BM1688/CV186AH SOPHONSDK 开发指南

发行版本 master

SOPHGO

2024 年 05 月 15 日

目录

CHAPTER 1

声明

法律声明

注意

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

技术支持

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

邮编
100094

网址
https://www.sophgo.com/

邮箱
sales@sophgo.com

电话
+86-10-57590723 +86-10-57590724

SOPHONSDK for BM1688/CV186AH 发布记录

版本	发布日期	说明
v24.01.01	2024.01.01	2024 年 1 月第一次发布。
v24.02.01	2024.02.20	更新 SDK v1.3 下载链接、新增内存修改工具使用说明、sophon-mw 更名为 sophon-media
v24.04.01	2024.04.16	更新 SDK v1.5 下载链接、TPU-MLIR 安装方法
v24.05.01	2024.05.15	更新 SDK v1.5.1 下载链接、新增 SDK 源码下载以及编译方法、新增 BM1684、BM1684X SDK 兼容性方法

CHAPTER 2

引言

2.1 术语解释

术语	说明
BM1688/CV186AH	算能科技面向深度学习领域推出的两款第五代张量处理器
BM1684X	算能科技面向深度学习领域推出的第四代张量处理器
BM1684	算能科技面向深度学习领域推出的第三代张量处理器
智能视觉深度学习处理器	BM1688/CV186AH/BM1684/BM1684X 中的神经网络运算单元
VPSS	BM1688/CV186AH 中的视频处理子系统，包括图形运算加速单元以及解码单元
VPU	BM1684/BM1684X 中的解码单元
VPP	BM1684/BM1684X 中的图形运算加速单元
JPU	BM1688/CV186AH/BM1684/BM1684X 中的图像 JPEG 编解码单元
SOPHONSDK	算能科技基于 BM1688/CV186AH/BM1684/BM1684X 的原创深度学习开发工具包
PCIe Mode	BM1684/BM1684X 的一种工作形态，作为加速设备来进行使用，客户算法运行于 x86 主机
SoC Mode	BM1688/CV186AH/BM1684/BM1684X 的一种工作形态，本身作为主机独立运行，客户算法可以直接运行其上
arm_pcie Mode	BM1684/BM1684X 的一种工作形态，搭载 BM1684/BM1684X 的板卡作为 PCIe 从设备插到 ARM 处理器的服务器上，客户算法运行于 ARM 处理器的主机上

续下页

表 2.1 – 接上页

BMCompiler	面向智能视觉深度学习处理器研发的深度神经网络的优化编译器，可以将深度学习框架的各种深度神经网络转化为处理器上运行的指令流
BMRuntime	智能视觉深度学习处理器推理接口库
BMCV	图形运算硬件加速接口库
BMLib	在内核驱动之上封装的一层底层软件库，设备管理、内存管理、数据搬运、API 发送、A53 使能、功耗控制
mlir	由 TPU-MLIR 生成的中间模型格式，用于迁移或者量化模型
BModel	面向智能视觉深度学习处理器的深度神经网络模型文件格式，其中包含目标网络的权重（weight）、指令流等
BMLang	面向智能视觉深度学习处理器的高级编程模型，用户开发时无需了解底层硬件信息
TPUKernel	基于智能视觉深度学习处理器原子操作（根据 BM1688/CV186AH/BM1684/BM1684X 指令集封装的一套接口）的开发库。
SAIL	支持 Python/C++ 接口的 SOPHON Inference 推理库，是对 BMCV、sophon-media、BMLib、BMRuntime 等的进一步封装
TPU-MLIR	智能视觉深度学习处理器编译器工程，可以将不同框架下预训练的神经网络，转化为可以在算能智能视觉深度学习处理器上高效运算的 bmodel

2.2 注意事项

- 推荐使用 cmake 中的 find_package 查找 libsophon 和 sophon-media 的库包，并进行链接。
- 部分头文件不包含于 libsophon，如 bm_wrapper.hpp、utils.hpp。
- SoC 模式下，SDK 动态库文件位于/opt/sophon/目录下。
- SoC 模式下，通过开机时的 systemd 服务，自动加载/opt/sophon/libsophon-current/data 下的 ko 内核模块。如用户定义的自启动服务，并且依赖 BM168X，请确保在 ko 加载后启动。

2.3 授权

2.4 帮助与支持

在使用过程中，如有关于 BM1688/CV186AH SOPHONSDK 的任何问题或者意见和建议，请联系相关技术支持。

CHAPTER 3

SDK 软件包

3.1 SDK 简介

目录

“BM1688/CV186AH SOPHONSDK” 是算能科技基于 BM1688/CV186AH 定制的深度学习 SDK，涵盖了神经网络推理阶段所需的模型优化、高效运行支持等能力，为深度学习应用开发和部署提供易用、高效的全栈式解决方案。

3.1.1 BM1688/CV186AH SOPHONSDK 文件清单

解压后的 SDK 文件结构如下：

```
BM1688/CV186AH SOPHONSDK
├─── doc
│    ├─── BMLib_Technical_Reference_Manual.pdf
│    ├─── BMLIB开发参考手册.pdf
│    ├─── LIBSOPHON_User_Guide.pdf
│    ├─── LIBSOPHON使用手册.pdf
│    ├─── SOPHON BMRuntime Technical Reference Manual.pdf
│    ├─── SOPHON-SAIL_en.pdf
```

(续下页)

(接上页)

```
│   ├── SOPHON-SAIL_zh.pdf
│   ├── TPU-MLIR_Quick_Start.pdf
│   ├── TPU-MLIR_Technical_Reference_Manual.pdf
│   ├── TPU-MLIR开发参考手册.pdf
│   ├── TPU-MLIR快速入门指南.pdf
│   ├── 算能边缘产品BMCV开发参考手册.pdf
│   ├── 算能边缘产品BMRUNTIME开发参考手册.pdf
│   ├── 算能边缘产品BSP开发参考手册_en.pdf
│   ├── 算能边缘产品BSP开发参考手册.pdf
│   ├── 算能边缘产品MULTIMEDIA常见问题手册.pdf
│   ├── 算能边缘产品MULTIMEDIA开发参考手册.pdf
│   └── 算能边缘产品多媒体使用参考指南.pdf
├── isp_tool
│   └── CviPQtool_20240403.zip
├── sophon-demo
│   ├── release_version.txt
│   ├── sophon-demo.MD5
│   └── sophon-demo_v0.1.10_6375a8be_20240403.tar.gz
├── sophon-img
│   ├── boot
│   │   ├── boot.itb
│   │   ├── boot.scr.emmc
│   │   ├── fip.bin
│   │   ├── fip_debug.bin
│   │   └── multi.its
│   ├── bsp-debs
│   │   ├── bmssm_soc_1.2.0_SDK.deb
│   │   ├── linux-headers-5.10.4-tag--00082-g91a1fb25c7fb.deb
│   │   ├── linux-image-5.10.4-tag--00082-g91a1fb25c7fb-dbg.deb
│   │   ├── linux-image-5.10.4-tag--00082-g91a1fb25c7fb.deb
│   │   ├── linux-libc-dev_5.10.4-tag--00082-g91a1fb25c7fb-1_arm64.deb
│   │   ├── qt5-base.deb
│   │   ├── sophgo-bsp-rootfs_1.1.0_arm64.deb
│   │   ├── sophgo-hdmi_1.0.0_arm64.deb
│   │   ├── sophliteos_soc_1.1.2_sdk.deb
│   │   ├── sophon-media-soc-sophon-ffmpeg_1.5.0_arm64.deb
│   │   ├── sophon-media-soc-sophon-opencv_1.5.0_arm64.deb
│   │   ├── sophon-media-soc-sophon-sample_1.5.0_arm64.deb
│   │   ├── sophon-soc-libisp_1.0.0_arm64.deb
│   │   ├── sophon-soc-libisp-dev_1.0.0_arm64.deb
│   │   ├── sophon-soc-libsophon_0.4.9_arm64.deb
│   │   └── sophon-soc-libsophon-dev_0.4.9_arm64.deb
│   ├── doc
│   │   ├── BMLib_Technical_Reference_Manual.pdf
│   │   ├── BMLIB开发参考手册.pdf
│   │   ├── LIBSOPHON_User_Guide.pdf
│   │   ├── LIBSOPHON使用手册.pdf
│   │   ├── SOPHON_BMCV_Technical_Reference_Manual.pdf
│   │   ├── SOPHON BMRuntime Technical Reference Manual.pdf
│   │   ├── 算能边缘产品BMRUNTIME开发参考手册.pdf
│   │   └── 算能边缘产品BSP开发参考手册.pdf
```

(续下页)

(接上页)

```
│   ├── libsophon_soc_0.4.9_aarch64.tar.gz
│   ├── release_version.txt
│   ├── sdcard.tgz
│   ├── sophon-img-edge-auto.MD5
│   ├── tftp.tgz
│   └── usb.tgz
├── sophon_media
│   ├── doc
│   │   ├── Multimedia_FAQ_en.pdf
│   │   ├── Multimedia_Guide_en.pdf
│   │   ├── Multimedia_Manual_en.pdf
│   │   ├── 算能边缘产品MULTIMEDIA常见问题手册.pdf
│   │   ├── 算能边缘产品MULTIMEDIA开发参考手册.pdf
│   │   └── 算能边缘产品多媒体使用参考指南.pdf
│   ├── release_version.txt
│   ├── sophon_media.MD5
│   ├── sophon-media-soc_1.5.0_aarch64.tar.gz
│   ├── sophon-media-soc-sophon-ffmpeg_1.5.0_arm64.deb
│   ├── sophon-media-soc-sophon-ffmpeg-dev_1.5.0_arm64.deb
│   ├── sophon-media-soc-sophon-opencv_1.5.0_arm64.deb
│   ├── sophon-media-soc-sophon-opencv-dev_1.5.0_arm64.deb
│   └── sophon-media-soc-sophon-sample_1.5.0_arm64.deb
├── sophon-sail
│   ├── release_version.txt
│   ├── sophon-sail_3.7.0.tar.gz
│   ├── SOPHON-SAIL_en.pdf
│   ├── sophon-sail.MD5
│   └── SOPHON-SAIL_zh.pdf
├── sophon-stream
│   ├── release_version.txt
│   ├── sophon-stream.MD5
│   └── sophon-stream_v0.0.4-rc4_9517722_20240402.tar.gz
├── tpu-mlir
│   ├── tpu_mlir-1.6.1-py3-none-any.whl
│   └── tpu-mlir_readme.txt
└── tpu-perf
    ├── tpu_perf-1.2.52-py3-none-manylinux2014_aarch64.whl
    └── tpu_perf-1.2.52-py3-none-manylinux2014_x86_64.whl
```

其中各 SDK 接口的说明如下：

- **doc**
  - BM1688/CV186AH SOPHONSDK 相关接口的开发参考手册
- **isp-tools**
  - 用于调试 isp 相关参数的调试工具
- **sophon-demo**
  - x86 和 SoC 环境下针对单模型或场景的综合例程，供用户在深度学习应用开发过程中参考。

- **sophon-img**
  - 边缘服务器（soc 模式）刷机包以及相关 bsp 文件
- **sophon-stream**
  - 面向 sophon 开发平台的数据流处理工具, 够将前处理/推理/后处理分别运行在 3 个线程上，最大化的实现并行。
- **sophon-media**
  - 封装了 SOPHON-OpenCV、SOPHON-FFmpeg 等库，用来驱动 VPSS 等硬件，支持 RTSP 流、GB28181 流的解析，视频图像编解码加速等，供用户进行深度学习应用开发。
- **sophon-sail**
  - 提供了支持 Python/C++ 的高级接口，是对 BMRuntime、BMCV、sophon-media、BMLib 等底层库接口的封装，供用户进行深度学习应用开发。
- **tpu-mlir**
  - 为 Tensor Processing Unit 编译器工程提供一套完整的工具链，可以将不同框架下预训练的神经网络，转化为可以在算能智能视觉深度学习处理器上高效运行的二进制文件 BModel。目前直接支持的框架包括 tflite、onnx 和 Caffe。
- **tpu-perf**
  - 用于模型性能和精度验证的完整工具包。

3.1.2 SDK 主要模块

在您在移植适配过程中将主要涉及到以下 4 个模块：

1. **硬件驱动及运行时库——LIBSOPHON**：动态库路径位于/opt/sophon/libsophon-current，包含 BMCV（张量运算及图像处理库）、BMRuntime（推理运行时）、BMLib（设备管理、数据搬运、A53 使能等）等，用来驱动 VPSS、智能视觉深度学习处理器等硬件，完成图像处理、张量运算、模型推理等操作。
2. **多媒体库——SOPHON-MEDIA**：动态库路径位于/opt/sophon/sophon-ffmpeg-latest、/opt/sophon/sophon-opencv-latest，支持 SOPHON 设备硬件加速的 SOPHON-OpenCV 和 SOPHON-FFmpeg，支持 RTSP 流、GB28181 流的解析，视频及图片的编解码。
3. **模型编译量化工具链——TPU-MLIR**：支持 Caffe、TFLite、ONNX 等框架模型的模型转换以及量化
4. **算丰深度学习处理加速库——SAIL**：支持 Python/C++ 的高级接口，是对 BMRuntime、BMCV、sophon-media 等底层库接口的封装。源码发布，用户可根据需求，编译生成特定的版本。

3.2 资料简介

目录

BM1688/CV186AH SOPHONSDK 是一个一站式 SDK，其中包含了模型转换、模型量化、算法移植等相关模块，我们提供了包括文档、视频、论坛、开源仓库等一系列资料帮助用户进行算法移植和开发工作。

请先阅读 GUIDE，其中包括：基本概念及 SDK 简介；资料简介、文档说明；SDK 的获取、安装、配置及更新；快速入门例子、模型转换及模型量化、示例代码的讲解等内容

当您在某个环节遇到问题时，可以在 SOPHONSDK 目录的 doc 文件夹下，查找对应问题的参考指南：

```
doc
├── BMLib_Technical_Reference_Manual.pdf
├── BMLIB开发参考手册.pdf
├── LIBSOPHON_User_Guide.pdf
├── LIBSOPHON使用手册.pdf
├── SOPHON BMRuntime Technical Reference Manual.pdf
├── SOPHON-SAIL_en.pdf
├── SOPHON-SAIL_zh.pdf
├── TPU-MLIR_Quick_Start.pdf
├── TPU-MLIR_Technical_Reference_Manual.pdf
├── TPU-MLIR开发参考手册.pdf
├── TPU-MLIR快速入门指南.pdf
├── 算能边缘产品BMCV开发参考手册.pdf
├── 算能边缘产品BMRUNTIME开发参考手册.pdf
├── 算能边缘产品BSP开发参考手册_en.pdf
├── 算能边缘产品BSP开发参考手册.pdf
├── 算能边缘产品MULTIMEDIA常见问题手册.pdf
├── 算能边缘产品MULTIMEDIA开发参考手册.pdf
└── 算能边缘产品多媒体使用参考指南.pdf
```

上述参考指南的相关说明如下所示

- **BMLIB 开发参考手册**
  - BMLIB 是在内核驱动之上封装的一层底层软件库，负责设备 Handle 的管理、内存管理、数据搬运、API 的发送和同步、A53 使能、设置智能视觉深度学习处理器工作频率等
- **LIBSOPHON 使用手册**
  - LIBSOPHON 包含 BMCV、BMRuntime、BMLib 等库，用来驱动 VPP、智能视觉深度学习处理器等硬件，完成图像处理、张量运算、模型推理等操作。
- **算能边缘产品 BMRUNTIME 开发参考手册**

  - BMRuntime 提供了丰富的接口，驱动 BModel 在 SOPHON 智能视觉深度学习处理器中执行

- **sophon-sail_zh**
  - sail 开发参考手册，包含编译，C++ 接口调用，python 接口调用方法等
- **TPU-MLIR 快速入门指南**
  - 基于 TPU-MLIR 工具的模型迁移方法以及使用示例
- **TPU-MLIR 开发参考手册**
  - Tensor Processing Unit 编译器工程
- **算能边缘产品 BMCV 开发参考指南**
  - 算能硬件图像处理加速库开发参考手册，包含常见图像处理接口函数
- **算能边缘产品 BSP 开发参考手册**
  - BM168X 系列深度学习处理器计算模组（含开发板）的详细手册
- **算能边缘产品 MULTIMEDIA 常见问题手册**
  - 多媒体方面的常见问题及解答
- **算能边缘产品 MULTIMEDIA 开发参考手册**
  - 驱动 VPSS 进行 RTSP 流、GB28181 流的解析以及视频、图像编解码等多媒体相关
- **算能边缘产品多媒体使用参考指南**
  - 多媒体库安装以及编译指导手册

其他相关的开发参考资料如下：

- **官网视频教程**：https://developer.sophgo.com/site/index/course/all/all.html，其中包括：智算卡、智算盒子、智算服务器等产品介绍视频；快速跑通 PCIe 模式的例程、快速跑通 SoC 模式的例程、SDK 算法移植介绍、BMCV 编程示例、编解码编程示例、BMLang 编程示例、TPUKernel 编程示例。
- **官网论坛**：https://developer.sophgo.com/forum/index/.html ，欢迎在官网论坛向我们发起技术支持提问帖。
- **云开发平台 (SOPHON NET)**：https://cloud.sophgo.com/developer/cloudSpace/index?lang=CN ，提供在线的开发软硬件环境。
- **开源仓库：**
  (1) 针对单模型或场景的综合示例 sophon-demo：https://github.com/sophgo/sophon-demo
  (2) BSP：https://gitee.com/sophon-ai/bsp-sdk
  (3) sophon-ffmpeg：https://github.com/sophgo/sophon_ffmpeg
  (4) sophon-stream：https://github.com/sophgo/sophon-stream

