

第三方算法接入 SDK2参考文档

本指南分为

1. bmodel编译指南
2. 1688平台APP编译指南
3. 1688平台ThingsSDK样例说明

三个步骤，描述了第三方算法接入到 `ThingSense平台` `SDK2` 框架的流程规范

1. bmodel编译指南

此部分基于[算能 TPU-MLIR开发参考手册](#) LTS23.09版本

此部分编译操作默认位于已经安装docker环境的Linux服务器上,使用docker容器进行编译

1.1 配置编译容器并启动

创建工作目录并进入 其中 `yourpath` 为自行设定的路径

```
mkdir -p /yourpath/sophgo_bmodel_compile && cd /yourpath/sophgo_bmodel_compile
```

1.1.1 使用 docker compose 构建容器

创建 `docker-compose.yml` 文件 和 `workspace` 数据目录

```
mkdir workspace && touch docker-compose.yml
```

将以下内容写入 `docker-compose.yml` 文件 其中 `** container_name` **为容器名 可自定义

```
version: "3"
services:
  tpuc_dev:
    image: sophgo/tpuc_dev:v3.2
    container_name: sophgo_bmodel_compile # 容器名可更改
    privileged: true
    volumes:
      - $(pwd)/workspace:/workspace:rw
    tty: true
    stdin_open: true
```

使用 命令启动容器

```
docker compose up -d
```

输出如下 容器即成功启动

```
[+] Running 2/2
✓ Network soph_default          Created           0.1s
✓ Container sophgo_bmodel_compile Started
```

1.1.2 使用 docker命令 构建容器

如果没有 `docker compose` 工具 可以采用 命令形式构建容器

创建 `workspace` 数据目录

```
mkdir workspace
```

使用命令启动容器 其中 `--name` 为容器名 可自定义

```
docker run -d --name sophgo_bmodel_compile --privileged -v "$(pwd)/data:/workspace:rw" sophgo/tpuc_dev:v3.2 tail -f /dev/null
```

使用命令查看 容器是否启动成功

```
docker ps -a | grep sophgo_bmodel_compile
```

输出如下 容器即成功启动

```
b19be297da3f sophgo/tpuc_dev:v3.2 "tail -f /dev/null" 6 seconds ago Up 4 seconds sophgo_bmodel_compile
```

1.2 进入编译容器并安装编译环境

使用命令 进入容器

```
docker exec -it sophgo_bmodel_compile bash
```

输出如下即进入容器成功

```
root@61a1b52e7044:/workspace#
```

以下操作默认在容器内部进行

复制 `tpu_mlir-1.11b0-py3-none-any.whl` 到 容器内部(文件已提供在压缩包内)

复制到宿主机的 /yourpath/sophgo_bmodel_compile/workspace 即可

查看复制进来的文件

```
root@e25bcb3f14c9:/workspace# ls
tpu_mlir-1.11b0-py3-none-any.whl
root@e25bcb3f14c9:/workspace#
```

安装

```
root@e25bcb3f14c9:/workspace# pip install tpu_mlir-1.11b0-py3-none-any.whl
Processing ./tpu_mlir-1.11b0-py3-none-any.whl
.....
Installing collected packages: tpu-mlir
Successfully installed tpu-mlir-1.11b0
root@e25bcb3f14c9:/workspace#
```

环境已经安装完成

1.3 模型编译

此例程基于yolov8 转换bmodel模型

[更多参数请参考算能文档](#)

1.3.1 ONNX模型转换mlir中间文件

使用 model_transform 进行转换

```
model_transform.py \
    --model_name test \
    --model_def ./test.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 output0 \  
--mlir test.mlir
```

输出如下

```
root@e25bcb3f14c9:/workspace# model_transform.py \  
--model_name test \  
--model_def ./test.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 output0 \  
--mlir test.mlir
```

```
2025/01/23 17:54:44 - INFO : TPU-MLIR v1.11.beta.0-20240918
```

```
2025/01/23 17:54:44 - INFO :
```

```
-----  
| preprocess:                                     |  
|   (x - mean) * scale                           |  
'-----'
```

config Preprocess args :

```
resize_dims      : same to net input dims  
keep_aspect_ratio : True  
keep_ratio_mode  : letterbox  
pad_value        : 0  
pad_type         : center
```

```
-----  
mean              : [0.0, 0.0, 0.0]  
scale             : [0.0039216, 0.0039216, 0.0039216]  
-----
```

```
pixel_format      : rgb
channel_format    : nchw
```

```
.....
2025/01/23 17:54:45 - INFO : Save mlir file: test_origin.mlir
[Running]: tpuc-opt test_origin.mlir --shape-infer --canonicalize --extra-optimize -o test.mlir
.....
[Success]: tpuc-opt test_origin.mlir --shape-infer --canonicalize --extra-optimize -o test.mlir
2025/01/23 17:54:45 - INFO : Mlir file generated:test.mlir
```

