PPL 快速入门指南

SOPHGO

2025 年 03 月 20 日

Contents

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

CHAPTER 1

PPL 简介

本文档旨在让开发者快速了解 PPL 的开发流程和步骤，及相关的测试步骤；不涉及具体的 API 使用参考，如需详细了解，可参考 doc 目录下的《PPL 开发参考手册.pdf》。

1.1 PPL 概述

PPL 针对 TPU 编程的专用编程语言 (DSL)。开发者可以通过 PPL 编写在设备（TPU）上运行的自定义算子。PPL 代码经过编译后，会生成在能在 TPU 上运行的 c 代码。用户可以将 PPL 生成的代码编译成 so，在主机的应用中利用 runtime 的动态加载接口进行调用，实现利用 TPU 加速计算的目的。

下面是 PPL 开发包的目录结构：

```
├─── bin/
|    ├─── ppl-compile    # PPL 编译工具：将 ppl 的 host 代码转换成 device 代码
├─── doc/
|    ├─── PPL快速入门指南.pdf
|    └─── PPL开发参考手册.pdf
├─── docker/            # PPL 项目镜像生成脚本目录
|    ├─── Dockerfile     # PPL 编译环境的 Dockerfile
|    └─── build.sh       # Dockerfile 的编译脚本
├─── envsetup.sh        # 环境初始化脚本，设定环境变量
├─── examples/          # PPL 代码示例
├─── inc/               # PPL 代码依赖的头文件
├─── python/            # 使用 python 开发的辅助工具
├─── runtime/           # 运行时库以及 ppl 生成代码依赖的辅助函数和脚本
|    ├─── bm1684x/       # bm1684x runtime 库
|    |    ├─── lib/       # device 代码依赖的库
|    |    ├─── libsophon/  # device 代码依赖的头文件和库
|    |    └─── TPU1686/    # device 代码依赖的头文件
|    ├─── bm1688/        # bm1688 runtime 库
|    |    ├─── lib/       # device 代码依赖的库
|    |    ├─── libsophon/  # device 代码依赖的头文件和库
|    |    └─── TPU1686/    # device 代码依赖的头文件
|    ├─── bm1690/        # bm1690 runtime 库
|    |    ├─── lib/       # device 代码依赖的库
|    |    ├─── TPU1686/    # device 代码依赖的头文件
|    |    └─── tpuv7-emulator/   # device 代码依赖的头文件和库
|    ├─── sg2380/        # sg2380 runtime 库
```

(续下页)

(接上页)

```
|   |   ├── include/   # device 代码依赖的头文件
|   |   ├── qemu/      # device 代码依赖的头文件和库
|   |   ├── samples/   # device 代码依赖的链接配置文件
|   |   ├── scripts/   # ppl 提供的代码运行脚本文件
|   |   ├── sifive_x280mc8/   # device 代码依赖的库
|   |   └── TPU1686/   # device 代码依赖的库
|   ├── customize/     # ppl 提供的 device、host 端辅助函数
|   ├── kernel/        # device 代码依赖的头文件
|   └── scripts/       # ppl 提供的 cmake 模块文件
└── samples/           # samples
```

温馨提示：如果您是在 vscode ide 上使用 ppl 开发算子，

可以安装 MLIR Highlighting for VSCode 插件，该插件的功能为高亮 pl 文件、提示语法错误等，便于算子开发工作。

1.2 配置 PPL 环境

1.2.1 使用 ppl 提供的 docker 环境

```
cd docker
# 当前已处于ppl/docker目录下
./build.sh

cd ..
# 当前已处于ppl/目录下

docker run --privileged -itd -v $PWD:/work --name ppl sophgo/ppl:latest
docker exec -it ppl bash
cd /work
source envsetup.sh
```

1.2.2 使用 tpu-mlir 提供的 docker 环境

ppl 也可以在 TPU-MLIR 的 docker 环境中使用；TPU-MLIR docker 环境部署具体参考 TPU-MLIR 相关文档

1.2.3 不使用 docker 环境，直接安装依赖库

```
apt-get install -qy zlib1g-dev
apt-get install -qy gcc-11 g++-11
apt-get install -qy clang-14
apt-get install -qy cmake
apt-get install -qy ninja-build


apt-get install -qy python3.10
ln -s /usr/bin/python3.10 /usr/bin/python
apt-get install -qy python3-pip
pip install numpy==1.24.4 -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install plotly==5.18.0 -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install pqdm==0.2.0 -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install scipy==1.10.1 -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install tqdm==4.66.1 -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple
```

1.3 PPL 算子开发快速入门

本章节将使用 ppl 快速实现一个简单的 tensor 与常量相加的算子。若要深入了解 ppl 的使用方法，请参阅 PPL 算子开发 COP、PPL 开发参考手册等文档。

示例代码存放在 ppl/examples/quick_start/

1.3.1 c++ 前端

简单算子示例

代码路径 ppl/examples/quick_start/add.pl

```
#include "ppl.h" // PPL 代码依赖的头文件
using namespace ppl;

// 算子函数调用的入口需要用__KERNEL__定义
// ptr_res是存放输出数据的指针，ptr_inp是存放输入数据的指针，数据的shape是[N, C,
// H, W]
__KERNEL__ void add_kernel(fp32 *ptr_res, fp32 *ptr_inp, float scalar_c,
                           const int C, int W, const int block_w) {
  dim4 global_shape = {1, C, 1, W};
  // 使用gtensor来抽象输入输出数据
  auto in_gtensor = gtensor<fp32>(global_shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp32>(global_shape, GLOBAL, ptr_res);

  // 使用tensor来抽象TPU上的数据
  // 定义TPU上每次运算数据的block shape，同时用来进行内存分配
  dim4 local_block_shape = {1, C, 1, block_w};
  // 在TPU上申请local_block_shape大小的内存，用来存放输入数据
  auto in = tensor<fp32>(local_block_shape);
  // 在TPU上申请local_block_shape大小的内存，用来存放输出数据
  auto res = tensor<fp32>(local_block_shape);

  for (int idx_w = 0; idx_w < W; ++idx_w) {
    // enable_pipeline();
    // 当前循环w维度实际要运算的数据量
    int real_w = min(block_w, W - idx_w);
    dim4 local_real_shape = {1, C, 1, real_w};

    // 从tensor取实际shape的视图（tensor占用的TPU内存不变）
    auto in_sub = in.view(local_real_shape);
    auto res_sub = res.view(local_real_shape);

    // DDR上存取数据的偏移
    dim4 offset = {0, 0, 0, idx_w};
    // 通过 dma 将数据从DDR载入到TPU上
    dma::load(in_sub, in_gtensor.sub_view(local_real_shape, offset));
    // 使用TPU进行加法运算
    tiu::fadd(res_sub, in_sub, scalar_c);
    print("res:%s", to_string(res_sub));
    // 通过 dma 将结果从TPU搬运到DDR上
    dma::store(res_gtensor.sub_view(local_real_shape, offset), res_sub);
  }
}
```