欢迎各位通过 github issues 向我们反馈您在使用过程中遇到的问题，并向我们提交 PR 共同参与开源仓库的建设。

3.3 获取 SDK

3.3.1 官网下载 SDK 及源码

算能官网目前提供最新版本的 BM1688/CV186AH SOPHONSDK 开发包以及源码下载链接，下载链接如下：

https://developer.sophgo.com/site/index/material/44/all.html

3.3.2 SDK 下载命令

获取 BM1688/CV186AH SOPHONSDK 开发包的下载命令如下：

```
pip3 install dfss --upgrade
python3 -m dfss --url=open@sophgo.com:/gemini-sdk/gemini_sdk_edge_v1.5.1_offical_release.tar.
↪gz
```

BM1688/CV186AH SOPHONSDK **目前仅支持 Linux 版本**，本文档后续内容均以 Linux 的操作为例。

> **注意：** **请使用 x86 架构的开发主机配合 tpu-mlir 与 docker 环境完成模型的编译量化**。
> 若遇到问题，请联系技术支持获取帮助。

3.3.3 SDK 源码下载命令

如果您需要获取 SDK 的源码，可以通过 repo 命令获取

```
repo init -u https://github.com/sophgo/manifest.git -m release/edge.xml
```

3.4 安装 SDK

目录

    * 交叉编译环境搭建
  - 环境配置-SoC
    * 开发环境配置
    * 运行环境配置

SOPHON BM1688/CV186AH 目前支持 SoC 工作模式，以下为 SoC 模式与 PCIe 模式之间的区别

BM168X	SoC 模式	PCIe 模式
独立运行	BM168X 作为独立主机	BM168X 作为 PCIe 加速卡
对外 IO 方式	千兆以太网	PCIe 接口
对应产品	微服务器/模组	PCIe 加速卡

对于 BM1688/CV186AH 的用户，推荐将开发环境与运行环境分离，其中开发环境是指用于模型转换或验证以及程序编译等开发过程的环境；运行环境是指在具备 SOPHON 设备的平台上实际使用设备进行算法应用部署的运行环境。

因此，您需要 **准备一台 x86 主机**作为开发环境，用于模型的编译量化以及程序的交叉编译，推荐使用 Ubuntu16.04/18.04/20.04 的 x86 主机，运行内存建议 12GB 以上

3.4.1 环境配置-Linux 开发环境

我们提供了 tpu-mlir 的 docker 环境供用户在 x86 主机上进行模型的编译量化，用户可以在 x86 主机上基于 tpu-mlir 和 libsophon 完成模型的编译量化与程序的交叉编译，部署时将编译好的程序拷贝至 SoC 平台（SE 微服务器/SM 模组）中执行。

使用 tar 命令对压缩包进行解压:

```
tar -zxvf gemini_sdk_edge_v1.5.1_offical_release.tar.gz
rm -rf gemini_sdk_edge_v1.5.1_offical_release.tar.gz
```

3.4.1.1 tpu-mlir 环境初始化

1. Docker 安装

若已安装 docker，请跳过本节。执行以下脚本安装 docker 并将当前用户加入 docker 组，获得 docker 执行权限。

```
# 安装docker
sudo apt-get install docker.io
sudo systemctl start docker
sudo systemctl enable docker
# docker命令免root权限执行
# 创建docker用户组，若已有docker组会报错，可忽略
sudo groupadd docker
```

(续下页)

(接上页)

```
# 将当前用户加入docker组
sudo usermod -aG docker $USER
# 重启docker服务
sudo service docker restart
# 切换当前会话到新group或重新登录重启会话
newgrp docker
```

2. 环境初始化

目前支持 **pip 命令**安装 tpu-mlir 库，并提供 docker 模型移植环境

注意，这里请使用 pip 命令安装，而不是 pip3 命令，否则可能会安装错误

(1) 创建 docker 容器并进入 docker

```
docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_dev:v3.2
```

(2) 安装 tpu_mlir

```
pip install tpu_mlir
```

(3) 安装 tpu_mlir 依赖

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

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

目前支持 6 种配置:onnx, torch, tensorflow, caffe, paddle, all。

如果您想要离线安装，可以从 https://github.com/sophgo/tpu-mlir/releases/ 或者 SDK 中的 tpu-mlir 目录中获取离线 whl 安装包。

3.4.1.2 交叉编译环境搭建

如果您希望使用 SOPHONSDK 搭建交叉编译环境，您需要用到 gcc-aarch64-linux-gnu 工具链，再将程序所依赖的头文件和库文件打包到 soc-sdk 目录中。

1. 首先安装工具链:

```
sudo apt-get install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
```

2. 解压 sophon-img 包里的 libsophon_soc_<x.y.z>_aarch64.tar.gz，将 lib 和 include 的所有内容拷贝到 soc-sdk 文件夹。

```
cd sophon-img
# 创建依赖文件的根目录
mkdir -p soc-sdk
# 解压sophon-img release包里的libsophon_soc_${x.y.z}_aarch64.tar.gz，其中x.y.z为版本号
tar -zxf libsophon_soc_<x.y.z>_aarch64.tar.gz
# 将相关的库目录和头文件目录拷贝到依赖文件根目录下
cp -rf libsophon_soc_<x.y.z>_aarch64/opt/sophon/libsophon-<x.y.z>/lib ${soc-sdk}
cp -rf libsophon_soc_<x.y.z>_aarch64/opt/sophon/libsophon-<x.y.z>/include ${soc-sdk}
```

3. 解压 sophon-media 包里的 sophon-media-soc_<x.y.z>_aarch64.tar.gz，将 sophon-media 下 lib 和 include 的所有内容拷贝到 soc-sdk 文件夹。

```
cd sophon-media_<date>_<hash>
# 解压sophon-media包里的sophon-media-soc_<x.y.z>_aarch64.tar.gz，其中x.y.z为版本号
tar -zxf sophon-media-soc_<x.y.z>_aarch64.tar.gz
# 将sophon-media目录下的库目录和头文件目录拷贝到依赖文件根目录下
cp -rf sophon-media-soc_<x.y.z>_aarch64/lib ${soc-sdk}
cp -rf sophon-media-soc_<x.y.z>_aarch64/include ${soc-sdk}
# 将ffmpeg和opencv的库目录和头文件目录拷贝到依赖文件根目录下
cp -rf sophon-media-soc_<x.y.z>_aarch64/opt/sophon/sophon-ffmpeg_<x.y.z>/lib ${soc-sdk}
cp -rf sophon-media-soc_<x.y.z>_aarch64/opt/sophon/sophon-ffmpeg_<x.y.z>/include ${soc-sdk}
cp -rf sophon-media-soc_<x.y.z>_aarch64/opt/sophon/sophon-opencv_<x.y.z>/lib ${soc-sdk}
cp -rf sophon-media-soc_<x.y.z>_aarch64/opt/sophon/sophon-opencv_<x.y.z>/include ${soc-sdk}
```

4. 若您需要使用第三方库，可以使用 qemu 在 x86 上构建虚拟环境安装，再将头文件和库文件拷贝到 soc-sdk 目录中，具体可参考 libsophon 使用手册（构建 qemu 虚拟环境）。

您可以执行以下命令，验证开发环境中的交叉编译工具链是否配置成功：

```
which aarch64-linux-gnu-g++
# 终端输出内容
# /usr/bin/aarch64-linux-gnu-g++
```

如果终端输出了 aarch64 编译的路径，则说明交叉编译工具链正确，开发环境是可以正常使用的。

若您需要使用 SAIL 模块，请参考 doc 目录下《sophon-sail_zh》中的安装说明。

3.4.2 环境配置-SoC

3.4.2.1 开发环境配置

对于 SoC 模式，**模型转换必须要在 x86 主机的 docker 开发容器中完成**；C/C++ 程序建议在 x86 主机上使用交叉编译工具链编译生成可执行文件后，再拷贝到 SoC 目标平台运行。

若您希望直接在 SoC 中进行 C/C++ 程序的编译，则 **需要先安装 sophon-soc-libsophon-dev_<x.y.z>_arm64.deb 工具包**，使用以下命令安装：

```
sudo dpkg -i sophon-soc-libsophon-dev_<x.y.z>_arm64.deb
```

再安装 sophon-media-soc-sophon-ffmpeg-dev_<x.y.z>_arm64.deb、sophon-media-soc-sophon-opencv-dev_<x.y.z>_arm64.deb 工具包，使用以下命令安装：

```
sudo dpkg -i sophon-media-soc-sophon-ffmpeg-dev_<x.y.z>_arm64.deb
sudo dpkg -i sophon-media-soc-sophon-opencv-dev_<x.y.z>_arm64.deb
```

注意:

1. 请先使用 bm_version 命令查看微服务器中的 SDK 版本与当前 SDK 版本是否一致，如果不一致，请参考下一小节对微服务器进行软件更新。
2. **SE 微服务器自带的操作系统并没有桌面系统，您需要使用 ssh 登录到微服务器终端内进行操作开发。**

3.4.2.2 运行环境配置

对于 SoC 平台，内部已经集成了相应的 libsophon、sophon-opencv 和 sophon-ffmpeg 运行库包，位于/opt/sophon/下。如果您需要使用我们提供 opencv-python，只需设置环境变量即可。

```
# 设置环境变量
export PYTHONPATH=$PYTHONPATH:/opt/sophon/sophon-opencv-latest/opencv-python
```

3.5 更新 SDK

目录

我们暂时通过 sftp 服务器发布新版本的 BM1688/CV186AH SOPHONSDK，通过以下命令获取最新版本的 SDK

```
pip3 install dfss --upgrade
python3 -m dfss --url=open@sophgo.com:/gemini-sdk/gemini_sdk_edge_v1.5.1_offical_release.tar.
↪gz
```

对于 SoC 用户，我们推荐使用 **SD 卡刷机的方式**实现。注意 **SD 卡刷机会重写整个 eMMC，使得 eMMC 的数据全部会丢失**，但是这种方式最为干净可靠，理论上只要您的 BM1688/CV186AH SoC 没有硬件损坏，都可以进行 SD 卡刷机。

以下为具体的操作步骤：

（1）SD 卡格式化为 **FAT32 格式**（如果 SD 卡上有多个分区，只能使用第一个分区），大小为 1GB 以上；

（2）解压 **sophon-img/sdcard.tgz** 到 SD 卡，并确保所有文件都解压到 SD 卡的 **根目录**；

（3）使用 type-c 连接串口线，安装 CP210x 驱动，使用串口连接工具，设置波特率为 115200；

（3）将设备断电，**确保指示灯熄灭后**，插入 SD 卡，设备重启上电，然后设备状态灯将变红，进入自动刷机流程；

（4）刷机通常耗时 3 分钟，您可以在 **串口或者指示灯处获取到刷机完成的信息**，结束后，拔掉 SD 卡并断电重启；

注意，**刷机后 Ubuntu 系统第一次启动时会进行文件系统初始化等关键动作，请勿随意断电**，待开机进入命令行后使用 sudo poweroff 命令关机

其他升级操作可以参考 doc 目录下的《算能边缘产品 BSP 开发参考手册》的“软件安装”章节。

3.6 SDK 更新记录

目录

2023.11.06 v1.1

初版发布，新增的功能如下：

- 支持 2DE,HDMI,PCIE,SATA,CAN,WIFI,USB3.0 驱动
- 支持 ion 内存管理
- 支持 opteed
- 支持 BM1688/CV186AH 支持双核单模型与双核双模型推理新增异步推理接口，支持用户指定某个核做推理
- 支持双核模型指令存储 tpu_model 支持查看模型核数
- 支持 BM1686/CV186AH firmware 现有算子支持 BM1686/CV186AH 支持 yolov3/yolov5 后处理算子

2023.12.06 v1.2

- 驱动新增 CAN 远程帧，CAN 扩展帧，CAN 扩展远程帧，CAN FD；
- tpu-mlir 优化多 batch 模型双核 TP 性能；
- tpu_perf 支持双核编译批量测试；
- 多媒体新增 sgbm、fgs、online、ive、dpu、ldc、dwa、blending 功能；
- 多媒体新增旋转、去畸变功能
- sophon-demo 新增 ppocr、deepsort 例程

2024.01.06 v1.3

- 增加 trng 驱动
- 多媒体新增 ldc genmesh、dwa 仿射、鱼眼、gemm、bgmodel 接口
- 多媒体新增 v4l2 的支持
- mlir 支持 pip 安装
- sophon-demo 新增例程适配 wenet, lprnet, yolov34, retinaface
- sophon-stream 新增车牌检测识别例程
- isp 新增 04a10、imx5852、pqtool 支持

2024.01.06 v1.3_0117

- 修复部分已知 bug

2024.02.06 v1.4

- 新增 WEB 设备管理页面，用于登录管理、状态查询、设备运维、日志管理
- 新增恢复出厂设备功能
- STAT 指示灯管理升级
- SSD 指示灯管理升级
- 多媒体新增 hist balance/quantify
- 多媒体 dwa 新增 dewarp_grid_info，gdc__grid_info，fisheye_grid_info
- tpu-runtime 支持单双核动态加载 bmodel
- TPU1688 支持单核 tpu-kenrel samples 动态加载
- sophon-demo 新增 YOLOv8、CenterNet、WeNet、YOLOX 例程
- sophon-stream 新增 PPOCR 例程
- isp 新增六路支持
- isp tool daemon 新增 HDMI 输出

2024.04.08 v1.5

- mlir 支持 pip 在线安装
- 修复动态加载引发的模型性能问题
- 新增 put_text/draw_line 接口
- 修复部分处理器型号显示错误问题
- 提升系统稳定性

2024.04.26 v1.5.1

- 修复微服务器有概率 reboot 失败

CHAPTER 4

快速入门

4.1 移植开发综述

4.1.1 算法移植流程

基于 SOPHON BM1688/CV186AH 进行产品开发会经历如下几个阶段：

1. **评测选型：**根据应用场景，确定使用的产品形态。
2. **模型迁移：**将原始深度学习框架下训练生成的模型转换为 BM1688/CV186AH 平台运行的 FP32/FP16/INT8/INT4 BModel 模型。
3. **算法移植：**基于 BM1688/CV186AH 硬件加速接口，对模型的前后处理及推理现有算法进行移植。
4. **程序移植：**移植任务管理、资源调度等算法引擎代码及逻辑处理、结果展示、数据推送等业务代码。
5. **测试调优：**网络性能与精度测试、压力测试，基于网络编译、量化工具、多卡多芯、任务流水线等方面的深度优化。
6. **部署联调：**将算法服务打包部署到 BM1688/CV186AH 硬件产品上，并在实际场景中与业务平台或集成平台进行功能联调。

4.1.2 典型视频深度学习分析任务

一个典型的深度学习视频分析任务通常包括：视频/图片解码、预处理、推理、后处理、视频/图片编码等环节。SOPHON 设备对各环节的硬件加速支持情况如下：

算法步骤	支持硬件加速	sophon-OpenCV	sophon-FFmpeg	Native 接口
视频/图片编解码	支持	Y	Y	BMCV（图片）/SAIL
输入预处理	支持	Y	N	BMCV/SAIL
模型推理	支持	N	N	BMRuntime/SAIL
输出后处理	部分支持	N	N	BMCV

- MLIR 模型支持情况和测试的模型案例，请参见 doc 目录下的《TPU-MLIR 快速入门指南》的附录七，已支持的算子章节的描述。
- 典型的图像分类、检测、分割算法实现可以参考 sophon-demo，关于使用多线程构建多模型推理任务可参考 sophon-stream。

4.2 重要概念

目录

4.2.1 工作模式

SOPHON BM168X 系列产品涵盖了从末端到边缘到中枢的多种产品形态，可以支持两种不同的工作模式，分别对应不同的产品形态，具体信息如下：

BM168X	SoC 模式	PCIe 模式
独立运行	作为主机独立运行	作为 PCIe 加速卡，安装在服务器上运行
对外 IO 方式	千兆以太网	PCIe 接口
对应产品	微服务器/模组	PCIe 加速卡

BM1688/CV186AH 目前仅支持 SoC 模式

4.2.2 硬件内存

内存是 BM168X 应用调试中经常会涉及的重要概念，特别地，有以下 3 个概念需要特别区分清楚：Global Memory、Host Memory、Device Memory。

对于 BM1688/CV186AH 的 SoC 工作模式，三种内存的解释如下

- **全局内存（Global Memory）**：指处理器的片外存储 DDR
- **设备内存（Device Memory）**：划分给 Tensor Computing Processor/VPSS 的内存
- **系统内存（Host Memory）**：划分给主控 Cortex A53 的内存

三者关系可以简单的理解为 **全局内存** = **设备内存** + **系统内存**.