使用 命令查看

```
root@e25bcb3f14c9:/workspace# ls -l
total 362804
-rw-r--r-- 1 root root 44733239 Jan 23 17:54 final_opt.onnx
-rw-r--r-- 1 root root 67450 Jan 23 17:54 test.mlir
-rw-rw-r-- 1 1012 1013 44733239 Jan 23 17:52 test.onnx
-rw-r--r-- 1 root root 173225 Jan 23 17:54 test_opt.onnx.prototxt
-rw-r--r-- 1 root root 81857 Jan 23 17:54 test_origin.mlir
-rw-r--r-- 1 root root 44701080 Jan 23 17:54 test_top_f32_all_weight.npz
-rwxrwxrwx 1 1012 1013 237001671 Jan 23 17:46 tpu_mlir-1.11b0-py3-none-any.whl
root@e25bcb3f14c9:/workspace#
```

其中 test.mlir 即为需要的中间文件

1.3.2 mlir中间文件转换bmodel(fp16,fp32)

使用 model_deploy 转换 fp32 精度 模型

```
model_deploy.py \
  --mlir test.mlir \
  --quantize F32 \
  --chip bm1688 \
  --model test_fp32.bmodel \
```

```
--num_core 2
```

输出如下

```
root@e25bcb3f14c9:/workspace# model_deploy.py \
--mlir test.mlir \
--quantize F32 \
--chip bm1688 \
--model test_fp32.bmodel \
--num_core 2
2025/01/23 18:00:59 - INFO : TPU-MLIR v1.11.beta.0-20240918
2025/01/23 18:00:59 - INFO :
load_config Preprocess args :
  resize_dims      : [640, 640]
  keep_aspect_ratio : True
  keep_ratio_mode   : letterbox
  pad_value         : 0
  pad_type          : center
  input_dims        : [640, 640]
-----
  mean              : [0.0, 0.0, 0.0]
  scale             : [0.0039216, 0.0039216, 0.0039216]
-----
  pixel_format      : rgb
  channel_format     : nchw

[Running]: tpuc-opt test.mlir --processor-assign="chip=bm1688 mode=F32
.....
[Success]: mv compiler_profile_0.[td][xa]t test_fp32.bmodel.compiler_profile_0.txt
[Running]: mv net_0.profile test_fp32.bmodel.net_0.profile
[Success]: mv net_0.profile test_fp32.bmodel.net_0.profile
root@e25bcb3f14c9:/workspace#
```

使用 命令查看

```

root@e25bcb3f14c9:/workspace# ls -l
total 458172
-rw-r--r-- 1 root root 963098 Jan 23 18:01 compiler_profile_1.txt
-rw-r--r-- 1 root root 44733239 Jan 23 17:54 final_opt.onnx
-rw-r--r-- 1 root root 67450 Jan 23 17:54 test.mlir
-rw-rw-r-- 1 1012 1013 44733239 Jan 23 17:52 test.onnx
-rw-r--r-- 1 root root 285739 Jan 23 18:01 test_bm1688_f32_final.mlir
-rw-r--r-- 1 root root 82578 Jan 23 18:00 test_bm1688_f32_tpu.mlir
drwxr-xr-x 2 root root 4096 Jan 23 18:01 test_fp32
-rw-r--r-- 1 root root 47943624 Jan 23 18:01 test_fp32.bmodel
-rw-r--r-- 1 root root 1008519 Jan 23 18:01 test_fp32.bmodel.compiler_profile_0.txt
-rw-r--r-- 1 root root 1385176 Jan 23 18:01 test_fp32.bmodel.json
-rw-r--r-- 1 root root 1265028 Jan 23 18:01 test_fp32.bmodel.net_0.profile
-rw-r--r-- 1 root root 173225 Jan 23 17:54 test_opt.onnx.prototxt
-rw-r--r-- 1 root root 81857 Jan 23 17:54 test_origin.mlir
-rw-r--r-- 1 root root 44701080 Jan 23 17:54 test_top_f32_all_weight.npz
-rw-r--r-- 1 root root 44701080 Jan 23 18:01 test_tpu_addressed_bm1688_f32_weight.npz
-rwxrwxrwx 1 1012 1013 237001671 Jan 23 17:46 tpu_mlir-1.11b0-py3-none-any.whl