验证算子输出

我们已经开发了一个 TPU 算子，现在我们校验下这个算子运算结果是否正确，我们可以添加一个测试函数调用算子，查看输出是否符合预期；测试函数与算子函数可以写在一个 pl 文件中。

```
// 测试函数需要加上__test__定义
__TEST__ void add() {
  int C = 8;
  int W = 64;
  int block_w = 32;
  float scalar_c = 0.25;
  dim4 shape = {1, C, 1, W};
  // 使用随机数填充输出
  fp32 *res = rand<fp32>(&shape);
  // 使用随机数范围[-32.21f, 32.32f]填充输入
  fp32 *inp = rand<fp32>(&shape, -32.21f, 32.32f);
  // 调用算子
  add_kernel(res, inp, scalar_c, C, W, block_w);
}
```

然后我们可以使用 ppl_compile.py 这个辅助脚本运行算子。

```
# 我们将前面的代码写入add.pl文件，使用 bm1684x 芯片测试算子
# --gen_test表示生成测试程序，源文件必须有__TEST__函数
ppl_compile.py --src add.pl --chip bm1684x --gen_test

# 也可以使用--gen_ref来测试
# ppl_compile.py --src add.pl --chip bm1684x --gen_ref
# --gen_ref选项会运行算子两次，第一次是经过编译优化后的算子，
# 第二次是未经过编译优化的算子，作为对照；如果两次运行结果不一样，
# 则说明存在问题，可能是算子写得不对，也可能是ppl编译存在问题
```

运行 ppl_compile.py 后，会在当前目录下生成 test_add 文件夹（" test_ " + pl 文件名不带后缀）；test_add/data 目录存放了输入输出数据，add_input.npz 为输入数据，add_tar.npz 为输出数据，如果使用的–gen_ref，还会生成对照程序输出 add_ref.npz；我们可以使用 npz_help.py 来查看 npz 数据

```
# npz_help.py dump xxx.npz会打印xxx.npz包含的数组名
# npz_help.py dump xxx.npz array_name可以打印数组数据
# 数组名是指针在kernel函数的序号，例如ptr_inp在kernel函数位置是1，则数组名为1
# 数组名只考虑指针的序号，不考虑其他参数，例如kernel函数参数为
# __KERNEL__ add_kernel(fp32 *ptr_res, float scale, fp32 *ptr_inp)
# 则ptr_inp的数组名还是1

# 查看add_kernel ptr_inp输入时的数据
npz_help.py dump test_add/data/add_input.npz 1

# 查看add_kernel ptr_res输出的数据
npz_help.py dump test_add/data/add_tar.npz 0
```

虽然可以使用 npz_help.py 来查看输入输出数据，但是对于复杂的运算还是不方便确认结果是否复合预期，所以我们通常写一个 python 脚本来进行验证，例如添加一个 add_check.py

```
# test_processor封装了npz读取、对比等功能
from tool.test_process import test_processor

# tpu_out即为add_input.npz内的数据；scalar和shape是用户自定义的参数，在调用时传入
@test_processor
def add_const(tpu_out, scalar, shape, **kwargs):
    # 获取ptr_inp的输入数据
    x = tpu_out["1"].reshape(shape)
    # 计算结果
    out = x + scalar
    # 将结果通过dict返回，因为需要比对ptr_res的结果，因此返回key为"0"
    # test_processor会将返回dict保存为torch.npz，然后与add_tar.npz进行比对
    return {"0": out}
```

（续下页）

(接上页)

```
# 调用python运算并进行比对，dir为add.pl去除后缀名
add_const(dir="add", scalar=0.25, shape=(1, 8, 1, 64))
```

运行

```
# 需要在test_add的存放目录下运行
python add_check.py
```

终端会打印 python 运算与 TPU 运算结果的余弦相似度和欧氏距离

如果 TPU 结果不正确，我们也可以在算子函数中，使用 print 函数打印中间结果

```
// 通过 dma 将数据从DDR载入到TPU上
dma::load(in_sub, in_gtensor.sub_view(local_real_shape, offset));
// 使用TPU进行加法运算
tiu::fadd(res_sub, in_sub, scalar_c);
print("res:%s", to_string(res_sub));
// 通过 dma 将结果从TPU搬运到DDR上
dma::store(res_gtensor.sub_view(local_real_shape, offset), res_sub);
```

```
ppl_compile.py --src add.pl --chip bm1684x --gen_test
```

这样就会在终端输出中间数据，方便确认哪一步结果不对

1.3.2 c++ 前端 pl+cpp 模式

如果 pl 文件是与 TPU-MLIR 或 Torch-TPU 框架集成的，我们可以使用 pl+cpp 模式来开发算子。这里 cpp 的作用主要是：

- 提供 tiling 函数；有时候需要对输入数据进行切分，寻找 TPU local 内存可以容纳的最大的 block size。
- 如果 pl 文件提供了多个算子，可以在 cpp 文件中选择需要的算子。
- 可以对输入参数做一些初始化功能。

实际上 pl+cpp 才是一个完整的算子，并且这个算子可以在 TPU-MLIR 和 Torch-TPU 框架中通用。

我们以 ppl/examples/cxx/pl_with_cpp 为例，介绍如何添加 cpp 文件。

cpp 文件示例

路径 pl_with_cpp/add_const.cpp

```
// 添加pl文件自动生成的头文件
#include "add_const_fp.h"

// 需要用extern C来定义入口函数
extern "C" {
// 如果pl文件提供了多个算子，可以提前定义函数指针，这样可以减少一些重复代码
// 注意pl文件中的指针类型需要用gaddr_t定义
using KernelFunc = int (*)(gaddr_t, gaddr_t, float, int, int, int, int, int,
                           bool);
// 添加入口函数，输入参数由用户自定义
int add_tiling(gaddr_t ptr_dst, gaddr_t ptr_src, float rhs, int N, int C, int H,
               int W, bool relu, int dtype) {
  KernelFunc func;
  // 根据输入数据类型，选择合适的算子
  if (dtype == 0) {
    func = add_const_f32;
  } else if (dtype == 1) {
```

(续下页)