内存同步问题是后续应用调试中经常会遇到的比较隐蔽的重要问题。我们在 sophon-opencv 和 sophon-ffmpeg 两个框架内都提供了内存同步操作的函数；而 BMCV API 只面向设备内存操作，因此不存在内存同步的问题，在调用 BMCV API 前，需要将输入数据在设备内存上准备好。

我们在 BMLib 中提供了接口，可以实现 Host Memory 和 Global Memory 之间、Global Memory 内部以及不同设备的 Global Memory 之间的数据搬运。

更多详细信息请参考《BMLib 开发参考手册》和《多媒体开发参考手册》。

4.2.3 BModel

BModel：是一种面向算能智能视觉深度学习处理器的 **神经网络模型文件格式**，其中包含目标网络的权重（weight）、指令流等等。

Stage：支持将同一个网络的不同 batch size 的模型 combine 为一个 BModel；同一个网络的不同 batch size 的输入对应着不同的 stage，推理时 BMRuntime 会根据输入 shape 的大小自动选择相应 stage 的模型。也支持将不同的网络 combine 为一个 BModel，通过网络名称来获取不同的网络。

动态编译和静态编译：支持模型的动态编译和静态编译，可在转换模型时通过参数设定。动态编译的 BModel，在 Runtime 时支持任意小于编译时设置的 shape 的输入 shape；静态编译的 BModel，在 Runtime 时只支持编译时所设置的 shape。

备注： 优先使用静态编译的模型：动态编译模型运行时需要 BM168X 内微控制器 ARM9 的参与，实时地根据实际输入 shape，动态生成智能视觉深度学习处理器运行指令。因此，动态编译的模型执行效率要比静态编译的模型低。若可以，应当优先使用静态编译的模型或支持多种输入 shape 的静态编译模型。

4.2.4 bm_image

```
typedef enum bm_image_format_ext_{
    FORMAT_YUV420P,
    FORMAT_YUV422P,
    FORMAT_YUV444P,
    FORMAT_NV12,
    FORMAT_NV21,
    FORMAT_NV16,
    FORMAT_NV61,
    FORMAT_RGB_PLANAR,
    FORMAT_BGR_PLANAR,
    FORMAT_RGB_PACKED,
    FORMAT_BGR_PACKED,
    PORMAT_RGBP_SEPARATE,
    PORMAT_BGRP_SEPARATE,
    FORMAT_GRAY,
    FORMAT_COMPRESSED
} bm_image_format_ext;

typedef enum bm_image_data_format_ext_{
    DATA_TYPE_EXT_FLOAT32,
    DATA_TYPE_EXT_1N_BYTE,
    DATA_TYPE_EXT_4N_BYTE,
    DATA_TYPE_EXT_1N_BYTE_SIGNED,
    DATA_TYPE_EXT_4N_BYTE_SIGNED,
}bm_image_data_format_ext;

// bm_image结构体定义如下
struct bm_image {
    int width;
    int height;
    bm_image_format_ext image_format;
    bm_data_format_ext data_type;
    bm_image_private* image_private;
};
```

BMCV API 均是围绕 bm_image 来进行的，一个 bm_image 对象对应于一张图片。

用户通过 bm_image_create 构建 bm_image 对象，attach 到 device memory；使用完需要调用 bm_image_destroy 销毁；手动 attach 到 device memory 的还需要手动释放。

不同 BMCV API 支持的 image_format 和 data_type 不同，注意格式要求，必要时需要进行格式转换。

更多详细信息请参考《算能边缘产品 BMCV 开发参考指南》。

另外，SAIL 库中将 bm_image 封装为 BMImage，相关信息请参考 doc 目录下的《sophon-sail_zh》。

4.3 快速跑通一个样例

目录

我们提供了模型转换及量化样例程序、多媒体相关样例程序、以及涵盖图像分类、目标检测、图像分割、文本检测识别等领域的单个模型算法移植样例程序，具体如下：

1. BM1688/CV186AH SOPHONSDK 中 doc 文件夹下的《TPU-MLIR 快速入门手册》提供了模型转换与量化样例；
2. BM1688/CV186AH SOPHONSDK 中 sophon-demo 提供了 CenterNet、LPRNet、PP-OCR、ResNet、RetinaFace、YOLOv3/v4、YOLOv5、YOLOX、YOLOv8 等单个模型的推理样例程序；
3. BM1688/CV186AH SOPHONSDK 中 sophon-stream 提供了多线程任务构建的样例程序，只需要编写配置文件就可以完成流水线的搭建；
4. BM1688/CV186AH SOPHONSDK 中 sophon-media 提供不同硬件架构下关于视频及图片编解码、bmcv 格式转换的多媒体相关样例程序，具体可以参考 doc 文件夹下的多媒体参考手册

下面以 YOLOv5 算法为例，介绍如何快速跑通一个样例；

（1）**将 sophon-sail、sophon-demo 压缩包复制到 SOPHON 边缘服务器并解压**

（2）**进入 sophon-sail 文件夹，安装 sophon-sail 的 python 库,whl 包位于 python_wheels/soc_BM1688 目录下，选择对应的版本进行安装**

```
pip3 install ./sophon_arm-3.7.0-py3-none-any.whl --force-reinstall
```

（3）**进入 sophon-demo 文件夹，下载官方模型以及数据集**

```
cd ./sample/YOLOv5/scripts
chmod +x /download.sh
./download.sh
```

（4）**执行 yolov5 的 python 检测代码**

```
cd ../python
# BM1688
python3 yolov5_bmcv.py --bmodel ../models/BM1688/yolov5s_v6.1_3output_int8_1b_2core.
↪bmodel --conf_thresh 0.45

# CV186AH
python3 yolov5_bmcv.py --bmodel ../models/CV186X/yolov5s_v6.1_3output_int8_1b.bmodel --
↪conf_thresh 0.45
```

测试结束后，会将预测的图片、json 文件保存在 results 目录下，同时会打印预测结果、推理时间等信息。

模型性能、精度、C++ 程序编译等更多信息请参考 YOLOv5/readme、YOLOv5/cpp/readme、YOLOv5/python/readme 等文件。

CHAPTER 5

网络模型迁移

5.1 MLIR-模型迁移流程

5.1.1 MLIR-迁移工具概述

TPU-MLIR 是算能的智能视觉深度学习处理器编译器工程，该工程提供了一套完整的工具链，其 **可以将不同框架下预训练的神经网络，转化为可以在算能智能视觉深度学习处理器上高效运算的模型文件 bmodel** 。

代码已经开源到 github: https://github.com/sophgo/tpu-mlir 。

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

算丰系列智能视觉深度学习处理器平台仅支持 BModel 模型加速，用户需要通过使用算丰 MLIR 工具链，可以把其他框架下训练好的模型转换为 BModel，使之能在算丰系列智能视觉深度学习处理器上运行。

MLIR 已直接支持绝大部分开源框架（Pytorch、ONNX、TFLite、Caffe 等）下的算子和模型，其他模型（TensorFlow、PaddlePaddle 等）需要转换为 ONNX 模型后，再进行转换，更多的网络层和模型也在持续支持中。

TPU-MLIR 的整体架构如下图所示:

对于 BM1688/CV186AH 平台来说，TPU-MLIR 支持 float32、float16、int8、以及 int4 模型

另外，TPU-MLIR 模型转换流程如下图:

迁移模型需要在指定的 docker 执行，主要分两步，一是通过 model_transform.py 将原始模型转换成 mlir 文件，二是通过 model_deploy.py 将 mlir 文件转换成 BModel。

其中，生成 FLOAT 模型时，model_deploy.py 工具支持输出 F32/F16/BF16 三种浮点数据类型。

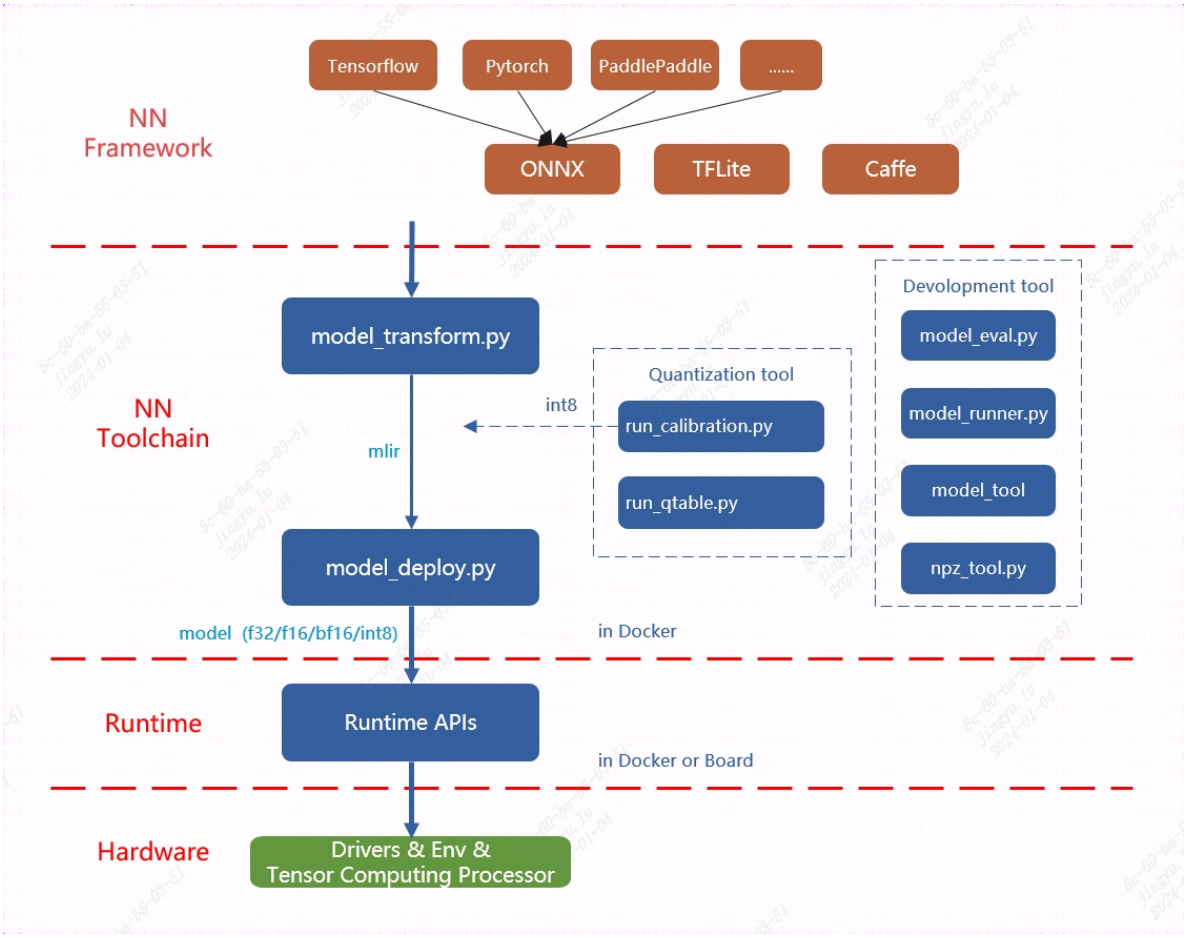


图 5.1: TPU-MLIR 的整体架构

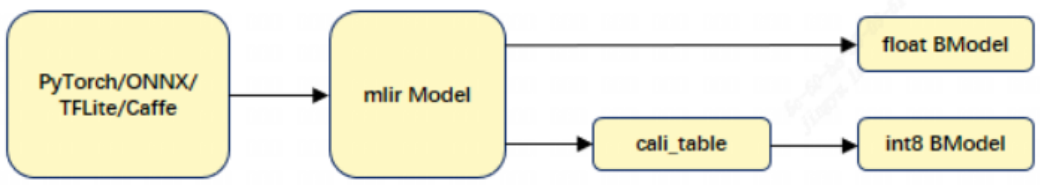


图 5.2: TPU-MLIR 模型转换流程

INT8/INT4 模型, 则需要准备量化数据集、调用 run_calibration.py 生成校准表, 然后将校准表传给 model_deploy.py。如果 INT8 模型不满足精度需要, 可以调用 run_qtable.py 生成量化表, 用来决定哪些层采用浮点计算，然后传给 model_deploy.py 生成混精度模型。具体请依次参考 5.1.3 INT8 模型生成章节。

具体移植以及量化步骤可参考本章第二、三小节，更详细的问题可参考 doc 目录下的《TPU-MLIR 快速入门指南》。

5.1.2 FLOAT 模型生成

算丰系列智能视觉深度学习处理器平台仅支持 BModel 模型加速, 用户需要首先进行模型迁移, 把其他框架下训练好的模型转换为 BModel 才能在算丰系列智能视觉深度学习处理器上运行。

TPU-MLIR 工具包

本文以 yolov5s.onnx 为例，介绍 TPU-MLIR 如何转换模型为 FP32 格式的 BModel 并部署。MLIR 环境的配置，可以参考tpu-mlir 环境初始化 ，**其他模型迁移请参考 doc 目录下的《TPU-MLIR 快速入门指南》**。

yolov5 的模型来自 yolov5 的官网 ，可点击本链接进行下载。

5.1.2.1 加载 tpu-mlir

参考tpu-mlir 环境初始化 , 进入 x86 主机的 Docker 环境，并通过 pip 命令安装 tpu-mlir

5.1.2.2 准备工作目录

建立 model_yolov5s 目录, 建议与 tpu-mlir 同级目录；并把模型文件和图片文件都放入 model_yolov5s 目录中。

操作如下:

```
$ mkdir yolov5s_onnx && cd yolov5s_onnx
$ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

这里的 $TPUC_ROOT 是环境变量, 对应 tpu-mlir_xxxx 目录。

5.1.2.3 ONNX 转 MLIR

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

$$y = (x - mean) \times scale$$

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

模型转换命令如下:

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

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

表 5.1: model_transform 参数功能

参数名	必选？	说明
model_name	是	指定模型名称
model_def	是	指定模型定义文件, 比如 ‘.onnx‘或 ‘.tflite‘或 ‘.prototxt‘文件
input_shapes	否	指定输入的 shape, 例如 [[1,3,640,640]]; 二维数组, 可以支持多输入情况
input_types	否	指定输入的类型, 例如 int32; 多输入用, 隔开; 不指定情况下默认处理为 float32
resize_dims	否	原始图片需要 resize 之后的尺寸; 如果不指定, 则 resize 成模型的输入尺寸
keep_aspect_ratio	否	在 Resize 时是否保持长宽比, 默认为 false; 设置时会对不足部分补 0
mean	否	图像每个通道的均值, 默认为 0.0,0.0,0.0
scale	否	图片每个通道的比值, 默认为 1.0,1.0,1.0
pixel_format	否	图片类型, 可以是 rgb、bgr、gray、rgbd 四种情况, 默认为 bgr
channel_format	否	通道类型, 对于图片输入可以是 nhwc 或 nchw, 非图片输入则为 none, 默认为 nchw
output_names	否	指定输出的名称, 如果不指定, 则用模型的输出; 指定后用该指定名称做输出
test_input	否	指定输入文件用于验证, 可以是图片或 npy 或 npz; 可以不指定, 则不会正确性验证
test_result	否	指定验证后的输出文件
excepts	否	指定需要排除验证的网络层的名称, 多个用, 隔开
mlir	是	指定输出的 mlir 文件名称和路径

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

将 mlir 文件转换成 f32 的 bmodel, 操作方法如下:

```
#bm1688
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize F32 \
    --chip bm1688 \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.99,0.99 \
    --num_core 2 \
    --model yolov5s_1688_f32_2core.bmodel

#cv186ah
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize F32 \
    --chip cv186x \
```

(续下页)

(接上页)