```

其中 `test_fp32.bmodel` 即为需要的bmodel文件

使用 `model_deploy` 转换 `fp16` 精度 模型

```

model_deploy.py \
  --mlir test.mlir \
  --quantize F16 \
  --chip bm1688 \
  --model test_fp16.bmodel \
  --num_core 2

```

输出和 转换 `fp32` 基本一致

使用 命令查看

```

root@e25bcb3f14c9:/workspace# ls -l

```



```
total 458980
-rw-r--r-- 1 root root 859941 Jan 23 18:07 compiler_profile_1.txt
-rw-r--r-- 1 root root 44733239 Jan 23 17:54 final_opt.onnx
-rw-r--r-- 1 root root 67450 Jan 23 17:54 test.mlir
-rw-rw-r-- 1 1012 1013 44733239 Jan 23 17:52 test.onnx
-rw-r--r-- 1 root root 373148 Jan 23 18:07 test_bm1688_f16_final.mlir
-rw-r--r-- 1 root root 116027 Jan 23 18:07 test_bm1688_f16_tpu.mlir
-rw-r--r-- 1 root root 285739 Jan 23 18:07 test_bm1688_f32_final.mlir
-rw-r--r-- 1 root root 82578 Jan 23 18:07 test_bm1688_f32_tpu.mlir
drwxr-xr-x 2 root root 4096 Jan 23 18:07 test_fp16
-rw-r--r-- 1 root root 25571048 Jan 23 18:07 test_fp16.bmodel
-rw-r--r-- 1 root root 908358 Jan 23 18:07 test_fp16.bmodel.compiler_profile_0.txt
-rw-r--r-- 1 root root 1809662 Jan 23 18:07 test_fp16.bmodel.json
-rw-r--r-- 1 root root 1358916 Jan 23 18:07 test_fp16.bmodel.net_0.profile
drwxr-xr-x 2 root root 4096 Jan 23 18:01 test_fp32
-rw-r--r-- 1 root root 47943624 Jan 23 18:07 test_fp32.bmodel
-rw-r--r-- 1 root root 1008519 Jan 23 18:07 test_fp32.bmodel.compiler_profile_0.txt
-rw-r--r-- 1 root root 1385176 Jan 23 18:07 test_fp32.bmodel.json
-rw-r--r-- 1 root root 1265028 Jan 23 18:07 test_fp32.bmodel.net_0.profile
-rw-r--r-- 1 root root 173225 Jan 23 17:54 test_opt.onnx.prototxt
-rw-r--r-- 1 root root 81857 Jan 23 17:54 test_origin.mlir
-rw-r--r-- 1 root root 44701080 Jan 23 17:54 test_top_f32_all_weight.npz
-rw-r--r-- 1 root root 22396380 Jan 23 18:07 test_tpu_addressed_bm1688_f16_weight.npz
-rw-r--r-- 1 root root 44701080 Jan 23 18:07 test_tpu_addressed_bm1688_f32_weight.npz
-rwxrwxrwx 1 1012 1013 237001671 Jan 23 17:46 tpu_mlir-1.11b0-py3-none-any.whl
```

其中 `test_fp16.bmodel` 即为需要的bmodel文件

1.4 对转换完成的bmodel进行查看

可使用 `model_tool` 查看bmodel的基本信息 包括模型的编译版本，编译日期，模型中网络名称，输入和输出参数等

```
model_tool --info test_fp16.bmodel
```

输出如下

```
root@e25bcb3f14c9:/workspace# model_tool --info test_fp16.bmodel
bmodel version: B.2.2+v1.11.beta.0-20240918
chip: BM1688
create time: Thu Jan 23 18:07:49 2025

kernel_module name: libbmtpulv60_kernel_module.so
kernel_module size: 2371984
=====
net 0: [test] static
-----
stage 0:
input: images, [1, 3, 640, 640], float32, scale: 1, zero_point: 0
output: output0_Concat_f32, [1, 6, 8400], float32, scale: 1, zero_point: 0

device mem size: 38998016 (weight: 22614016, instruct: 0, runtime: 16384000)
host mem size: 0 (weight: 0, runtime: 0)
root@e25bcb3f14c9:/workspace#
```