(接上页)

```
    func = add_const_f16;
  } else if (dtype == 2) {
    func = add_const_bf16;
  } else {
    assert(0 && "unsupported dtype");
  }

  // 计算block size，可以将block size对齐到EU_NUM，
  // 这样可以减少内存分配失败的次数，并且因为TPU上的内存大部分是按照EU_NUM对齐的，
  // 所以不会影响到内存分配
  int block_w = align_up(N * C * H * W, EU_NUM);
  int ret = -1;
  while (block_w > 1) {
    ret = func(ptr_dst, ptr_src, rhs, N, C, H, W, block_w, relu);
    if (ret == 0) {
      return 0;
    } else if (ret == PplLocalAddrAssignErr) {
      // 当错误类型为PplLocalAddrAssignErr时，说明block size太大，
      // local 内存放不下，需要减小block size
      block_w = align_up(block_w / 2, EU_NUM);
      continue;
    } else if (ret == PplL2AddrAssignErr) {
      // 当错误类型为PplL2AddrAssignErr时，说明block size太大，
      // L2 内存放不下，需要减小block size，本示例没有分配L2内存，
      // 因此不会出现这个错误
      assert(0);
    } else {
      // 其他错误，需要debug
      assert(0);
      return ret;
    }
  }
  return ret;
}
}
```

注意事项

- add_const_fp.h 头文件中包含了一些错误码和芯片相关的参数定义：
- pl 文件中的指针需要使用 gaddr_t 类型定义

表 1: 内置错误码

参数名	说明
PplLocalAddrAssignErr	Local 内存分配失败
FileErr	
LlvmFeErr	
PplFeErr	AST 转 IR 失败
PplOpt1Err	优化 pass opt1 失败
PplOpt2Err	优化 pass opt2 失败
PplFinalErr	优化 pass final 失败
PplTransErr	代码生成失败
EnvErr	环境变量异常
PplL2AddrAssignErr	L2 内存分配失败
PplShapeInferErr	shape 推导失败
PplSetMemRefShapeErr	
ToPplErr	
PplTensorConvErr	
PplDynBlockErr	

表 2: 内置芯片参数

参数名	说明
EU_NUM	EU 数量
LANE_NUM	LANE 数量

如果我们想验证 pl+cpp 模式算子的结果，需要再加一个测试 cpp 文件

测试文件示例

路径 pl_with_cpp/add_const_test.cpp

```
// pl文件自动生成的头文件，必须放在最前面，包含了芯片类型的宏定义
#include "add_const_fp.h"
// 测试辅助函数
#include "host_test_utils.h"
// npz文件操作函数
#include "npz_helper.h"

// 需要extern申明pl+cpp模式算子的入口函数
extern "C" {
extern int add_tiling(gaddr_t ptr_dst, gaddr_t ptr_src, float rhs, int N, int C,
                      int H, int W, bool relu, int dtype);
}

int main() {
  int ret = 0;
  // 初始化设备
  ret = init_device();
  if (ret != 0) {
    return -1;
  }

  int N = 1;
  int C = 32;
  int H = 1;
  int W = 1024;

  dim4 src_shape = {N, C, H, W};
  // 申明输入输出npz文件
  cnpy::npz_t npz_in;
  cnpy::npz_t npz_out;

  // 申请内存
  mem_t src_mem = alloc_rand_mem(src_shape, DT_FP32, 0.0, 1000.0);
  mem_t dst_mem = alloc_rand_mem(src_shape, DT_FP32, 0.0, 0.0);

  // 将输入数据加入到输入npz，注意key为1，因为pl文件中ptr_src的在所有指针中索引是1
  npz_add(npz_in, "1", src_mem);
  // dump输入npz
  npz_save_input(npz_in);
  // 调用算子
  ret = add_tiling(dst_mem.dev_mem, src_mem.dev_mem, 1.0, N, C, H, W, 0, 0);
  if (ret != 0) {
    printf("add_tiling failed\n");
    return -1;
  }
  // 将数据从device端搬运到host端
  D2S(dst_mem);
  // 将输出数据加入到输出npz
  npz_add(npz_out, "0", dst_mem);
  // dump输出npz
```

(续下页)

(接上页)

```
  npz_save_output(npz_out);
  // 释放内存
  free_mem(src_mem);
  free_mem(dst_mem);
  // 释放设备
  release_device();
  return 0;
}
```

特别要注意，将输入输出数据加入到 npz 文件时，key 必须是指针在 pl 文件中的索引，例如 ptr_src 的索引是 1，ptr_res 的索引是 0；索引是去除非指针参数后的参数索引，例如：

```
//ptr_src的索引是1，ptr_dst的索引是0
int add_tiling(gaddr_t ptr_dst, gaddr_t ptr_src, float rhs, int N, int C,
            int H, int W, bool relu, int dtype)
//ptr_src的索引是1，ptr_dst的索引是0
int add_tiling(gaddr_t ptr_dst, float rhs, gaddr_t ptr_src, int N, int C,
            int H, int W, bool relu, int dtype)
```

数据类型定义如下：

```
typedef enum {
   DT_INT8   = (0 << 1) | 1,
   DT_UINT8  = (0 << 1) | 0,
   DT_INT16  = (3 << 1) | 1,
   DT_UINT16 = (3 << 1) | 0,
   DT_FP16   = (1 << 1) | 1,
   DT_BFP16  = (5 << 1) | 1,
   DT_INT32  = (4 << 1) | 1,
   DT_UINT32 = (4 << 1) | 0,
   DT_FP32   = (2 << 1) | 1,
   DT_INT4   = (6 << 1) | 1,
   DT_UINT4  = (6 << 1) | 0,
   DT_FP8E5M2 = (0 << 5) | (7 << 1) | 1,
   DT_FP8E4M3 = (1 << 5) | (7 << 1) | 1,
   DT_FP20   = (8 << 1) | 1,
   DT_TF32   = (9 << 1) | 1,
} data_type_t;
```

编译运行

同样可以通过 ppl_compile.py 来编译运行

```
ppl_compile.py --src examples/cxx/arith/add_const_fp.pl:examples/cxx/pl_with_cpp/add_const.
↪cpp:examples/cxx/pl_with_cpp/add_const_test.cpp --chip bm1684x
```

可以将 pl 和 cpp 文件路径用 “:” 连接起来传入 ppl_compile.py，ppl_compile.py 会自动将 pl 和 cpp 文件编译成一个 so 文件，并且会生成测试程序。此模式还支持一些其他参数，如下：

表 3: 参数

参数名	说明
–extra_cflags	额外的 cflags，使用 “:” 连接多个参数，例如：–extra_cflags “-DDEBUG:-O0”
–extra_ldflags	额外的 ldflags，使用 “:” 连接多个参数
–extra_include_paths	额外的 include 路径，使用 “:” 连接多个路径
–extra_link_paths	额外的 link 路径，使用 “:” 连接多个路径