```
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.99,0.99 \
    --model yolov5s_cv186x_f32.bmodel
```

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

表 5.2: model_deploy 参数功能

参数名	必选？	说明
mlir	是	指定 mlir 文件
quantize	是	指定默认量化类型, 支持 F32/F16/BF16/INT8
chip	是	指定模型将要用到的平台, 支持 bm1684x/bm1684/cv183x/cv182x/cv181x/cv180x
calibration_table	否	指定校准表路径, 当存在 INT8 量化的时候需要校准表
tolerance	否	表示 MLIR 量化后的结果与 MLIR fp32 推理结果相似度的误差容忍度
test_input	否	指定输入文件用于验证, 可以是图片或 npy 或 npz; 可以不指定, 则不会正确性验证
test_reference	否	用于验证模型正确性的参考数据 (使用 npz 格式)。其为各算子的计算结果
compare_all	否	验证正确性时是否比较所有中间结果, 默认不比较中间结果
excepts	否	指定需要排除验证的网络层的名称, 多个用, 隔开
num_core	否	仅支持 BM1688, 选择并行计算的核心数量, 默认设置为 1 个计算核心
model	是	指定输出的 model 文件名称和路径

仅 BM1688 支持 num_core 选项，用于指定生成适用于 BM1688 双核高性能模型。

编译完成后，会生成名为 yolov5s_1684x_f32_2core.bmodel 或者 yolov5s_cv186x_f32.bmodel 的文件。

如果想要生成 float16 的模型，则 **仅需要在 model_deploy.py 的时候指定 quantize 为 F16** ，其余步骤相同。

5.1.3 INT 模型生成

目录

- ONNX 转 MLIR
- MLIR 转 INT8 模型
  - 生成校准表
  - 编译为 INT8 对称量化模型
- 效果对比
- 模型性能测试

TPU-MLIR 的整体架构如下图所示:

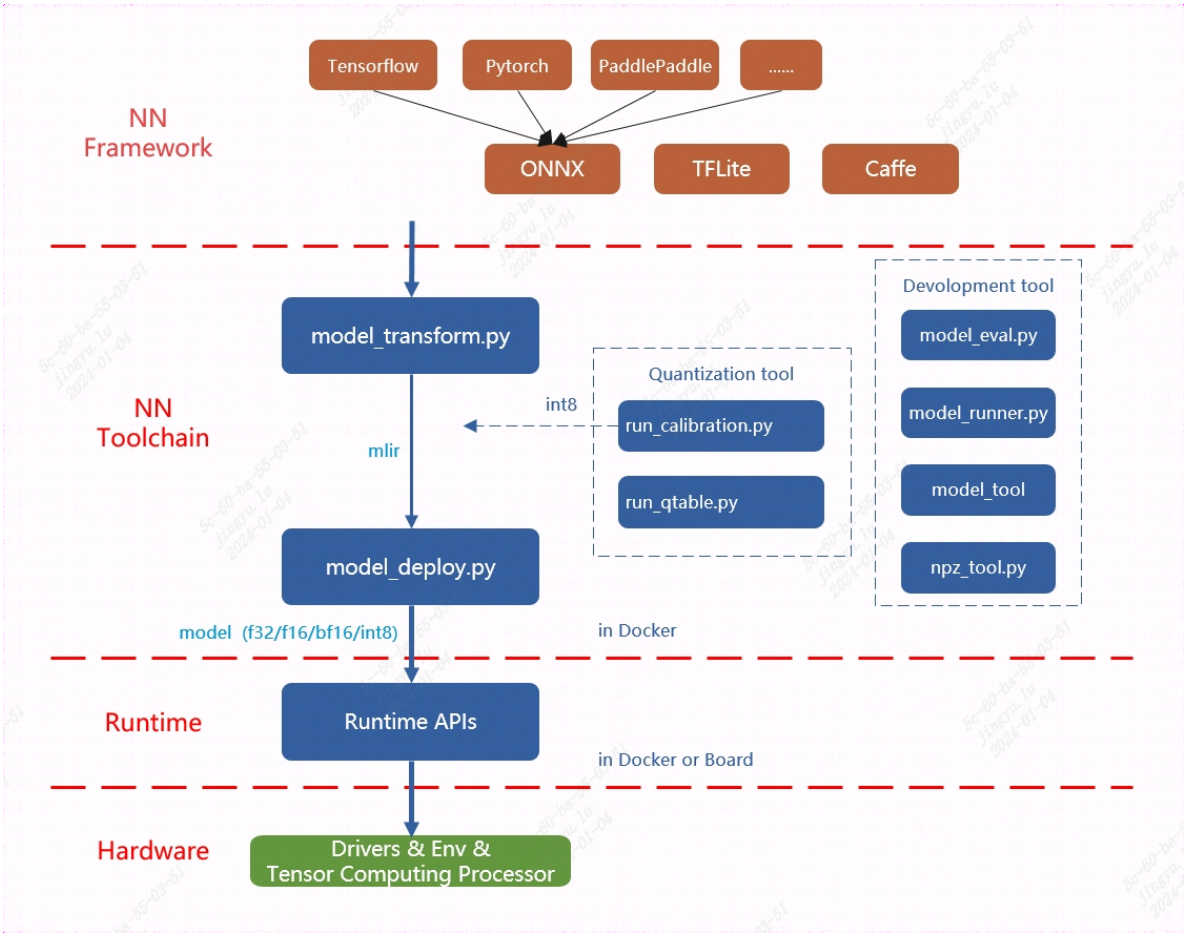


图 5.3: TPU-MLIR 的整体架构

相比于迁移 float 模型，如果要转 INT8 模型，则需要调用 run_calibration.py 生成校准表，然后传给 model_deploy.py。

具体更多细节请参考 doc 目录下的《TPU-MLIR 快速入门指南》

本文以 yolov5s.onnx 为例，介绍 TPU-MLIR 如何转换模型为 INT8 格式的 BModel 并部署。开发环境的配置，可以参考tpu-mlir 环境初始化，**其他模型迁移请参考 doc 目录下的《TPU-MLIR 快速入门指南》**。

yolov5 的模型来自 yolov5 的官网，可点击本链接进行下载。

5.1.3.1 加载 tpu-mlir

参考tpu-mlir 环境初始化，进入 x86 主机的 Docker 环境，并通过 pip 命令安装 tpu-mlir

5.1.3.2 准备工作目录

建立 model_yolov5s 目录，建议与 tpu-mlir 同级目录；并把模型文件和图片文件都放入 model_yolov5s 目录中。

```
$ mkdir yolov5s_onnx && cd yolov5s_onnx
$ wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.onnx
$ cp -rf $TPUC_ROOT/regression/dataset/COCO2017 .
$ cp -rf $TPUC_ROOT/regression/image .
$ mkdir workspace && cd workspace
```

这里的 $TPUC_ROOT 是环境变量，对应 tpu-mlir_xxxx 目录。

5.1.3.3 ONNX 转 MLIR

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

$$y = (x - mean) \times scale$$

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

模型转换命令如下：

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

model_transform.py 的相关参数已在上一小节说过，这里不再赘述。

对于 INT8 模型的量化与迁移，需要确保前处理参数准确无误，否则生成的 INT8 可能会有严重的精度损失。

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

5.1.3.4 MLIR 转 INT8 模型

5.1.3.4.1 生成校准表

转 INT8 模型前需要跑 calibration, 得到校准表; 输入数据的数量根据情况准备 100~1000 张左右，需要 **保证量化数据集与测试数据集的数据分布保持一致**。

然后用校准表, 生成对称或非对称 bmodel。如果对称量化精度符合需求, **一般不建议用非对称量化**，因为非对称量化的模型性能会略差于对称模型。

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

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

运行完成后会生成名为 yolov5s_cali_table 的文件, 该文件用于后续编译 INT8 模型的量化表文件。

5.1.3.4.2 编译为 INT8 对称量化模型

转成 INT8 对称量化模型, 执行如下命令:

```
# bm1688
$ model_deploy.py \
    --mlir yolov5s.mlir \
    --quantize INT8 \
    --calibration_table yolov5s_cali_table \
    --chip bm1688 \
    --test_input yolov5s_in_f32.npz \
    --test_reference yolov5s_top_outputs.npz \
    --tolerance 0.85,0.45 \
    --num_core 2 \
    --model yolov5s_1688_int8_sym_2core.bmodel

# cv186ah
$ model_deploy.py \
   --mlir yolov5s.mlir \
   --quantize INT8 \
   --calibration_table yolov5s_cali_table \
   --chip cv186x \
   --test_input yolov5s_in_f32.npz \
   --test_reference yolov5s_top_outputs.npz \
   --tolerance 0.85,0.45 \
   --num_core 2 \
   --model yolov5s_cv186ah_int8_sym.bmodel
```

编译完成后，会生成名为 yolov5s_1688_int8_sym_2core.bmodel 或者 yolov5s_cv186ah_int8_sym.bmodel 的文件。

5.1.3.5 效果对比

在本发布包中有用 python 写好的 yolov5 用例, 源码路径 $TPUC_ROOT/python/samples/detect_yolov5.py , 使用示例代码分别来验证 onnx/f16/int8 的执行结果。

onnx 模型的执行方式如下, 得到 dog_onnx.jpg :

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

对于 bmodel 模型文件，由于开发环境没有相关设备，因此将会使用 x86 的主机 cpu 模拟运行 bmodel 模型，仅用于准确性验证。

f16 bmodel 的执行方式如下, 得到 dog_f16.jpg :

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_1688_f16.bmodel \
    --output dog_f16.jpg
```

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

```
$ detect_yolov5.py \
    --input ../image/dog.jpg \
    --model yolov5s_1688_int8_sym_2core.bmodel \
    --output dog_int8_sym.jpg
```

对比结果如下:

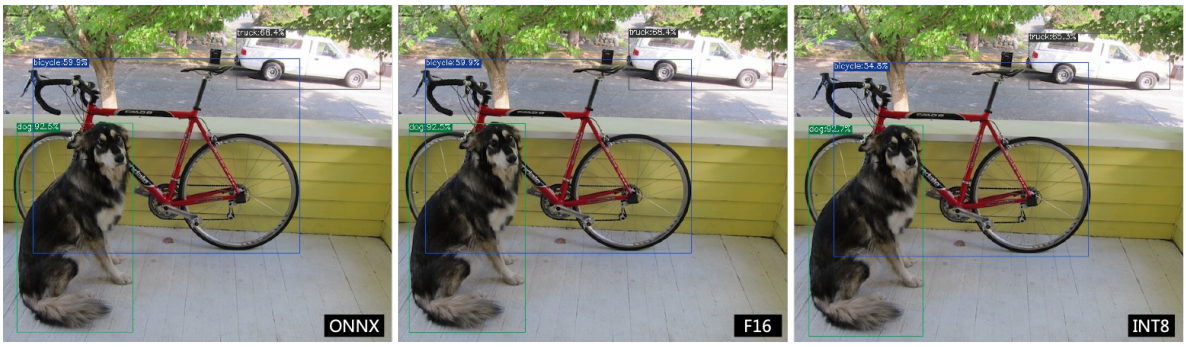


图 5.4: TPU-MLIR 对 YOLOv5s 编译效果对比

由于运行环境不同, 最终的效果和精度与yolov5s_result 会有些差异。

5.1.3.6 模型性能测试

以下操作需要在边缘服务器内执行，默认的出厂环境或者刷机完成后，可以使用 bmrt_test 来测试编译出的 bmodel 的正确性及性能。

可以根据 “bmrt_test “ 输出的性能结果，来估算模型最大的 fps，来选择合适的模型。

```
# 下面测试上面编译出的bmodel
# --bmodel参数后面接bmodel文件，

$ cd $TPUC_ROOT/../model_yolov5s/workspace
# BM1688
$ bmrt_test --bmodel yolov5s_1688_f16.bmodel
$ bmrt_test --bmodel yolov5s_1688_int8_sym_2core.bmodel
```

以最后一个命令输出为例 (此处对日志做了部分截断处理):

```
[BMRT][load_bmodel:1501] INFO:pre net num: 0, load net num: 1
[BMRT][show_net_info:1810] INFO: ############################
[BMRT][show_net_info:1814] INFO: NetName: yolov5s_v6.1_3output, Index=0
[BMRT][show_net_info:1817] INFO: ---- stage 0 ----
[BMRT][show_net_info:1827] INFO:   Input 0) 'images' shape=[ 1 3 640 640 ] dtype=FLOAT32 scale=1 zero_point=0 device_id=0
[BMRT][show_net_info:1838] INFO:   Output 0) 'output_Transpose_f32' shape=[ 1 3 80 80 85 ] dtype=FLOAT32 scale=1 zero_point=0 device_id=0
[BMRT][show_net_info:1838] INFO:   Output 1) '365_Transpose_f32' shape=[ 1 3 40 40 85 ] dtype=FLOAT32 scale=1 zero_point=0 device_id=0
[BMRT][show_net_info:1838] INFO:   Output 2) '385_Transpose_f32' shape=[ 1 3 20 20 85 ] dtype=FLOAT32 scale=1 zero_point=0 device_id=0
[BMRT][show_net_info:1841] INFO: ############################
[BMRT][bmrt_test:844] INFO:==> running network #0, name: yolov5s_v6.1_3output, loop: 0
[BMRT][bmrt_test:899] INFO:reading input #0, bytesize=4915200
[BMRT][print_array:755] INFO:  --> input_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1228800
[BMRT][bmrt_test:1030] INFO:reading output #0, bytesize=6528000
[BMRT][print_array:755] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=1632000
[BMRT][bmrt_test:1030] INFO:reading output #1, bytesize=1632000
[BMRT][print_array:755] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=408000
[BMRT][bmrt_test:1030] INFO:reading output #2, bytesize=408000
[BMRT][print_array:755] INFO:  --> output ref_data: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... > len=102000
[BMRT][bmrt_test:1065] INFO:net[yolov5s_v6.1_3output] stage[0], launch total time is 7083 us (npu 6387 us [6387 us, 2391 us], cpu 696 us), (issue 3000 us, sync 4136 us)
[BMRT][bmrt_test:1068] INFO:+++ The network[yolov5s_v6.1_3output] stage[0] output_data +++
[BMRT][print_array:755] INFO:output data #0 shape: [1 3 80 80 85 ] < -0.145839  ...
[BMRT][print_array:755] INFO:output data #1 shape: [1 3 40 40 85 ] < -0.726845  ...
[BMRT][print_array:755] INFO:output data #2 shape: [1 3 20 20 85 ] < 0.563809  ...
[BMRT][bmrt_test:1129] INFO:load input time(s): 0.018420
[BMRT][bmrt_test:1130] INFO:calculate  time(s): 0.007136
```

(续下页)

(接上页)

```
26 [BMRT][bmrt_test:1131] INFO:get output time(s): 0.020361
27 [BMRT][bmrt_test:1132] INFO:compare    time(s): 0.021095
```

从上面输出可以看到以下信息:

1. 05-08 行是 bmodel 的网络输入输出信息
2. 19 行是在智能视觉深度学习处理器上运行的时间, 其中智能视觉深度学习处理器用时 7136us, central processing unit 用时 696us。这里 central processing unit 用时主要是指在 HOST 端调用等待时间
3. 24 行是加载数据到 NPU 的 DDR 的时间
4. 25 行相当于 12 行的总时间
5. 26 行是输出数据取回时间

CHAPTER 6

算法移植

6.1 算法移植概述

目录

对于基于深度学习的视频/图片分析任务来说，通常都包括如下几个步骤：

1. 视频/图片解码
2. 输入预处理
3. 模型推理
4. 输出后处理
5. 视频/图片编码

实际任务中，算法往往还会包含多个不同神经网络模型，因此，步骤 2-4 会根据需要反复执行多次。

6.1.1 硬件加速支持情况

实践证明，单纯针对神经网络运算进行加速，已经无法满足真实场景的需求。为了提高算法运行效率，BM1688、CV186AH 在硬件设计中，还集成了编解码、图像处理等操作的硬件加速模块。

用户通过 SOPHONSDK 中提供的相应软件接口库，可以对如上几个步骤进行针对性地加速，从而便捷地开发出高效的算法和应用。

为了满足客户对不同风格接口使用的偏好，我们还对硬件加速接口库进行了多次封装，用户可以自行选取合适的接口库进行开发，具体情况总结如下：

任务流程	是否支持硬件加速	SAIL 高级接口库	OPENCV 接口库	FFMPEG 接口库	Native 接口库
视频/图片解码	支持	sail::Decoder	Y	Y	BMCV(图片)
输入预处理	支持	sail::Bmcv	Y	N	BMCV
模型推理	支持	sail::Engine	N	N	BMruntime
输出后处理	部分支持	sail::Bmcv	N	N	BMCV
视频/图片编码	支持	sail::Bmcv	Y	Y	BMCV(图片)

值得一提的是，为了提高算法效率以及硬件特性的要求，用户在调用硬件加速接口的时候需要注意以下几个方面，后续的文档会通过实例来进行具体阐述：

1. **内存零 copy**
2. **申请物理连续内存**
3. **将多个预处理步骤进行合并**

6.1.2 C/C++/Python 三种编程接口

目前支持 C/C++/Python 三种编程接口：

BMRuntime 模块支持 C/C++ 接口编程，BMCV、BMLib 支持 C++ 接口编程；Python/C++ 编程接口基于 SAIL 库实现。

SAIL（SOPHON Artificial Intelligent Library），是 **对 SOPHONSDK 中的 BMRuntime、BMCV、BMLib、sophon-media 库的高级封装**，将 SOPHONSDK 中原有的“加载 BModel 并驱动智能视觉深度学习处理器推理”、“驱动视频处理子系统 VPSS 做图像处理以及编解码”等功能抽象成更为简单的 C++ 接口对外提供；并且使用 pybind11 再次封装，提供了简洁易用的 Python 接口。目前，SAIL 模块中所有的类、枚举、函数都在“sail”命名空间下。

关于 SAIL C++/Python 接口详细内容请阅读 doc 目录下的《sophon-sail_zh》。

6.2 C/C++ 编程详解

目录

这个章节将会选取 sophon-demo 中的 YOLOV5 检测算法作为示例，说明各个步骤的接口调用和注意事项，样例代码路径：sophon-demo/sample/YOLOV5。

这个示例程序采用了 **OPENCV 解码** +**BMCV 图片预处理**的组合进行开发，您也可以根据您的需要采用不同的接口开发

我们按照算法的执行先后顺序展开介绍：

1. **加载 bmodel 模型**
2. **预处理**
3. **推理**
4. **注意事项**

6.2.1 加载 bmodel

```
...

BMNNContext(BMNNHandlePtr handle, const char* bmodel_file):m_handlePtr(handle){

    bm_handle_t hdev = m_handlePtr->handle();

    // init bmruntime contxt
    m_bmrt = bmrt_create(hdev);
```

(续下页)

(接上页)

```
    if (NULL == m_bmrt) {
    std::cout << "bmrt_create() failed!" << std::endl;
    exit(-1);
    }

    // load bmodel from file
    if (!bmrt_load_bmodel(m_bmrt, bmodel_file)) {
    std::cout << "load bmodel(" << bmodel_file << ") failed" << std::endl;
    }

    load_network_names();

}

...

void load_network_names() {

    const char **names;
    int num;

    // get network info
    num = bmrt_get_network_number(m_bmrt);
    bmrt_get_network_names(m_bmrt, &names);

    for(int i=0;i < num; ++i) {
    m_network_names.push_back(names[i]);
    }

    free(names);
}

...

BMNNNetwork(void *bmrt, const std::string& name):m_bmrt(bmrt) {
    m_handle = static_cast<bm_handle_t>(bmrt_get_bm_handle(bmrt));

    // get model info by model name
    m_netinfo = bmrt_get_network_info(bmrt, name.c_str());

    m_max_batch = -1;
    std::vector<int> batches;
    for(int i=0; i<m_netinfo->stage_num; i++){
        batches.push_back(m_netinfo->stages[i].input_shapes[0].dims[0]);
        if(m_max_batch<batches.back()){
            m_max_batch = batches.back();
        }
    }
    m_batches.insert(batches.begin(), batches.end());
    m_inputTensors = new bm_tensor_t[m_netinfo->input_num];
    m_outputTensors = new bm_tensor_t[m_netinfo->output_num];
```

