknn.api.rknn_runtime.RKNNRuntime.build_graph  
E init_runtime: Exception: RKNN init failed. error code:  
RKNN_ERR_MODEL_INVALID
```

出现该错误一般有以下几种情况：

- 在生成 rknn 模型时，不同版本的 RKNN-Toolkit2 和驱动是有对应关系的，建议将 RKNN-Toolkit2 / RKNPU2 和开发板的固件都升级到最新的版本。
- 没有正确设置 target_platform。例如不设置 rknn.config()接口中的 target_platform 时，生成的 RKNN 模型只能在 RK3566/RK3568 上运行。如果要在其他平台上运行（如 RK3588/RK3588S/RV1103/RV1106/RK3562），则需要在调用 rknn.config()接口时设置相应的 target_platform。
- 如果是在 Docker 容器中推理时出现该问题，有可能是因为宿主机上的 npu_transfer_proxy 进程没有结束，导致通信异常。可以先退出 Docker 容器，将宿主机上的 npu_transfer_proxy 进程结束掉，然后再进入容器执行推理脚本。
- 也可能是 RKNN 模型本身有问题。此时可以用串口连到开发板，在设置环境变量 RKNN_LOG_LEVEL=5 后执行 restart_rknn.sh，然后重跑程序并将产生的详细日志记录下来，反馈给瑞芯微 NPU 团队。

- 连板调试时，rknn_init()失败，返回设备不可用的错误

错误信息如下：


```
E RKNNAPI: rknn_init, msg_ioctl_ack fail, data_len = 104985,
except 102961!
D NPUTransfer: Transfer client closed, fd = 3
E init_runtime: Catch exception when init_runtime!
E init_runtime: Traceback (most recent call last):
E init_runtime: File "rknn/api/rknn_base.py", line 1961, in
rknn.api.rknn_base.RKNNBase.init_runtime
E init_runtime: File "rknn/api/rknn_runtime.py", line 360, in
rknn.api.rknn_runtime.RKNNRuntime.build_graph
E init_runtime: Exception: RKNN init failed. error code:
RKNN_ERR_DEVICE_UNAVAILABLE
```

该问题的原因比较复杂，请按以下方式排查：

确保 RKNN-Toolkit2 / RKNPU2 及开发板的系统固件都已经升级到最新版本。各组件版本查询方法请参考 [2.2](#) 章节。

另外，需要确保 adb devices 或 rknn.list_devices() 都能看到设备，并且 rknn.init_runtime() 的 target 和 device_id 设置正确。

● Runtime 出现 "Invalid RKNN format" 报错

Runtime 上出现以下报错：

```
Loading model ...
E RKNN: [06:28:39.048] parseRKNN from buffer: Invalid RKNN
format!
E RKNN: [06:28:39.049] rknn_init, load model failed!
rknn_init error ret=-1
```

原因：

- 1) 可能是模型转换时的 rknn.config() 的 target_platform 没有设置对，或没有设置（如没有设置默认是 RK3566）。
- 2) Runtime 版本与 RKNN-Toolkit2 不兼容。

建议：

- 1) 设置正确的 target_platform。
- 2) RKNN-Toolkit2 与 Runtime 要一起更新到同一个版本。

● rknn.inference() 耗时与 rknn.eval_perf() 理论速度不一致

因为 rknn.inference() 使用 PC + adb 的方式进行连板推理，存在着一些固定的数据传输开销，因此与 rknn.eval_perf() 理论速度不一致。

对于更真实的帧率，建议直接在开发板上使用 RKNPU2 C API 进行测试。

● rknn.inference() 对多 batch 的支持