校验结果

可以使用 pl 文件验证时的 python 脚本来校验

```
python examples/cxx/pl_with_cpp/add_const_fp.py
```

1.3.3 python 前端

代码路径 ppl/examples/quick_start/add_ppl.pl

简单算子示例

```
import torch
import ppl
import ppl.language as pl
import numpy as np
import functools


# 算子函数需要加上ppl.jit装饰器
@ppl.jit(debug=False)
def add_kernel(ptr_res, ptr_inp, scalar_c, C: pl.constexpr, W,
               block_w: pl.constexpr):

    # global tensor shape
    global_shape = [1, C, 1, W]

    # 使用gtensor来抽象输入输出数据
    in_gtensor = pl.gtensor(global_shape, pl.GLOBAL, ptr_inp)
    res_gtensor = pl.gtensor(global_shape, pl.GLOBAL, ptr_res)

    local_block_shape = [1, C, 1, block_w]
    for idx_w in range(0, W, block_w):
        # pl.enable_pipeline()
        real_w = min(block_w, W - idx_w)

        # local tensor real shape
        local_real_shape = [1, C, 1, real_w]

        # 使用tensor来抽象TPU上的数据
        # 设置TPU上每次运算数据的block shape，用来进行内存分配，以及实际shape
        src = pl.make_tensor(local_block_shape, ptr_inp.dtype,
                             local_real_shape)
        res = pl.make_tensor(local_block_shape, ptr_inp.dtype,
                             local_real_shape)
        # DDR上存取数据的偏移
        offset = [0, 0, 0, idx_w]
        # 通过 dma 将数据从DDR载入到TPU上
        pl.dma.load(src, in_gtensor.sub_view(local_real_shape, offset))
        # 使用TPU进行加法运算
        pl.fadd(res, src, scalar_c)
        # out = src + scalar_c
        # 打印res tensor内容
        print("res:%s", res)
        pl.dma.store(res_gtensor.sub_view(local_real_shape, offset), res)
```

验证输出

python 前端可以直接使用 torch 进行验证，我们在 py 文件中继续添加 CPU 运算，以及结果校验代码

```
# CPU 运算
def add_const_cpu(input, rhs):
    return input + rhs
```

(续下页)

(接上页)

```
atol = 1e-3  # 绝对容忍度
rtol = 1e-3  # 相对容忍度
torch.manual_seed(0)

# 输入参数设置
input_shape = [8, 64]
rhs = 0.25
block_w = 32
input_tensor = torch.randn(input_shape, dtype=torch.float)

# 获取torch CPU运算结果
output_cpu = add_const_cpu(input_tensor, rhs)
output_tpu = torch.empty(input_shape, dtype=torch.float)

# 调用TPU运行
# [(1,)]表示使用一个core来运算
add_kernel[(1, )](output_tpu, input_tensor, rhs, input_shape[0],
                  input_shape[1], 32)
# print(output_tpu)
# print(output_torch)
# 对比CPU与TPU结果
assert torch.allclose(output_cpu, output_tpu, atol=atol,
                      rtol=rtol), "add_const_fp result cmp failed"
print("add const compare success!")
```

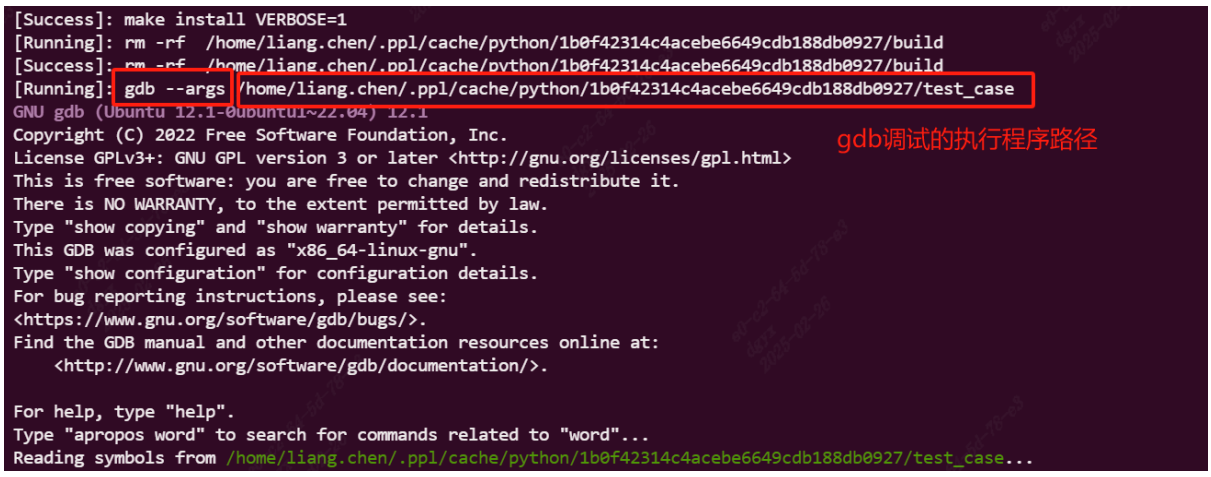
然后运行：

```
# 指定运行芯片
CHIP=bm1684x python add_ppl.py
```

如果要对算子的 tpu-kernel 代码进行调试，可以如下设置:

```
# 设置debug=True，当在cmodel环境下运行，进入算子函数时，将开启gdb
@ppl.jit(debug=True)
def add_kernel(ptr_res, ptr_inp, scalar_c, C: pl.constexpr, W,
               block_w: pl.constexpr):
```

然后使用 python 运行，当进入算子入口时，会自动启动 gdb。gdb 命令会打印在终端上，如下图所示：



gdb 调试的执行程序路径为 ${PPL_CACHE_PATH}/.ppl/cache/xxx/（PPL_CACHE_PATH 为缓存路径环境变量，用户可以自行设置），xxx 为 hash code；目录下有以下文件：

```
├─── data/      #输入输出数据
├─── device/    # device端代码
├─── host/      # host端代码
├─── include/   # 头文件
├─── lib/       # 算子动态库
├─── src/       # main函数
├─── *.mlir     # 中间IR
├─── CMakeLists # 编译脚本
```

1.4 Samples 使用说明

若开发者不通过 TPU-MLIR 或 Torch-TPU 框架接入 ppl 算子，而是直接集成到自己工程中，可参考 samples 进行开发。此时需重点关注以下内容：

1. 熟悉 TPU Runtime API
2. 将 ppl 算子交叉编译（PCIE/SOC 模式）成动态库
3. 用户自己管理动态库加载及算子运行调度

1.4.1 工程布局与编译

```
├─── CMakeLists.txt # host程序和动态库自动构建脚本
├─── ppl/           # ppl代码
├─── src/           # 应用程序的代码
├─── include/       #
```