(续下页)

(接上页)

```
    for(int i = 0; i < m_netinfo->input_num; ++i) {

        // get data type
        m_inputTensors[i].dtype = m_netinfo->input_dtypes[i];
        m_inputTensors[i].shape = m_netinfo->stages[0].input_shapes[i];
        m_inputTensors[i].st_mode = BM_STORE_1N;
        m_inputTensors[i].device_mem = bm_mem_null();
    }

...

}

...
```

这个几个函数的用法比较简单和固定，用户可以参考 doc 目录下的《算能边缘产品 BMRUNTIME 开发参考手册》了解更详细的信息。

唯一需要强调的是 name 字符串变量的用法：在推理代码中，模型的唯一标识就是他的 name 字符串，这个 name 需要在 compile 阶段就进行指定，算法程序也需要基于这个 name 开发；例如，在调用 inference 接口时，需要使用模型的 name 作为入参，让 runtime 作为索引去查询对应的模型，错误的 name 会造成 inference 失败。

6.2.2 预处理

6.2.2.1 预处理初始化

预处理初始化时，需要提前创建适当的 bm_image 对象保存中间结果，这样可以节省反复内存申请释放造成的开销，提高算法效率，具体代码如下：

```
...

int aligned_net_w = FFALIGN(m_net_w, 64);
int strides[3] = {aligned_net_w, aligned_net_w, aligned_net_w};
for(int i=0; i<max_batch; i++){

    // init bm images for storing results
    auto ret= bm_image_create(m_bmContext->handle(), m_net_h, m_net_w,
        FORMAT_RGB_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &m_resized_imgs[i], strides);
    assert(BM_SUCCESS == ret);
}
bm_image_alloc_contiguous_mem (max_batch, m_resized_imgs.data());

// bm images for storing inference inputs
bm_image_data_format_ext img_dtype = DATA_TYPE_EXT_FLOAT32;   //FP32

```

(续下页)

(接上页)

```

if (tensor->get_dtype() == BM_INT8) {   // INT8
    img_dtype = DATA_TYPE_EXT_1N_BYTE_SIGNED;
}

auto ret = bm_image_create_batch(m_bmContext->handle(), m_net_h, m_net_w,
    FORMAT_RGB_PLANAR,
    img_dtype,
    m_converto_imgs.data(), max_batch);
assert(BM_SUCCESS == ret);

...
```

不同于 bm_image_create() 函数只创建一个 bm_image 对象，bm_image_create_batch() 会根据最后一个参数 batch，创建一组 bm_image 对象，而且这组对象所使用的 data 域是物理连续的。使用物理连续的内存是硬件加速器的特殊需求，在析构函数，可以使用 bm_image_destroy_batch() 对内存进行释放。

接下来是输入的处理过程，这个示例算法同时支持图片和视频作为输入，在 main.cpp 的 main() 函数中，我们以视频为例，详细的写法如下：

6.2.2.2 打开视频流

```
...

// open stream
cv::VideoCapture cap(input_url, cv::CAP_ANY, dev_id);
if (!cap.isOpened()) {
  std::cout << "open stream " << input_url << " failed!" << std::endl;
  exit(1);
}

// get resolution
int w = int(cap.get(cv::CAP_PROP_FRAME_WIDTH));
int h = int(cap.get(cv::CAP_PROP_FRAME_HEIGHT));
std::cout << "resolution of input stream: " << h << "," << w << std::endl;

...
```

上面这段代码和标准的 opencv 处理视频流程几乎相同。

6.2.2.3 解码视频帧

```
...

// get one mat
cv::Mat img;
if (!cap.read(img)) { //check
    std::cout << "Read frame failed or end of file!" << std::endl;
    exit(1);
}

std::vector<cv::Mat> images;
images.push_back(img);

...
```

6.2.2.4 Mat 转换 bm_image

由于 BMCV 预处理接口和网络推理接口都需要使用 bm_image 对象作为输入，因此解码后的视频帧需要转换到 bm_image 对象。推理完成之后，再使用 bm_image_destroy() 接口进行释放。需要注意的是，这个转换过程没有发生内存拷贝。

```
...

// mat -> bm_image
CV_Assert(0 == cv::bmcv::toBMI((cv::Mat&)images[i], &image1, true));

...

//destroy
bm_image_destroy(image1);

...
```

6.2.2.5 预处理

bmcv_image_vpp_convert_padding() 函数使用 VPP 硬件资源，是预处理过程加速的关键，需要配置参数 padding_attr。bmcv_image_convert_to() 函数用于进行线性变换，需要配置参数 converto_attr。

```
...

// set padding_attr
bmcv_padding_atrr_t padding_attr;
memset(&padding_attr, 0, sizeof(padding_attr));
padding_attr.dst_crop_sty = 0;
padding_attr.dst_crop_stx = 0;
padding_attr.padding_b = 114;
```

(续下页)

(接上页)

```
padding_attr.padding_g = 114;
padding_attr.padding_r = 114;
padding_attr.if_memset = 1;
if (isAlignWidth) {
  padding_attr.dst_crop_h = images[i].rows*ratio;
  padding_attr.dst_crop_w = m_net_w;

  int ty1 = (int)((m_net_h - padding_attr.dst_crop_h) / 2);
  padding_attr.dst_crop_sty = ty1;
  padding_attr.dst_crop_stx = 0;
}else{
  padding_attr.dst_crop_h = m_net_h;
  padding_attr.dst_crop_w = images[i].cols*ratio;

  int tx1 = (int)((m_net_w - padding_attr.dst_crop_w) / 2);
  padding_attr.dst_crop_sty = 0;
  padding_attr.dst_crop_stx = tx1;
}

// do not crop
bmcv_rect_t crop_rect{0, 0, image1.width, image1.height};

auto ret = bmcv_image_vpp_convert_padding(m_bmContext->handle(), 1, image_aligned, &m_
↪resized_imgs[i],
    &padding_attr, &crop_rect);

...

// set converto_attr
float input_scale = input_tensor->get_scale();
input_scale = input_scale* (float)1.0/255;
bmcv_convert_to_attr converto_attr;
converto_attr.alpha_0 = input_scale;
converto_attr.beta_0 = 0;
converto_attr.alpha_1 = input_scale;
converto_attr.beta_1 = 0;
converto_attr.alpha_2 = input_scale;
converto_attr.beta_2 = 0;

// do converto
ret = bmcv_image_convert_to(m_bmContext->handle(), image_n, converto_attr, m_resized_
↪imgs.data(), m_converto_imgs.data());

// attach to tensor
if(image_n != max_batch) image_n = m_bmNetwork->get_nearest_batch(image_n);
bm_device_mem_t input_dev_mem;
bm_image_get_contiguous_device_mem(image_n, m_converto_imgs.data(), &input_dev_mem);
input_tensor->set_device_mem(&input_dev_mem);
input_tensor->set_shape_by_dim(0, image_n);  // set real batch number

...
```

6.2.3 推理

预处理过程的 output 是推理过程的 input，当推理过程的 input 数据准备好后，就可以进行推理。

```
...

ret = m_bmNetwork->forward();
...
```

6.2.4 后处理

后处理的过程因模型而异，而且大部分是中央处理器执行的代码，就不再这里赘述。需要注意的是我们在 BMCV 中也提供了一些可以用于加速的接口，如 bmcv_sort、bmcv_nms 等，对于其他需要使用硬件加速的情况，可根据需要使用 TPUKernel 进行开发。

以上就是 YOLOV5 示例的简单描述，关于涉及到的接口更加详细的描述，请查看相应模块文档。

6.2.5 算法开发注意事项汇总

根据上面的讨论，我们把一些注意事项汇总如下：

- 视频解码需要注意：

备注:

1. 我们支持使用 YUV 格式作为缓存原始帧的格式，解码后可通过 cap.set() 接口设置 YUV 格式。

- 预处理过程需要注意：

备注:

1. 预处理的操作对象是 bm_image，bm_image 对象可以类比 Mat 对象。
2. 预处理流程中 scale 缩放是针对 int8 模型。在推理数据输入前需要乘 scale 系数。scale 系数是在量化的过程中产生。
3. 为多个 bm_image 对象申请连续物理内存：bm_image_create_batch()。
4. resize 默认双线性插值算法，具体参考 bmcv 接口说明。

- 推理过程需要注意：

备注:

1. 推理过程在 device memory 上进行，所以推理前输入数据必须已经存储在 input tensors 的 device mem 里面，推理结束后的结果数据也是保存在 output tensors 的 device mem 里面。

6.3 Python 编程详解

目录

BM1688/CV186AH SOPHONSDK 通过 SAIL 库向用户提供 Python 编程接口。

这个章节将会选取 YOLOV5 检测算法作为示例，来介绍 python 接口编程, 样例代码路径位于 sophon-demo/sample/YOLOV5

关于 sail 接口的详细信息，请阅读 doc 目录下的《sophon-sail_zh》。

本章主要介绍以下三点内容：

- **加载模型**
- **预处理**
- **推理**

6.3.1 加载模型

```
import sophon.sail as sail

...

engine = sail.Engine(model_path, device_id, io_mode)

...
```

6.3.2 预处理

```python
class PreProcess:
    def __init__(self, width, height, batch_size, img_dtype, input_scale=None):

        self.std = np.array([255., 255., 255.], dtype=np.float32)
        self.batch_size = batch_size
        self.input_scale = float(1.0) if input_scale is None else input_scale
        self.img_dtype = img_dtype

        self.width = width
        self.height = height
        self.use_resize_padding = True
        self.use_vpp = False
        ...

    def resize(self, img, handle, bmcv):

        if self.use_resize_padding:
            img_w = img.width()
            img_h = img.height()
            r_w = self.width / img_w
            r_h = self.height / img_h

            if r_h > r_w:
                tw = self.width
                th = int(r_w * img_h)
                tx1 = tx2 = 0
                ty1 = int((self.height - th) / 2)
                ty2 = self.height - th - ty1

            else:
                tw = int(r_h * img_w)
                th = self.height
                tx1 = int((self.width - tw) / 2)
                tx2 = self.width - tw - tx1
                ty1 = ty2 = 0

            ratio = (min(r_w, r_h), min(r_w, r_h))
            txy = (tx1, ty1)
            attr = sail.PaddingAtrr()
            attr.set_stx(tx1)
            attr.set_sty(ty1)
            attr.set_w(tw)
            attr.set_h(th)
            attr.set_r(114)
            attr.set_g(114)
            attr.set_b(114)

            tmp_planar_img = sail.BMImage(handle, img.height(), img.width(),
                                          sail.Format.FORMAT_RGB_PLANAR, sail.DATA_TYPE_EXT_
```

(续下页)

(接上页)

```
↪1N_BYTE)
            bmcv.convert_format(img, tmp_planar_img)
            preprocess_fn = bmcv.vpp_crop_and_resize_padding if self.use_vpp else bmcv.crop_
↪and_resize_padding
            resized_img_rgb = preprocess_fn(tmp_planar_img,
                                            0, 0, img.width(), img.height(),
                                            self.width, self.height, attr)
        else:
            r_w = self.width / img.width()
            r_h = self.height / img.height()
            ratio = (r_w, r_h)
            txy = (0, 0)
            tmp_planar_img = sail.BMImage(handle, img.height(), img.width(),
                                          sail.Format.FORMAT_RGB_PLANAR, sail.DATA_TYPE_EXT_
↪1N_BYTE)
            bmcv.convert_format(img, tmp_planar_img)
            preprocess_fn = bmcv.vpp_resize if self.use_vpp else bmcv.resize
            resized_img_rgb = preprocess_fn(tmp_planar_img, self.width, self.height)
        return resized_img_rgb, ratio, txy

    ...

    def norm_batch(self, resized_images, handle, bmcv):

        bm_array = eval('sail.BMImageArray{}D'.format(self.batch_size))

        preprocessed_imgs = bm_array(handle,
                                     self.height,
                                     self.width,
                                     sail.FORMAT_RGB_PLANAR,
                                     self.img_dtype)

        a = 1 / self.std
        b = (0, 0, 0)
        alpha_beta = tuple([(ia * self.input_scale, ib * self.input_scale) for ia, ib in zip(a, b)])

        # do convert_to
        bmcv.convert_to(resized_images, preprocessed_imgs, alpha_beta)
        return preprocessed_imgs
```

6.3.3 推理

```
class SophonInference:
    def __init__(self, **kwargs):

        ...

        self.io_mode = sail.IOMode.SYSIO
        self.engine = sail.Engine(self.model_path, self.device_id, self.io_mode)
```

(续下页)

(接上页)

```
        self.handle = self.engine.get_handle()
        self.graph_name = self.engine.get_graph_names()[0]
        self.bmcv = sail.Bmcv(self.handle)

        ...

        input_names = self.engine.get_input_names(self.graph_name)
        for input_name in input_names:

            input_shape = self.engine.get_input_shape(self.graph_name, input_name)
            input_dtype = self.engine.get_input_dtype(self.graph_name, input_name)
            input_scale = self.engine.get_input_scale(self.graph_name, input_name)
            ...
            if self.input_mode:
                input = sail.Tensor(self.handle, input_shape, input_dtype, True, True)
            ...
            input_tensors[input_name] = input
            ...

        output_names = self.engine.get_output_names(self.graph_name)

        for output_name in output_names:

            output_shape = self.engine.get_output_shape(self.graph_name, output_name)
            output_dtype = self.engine.get_output_dtype(self.graph_name, output_name)
            output_scale = self.engine.get_output_scale(self.graph_name, output_name)
            ...
            if self.input_mode:
                output = sail.Tensor(self.handle, output_shape, output_dtype, True, True)
            ...
            output_tensors[output_name] = output
            ...
    def infer_bmimage(self, input_data):
        self.get_input_feed(self.input_names, input_data)

        #inference
        self.engine.process(self.graph_name, self.input_tensors, self.output_tensors)
        outputs_dict = OrderedDict()
        for name in self.output_names:
            outputs_dict[name] = self.output_tensors[name].asnumpy().copy() * self.output_
↪scales[name]
        return outputs_dict
```

6.4 解码模块

目录

6.4.1 OpenCV 解码

OpenCV 支持 YUVI420/BGR 格式输出，为了提高性能，示例中解码输出设置 yuv 格式数据。

简单示例如下：

```
cv::VideoCapture cap;
  if (!cap.isOpened()) {
    cap.open(input_url);
}
cap.set(cv::CAP_PROP_OUTPUT_YUV, 1.0); //
↪设置输出YUVI420格式数据，如选择BGR输出则注释掉此行代码
cv::Mat *img = new cv::Mat;
cap.read(*img);
//do something
......
//end
delete img;
```

cap.set 接口函数对输出格式设置，cap::read 获取 cv::Mat 对象 img，img 数据接下来需要通过图像运算加速接口（bmcv 模块）对数据进行推理前的预处理操作。

如果需要使用 python-opencv，请正确设置环境变量

```
export PYTHONPATH=/opt/sophon/sophon-opencv-latest/opencv-python:$PYTHONPATH
```

6.4.2 FFmpeg 解码

- C 编程接口初始化配置:

```
// ffmpeg默认输出NV12压缩格式数据，初始化解码器配置方法如下：
/*set compressed output*/
av_dict_set(&opts, "output_format", "101", 0);

if ((ret = avcodec_open2(*dec_ctx, dec, &opts)) < 0) {
fprintf(stderr, "Failed to open %s codec\n",
```

(续下页)

(接上页)

```
        av_get_media_type_string(type));
return ret;
}
```

- Python 编程接口

```
import sophon.sail as sail
decoder = sail.Decoder(filename)
img0 = decoder.read(handle)   #默认输出yuv i420格式
```

6.5 图形运算加速模块

目录

BMCV 提供了一套基于算丰深度学习处理器优化的机器视觉库，目前可以完成色彩空间转换、尺度变换、仿射变换、投射变换、线性变换、画框、JPEG 编码、BASE64 编码、NMS、排序、特征匹配等操作。

C++ 接口实现请参考《算能边缘产品 BMCV 开发参考指南》。

Python 接口的实现请参考《sophon-sail_zh》。

BMCV API 均是围绕 bm_image 来进行的。一个 bm_image 结构对应于一张图片。

6.5.1 C++ 语言编程接口

bm_image 结构体

```
struct bm_image {
    int width;
    int height;
    bm_image_format_ext image_format;
    bm_data_format_ext data_type;
    bm_image_private* image_private;
};
```

bm_image 结构成员包括图片的宽高，图片格式，图片数据格式，以及该结构的私有数据。

图片格式 image_format 枚举类型

```
typedef enum bm_image_format_ext_{
    FORMAT_YUV420P,
    FORMAT_NV12,
    FORMAT_NV21,
    FORMAT_NV16,
    FORMAT_NV61,
    FORMAT_RGB_PLANAR,
    FORMAT_BGR_PLANAR,
    FORMAT_RGB_PACKED,
    FORMAT_BGR_PACKED,
    FORMAT_GRAY,
    FORMAT_COMPRESSED
}bm_image_format_ext;
```