2. 1688平台APP编译指南

此部分默认在1688 SOC 模式下进行开发 且已经获取 SDK2 1688平台开发包
目录结构如下

```
1688_comm
├─ framework
│   └─ app/ 业务代码
│   └─ config/ 配置文件
│   └─ docs/ 文档
│   └─ includes/ 依赖文件
│   └─ libs/ 编译后的whl包
```

```
|   |— mods/ 使用的模型文件(模型文件请替换)
|   |— utils/ 对应的平台的依赖库文件
|— tools/
|   |— build_tools/ 编译工具
```

2.1 开发APP

修改app/ 目录下的业务代码

修改config/ 目录下的配置文件

SDK 的使用 和 config 配置参考下文

1688平台ThingsSDK样例说明

2.2 打包/测试APP

业务逻辑修改完成后 参考下文使用编译工具打包并测试APP

2.2.1 打包APP

工具位于

```
tools/build_tools/app_docker_build.sh
```

进入工具目录

```
cd tools/build_tools/
```

目录结构如下

```
tools/build_tools/  
├─ app.dockerfile  
├─ app_docker_build.sh  
├─ app_start.sh  
└─ docker_compose.yml
```

其中 docker_compose.yml 内容 需要修改

```
services:  
  test_image:      # 修改为实际的service名称  
    image: test_image:v1.0 # 修改为实际的APP名称和版本号  
    privileged: true  
    build:  
      context: ../../  
      dockerfile: tools/build_tools/app.dockerfile
```

执行打包命令

```
./app_docker_build.sh
```

输出如下

```
/your/path/1688_comm  
开始编译 app镜像  
[+] Building 0/0  
[+] Building 0/1image  
Building  
...  
[+] Building 42.2s (17/17)  
✓ Service your_app_name Built  
app镜像 编译完成
```

编译完成后 可以使用 `docker images` 命令

```
docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
your_app_name	v1.0	5b8ae48cf5c6	1 minutes ago	911MB

查看编译后的镜像

2.2 测试APP

通过上文的 `docker images` 命令 可以看到编译后的镜像

使用 如下命令 启动APP 测试

```
docker run -it --rm --privileged your_app_name:your_app_version
```

如果使用的是官方samples且config 配置正确 则输出预期如下

```
INFO 2025-01-03 15:52:27,283 : 启动ws线程
INFO 2025-01-03 15:52:27,284 : ws://xxx
INFO 2025-01-03 15:52:27,285 : 开始连接
decoder ctor: filepath=rtsp:your_rtsp_url
open filepath=rtsp:your_rtsp_url
INFO 2025-01-03 15:52:28,648 : loading model your_model_path...
INFO 2025-01-03 15:52:29,307 : model loaded successfully, class names:['your_class_name']
INFO 2025-01-03 15:52:29,346 : ws连接成功
INFO 2025-01-03 15:52:29,565 : 检测结果: [YoloResult(...)]
INFO 2025-01-03 15:52:29,569 : 发送报警数据 xxx
INFO 2025-01-03 15:52:29,601 : 发送图片数据 0
INFO 2025-01-03 15:52:29,604 : 发送图片数据 1
INFO 2025-01-03 15:52:29,607 : 发送图片数据 2
INFO 2025-01-03 15:52:29,610 : 发送图片数据 3
```

2.3 发布APP

本地编译测试通过后,发布APP到对应的官方仓库

使用命令

```
docker tag your_app_name:your_app_version HARBOR_URL/your_app_name:your_app_version
```

```
docker push HARBOR_URL/your_app_name:your_app_version
```

对应的HARBOR 仓库地址 请联系官方获取

3. 1688平台ThingsSDK样例说明

以官方sample 为例

业务代码 `app/start.py`

配置文件 `config/app.yml`

注 使用管理云下发部署时, 配置文件会默认位于容器内/`ths/config/xxx`(用户上传的配置文件名).`yml`, 此时配置默认读取路径为/`ths/config/app.yml` 入口文件为 `/workspace/app/start.py` 如需变更请自行修改代码

3.1 config 配置

目录的基本结构:

```
config/  
├─ app.yml
```

app.yml是app的配置文件

该文件包含 算法的输入，输出，自定义参数，其结构如下：

注: log config input output 为固定字段，请勿删除

形如 #desc:算法输出数据点位列表,type:checkbox,model:checkbox,editable:no,isNull:no 的注释
代表 在上传配置文件后,使用 管理云 进行配置时, 该字段的渲染描述文件 其含义如下

#desc:代表该字段的含义 eg. 算法输出数据点位列表

#type:代表该字段的类型 eg. checkbox 多选框|list 下拉菜单|input 输入框|switch 开关项

#model:代表该字段的操作模式 eg. text 文本编辑|switch 开关|single 单选|multiply 多选

#editable:代表该字段是否可编辑 eg. no 不可编辑|yes 可编辑

#isNull:代表该字段是否可为空 eg. no 不可为空|yes 可为空

log: (固定字段，请勿删除)

config: (用户自定义字段 按照yaml 语法进行配置 其中alarm 代表为算法所能检测的所有目标名 如果无可忽略)

#desc:算法输出数据点位列表,type:checkbox,model:checkbox,editable:no,isNull:no

alarm: #算法所能检测的所有目标名

- person
- smoke
- face_ok
- fire
- smoke1
- invade
- face_invalid

output: (固定字段，代表算法输出的数据点位 此字段在管理云选择后会自动生成，根据实际情况进行修改)

#desc:数据点位服务,type:list,model:multiply,editable:no,isNull:no

data_point:

- point_name: point-01 (数据点位名)

tgt_addr: ws://192.168.2.103:8011/iotdt-compute/websocket/report/776d4af0f90bed19c0589a328e756de5/test (数据点位服务地址)

input: (固定字段，代表算法输入的摄像头信息 此字段在管理云选择后会自动生成，根据实际情况进行修改)

```
#desc:摄像头与节点映射ID列表,type:list,model:multiply,editable:no,isnull:no
video: (摄像头信息列表 固定结构 三种不同的输入方式 RTSP/CAM_ID/GB)
- camera_name: camera-01
  src_addr: rtsp://uuuuu:admin*123@192.168.59.230:554/video1
  src_type: RTSP
  alarm: #此摄像头所需要检测的目标(可选字段 是 config/alarm 字段的子集)
    - person
    - smoke
- camera_name: camera-02
  src_addr: 1
  src_type: CAM_ID
  uni_decode: true (是否为统一解码 此字段仅在CAM_ID模式下有效!)
  addr_variable: true (输入地址流是否可变 此字段仅在CAM_ID模式下有效!)
- camera_name: camera-03
  src_addr: '{"deviceCode':"51010500002000000003',"channelCode':"51010500002000000003"}'
  src_type: GB
```

3.2 ThingsSDK 调用

以官方sample 为例

```
from sophon.sail import Bmcv
from sophon.sail import Handle

from ThingsSDK import Configer, logger
from ThingsSDK import Predict

def AppMain():
    # config = Configer(useCloud=True) # 使用管理云下发部署时 使用此行
    config = Configer(useCloud=True, local_config_path='/workspace/config/app.yml') # 使用本地测试时 使用local_config_path
    指定配置文件路径

    # 获取视频对象
    vhd11 = config.getVideoHdl('camera-01') # 此处的camera-01 为app.yml 中的camera_name
```



```

# 获取输出对象
o1 = config.getOutput('point-01') # 此处的point-01 为app.yml 中的data_point 中的point_name

handle = Handle(0)
bmcv = Bmcv(handle)

# 加载模型
model1 = Predict()
model1.LoadModel('/workspace/mods/xxx.bmodel',
                 '/workspace/mods/xxx.txt')

while True:
    # 每隔18帧获取一次视频帧
    for i in range(17):
        vhd1.get_frame()
    # 获取视频帧
    frame = vhd1.get_frame()
    # 预测
    result = model1.predict(frame)
    # 输出结果
    logger.info(f'检测结果: {result}')
    if result:
        # 上报结果到计算云
        o1.push('TEXT', result)
        o1.push('BINARY', frame)

    # 保存图片 # 此行代码为保存图片到本地 如需保存请自行修改
    bmcv.imwrite(f'./{frame.camera}.{frame.frame_name}.jpg', frame)
if __name__ == '__main__':
    AppMain()

```

用到的api 如下

```

# config
# 获取配置对象 useCloud 为True时 自动加载config中配置的输入/输出对象

```

```
config = Configer(useCloud=True,local_config_path='/workspace/config/app.yml')

config = Configer(useCloud=True) # 使用管理云下发部署时 使用此行
# input
vhd11 = config.getVideoHdl('camera-01') # 获取摄像头对象
frame = vhd11.get_frame() # 获取视频帧
# predict
model1 = Predict() # 获取推理对象
model1.LoadModel('/workspace/mods/xxx.bmodel', '/workspace/mods/xxx.txt') # 加载模型
result = model1.predict(frame) # 推理
# output
o1 = config.getOutput('point-01') # 获取输出对象
o1.push('TEXT', result) # 上报推理结果到计算云
o1.push('BINARY', frame) # 上报图片到计算云
```