1.4.2 CMODEL 模式编译

```
cd samples/add_pipeline
# 当前已处于samples/add_pipeline目录下

chmod +x build.sh
./build.sh bm1690 cmodel
# 会在当前目录下编译生成test_case执行文件, 如果编译bm1684x(或bm1688)下运行的程序,
# 使用./build.sh bm1684x(或bm1688) cmodel
# 如果使用自己安装的runtime库可以修改build.sh 在cmake命令里面加上-DRUNTIME_PATH=xxx
# (runtime根目录路径)

chmod +x run.sh
./run.sh bm1690 cmodel
# 运行sample
# 如果使用自己安装的runtime库可以修改LD_LIBRARY_PATH指向自己的runtime lib路径
```

1.4.3 PCIE 模式编译

编译 pcie 模式下的应用程序并运行（需要确保环境中安装了 libsophon）编译 device 端程序需要额外下载交叉编译工具或者通过 build.sh 自动下载

bm1684x 需要下载：

- gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu

bm1688 和 bm1690 需要下载:

- Xuantie-900-gcc-linux-5.10.4-glibc-x86_64-V2.6.1

下载后将编译器解压放到 toolchains_dir 下

```
# 下载交叉编译器（只需要执行一次）
cd samples

# 设置交叉编译器路径（通过build.sh自动下载编译器可以不用设置）
export CROSS_TOOLCHAINS=path/to/toolchains_dir

cd add_pipeline
# 当前已处于samples/add_pipeline目录下

chmod +x build.sh
./build.sh bm1684x pcie
# 会在当前目录下编译生成test_case执行文件。

chmod +x run.sh
./run.sh bm1684x pcie
# 运行sample
```

1.4.4 SOC 模式编译

SOC 模式开发适用于 SOC 的 TPU 设备。基本工作流程是在 x86 的机器上交叉编译出应用和对应供 a53 Linux 加载的动态库，然后复制到 SOC 设备上，先加载动态库，再运行应用程序。

在此模式下，需要按照《LIBSOPHON 使用手册》的“**使用 libsophon 开发**”一章中的“**SOC MODE**”进行环境初始化，并将 soc_sdk 的目录路径设置到 SOC_SDK 环境变量中，来指定 soc 的 sdk 位置。

编译 host 端程序和 device 端程序需要额外下载交叉编译工具或者通过 build.sh 自动下载。

bm1684x 需要下载：

- gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu
- gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu

bm1688 需要下载：

- Xuantie-900-gcc-linux-5.10.4-glibc-x86_64-V2.6.1
- gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu

bm1690 需要下载：

- Xuantie-900-gcc-linux-5.10.4-glibc-x86_64-V2.6.1

下载后将编译器解压放到 toolchains_dir 下

```
cd samples

# 设置交叉编译器路径（通过build.sh自动下载编译器可以不用设置）
export CROSS_TOOLCHAINS=path/to/toolchains_dir

# 下载libsophon soc，设置SOC_SDK
# /path_to_sdk/soc_sdk仅是示例，需要根据实际的设置进行修改
mkdir soc_sdk && cd soc_sdk
cp -rf libsophon_soc_0.x.x_aarch64/opt/sophon/libsophon-0.x.x/lib/. soc_sdk/.
cp -rf libsophon_soc_0.x.x_aarch64/opt/sophon/libsophon-0.x.x/include/. soc_sdk/.
export SOC_SDK=/path_to_sdk/soc_sdk

cd add_pipeline
# 当前已处于samples/add_pipeline目录下

chmod +x build.sh
./build.sh bm1684x soc
# 或者执行（./build.sh bm1688 soc），会在当前目录下编译生成test_case执行文件。
```

（续下页）

(接上页)

```
# 运行sample
# 将test_case、lib、run.sh目录复制到soc机器上直接执行 ./run.sh bm1684x soc
```

1.4.5 编译生成的文件

```
├─── build/              # 编译目录
|    ├─── device/             # device端代码目录
|    |    ├─── add_pipeline.c    # 编译后生成的kernel 函数代码
|    ├─── host/               # host端代码目录
|    |    ├─── add_pipeline.cpp  # 编译后生成的host端函数代码,
|    |                      # 内部封装了使用runtime调用kernel函数的过程
|    ├─── include/            #
|    |    ├─── add_pipeline.h    # 编译后生成的host端函数头文件
├─── ppl/                   # ppl代码
├─── src/                   # 应用程序的代码
├─── lib/
|    ├─── libkernel.so|libcmodel.so     # 编译后生成的device端包含kernel函数的动态库
| ─── test_case              # 应用程序
```

1.4.6 应用程序的代码解析

Sample 中应用程序的代码在 samples/add_pipeline/src/main.cpp 中，主要作用是不同芯片如何使用 PPL 生成的 kernel 端代码。PPL 目前支持算能科技的三种不同的芯片类型，bm1684x, bm1688 和 bm1690。其中 bm1684x 和 bm1688 使用的 runtime 基于 libsophon；而 bm1690 使用的 runtime 基于 tpuv7-runtime。在示例程序中可以看到使用了 __bm1684x__，__bm1688__，__bm1690__ 三个宏定义，针对不同芯片使能了不同的代码段。

- device 端 ppl 编译器内置了这个宏定义，因此可以直接在 ppl 代码中使用这个宏定义；
- host 端 ppl 编译生成的头文件（include/add_pipeline.h）定义了这个宏。

如果开发人员不需要兼容多款芯片，只需要查阅对应芯片宏定义有效范围内的代码。

```
#ifdef __bm1684x__
// 定义常用的数据结构和枚举
#include "tpu_defs.h"
// 基于 libsophon 的 runtime 的头文件，适用于 bm1684x和bm1688
#include "bmlib_runtime.h"
// 用于 bm1684x 和 bm1688 在PCIE模式下动态加载库文件
#include "kernel_module_data.h"
#endif
#ifdef __bm1690__
// 基于 tpuv7-runtime 的runtime的头文件，适用于 bm1690
#include <tpuv7_rt.h>
#endif

#ifdef __bm1688__
#include "bmlib_runtime.h"
#include "kernel_module_data.h"
#include "tpu_defs.h"
#endif
// 测试工具，用于随机生成数据和dump数据
#include "host_test_utils.h"
// host端 函数的头文件
#include "add_pipeline.h"
#include <cstring>
#include <string>
```

(续下页)