格式	说明
FORMAT_YUV420P	表示预创建一个 YUV420 格式的图片，有三个 plane
FORMAT_NV12	表示预创建一个 NV12 格式的图片，有两个 plane
FORMAT_NV21	表示预创建一个 NV21 格式的图片，有两个 plane
FORMAT_RGB_PLANAR	表示预创建一个 RGB 格式的图片，RGB 分开排列，有一个 plane
FORMAT_BGR_PLANAR	表示预创建一个 BGR 格式的图片，BGR 分开排列，有一个 plane
FORMAT_RGB_PACKED	表示预创建一个 RGB 格式的图片，RGB 交错排列，有一个 plane
FORMAT_BGR_PACKED	表示预创建一个 BGR 格式的图片，BGR 交错排列，有一个 plane
FORMAT_GRAY	表示预创建一个灰度图格式的图片，有一个 plane
FORMAT_COMPRESSED	表示预创建一个 VPU 内部压缩格式的图片，有四个 plane

数据存储格式枚举

```
typedef enum bm_image_data_format_ext_{
    DATA_TYPE_EXT_FLOAT32,
    DATA_TYPE_EXT_1N_BYTE,
    DATA_TYPE_EXT_4N_BYTE,
    DATA_TYPE_EXT_1N_BYTE_SIGNED,
    DATA_TYPE_EXT_4N_BYTE_SIGNED,
}bm_image_data_format_ext;
```

数据格式	说明
DATA_TYPE_EXT_FLOAT32	表示所创建的图片数据格式为单精度浮点数
DATA_TYPE_EXT_1N_BYTE	表示所创建图片数据格式为普通带符号 1N INT8
DATA_TYPE_EXT_4N_BYTE	表示所创建图片数据格式为 4N INT8，即四张带符号 INT8 图片数据交错排列
DATA_TYPE_EXT_1N_BYTE_SIGNED	表示所创建图片数据格式为普通无符号 1N UINT8
DATA_TYPE_EXT_4N_BYTE	表示所创建图片数据格式为 4N UINT8，即四张无符号 INT8 图片数据交错排列

更多 BMCV 接口使用方法请参考《算能边缘产品 BMCV 开发参考指南》

6.5.2 Python 语言编程接口

本章节只简要介绍了用例 YOLOv5 所用到的接口函数。

更多接口定义请查阅《sophon-sail_zh 》。

- **init**

```
def __init__(handle):
""" Constructor.
Parameters
---------
handle : sail.Handle Handle instance
"""
```

- **convert_to**

```
def convert_to(input, alpha_beta):
""" Applies a linear transformation to an image.
Parameters
---------
input : sail.BMImage Input image
alpha_beta: tuple (a0, b0), (a1, b1), (a2, b2) factors
Returns
---------
output : sail.BMImage Output image
"""
```

- **convert_format**

```
def convert_format(input, output):
"""Convert input to output format.
```

(续下页)

(接上页)

```
3
4   Parameters
5   ----------
6   input : sail.BMImage
7       BMimage instance
8   output : sail.BMImage
9       output image
10  """
```

- **vpp_crop_and_resize_padding**

```
1   def vpp_crop_and_resize_padding(input, crop_x0, crop_y0, crop_w, crop_h, resize_w, resize_h,
    ↪ padding):
2   """ Crop then resize an image using vpp.
3
4   Parameters
5   ----------
6   input : sail.BMImage
7       Input image
8   crop_x0 : int
9       Start point x of the crop window
10  crop_y0 : int
11      Start point y of the crop window
12  crop_w : int
13      Width of the crop window
14  crop_h : int
15      Height of the crop window
16  resize_w : int
17      Target width
18  resize_h : int
19      Target height
20  padding : PaddingAtrr
21      padding info
22
23  Returns
24  ----------
25  output : sail.BMImage
26      Output image
27  """
```

6.6 模型推理

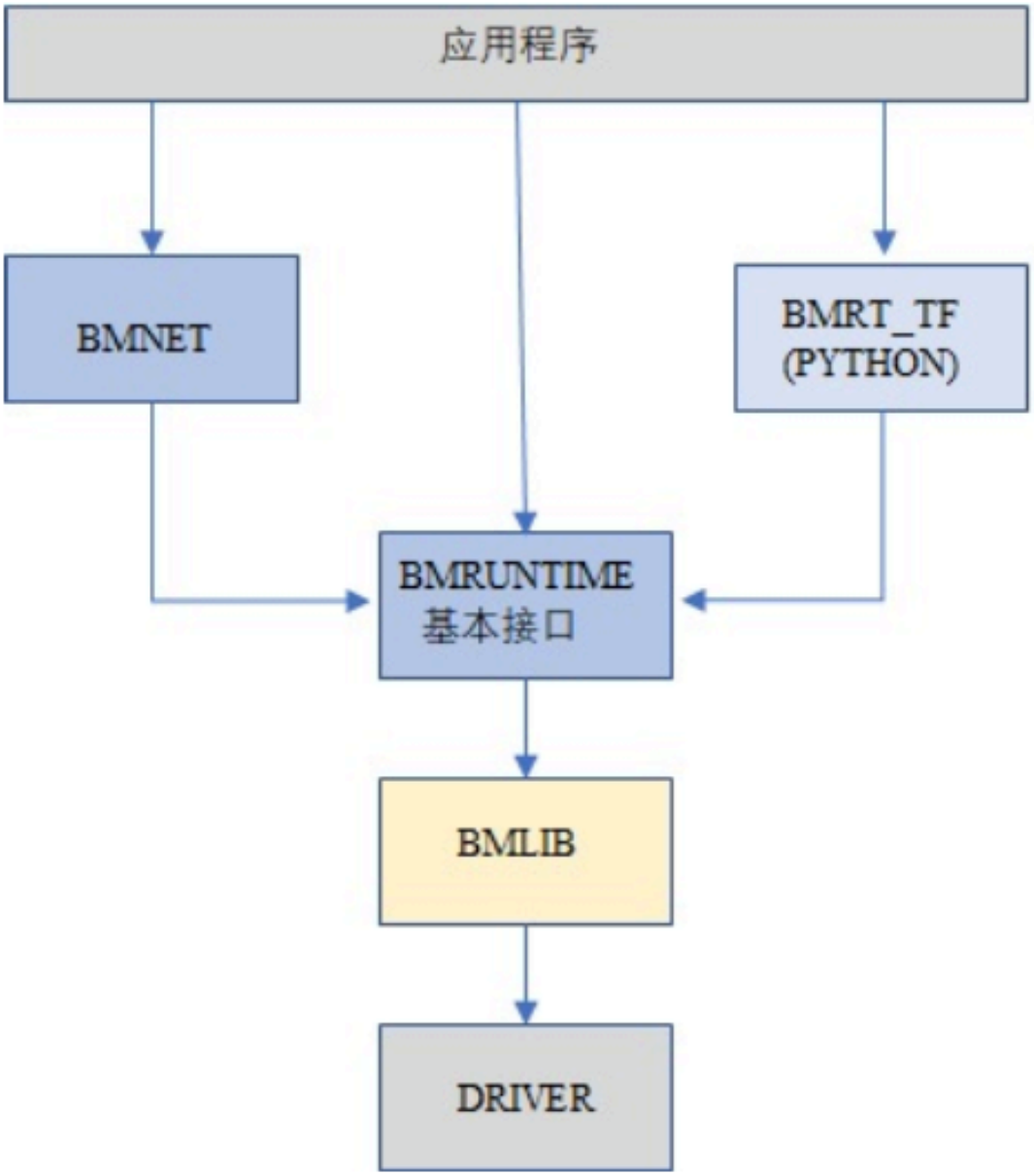
目录

- BMRuntime 模块 C 接口介绍
- Python 接口

C 接口详细介绍请阅读《算能边缘产品 BMRUNTIME 开发参考手册》。

Python 接口详细介绍请阅读《sophon-sail_zh 》。

BMRuntime 用于读取 BMCompiler 的编译输出 (.bmodel)，驱动其在 SOPHON 智能视觉深度学习处理器中执行。BMRuntime 向用户提供了丰富的接口，便于用户移植算法，其软件架构如下:



BMRuntime 实现了 C/C++ 接口，SAIL 模块基于对 BMRuntime 和 BMLib 的封装实现了 Python 接口。本章主要介绍 C 和 Python 常用接口，主要内容如下：

- BMLib 接口：负责设备 Handle 的管理、内存管理、数据搬运、API 的发送和同步、A53 使能、设置智能视觉深度学习处理器工作频率等
- BMRuntime 的 C 语言接口
- BMLib 和 BMRuntime 的 Python 接口介绍

6.6.1 BMLib 模块 C 接口介绍

BMLIB 接口

- 用于设备管理，不属于 BMRuntime，但需要配合使用，所以先介绍。

  BMLIB 接口是 C 语言接口，对应的头文件是 bmlib_runtime.h，对应的 lib 库为 libbmlib.so。

  BMLIB 接口用于设备管理，包括设备内存的管理。

  BMLIB 的接口很多，这里介绍应用程序通常需要用到的接口。

- **bm_dev_request**

用于请求一个设备，得到设备句柄 handle。其他设备接口，都需要指定这个设备句柄。其中 devid 表示设备号，在 PCIe 模式下，存在多个设备时可以用于选择对应的设备；在 SoC 模式下，请指定为 0。

```
/**
 * @name    bm_dev_request
 * @brief   To create a handle for the given device
 * @ingroup bmlib_runtime
 *
 * @param [out] handle  The created handle
 * @param [in]  devid   Specify on which device to create handle
 * @retval  BM_SUCCESS  Succeeds.
 *          Other code  Fails.
 */
bm_status_t bm_dev_request(bm_handle_t *handle, int devid);
```

- **bm_dev_free**

用于释放一个设备。通常应用程序开始需要请求一个设备，退出前释放这个设备。

```
/**
 * @name    bm_dev_free
 * @brief   To free a handle
 * @param [in] handle  The handle to free
 */
void bm_dev_free(bm_handle_t handle);
```

6.6.2 BMRuntime 模块 C 接口介绍

对应的头文件为 bmruntime_interface.h，对应的 lib 库为 libbmrt.so。

用户程序使用 C 接口时建议使用该接口，该接口支持多种 shape 的静态编译网络，支持动态编译网络。

- **bmrt_create**

```
/**
 * @name    bmrt_create
 * @brief   To create the bmruntime with bm_handle.
 * This API creates the bmruntime. It returns a void* pointer which is the pointer
 * of bmruntime. Device id is set when get bm_handle;
 * @param [in] bm_handle     bm handle. It must be initialized by using bmlib.
 * @retval void* the pointer of bmruntime
 */
void* bmrt_create(bm_handle_t bm_handle);
```

- **bmrt_destroy**

```
/**
 * @name    bmrt_destroy
 * @brief   To destroy the bmruntime pointer
 * @ingroup bmruntime
 * This API destroy the bmruntime.
 * @param [in]     p_bmrt        Bmruntime that had been created
 */
void bmrt_destroy(void* p_bmrt);
```

- **bmrt_load_bmodel**

加载 bmodel 文件，加载后 bmruntime 中就会存在若干网络的数据，后续可以对网络进行推理。

```
/**
 * @name    bmrt_load_bmodel
 * @brief   To load the bmodel which is created by BM compiler
 * This API is to load bmodel created by BM compiler.
 * After loading bmodel, we can run the inference of neuron network.
 * @param   [in]   p_bmrt        Bmruntime that had been created
 * @param   [in]   bmodel_path   Bmodel file directory.
 * @retval true    Load context sucess.
 * @retval false   Load context failed.
 */
bool bmrt_load_bmodel(void* p_bmrt, const char *bmodel_path);
```

- **bmrt_load_bmodel_data**

加载 bmodel，不同于 bmrt_load_bmodel，它的 bmodel 数据存在内存中

```
/*
Parameters: [in] p_bmrt      - Bmruntime that had been created.
            [in] bmodel_data - Bmodel data pointer to buffer.
            [in] size        - Bmodel data size.
Returns:    bool             - true: success; false: failed.
*/
bool bmrt_load_bmodel_data(void* p_bmrt, const void * bmodel_data, size_t size);
```

- **bmrt_get_network_info**

bmrt_get_network_info 根据网络名，得到某个网络的信息

```
/* bm_stage_info_t holds input shapes and output shapes;
every network can contain one or more stages */
typedef struct {
bm_shape_t* input_shapes;   /* input_shapes[0] / [1] / ... / [input_num-1] */
bm_shape_t* output_shapes;  /* output_shapes[0] / [1] / ... / [output_num-1] */
} bm_stage_info_t;

/* bm_tensor_info_t holds all information of one net */
typedef struct {
const char* name;              /* net name */
bool is_dynamic;               /* dynamic or static */
int input_num;                 /* number of inputs */
char const** input_names;      /* input_names[0] / [1] / .../ [input_num-1] */
bm_data_type_t* input_dtypes;  /* input_dtypes[0] / [1] / .../ [input_num-1] */
float* input_scales;           /* input_scales[0] / [1] / .../ [input_num-1] */
int output_num;                /* number of outputs */
char const** output_names;     /* output_names[0] / [1] / .../ [output_num-1] */
bm_data_type_t* output_dtypes; /* output_dtypes[0] / [1] / .../ [output_num-1] */
float* output_scales;          /* output_scales[0] / [1] / .../ [output_num-1] */
int stage_num;                 /* number of stages */
bm_stage_info_t* stages;       /* stages[0] / [1] / ... / [stage_num-1] */
} bm_net_info_t;
```

bm_net_info_t 表示一个网络的全部信息，bm_stage_info_t 表示该网络支持的不同的 shape 情况。

```
/**
 * @name    bmrt_get_network_info
 * @brief   To get network info by net name
 * @param [in]     p_bmrt        Bmruntime that had been created
 * @param [in]     net_name      Network name
 * @retval  bm_net_info_t*       Pointer to net info, needn't free by user; if net name not found,
↪ will return NULL.
 */
const bm_net_info_t* bmrt_get_network_info(void* p_bmrt, const char* net_name);
```

示例代码：

```
const char *model_name = "VGG_VOC0712_SSD_300X300_deploy"
```

(续下页)

(接上页)

```
const char **net_names = NULL;
bm_handle_t bm_handle;
bm_dev_request(&bm_handle, 0);
void * p_bmrt = bmrt_create(bm_handle);
bool ret = bmrt_load_bmodel(p_bmrt, bmodel.c_str());
std::string bmodel; //bmodel file
int net_num = bmrt_get_network_number(p_bmrt, model_name);
bmrt_get_network_names(p_bmrt, &net_names);
for (int i=0; i<net_num; i++) {
//do somthing here
......
}
free(net_names);
bmrt_destroy(p_bmrt);
bm_dev_free(bm_handle);
```

- **bmrt_shape_count**

接口声明如下：

```
/*
number of shape elements, shape should not be NULL and num_dims should not large than BM_
↪MAX_DIMS_NUM
*/
uint64_t bmrt_shape_count(const bm_shape_t* shape);
```

可以得到 shape 的元素个数。

比如 num_dims 为 4，则得到的个数为 dims[0]*dims[1]*dims[2]*dims[3]

bm_shape_t 结构介绍：

```
typedef struct {
int num_dims;
int dims[BM_MAX_DIMS_NUM];
} bm_shape_t;
```

bm_shape_t 表示 tensor 的 shape，目前最大支持 8 维的 tensor。其中 num_dims 为 tensor 的实际维度数，dims 为各维度值，dims 的各维度值从 [0] 开始，比如 (n, c, h, w) 四维分别对应 (dims[0], dims[1], dims[2], dims[3])。

如果是常量 shape，初始化参考如下：

```
bm_shape_t shape = {4, {4,3,228,228}};
bm_shape_t shape_array[2] = {
{4, {4,3,28,28}}, // [0]
{2, {2,4}}, // [1]
}
```

- **bm_image_from_mat**

```
//if use this function you need to open USE_OPENCV macro in include/bmruntime/bm_wrapper.hpp
/**
* @name    bm_image_from_mat
* @brief   Convert opencv Mat object to BMCV bm_image object
* @param [in]     in          OPENCV mat object
* @param [out]    out         BMCV bm_image object
* @retval true    Launch success.
* @retval false   Launch failed.
*/
static inline bool bm_image_from_mat (cv::Mat &in, bm_image &out)
```

```
//* @brief   Convert opencv multi Mat object to multi BMCV bm_image object
static inline bool bm_image_from_mat (std::vector<cv::Mat> &in, std::vector<bm_image> &out)
```