RKNN-Toolkit2 1.4.0 及之后的版本，可以在构建 RKNN 模型时就指定输入图片的数量，详细用法参考 RKNN-Toolkit2 API 手册中关于 rknn.build 接口的说明。

另外，当 rknn_batch_size 大于 1（如等于 4 时），Python 里推理的调用要由：

```
outputs = rknn.inference(inputs=[img])
```

修改为：

```
img = np.expand_dims(img, 0)
img = np.concatenate((img, img, img, img), axis=0)
outputs = rknn.inference(inputs=[img])
```

完整示例请参考：examples/functions/multi_batch/

- **运行多个 RKNN 模型**

运行两个或多个模型时，需要创建多个 RKNN 对象。一个 RKNN 对象对应一个模型，类似一个上下文。每个模型在各自的上下文里初始化模型，推理，获取推理结果，互不干涉。这些模型在 NPU 上推理时是串行进行的。

- **模型推理的耗时非常长，而且得到的结果错误**

如果推理耗时超过 20s，且结果错误，这通常是 NPU 出现了 NPU Hang 的 BUG。如果遇到该问题，可以尝试更新 RKNN-Toolkit2 / RKNPU2 到最新版本。

- **模型输入为 3 维情况下，连板推理结果错误**

模型的输入为 3 维情况下，如出现 Simulator 的仿真结果正确，但连板推理结果错误的情况。原因可能是当前 NPU 的输入 3 维支持还不完善，后面会完善 3 维的支持。

建议：

- 先将模型输入改为 4 维。
- 更新 RKNN-Toolkit2 / RKNPU2 至最新版本进行尝试。

- **连板推理结果错误，并且每次都不一致**

ONNX 模型转 RKNN 后，用 Simulator 的仿真结果正确，并且每次结果都一致。但在连板推理时结果错误，并且每次都不一致。这种问题可能是板端 NPU 内核驱动 bug 导致，此时需要更新板端的 NPU 内核驱动，并且需要一并更新最新的 RKNN-Toolkit2 / RKNPU2。

- **模型存在较多的 Resize OP 时，出现精度下降问题**

当 ONNX 模型里存在较多的 Resize OP 时，转换为 RKNN 后出现精度下降。可能的原因是：

1) 精度下降是因为 NPU 目前还不支持硬件级别 Resize（后续会支持），转换工具会将 Resize

转为 ConvTranspose，会导致一点点的精度丢失。

2) 如模型有多个串联的 Resize，则可能会累积了太多误差导致精度下降比较多。

建议：

1) 目前尽量避免 Resize 的使用（如将 Resize 改为 ConvTranspose 再进行训练）

2) 可以在 rknn.config()里加入 optimization_level=2 的参数，此时 Resize Op 会走 cpu，精度不会掉，但会导致性能下降。

- **do_quantization 设为 False 以后推理结果都为 nan**

rknn.build()接口中的 do_quantization 设为 True 时推理结果没有异常，但设为 False 以后推理结果就都变为 nan 了。原因可能是 do_quantization=False 时，RKNN 模型的运算类型是 fp16 的，但该模型的中间层（如卷积）输出的范围可能超出了 fp16（65536）的范围（如-51597~75642）。

建议：

训练的时候需要保证中间层的输出不超过 fp16 的表达范围（一般通过添加 BN 层来解决该问题）。

- **QAT 模型与 RKNN 模型结果不一致**

在 Pytorch 框架下使用 QAT 训练了一个分类模型并转为 RKNN 模型，对该模型使用 Pytorch 和 RKNN 分别进行推理，发现得到的结果不一样，原因可能是 Pytorch 的推理没有设置 engine='qnnpack'，因为 RKNN 的推理方式与 qnnpack 更为贴近。

- **怎么获取模型运行时候内存占用率**

可以使用 rknn.eval_memory()接口，输出的日志里有个 Total 项，就是总的占用大小。

- **性能评估时，开启或关闭 rknn.init_runtime()的 perf_debug 参数，性能数据的差异**

开启 perf_debug 时，为了收集每层的信息，会添加一些调试代码，并且可能禁用一些并行的机制，因此耗时比 perf_debug=False 时多一些。

开启 perf_debug 的主要作用是看模型中是否有耗时占比比较多的层，以此为依据来设计优化方案。