(接上页)

```
#include <vector>

#if defined(__bm1684x__) || defined(__bm1688__)
// 定义设备的句柄
bm_handle_t handle;
// 定义kernel 动态库的句柄
tpu_kernel_module_t tpu_module;
#endif

#ifdef __bm1690__
// 定义 runtime stream
tpuRtStream_t stream;
// 定义 kernel module 的句柄
tpuRtKernelModule_t tpu_module;
#endif

int main() {
  std::string func_name = "add_pipeline";
  float v1 = (float)1.000000000e+00;
  float v2 = (float)-1.000000000e+00;
  int32_t N = 1;
  int32_t C = 1;
  int32_t H = 1;
  int32_t W = 32;
  dim4 input_shape = {N, C, H, W};
  size_t data_size = N * C * H * W * DtypeSize(DT_FP32);
#ifdef __bm1690__
  // tpuv7_runimte的状态变量
  tpuRtStatus_t ret;
  // 计算设备runtime初始化
  ret = tpuRtInit();
  if (ret != tpuRtSuccess) {
    printf("tpuRtInit failed\n)");
    return -1;
  }
  printf("tpuRtInit success\n)");
  // 设置当前进程使用的计算设备
  tpuRtSetDevice(0);
  // 创建一个stream
  tpuRtStreamCreate(&stream);

  // 通过环境变量设置 ppl 代码生成的 deivce 端动态库存放路径
  auto kernel_dir = getenv("PPL_KERNEL_PATH");
  if (!kernel_dir) {
    printf("Please set env PPL_KERNEL_PATH to libkernel.so path");
    return -2;
  }

  // 载入 device 端动态库
  tpu_module = tpuRtKernelLoadModuleFile(kernel_dir, stream);
  if (NULL == tpu_module) {
    printf("tpuRtKernelLoadModuleFile failed");
    return -2;
  }
  char *input_data = new char[data_size];
  char *output_data = new char[data_size];

  // 初始化输入数据
  printf("input_data:\n");
  for (int i = 0; i < N * C * H * W; ++i) {
    float val = 0.5f + i;
```

(续下页)

(接上页)

```
    ((float *)input_data)[i] = val;
    printf("%f, ", val);
  }
  printf("\n");
  void *dev_input_data;
  void *dev_output_data;
  // 向计算设备申请memory，第三个参数为并行度
  tpuRtMalloc((void **)(&dev_input_data), data_size, 0);
  tpuRtMalloc((void **)(&dev_output_data), data_size, 0);
  //  init input_data
  // 从host memory复制数据到计算设备memory
  tpuRtMemcpyS2D(dev_input_data, input_data, data_size);

  // 调用 自动生成的 host 端函数，此函数内部会调用 device 端的 kernel函数
  int rst = add_pipeline((unsigned long long)dev_output_data,
                         (unsigned long long)dev_input_data, W);

  // 从计算设备memory复制数据到host memory
  tpuRtMemcpyD2S(output_data, dev_output_data, data_size);

  // 打印运行结果
  printf("output data:\n");
  for (int i = 0; i < N * C * H * W; ++i) {
    printf("%f, ", ((float *)output_data)[i]);
  }
  printf("\n");
  delete[] input_data;
  delete[] output_data;
  // 释放计算设备的memory
  tpuRtFree(&dev_input_data, 0);
  tpuRtFree(&dev_output_data, 0);

  tpuRtKernelUnloadModule(tpu_module, stream);
  tpuRtStreamDestroy(stream);
#endif
#if defined(__bm1684x__) || defined(__bm1688__)
  // 返回状态
  bm_status_t ret = BM_SUCCESS;
  // 请求一个设备，得到设备句柄handle
  ret = bm_dev_request(&handle, 0);
  if (ret != BM_SUCCESS)
    throw("bm_dev_request failed");
  printf("bm_dev_request success\n");

  // kernel_module_data 定义在kernel_module_data.h中
  // 后续在cmake 架构章节中会看到，这是将 kernel
  // 库文件使用hexdump以文本形式打包到头文件中
  const unsigned int *p = kernel_module_data;
  size_t length = sizeof(kernel_module_data);
  // 动态加载 kernel module
  tpu_module = tpu_kernel_load_module(handle, (const char *)p, length);
  if (!tpu_module) {
    printf("tpu_kernel_load_module failed\n");
    return -1;
  }
  printf("tpu_module load success\n");
  // module 句柄
  bm_device_mem_t dev_input_data;
  bm_device_mem_t dev_output_data;
  char *input_data;
  char *output_data;
```

(续下页)

(接上页)

```
  // 申请指定大小的device mem和host mem，size为device mem的字节大小
  MallocWrap(handle, &dev_input_data, (u64 *)&input_data, data_size);
  MallocWrap(handle, &dev_output_data, (u64 *)&output_data, data_size);
  // 初始化输入数据
  printf("input_data:\n");
  for (int i = 0; i < N * C * H * W; ++i) {
    float val = 0.5f + i;
    ((float *)input_data)[i] = val;
    printf("%f, ", val);
  }
  printf("\n");

  // 将在系统内存上的数据拷贝到device mem
  // MemcpyS2D还有一个默认参数offset，默认为0，从src的offset偏移开始拷贝
  MemcpyS2D(handle, &dev_input_data, input_data, data_size);

  bm_profile_t start, end;
  bm_get_profile(handle, &start);
  // 通过自动生成的 host 端封装函数调用 kernel 函数
  int rst = add_pipeline(bm_mem_get_device_addr(dev_output_data),
                         bm_mem_get_device_addr(dev_input_data), W);
  bm_get_profile(handle, &end);
  if (ret) {
    printf("tpu_kernel_launch failed\n");
  } else {
    size_t npu_time = end.tpu_process_time - start.tpu_process_time;
    std::cout << "npu time = " << npu_time << "(us) --> ";
    printf("tpu_kernel_launch success\n");
  }
  // 将在 device mem 上的数据拷贝到 系统内存
  MemcpyD2S(handle, &dev_output_data, output_data, data_size);
  // 打印运行结果
  printf("output data:\n");
  for (int i = 0; i < N * C * H * W; ++i) {
    printf("%f, ", ((float *)output_data)[i]);
  }
  printf("\n");
  // device mem和host mem
  FreeWrap(handle, &dev_input_data, input_data);
  FreeWrap(handle, &dev_output_data, output_data);
  // 释放设备句柄
  tpu_kernel_free_module(handle, tpu_module);
  bm_dev_free(handle);
#endif
  return 0;
}
```

1.4.7 自动生成的 host 端代码解析

自动生成的 host 端代码在 samples/add_pipeline/host/add_pipeline.cpp 中，主要作用封装通过 runtime 调用 kernel 函数的流程，以 bm1684x 上生成的 host 端代码为例：

```
#include "add_pipeline.h"
#include <cstdio>
#include "tpu_defs.h"
#include "bmlib_runtime.h"
// 用于 bm1684x 和 bm1688 在PCIE模式下动态加载库文件
#include "kernel_module_data.h"
```