- **bm_image_from_frame**

```
/**
 * @name    bm_image_from_frame
 * @brief   Convert ffmpeg a avframe object to a BMCV bm_image object
 * @ingroup bmruntime
 *
 * @param [in]     bm_handle   the low level device handle
 * @param [in]     in          a read-only avframe
 * @param [out]    out         an uninitialized BMCV bm_image object
                   use bm_image_destroy function to free out parameter until you no longer useing it.
 * @retval true    change success.
 * @retval false   change failed.
 */

static inline bool bm_image_from_frame (bm_handle_t       &bm_handle,
                                        AVFrame           &in,
                                        bm_image          &out)
```

```
/**
 * @name    bm_image_from_frame
 * @brief   Convert ffmpeg avframe  to BMCV bm_image object
 * @ingroup bmruntime
 *
 * @param [in]     bm_handle   the low level device handle
 * @param [in]     in          a read-only ffmpeg avframe vector
 * @param [out]    out         an uninitialized BMCV bm_image vector
                   use bm_image_destroy function to free out parameter until you no longer useing it.
 * @retval true    change success.
 * @retval false   chaneg failed.
 */
static inline bool bm_image_from_frame (bm_handle_t               &bm_handle,
                                        std::vector<AVFrame>      &in,
                                        std::vector<bm_image>     &out)
```

- **bm_inference**

```
//if use this function you need to open USE_OPENCV macro in include/bmruntime/bm_wrapper.hpp
/**
* @name     bm_inference
* @brief    A block inference wrapper call
* @ingroup bmruntime
*
* This API supports the neuron nework that is static-compiled or dynamic-compiled
* After calling this API, inference on TPU is launched. And the CPU
* program will be blocked.
* This API support single input && single output, and multi thread safety
*
* @param [in]    p_bmrt         Bmruntime that had been created
* @param [in]    input          bm_image of single-input data
* @param [in]    output         Pointer of  single-output buffer
* @param [in]    net_name       The name of the neuron network
* @param [in]    input_shape    single-input shape
*
* @retval true    Launch success.
* @retval false   Launch failed.
*/
static inline bool bm_inference (void          *p_bmrt,
                                 bm_image      *input,
                                 void          *output,
                                 bm_shape_t input_shape,
                                 const char   *net_name)
```

```
// * This API support single input && multi output, and multi thread safety
static inline bool bm_inference (void                    *p_bmrt,
                                 bm_image                *input,
                                 std::vector<void*>       outputs,
                                 bm_shape_t            input_shape,
                                 const char            *net_name)
```

```
// * This API support multiple inputs && multiple outputs, and multi thread safety
static inline bool bm_inference (void                          *p_bmrt,
                                 std::vector<bm_image*>         inputs,
                                 std::vector<void*>           outputs,
                                 std::vector<bm_shape_t>   input_shapes,
                                 const char                *net_name)
```

6.6.3 Python 接口

本章节只简要介绍了 YOLOv5 用例中所用的接口函数。

更多接口定义请查阅《sophon-sail 使用手册 》。

- **Engine**

```
def __init__(tpu_id):
""" Constructor does not load bmodel.
Parameters
---------
tpu_id : int TPU ID. You can use bm-smi to see available IDs
"""
```

- **load**

```
def load(bmodel_path):
"""Load bmodel from file.
Parameters
---------
bmodel_path : str Path to bmode
"""
```

- **set_io_mode**

```
def set_io_mode(mode):
""" Set IOMode for a graph.
Parameters
---------
mode : sail.IOMode Specified io mode
"""
```

- **get_graph_names**

```
def get_graph_names():
""" Get all graph names in the loaded bmodels.
Returns
------
graph_names : list Graph names list in loaded context
"""
```

- **get_input_names**

```
def get_input_names(graph_name):
""" Get all input tensor names of the specified graph.
Parameters
---------
graph_name : str Specified graph name
Returns
------
input_names : list All the input tensor names of the graph
"""
```

- **get_output_names**

```
def get_output_names(graph_name):
""" Get all output tensor names of the specified graph.
Parameters
---------
graph_name : str Specified graph name
Returns
------
input_names : list All the output tensor names of the graph
"""
```

- **sail.IOMode**

```
# Input tensors are in system memory while output tensors are in device memory sail.IOMode.SYSI
# Input tensors are in device memory while output tensors are in system memory.
sail.IOMode.SYSO
# Both input and output tensors are in system memory.
sail.IOMode.SYSIO
# Both input and output tensors are in device memory.
ail.IOMode.DEVIO
```

- **sail.Tensor**

```
def __init__(handle, shape, dtype, own_sys_data, own_dev_data):
""" Constructor allocates system memory and device memory of the tensor.
Parameters
---------
handle : sail.Handle Handle instance
shape : tuple Tensor shape
dytpe : sail.Dtype Data type
own_sys_data : bool Indicator of whether own system memory
own_dev_data : bool Indicator of whether own device memory
"""
```

- **get_input_dtype**

```
def get_input_dtype(graph_name, tensor_name):
""" Get scale of an input tensor. Only used for int8 models.
Parameters
---------
graph_name : str The specified graph name tensor_name : str The specified output tensor name
Returns
------
scale: sail.Dtype Data type of the input tensor
"""
```

- **get_output_dtype**

```
def get_output_dtype(graph_name, tensor_name):
""" Get the shape of an output tensor in a graph.
Parameters
```

(续下页)

(接上页)

```
---------
graph_name : str The specified graph name tensor_name : str The specified output tensor name
Returns
------
tensor_shape : list The shape of the tensor
"""
```

- **process**

```
def process(graph_name, input_tensors, output_tensors):
""" Inference with provided input and output tensors.
Parameters
---------
graph_name : str The specified graph name
input_tensors : dict {str : sail.Tensor} Input tensors managed by user
output_tensors : dict {str : sail.Tensor} Output tensors managed by user
"""
```

- **get_input_scale**

```
def get_input_scale(graph_name, tensor_name):
""" Get scale of an input tensor. Only used for int8 models.
Parameters
---------
graph_name : str The specified graph name tensor_name : str The specified output tensor name
Returns
------
scale: float32 Scale of the input tensor
"""
```

- **get_output_scale**

```
def get_output_scale(graph_name, tensor_name)
""" Get scale of an output tensor. Only used for int8 models.

Parameters
----------
graph_name : str
    The specified graph name
tensor_name : str
    The specified output tensor name

Returns
-------
scale: float32
    Scale of the output tensor
"""
```

- **get_input_shape**

```
def get_input_shape(graph_name, tensor_name):
""" Get the maximum dimension shape of an input tensor in a graph.
    There are cases that there are multiple input shapes in one input name,
    This API only returns the maximum dimension one for the memory allocation
    in order to get the best performance.

Parameters
----------
graph_name : str
    The specified graph name
tensor_name : str
    The specified input tensor name

Returns
-------
tensor_shape : list
    The maxmim dimension shape of the tensor
"""
```

- **get_output_shape**

```
def get_output_shape(graph_name, tensor_name):
""" Get the shape of an output tensor in a graph.

Parameters
----------
graph_name : str
    The specified graph name
tensor_name : str
    The specified output tensor name

Returns
-------
tensor_shape : list
    The shape of the tensor
"""
```

CHAPTER 7

附录

7.1 微服务器定制化软件包

目录

7.1.1 bm1688/CV186AH socbak 使用说明

1. sudo su 进入 root 权限
2. 创建/socrepack 目录，并且将外部存储设备挂载到该位置
3. 下载最新的 socbak 工具，然后将 socbak.sh 脚本拷贝到该目录下
4. cd 到这个目录下，./socbak.sh 执行环境镜像
5. 下载最新的 bm1688_repack 包，解压并进入
6. 建立目录 pack, 然后将原始刷机包解压到这个目录下
7. cd 到 depack.sh 文件同目录, 然后执行./depack.sh 脚本进行解包
8. 生成的各个分区压缩包会在 update 目录下生成, 此时将 socbak 生成的所有文件都拷贝到这个目录下
9. 修改完毕后 cd 到 enpack.sh 文件同目录, 然后执行./enpack.sh sdcard 或者./enpack.sh tftp 命令进行打包

10. 生成的新的刷机包在 update/sdcard 和 update/tftp

7.2 SoC 模式内存修改工具

目录

备注: SoC 模式下的内存修改脚本与远程内存修改工具使用时，均需要目标 SoC 服务器上 rootfs-rw 分区（即根目录），有 200M 以上空间。同时，内存修改脚本在目标 SoC 的存放目录也需要有 200M 以上空间；使用远程内存修改工具时，目标 SoC 的/data 分区需要 200M 空间。使用如下命令查看目录的空间大小:

```
# 查看根目录
sudo df -h /
# 查看/data分区
sudo df -h /data
# 查看当前目录
sudo df -h .
```

7.2.1 工具 1: 在 SoC 上使用脚本进行修改

说明：本脚本用于在 SoC 运行环境中修改 SE9 在 SoC 模式下的 NPU、VPP 的内存分配。同时，调整 NPU、VPP 后，SoC 的操作系统将使用除了 NPU、VPP 、uboot 固件之外的所有内存。

备注: 本脚本默认适配 v1.3release 及之后版本。之前的版本修改后有启动失败的风险如果发现设备树没有自动识别正确，请在命令末尾增加一个设备树文件名用于指定设备树

使用教程:

1. 使用浏览器打开并下载 https://sophon-file.sophon.cn/sophon-prod-s3/drive/23/09/11/13/DeviceMemoryModificationKit.tgz ，选择” memory_edit_v<x.x>.tar.xz” 压缩包。
2. 登录微服务器，并将脚本压缩包拷贝到 SoC 微服务器上，执行解压命令并进入该压缩包；并执行命令检查微服务器 NPU、VPU、VPP 当前可以配置的最大内存大小和当前配置的内存大小。

```
tar -xaf memory_edit_vx.x.tar.xz
cd memory_edit
./memory_edit.sh -p
```

其中类似如下的输出显示了可配置的最大内存。**其中, MiB（Mebibyte）是计算机存储容量的单位，1 MiB = 1024 KiB = 1024*1024 Byte**。

```
Info: get max memory size ...
Info: max npu+vpp size: 0x97800000 [2424 MiB]
Info: max npu size: 0x97800000 [2424 MiB]
Info: max vpp size: 0x97800000 [2424 MiB]
```

其中类似如下的输出显示了当前配置的内存大小。

```
Info: get now memory size ...
Info: now npu size: 0x30000000 [768 MiB]
Info: now vpp size: 0x50000000 [1280 MiB]
```

请检查输出中是否有 Error ，如果有请检查 SoC 运行环境是否支持内存修改。

3. 进行内存布局的修改，其中输入的三个参数是需要 NPU、VPP 配置的大小的十进制数字，单位 MiB；或者为十六进制数值，单位 Byte。**SE9 默认没有 VPU 内存，配置为 0 即可**。

```
# 十进制，单位MiB
./memory_edit.sh -c -npu 300 -vpu 0 -vpp 300
# 十六进制，单位Byte
./memory_edit.sh -c -npu 0x12C00000 -vpu 0x0 -vpp 0x12C00000
```

请检查输出中是否有 Error ，以及类似于如下输出中三个部分的大小是否与您所需要配置的大小相同。

```
Info:F
↪==========================================================================
Info: output configuration results ...
Info: ion npu mem area(ddr1): 0x12c00000 [300 MiB] 0x68800000 -> 0x7b3fffff
Info: ion vpp mem area(ddr1): 0x12c00000 [300 MiB] 0xed400000 -> 0xffffffff
Info:F
↪==========================================================================
Info: start check memory size ...
Info: check npu size: 0x12c00000 [300 MiB]
Info: check vpp size: 0x12c00000 [300 MiB]
Info: check edit size ok
```

4. 如果检查无误，请保存当前工作， 将修改后的 boot.itb 文件替换启动分区中的启动映像，并重启机器使修改生效。

```
sudo cp boot.itb /boot && sync
sudo reboot
```

备注: 如果需要迁移修改配置，请导出这个 boot.itb 文件，拷贝到其他需要修改的微服务器的/boot 目录下 **要求：与修改时使用的微服务器型号、刷机包版本、SDK 版本相同，可以通过 bm_version 命令查看**。

7.2.2 工具 2: 使用图像化程序远程修改

说明：本工具是一个图形化程序，可以通过远程 SSH 修改工作在 SoC 模式下的 BM1688、CV186AH 以及 BM1684/BM1684X 的 NPU、VPU、VPP 占用空间。默认适配环境为 64 位 win10 操作系统或者带有桌面环境的 linux 系统。

备注: 该工具适用于 BM1688/CV186AH SOPHONSDK v1.3 及以上版本、BM1684/BM1684X SOPHONSDK V22.09.02 及以上 SoC 版本。当前默认适配桌面系统如下: amd64-win10、amd64-linux(带有完整桌面，如 xfce4)、aarch64-linux(带有完整桌面，如 xfce4)。

单远程目标

使用教程:

1. 使用浏览器打开 https://sophon-file.sophon.cn/sophon-prod-s3/drive/23/09/11/13/DeviceMemoryModificationKit.tgz 并下载最新的远程内存修改工具，选择”qt_mem_edit_V<x.x.x>.” 文件，其中.exe 文件适用于 win 系统，.AppImage 适用于带有桌面环境的 linux 系统。
2. windows 系统双击程序即可运行，linux 系统可能需要使用 chmod +x qt_mem_edit_Vx.x.x-架构名.AppImage 命令使能其具有运行权限后才能运行该程序。该程序正常运行时如下图。
3. 在该程序中配置微服务器远程 SSH 链接需要的 IP、端口、用户名和密码。
4. 点击“修改目录”按钮配置执行文件目录，该目录会在当前操作的 host 主机上存放远程修改内存的所有过程文件（默认目录为远程内存修改程序所在目录）。
5. 点击“获取信息”按钮获取微服务器当前 NPU、VPP 配置的内存大小信息和最大可以配置的内存大小信息，这些信息将在“获取信息”按钮左侧的文本框中显示。
6. 根据您的需要修改“进行配置”按钮等高的两个个数字框（SE9 没有 VPU 内存，配置为 0 即可），它们代表了需要配置的 NPU、VPP 各个部分的内存大小（单位 MiB，十进制）。
7. 点击“进行配置”按钮，配置完成后，程序输出如下图。
8. 查看“远程执行信息”中是否有 Error 输出，如果没有错误输出，请登录微服务器，保存微服务器上当前工作，并重启微服务器使您的修改生效。

```
sudo reboot
```

图 7.1: 远程内存修改工具运行初始界面

图 7.2: 远程内存修改工具配置完成效果

9. 进行操作后执行文件目录下会存放从微服务器上获取的内存修改的所有过程文件，文件名以 memory_edit_p_ 为前缀代表该文件是”获取信息“操作的过程文件，文件名以 memory_edit_c_ 为前缀代表该文件是“进行配置”操作的过程文件。

文件名以 memory_edit_p_ 为前缀的获取信息压缩包中文件目录如下：

```
└─── memory_edit
    ├─── bintools
    ├─── boot.itb
    ├─── log.txt
    ├─── memory_edit.sh
    ├─── multi.its
    ├─── memory_edit_p.log
    └─── output
```

文件名以 memory_edit_c_ 为前缀的获取信息压缩包中文件目录如下：

```
└─── memory_edit
    ├─── bintools
    ├─── boot.itb
    ├─── log.txt
    ├─── memory_edit.sh
    ├─── multi.its
    ├─── memory_edit_c.log
    └─── output
```

备注： 在“进行配置”操作的过程文件压缩包中，有一个名为 boot.itb 的文件，如果需要迁移修改配置，请解压并导出这个 boot.itb 文件，拷贝到其他需要修改的微服务器的/boot 目录下。**要求：与修改时使用的微服务器型号、刷机包版本、SDK 版本相同，可以通过 bm_version 命令查看上述信息**

多远程目标

使用教程：

1. 使用浏览器打开 https://sophon-file.sophon.cn/sophon-prod-s3/drive/23/09/11/13/DeviceMemoryModificationKit.tgz 并下载最新的远程内存修改工具，选择”qt_mem_edit_V<x.x.x>.” 文件，其中.exe 文件适用于 win 系统，.AppImage 适用于带有桌面环境的 linux 系统。

2. windows 系统双击程序即可运行，linux 系统可能需要使用 chmod +x qt_mem_edit_Vx.x.x-架构名.AppImage 命令使能其具有运行权限后才能运行该程序。

3. 修改“进行配置”按钮等高的两个数字框，它们代表了需要配置的 NPU、VPP 各个部分的内存大小（单位 MiB，十进制），配置 SE9 VPU 默认为空即可。

1. 点击远程内存布局修改工具中的“批量配置”按钮。

2. 配置好需要修改内存的批量远程目标后，点击“批量配置”，在弹出的窗口中选择“yes”，在多台远程设备内存修改时，远程微服务器将会自动重启。执行文件目录下会存放所有