- **环境用的 docker，之前连板推理正常，重启 docker 后，推理时卡在初始化环境阶段？**

因为 docker 重启时 npu_transfer_proxy 类似于异常退出的状态，导致开发板上的 RKNN Server

无法检测到上端连接已经断开，这时需要重启下开发板，重置 RKNN Server 的连接状态。

10.9 C API 使用常见问题

- **rknn_outputs_release()是否会释放 rknn_output 数组**

rknn_outputs_release()与 rknn_outputs_get()配合调用，它只释放 rknn_output 数组里的 buf。类似的情况还有 rknn_destroy_mem()。

- **rknn_create_mem 如何创建合适的大小的内存？**

对于输入而言，一般原则是：如果是量化 RKNN 模型，rknn_create_mem()使用 rknn_tensor_attr 的 size_with_stride 分配内存；非量化模型 rknn_create_mem()使用用户填充的数据类型的字节数*n_elems 分配内存。

对于输出而言，rknn_create_mem()使用用户填充的数据类型的字节数*n_elems 分配内存。

- **输入数据如何填充？**

如果使用通用 API,对于四维形状输入，fmt=NHWC，即数据填充顺序为[batch, height, width, channel]。非四维输入形状，fmt=UNDEFINED，按照模型的原始形状填充数据。

如果使用零拷贝 API,对于四维形状输入，fmt=NHWC/NC1HWC2。fmt=NC1HWC2 时如何填充数据请参考《RKNN Runtime 零拷贝调用》章节。非四维输入形状，fmt=UNDEFINED，按照模型的原始形状填充数据。

- **pass_through 如何使用？**

输入数据格式由 rknn_query()的 RKNN_NATIVE_INPUT_ATTR 命令获取。如果是 4 维形状：对于通道值为 1、3、4，layout 要求使用 NHWC，其他通道值要求使用 NC1HWC2；如果是非 4 维形状，建议指定 layout=UNDEFINED。

另外，在 pass_through 模式下，量化模型通常指定输入 Tensor 的 dtype 为 INT8，非量化模型通常指定输入 Tensor 的 dtype 为 FLOAT16。

- **出现"failed to submit"错误如何处理？**

如果错误出现在第一层卷积并且使用零拷贝接口，可能的原因是输入 tensor 内存分配不够导致，此时应该使用 tensor 属性中的 size_with_stride 分配内存。

如果错误出现在中间的 NPU 层，可能的原因是模型配置出错，此时可在错误日志中找到最新的 SDK 网盘链接，建议升级最新工具链或者在转换 RKNN 模型时将该层指定到 CPU 上运行。

- **出现"Meet unsupport xxx operator"错误如何处理？**

在板端运行 demo 出现类似的报错时，一般是板端的 Runtime (librknnrt.so) 不支持该算子。建议用户先更新 RKNN 相关工具链到最新版本，再重新转换模型，并在板端重跑 demo。

如果最新的工具链还出现同样报错，则用户需要自行添加该算子的实现，可以参考 [5.5](#) 章节来自定义实现算子，或者通过 redmine 上报给 RKNN 团队。

- **动态 shape 模型是否支持在零拷贝流程中使用外部分配的内存？**

1.6.0 之前版本不支持，1.6.0 版本开始支持使用 RKNN_FLAG_MEM_ALLOC_OUTSIDE 标志初始化上下文。

- **自定义算子性能评估(runtime)**

自定义算子的性能可以在板端上设定 RKNN_LOG_LEVEL=4 或 5 以上，并运行测试 demo，runtime 库就会自动打印出每层耗时信息，包含自定义 OP 的耗时。

注意：这层自定义 OP 的耗时包含 compute 回调函数在内的所有耗时，以 GPU OP 为例子，耗时会包含 compute 回调函数在内，包括 clImportMemoryARM 函数的，以及数据格式和类型的转换的耗时(包含给输入输出的转换)和最后 GPU OP 的 kernel 运行耗时。

11 相关资源

RKNN: <https://github.com/airockchip/rknn-toolkit2>。

Model Zoo: https://github.com/airockchip/rknn_model_zoo。

RGA: <https://github.com/airockchip/librga>。

Rockchip