(续下页)

(接上页)

```
// device 端 kernel 函数传参使用的结构体
typedef struct {
  unsigned long long v1;
  unsigned long long v2;
  int32_t v3;
} tpu_kernel_api_add_pipeline_t;

// 设备的句柄，在main.cpp初始化
extern bm_handle_t handle;
// device端 kernel module句柄，在main.cpp初始化
extern tpu_kernel_module_t tpu_module;

// host端函数，与pl中的kernel函数同名同参数
int add_pipeline(unsigned long long v1, unsigned long long v2, int32_t v3) {
  // 避免多次重复加载函数增加耗时
  static int func_id = -1;
  if (func_id < 0) {
    // 获取函数句柄
    func_id = tpu_kernel_get_function(handle, tpu_module, "add_pipeline");
    if (func_id < 0) {
      printf("load kernel function failed!\n");
      return -2;
    }
  }
  tpu_kernel_api_add_pipeline_t add_pipeline_api;
  add_pipeline_api.v1 = v1;
  add_pipeline_api.v2 = v2;
  add_pipeline_api.v3 = v3;

  // 调用 kernel 函数
  int ret = tpu_kernel_launch(handle, func_id, &add_pipeline_api, sizeof(add_pipeline_api));
  // 同步
  bm_thread_sync(handle);
  return ret;
}
```

1.4.8 cmake 架构

从 1.5.3 CMODEL 模式编译章节可以看到，主要的编译接口为 build.sh，该脚本主要传递三个参数，顺序为芯片类型（bm1684x/bm1688/bm1690），运行模式（cmodel/pcie），以及对应的 PPL 代码，默认为 samples/add_pipeline/ppl/add_pipeline.pl，并分别将这三个值赋给环境变量 CHIP，DEV_MODE，FILE。CMakeList.txt 中主要是设置对应的编译选项，并检查 CHIP，DEV_MODE 是否有定义，如果没有正确定义则报错。最后根据 DEV_MODE 选择 cmodel.cmake 或 pcie.cmake 。

cmodel 仿真模式

cmodel.cmake 主要适用于构建 cmodel 仿真时的应用。下方代码为 cmodel.cmake 中的关键步骤。

路径配置

针对芯片类型不同，需要调用不同的 runtime 库。在实际构建工程时，进需要根据芯片类型配置 RUNTIME_TOP 和 BMLIB_CMODEL_PATH。实际工程还需要使用到后端库信息以及 PPL 提供的 device/host 的辅助函数，相关路径也是必须要完成相关配置的。

```
# 配置runtime库相关路径
# bm1684x 和 bm1688 使用的 runtime 基于 libsophon
# bm1690 使用的 runtime 基于 tpuv7-runtime
if(DEFINED RUNTIME_PATH)
    set(RUNTIME_TOP ${RUNTIME_PATH})
  message(NOTICE "RUNTIME PATH: ${RUNTIME_PATH}")
else()
    if(${CHIP} STREQUAL "bm1690")
        set(RUNTIME_TOP ${PPL_TOP}/runtime/bm1690/tpuv7-runtime-emulator)
        set(BMLIB_CMODEL_PATH ${RUNTIME_TOP}/lib/libtpuv7_emulator.so)
    elseif(${CHIP} STREQUAL "bm1684x")
        set(RUNTIME_TOP ${PPL_TOP}/runtime/bm1684x/libsophon/bmlib)
        set(BMLIB_CMODEL_PATH ${PPL_TOP}/runtime/bm1684x/lib/libcmodel_firmware.so)
    elseif(${CHIP} STREQUAL "bm1688")
        set(RUNTIME_TOP ${PPL_TOP}/runtime/bm1688/libsophon/bmlib)
        set(BMLIB_CMODEL_PATH ${PPL_TOP}/runtime/bm1688/lib/libcmodel_firmware.so)
    else()
    message(FATAL_ERROR "Unknown chip type:${CHIP}")
    endif()
endif()
# TPUKERNEL_TOP 为指定芯片类别的后端相关信息
set(TPUKERNEL_TOP ${PPL_TOP}/runtime/${CHIP}/TPU1686)
# KERNEL_TOP 为后端公共头文件，主要是常用数据类型以及类型转换等
set(KERNEL_TOP ${PPL_TOP}/runtime/kernel)
# CUS_TOP内为ppl提供的device、host端辅助函数
set(CUS_TOP ${PPL_TOP}/runtime/customize)
```

头文件搜索路径配置

头文件主要需要包含以下几个部分：PPL 生成的 kernel 函数的头文件，rumtime 相关头文件，芯片后端相关头文件，后端公共头文件，以及 PPL 的辅助函数相关头文件。对于 bm1684x 和 bm1688 芯片，会在构建工程时生成 kernel_module_data.h 用于动态加载，这个目前是在 ${CMAKE_BINARY_DIR} 路径下。

```
# ppl 在输出路径自动生成的inculde文件夹，包含kernel函数参数的定义
include_directories(${CMAKE_CURRENT_BINARY_DIR}/include)
# 指定芯片的tpu-kernel指令调用接口头文件，以及和TPU相关的通用定义
include_directories(${TPUKERNEL_TOP}/kernel/include)
# KERNEL_TOP 为后端公共头文件，主要是常用数据类型以及类型转换等
include_directories(${KERNEL_TOP})
# runtime 的头文件
include_directories(${RUNTIME_TOP}/include)
# CUS_TOP内为ppl提供的device、host端辅助函数
include_directories(${CUS_TOP}/include)
if(${CHIP} STREQUAL "bm1684x" OR ${CHIP} STREQUAL "bm1688")
# 使用基于libsophon的runtime时，cmake构建时会在${CMAKE_BINARY_DIR}
# 路径下生成kernel_module_data.h
  include_directories(${CMAKE_BINARY_DIR})
endif()
```

库文件搜索路径设置

库文件需要加载 runtime 和后端的相关动态链接库。

```
# 链接用于bmruntime的cmodel库
link_directories(${PPL_TOP}/runtime/${CHIP}/lib)
# 链接 runtime 的库
link_directories(${RUNTIME_TOP}/lib)
```