图 7.3: 远程内存修改工具进行配置

远程设备的执行文件。

7.3 BM1688/CV186AH SOPHONSDK 源码下载以及编译方法

目前，BM1688/CV186AH SOPHONSDK 源码已在 github 开源发布，您可以下载源码编译生成刷机包，可用于软件定制。

7.3.1 编译环境配置

编译 BM1688/CV186AH SOPHONSDK 源码需要再 linux 系统上进行，推荐使用 ubuntu20.04 或者 22.04。

在 github 建立个人账号，并配置好 ssh key，下载代码需要用到个人 github 账号。

7.3.2 安装 repo

（1）配置 repo 安装路径

```
mkdir ~/bin
```

（2）增加环境变量

```
vim ~/.profile
```

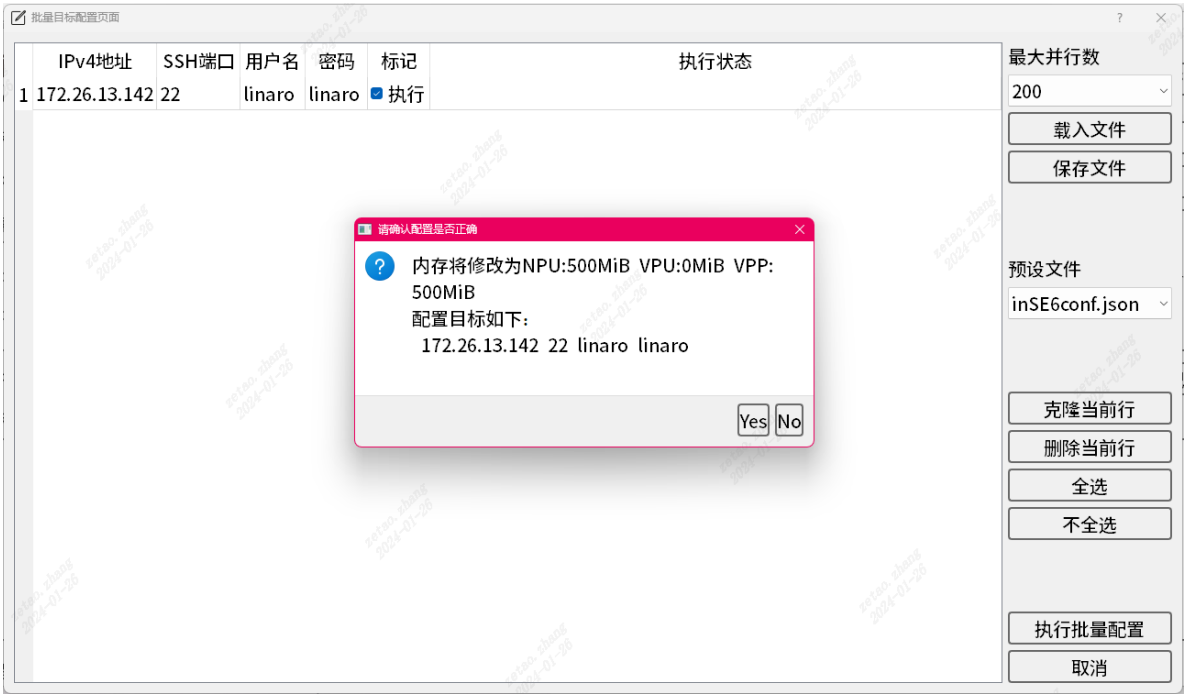


图 7.4: 远程内存修改工具批量配置

在文件最后增加

```
PATH=~/bin:$PATH
```

（3）安装 repo

```
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
```

7.3.3 下载代码

请使用以下命令下载 SDK 源码

```
# 创建工作目录
mkdir /your/workspace/path && cd /your/workspace/path
# 下载代码
repo init -u https://github.com/sophgo/manifest.git -m release/edge.xml
```


7.3.4 docker 配置

使用以下命令下载 docker 镜像文件并导入

```
sudo apt-get install python3-pip
pip3 install dfss --upgrade
python -m dfss --url=open@sophgo.com:/gemini-sdk/docker/ubuntu_bm1688.tar

#导入docker镜像，有导入过的可以不用操作
sudo cat ubuntu_bm1688.tar | docker import - bm1688
```

将以下命令添加到 ~/.bashrc

```
function run_docker() {
    docker run  -e LOCAL_USER_ID=`id -u $USER_ID` --privileged -itd -v $2:/project/$1 --
↪name $1  bm1688:latest /bin/bash
}
```

重新 source 文件，使改动生效

```
source ~/.bashrc
```

调用 run_docker 命令，进入 docker 编译环境

```
# 在进入docker前将代码下载到 /your/workspace/path
run_docker SDK_build  /your/workspace/path
# 进入container
docker run -it SDK_build /bin/bash
pip3 install jinja2
```

7.3.5 编译源码

请使用以下命令编译 BM1688/CV186AH SOPHONSDK 源码

```
defconfig  bm1688_wevb_emmc
#需要有root权限或者使用docker
build_bm1688_all
```

生成的升级包在 install/soc_bm1688_wevb_emmc/package_edge/sdcard

7.4 BM1684(X)_to_BM1688(CV186AH) 兼容性文档

该文档面向有一定 SOPHONSDK 使用经验的用户，它还需丰富和完善，如果您发现了新的问题或有改进建议，可以反馈给我们的工作人员。

- 获取 BM1688/CV186AH SOPHONSDK
- 准备 BM1688/CV186AH 的 bmodel
- 准备编译依赖的 SDK
- 参照接口变化修改部分代码:
  - 头文件改动
  - bmcv 改动
  - ffmpeg 改动
  - bmlib 改动
  - bmrt 改动
- 准备 BM1688/CV186AH 运行环境
- 其他注意事项

7.4.1 获取 BM1688/CV186AH SOPHONSDK

BM1688/CV186AH 的 SOPHONSDK 与 BM1684/BM1684X 的 SOPHONSDK 不是同一套，所以需要另外获取:

```
pip3 install dfss --upgrade
python3 -m dfss --url=open@sophgo.com:/gemini-sdk/gemini_sdk_edge_v1.5.1_offical_release.tar.
↪gz
```

下载后使用工具解压，可以得到各个模块的安装包、示例程序，SDK 目录结构如下:

文件夹名	备注
sophon-img	SoC 模式安装包等
sophon_media	支持 SOPHON 设备硬件加速的多媒体库
tpu-mlir	TPU 编译器工具链
tpu-perf	模型性能和精度验证工具包
sophon-stream	基于 pipeline 的高性能推理框架
sophon-demo	针对单模型或者场景的综合例程
sophon-sail	对底层接口进行 C++/Python API 封装的接口库
isp-tools	ISP 各个模块的参数调节工具
doc	各个模块的文档资料合集

7.4.2 准备 BM1688/CV186AH 的 bmodel

重新编译能在 BM1688/CV186AH 上运行的 bmodel，需要使用能够支持 BM1688/CV186AH 的 tpu-mlir。

一般来说，有以下几个步骤：

1. 准备一台 x86 ubuntu，相应的规格参数，取决于需要编译的模型大小，建议运行内存 32GB 以上，这样足够编译绝大多数模型。

2. 运行如下命令，搭建 tpu-mlir 环境：

```
# 拉取latest版本docker，目前对应tpuc_dev:v3.
↪2，如果您的环境已经有这个image了，也需要重新执行以下命令。
docker pull sophgo/tpuc_dev:latest

# 这里将本级目录映射到docker内的/workspace目录，
↪用户需要根据实际情况将demo的目录映射到docker里面
# myname只是举个名字的例子，请指定成自己想要的容器的名字
docker run --privileged --name myname -v $PWD:/workspace -it sophgo/tpuc_
↪dev:latest

# 此时已经进入docker，并在/workspace目录下

# 通过pip下载tpu_mlir，如果下不下来，可以使用清华源加速。
pip install tpu_mlir

# TPU-
↪MLIR在对不同框架模型处理时所需的依赖不同。对于onnx或torch生成的模型文件，使用类似下面命令安装
pip install tpu_mlir[onnx]
pip install tpu_mlir[torch]

# 目前支持五种配置：onnx, torch, tensorflow, caffe,F
↪paddle。可使用一条命令安装多个配置，也可直接安装全部依赖环境：
pip install tpu_mlir[onnx,torch]
pip install tpu_mlir[all]
```

3. 编译模型：

   只要将以前的编译命令或脚本中 model_deploy.py 部分的 --chip 或 --processor ，参数值更改成 bm1688/cv186x，其他的部分都不用改动，运行之后就可以获取能在 BM1688/CV186AH 上运行的 bmodel。

4. 编译 BM1688 双核模型：

   BM1688 由两个相同的核组成，支持一个模型的数据切分到两个核上面跑，tpu-mlir model_deploy.py 为 BM1688 新增了 --num_core 参数，如果您想要一个模型跑两个核，可以在 model_deploy.py 中新增 --num_core 2 参数，这样就编译出来的模型，会由两个核来执行。

5. 量化注意事项：

   以前的 calibration_table 和 qtable 均是可以复用的，如果在 mlir 编译 bmodel 的过程中出现类似算子不支持的报错，或者量化出来的模型精度不好，再重

新生成 calibration_table 或 qtable。

7.4.3 准备编译依赖的 SDK

如果您使用 C++ 编程，您仍然需要获取 sophon-img 目录下的 libsophon_soc_${x.y.z}/_aarch64.tar.gz 和 sophon_media 目录下的 sophon-media-soc_${x.y.z}/_aarch64.tar.gz 发布包，作为交叉编译依赖的头文件和库文件。

如果以前有手动链接 **bmion/bmjpulite/bmjpuapi/bmvpulite/bmvpuapi/bmvideo/bmvppapi** 等库，可以直接去掉，这些库其实并没有暴露接口给用户，它们是供底层调用的。

和 BM1684/BM1684X 有一点不同的是，目前 sophon_media 中提供的库还依赖 libisp 模块，如果您使用 BM1688/CV186AH 的 GeminiSDK1.3 以上版本，您还需要做这些操作：

从 sdk 中获取 sophon-img/bsp-debs/目录下的 sophon-soc-libisp_${x.y.z}_arm64.deb，然后运行如下命令：

```
dpkg -x sophon-soc-libisp_${x.y.z}_arm64.deb sophon-libisp
cp -rf sophon-libisp/opt/sophon/sophon-soc-libisp_${x.y.z}/lib ${soc-sdk}
↪#将libisp的库添加到已有的依赖库中。
```

如果您使用 Python 编程，那只需要重新编译 sophon-sail 的.whl 安装包，编译流程与之前相同，参考 sophon-sail 开发指南即可。BM1688 SoC 上自带 libsophon 和 sophon_media 的 runtime。

7.4.4 参照接口变化修改部分代码：

BM1688/CV186AH SOPHONSDK 提供的接口相比 BM1684/BM1684X SDK 有一些变化，下面将分模块分别进行说明，并给出一套应用层代码兼容两套 SDK 的兼容性解决办法：

7.4.4.1 头文件改动

1. bmcv_api.h 已废弃，兼容性解决办法如下：

```
#if !(BMCV_VERSION_MAJOR > 1)
#include <bmcv_api.h>
#endif
```

7.4.4.2 bmcv 改动

1. 函数参数变为指针：

```
DECL_EXPORT bm_status_t bm_image_destroy(bm_image image)
```

变为

```
DECL_EXPORT bm_status_t bm_image_destroy(bm_image *image)
```

兼容性解决办法如下：

1. 在公共头文件中添加如下代码：

```
#if BMCV_VERSION_MAJOR > 1
   static inline bm_status_t bm_image_destroy(bm_image& image){
   return bm_image_destroy(&image);
   }
#endif
```

2. 函数名称变化：

```
bm_image_dev_mem_alloc
```

变为

```
bm_image_alloc_dev_mem
```

兼容性解决办法：

1. 在公共头文件中封装一个相同功能的旧接口。

3. 接口名称修正：

```
bm_image_dettach_contiguous_mem
```

修正为

```
bm_image_detach_contiguous_mem
```

兼容性解决办法：

1. 在公共头文件中封装一个相同功能的旧接口。

4. 结构体名称修正

```
bmcv_padding_atrr_t
```

修正为

```
bmcv_padding_atrr_t
```

兼容性解决办法：

1. 在公共头文件中添加 typedef bmcv_padding_attr_t bmcv_padding_atrr_t; 。

5. 接口废弃：

```
bmcv_image_crop
```

兼容性解决办法：

1. 使用 bmcv_image_vpp_convert 来代替。
2. 或者在公共头文件使用 bmcv_image_vpp_convert 封装一个 bmcv_image_crop 接口。

6. 数据类型废弃：

```
DATA_TYPE_EXT_4N_BYTE_SIGNED
DATA_TYPE_EXT_4N_BYTE
```

兼容性解决办法：

1. 4N 数据类型在以前的 SDK 中也很少使用，不建议再使用 4N 数据类型，改用 1N 数据类型。

7. 内存布局改动导致的接口功能改动：

```
bm_image_alloc_dev_mem(heap_id = 2)
bm_image_alloc_dev_mem_heap_mask(heap_mask = 1 << 2)
bm_image_alloc_contiguous_mem_heap_mask(heap_mask = 1 << 2)
```

BM1688/CV186AH 上只有两个 heap，原来放在 heap2 的内存需要挪到 heap1，否则分配失败。如上接口的 heap_id/heap_mask 参数需要改为：

```
bm_image_alloc_dev_mem(heap_id = 1)
bm_image_alloc_dev_mem_heap_mask(heap_mask = 1 << 1)
bm_image_alloc_contiguous_mem_heap_mask(heap_mask = 1 << 1)
```

8. 暂未支持的接口：

```
bmcv_base64_enc()
bmcv_base64_dec()
bmcv_faiss_indexflatIP()
bmcv_nms()
bmcv_nms_ext()
bmcv_fft_1d_create_plan()
bmcv_fft_2d_create_plan()
bmcv_fft_execute()
bmcv_fft_execute_real_input()
bmcv_fft_destroy_plan()
```

7.4.4.3 ffmpeg 改动

ffmpeg 版本从 4.1 升级到了 6.0，由此引入了一些 ffmpeg 本身的改动，如下：

1. 接口废弃：

```
av_register_all
```

兼容性解决办法：

1. 去掉
2. 或者封装一个 av_register_all。

```
avcodec_decode_video2
avcodec_encode_video2
```

兼容性解决办法：

1. 按照 ffmpeg 文档，改成 avcodec_send_packet 和 avcodec_receive_frame 的形式。
2. 可以自己封装一个 avcodec_decode_video2，参考 https://github.com/sophgo/sophon-demo/blob/release/include/ff_decode.hpp
3. 可以自己封装一个 avcodec_encode_video2，参考 sophon-sail src/internal.h

2. 结构体或函数类型变为 const：

```
struct AVOutputFormat *oformat;
AVInputFormat* av_find_input_format
AVCodec* avcodec_find_decoder
AVOutputFormat* av_guess_format
AVCodec* avcodec_find_decoder_by_name
AVCodec* avcodec_find_decoder_by_name
```

变为

```
const struct AVOutputFormat *oformat
const AVInputFormat* av_find_input_format
const AVCodec* avcodec_find_decoder
const AVOutputFormat* av_guess_format
const AVCodec* avcodec_find_decoder_by_name
const AVCodec* avcodec_find_decoder_by_name
```

兼容性解决办法：

1. 使用 const_cast 去掉 const 修饰符。

3. 结构体成员变量变化：

```
AVStream->codec
```

变为

```
AVStream->codecpar
```

兼容性解决办法：

1. avcodec_parameters_to_context 重新获取上下文，avcodec_parameters_from_context 把上下文内容重新拷贝到流中。

7.4.4.4 bmlib 改动

1. 新增接口：

```
DECL_EXPORT bm_status_t bm_thread_sync_from_core(bm_handle_t handle, int core_id);
```

功能描述：

可以指定只 sync 某个核，在每个核各跑一个模型时会用到。

2. 内存布局改动导致的接口功能改动：

```
bm_malloc_device_byte_heap(heap_id = 2)
bm_malloc_device_byte_heap_mask(heap_mask = 1 << 2)
```

BM1688/CV186AH 上只有两个 heap，原来放在 heap2 的内存需要挪到 heap1，否则分配失败。如上接口的 heap_id/heap_mask 参数需要改为：

```
bm_malloc_device_byte_heap(heap_id = 1)
bm_malloc_device_byte_heap_mask(heap_mask = 1 << 1)
```

7.4.4.5 bmrt 改动

1. 新增接口：

```
DECL_EXPORT bool bmrt_launch_tensor_multi_cores(void *p_bmrt,
                                                const char *net_name,
                                                const bm_tensor_t input_tensors[],
                                                int input_num,
                                                bm_tensor_t output_tensors[],
                                                int output_num,
                                                bool user_mem,
                                                bool user_stmode,
                                                const int *core_list,
                                                int core_num);
```

功能描述：

该接口可以将单核模型指定在不同的核上运行，具体请查看 SOPHONSDK 的 bmruntime 接口文档或源码。

7.4.5 准备 BM1688/CV186AH 运行环境

BM1688/CV186AH 的刷机步骤和 BM1684/BM1684X 相同，具体可以看相关的产品使用手册，在获取 BM1688/CV186AH SOPHONSDK 章节下载的 SDK 中，sophon-img/sdcard.tgz 文件就是刷机包。

准备好运行环境之后，就可以将编译好的程序拷贝到 BM1688/CV186AH SoC 上执行了。

7.4.6 其他注意事项

1. bmrt、bmcv 相关的操作，最好把各自分配的内存隔离开，bmrt 需要用到的内存放在 heap0（也就是 npu heap），bmcv 需要用到的内存则放在 heap1（也就是 vpp heap）。
2. 由于 BM1688 的内存减小，ddr 是全 interleave 的，没有带宽分配问题，因此不再预留 vpu heap，多媒体模块共用 vpp heap。
3. bm1688 的 sdk 目前只把 ddr 划分成 npu、vpp 两个 heap，查看方法改为：

   ```
   sudo cat /sys/kernel/debug/ion/cvi_vpp_heap_dump/summary
   sudo cat /sys/kernel/debug/ion/cvi_npu_heap_dump/summary
   ```

4. 内存布局修改工具：https://doc.sophgo.com/sdk-docs/v23.09.01-lts/docs_latest_release/docs/SophonSDK_doc/zh/html/appendix/2_mem_edit_tools.html

   可以兼容 BM1688/CV186AH Gemini v1.5 以上 SDK，vpu heap 需要设置为 0。