生成 libsophon 动态加载的 kernel_module_data.h

这一步骤仅用于 bm1684x 和 bm1688 芯片。cmodel 仿真过程中，并不需要将 kernel 库文件写入头文件中，为了保持和 PCIE 模式代码统一，保留了这一步骤。

```
# 在${CMAKE_BINARY_DIR}路径下生成kernel_module_data.h
# cmodel模式下不需要动态加载，所以置空
if(${CHIP} STREQUAL "bm1684x"
    OR ${CHIP} STREQUAL "bm1688")
  set(KERNEL_HEADER "${CMAKE_BINARY_DIR}/kernel_module_data.h")
  add_custom_command(
    OUTPUT ${KERNEL_HEADER}
    COMMAND echo "const unsigned int kernel_module_data[] = {0}\;" > ${KERNEL_
↪HEADER}
  )
  add_custom_target(gen_kernel_module_data_target DEPENDS ${KERNEL_HEADER})
endif()
```

AddPPL 编译生成 kernel 代码

基于 AddPPL.cmake 使用 ppl_gen 可以快速将用户编写的 pl 文件转换为能在 TPU 上运行的 C 代码，详细信息可以参考”PPL 开发参考手册”。开发人员也可以根据工程管理方式不同，选择离线生成 kernel C 代码。

```
set(SCRIPTS_CMAKE_DIR "${PPL_TOP}/runtime/scripts/")
list(APPEND CMAKE_MODULE_PATH "${SCRIPTS_CMAKE_DIR}")
include(AddPPL)  #AddPPL.cmake including pplgen

# 将 PPL 编写的 kernel 代码进行编译
file(GLOB PPL_SOURCE ppl/*.pl)
foreach(ppl_file ${PPL_SOURCE})
  set(input ${ppl_file})
  # PPL 编译输出路径为 当前目录的 build 下
  set(output ${CMAKE_CURRENT_BINARY_DIR})
  ppl_gen(${input} ${CHIP} ${output} ${OPT_LEVEL})
endforeach()
```

生成可执行文件

开发人员需要根据实际芯片类型，链接不同的 runtime 库。如果使用的是 bm1684x 或 bm1688 芯片，需要增加对 kernel_module_data.h 的依赖。

```
# 将 PPL 编译生成的 host 目录下的源文件构建为共享库，然后安装到指定目录中
aux_source_directory(${CMAKE_CURRENT_BINARY_DIR}/host PPL_SRC_FILES)
aux_source_directory(src SRC_FILES)
add_executable(test_case ${PPL_SRC_FILES} ${SRC_FILES})
if(${CHIP} STREQUAL "bm1684x" OR ${CHIP} STREQUAL "bm1688")
  target_link_libraries(test_case PRIVATE bmlib pthread)
```

(续下页)

(接上页)

```
  add_dependencies(test_case gen_kernel_module_data_target)
else()
  target_link_libraries(test_case PRIVATE tpuv7_rt cdm_daemon_emulator pthread)
endif()
install(TARGETS test_case DESTINATION ${CMAKE_CURRENT_SOURCE_DIR})
```

kernel 函数构建动态库

对于 bm1690 芯片，是显式的去加载编译生成的动态库，可以根据需求去修改动态库的名称；但是对于 bm1684x 或 bm1688，cmodel 仿真是默认加载 libcmodel.so，如果需要修改名称，需要在仿真环境中给环境变量 TPUKERNEL_FIRMWARE_PATH 中设置为对应的库的名称。

```
# 将 PPL 编译生成的 device 目录下的源文件构建为共享库，然后安装到指定目录中
aux_source_directory(${CMAKE_CURRENT_BINARY_DIR}/device KERNEL_SRC_FILES)
add_library(kernel SHARED ${KERNEL_SRC_FILES} ${CUS_TOP}/src/ppl_helper.c)
target_include_directories(kernel PRIVATE
    include
    ${PPL_TOP}/include
    ${CUS_TOP}/include
    ${TPUKERNEL_TOP}/common/include
    ${TPUKERNEL_TOP}/kernel/include
)
target_link_libraries(kernel PRIVATE ${BMLIB_CMODEL_PATH} m)
if(${CHIP} STREQUAL "bm1684x"
  OR ${CHIP} STREQUAL "bm1688")
    set_target_properties(kernel PROPERTIES OUTPUT_NAME cmodel)
endif()
install(TARGETS kernel DESTINATION ${CMAKE_CURRENT_SOURCE_DIR}/lib)
```

1.4.9 PCIE 运行模式

pcie.cmake 主要适用于构建用于在 pcie 上实际运行时的应用。目前仅支持 bm1684x 芯片使用 PCIE 运行模式自动构建工程。下方代码为 pcie.cmake 中的关键信息。

额外头文件，库文件配置

如果开发人员工程设计需要引入其他的头文件或者库的搜索路径，可以通过 additional_include 和 additional_link 进行设置。开发人员也可以手动使用 cmake 命令进行相关路径的配置。

```
set(opt_level "2")
set(additional_include "path1;path2,path3 path4")
set(additional_link "")

string(REPLACE " " ";" additional_include "${additional_include}")
string(REPLACE "," ";" additional_link "${additional_link}")
```

工具链配置

TPU-KERNEL 开发需要用到 gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu 交叉工具链，可以使用 download_toolchain.sh 脚本下载工具链并完成相关配置。

```
# try download cross toolchain
if(NOT DEFINED ENV{CROSS_TOOLCHAINS})
    message("CROSS_TOOLCHAINS was not defined, try source download_toolchain.sh")
    execute_process(
        COMMAND bash -c "source $ENV{PPL_PROJECT_ROOT}/samples/scripts/download_
↪toolchain.sh && env"
        RESULT_VARIABLE result
        OUTPUT_VARIABLE output
    )
    if(NOT result EQUAL "0")
        message(FATAL_ERROR "Not able to source download_toolchain.sh: ${output}")
    endif()
    string(REGEX MATCH "CROSS_TOOLCHAINS=([^\n]*)" _ ${output})
    set(ENV{CROSS_TOOLCHAINS} "${CMAKE_MATCH_1}")
endif()
# Set the C compiler
set(CMAKE_C_COMPILER $ENV{CROSS_TOOLCHAINS}/gcc-arm-10.3-2021.07-x86_64-
↪aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-gcc)
```

路径配置

和 cmodel 仿真模式的路径配置方法类似，需要注意的是，这里不用配置 BMLIB_CMODEL_PATH，因为这是仅用于 runtime cmodel 仿真的。

头文件搜索路径配置

和 cmodel 仿真模式类似，多出来了对 additional_include 中路径的相关配置。

AddPPL 编译生成 kernel 代码

和 cmodel 仿真模式类似。

生成 libsophon 动态加载的 kernel_module_data.h

和 cmodel 仿真模式将 kernel 函数的动态库指定加载不同，对于 pcie 运行模式，需要先基于 kernel 函数构建动态库，并使用 hex2dump 将其打包到 kernel_module_data.h 中。

```
# Set the library directories for the shared library (link lib${CHIP}.a)
# 供设备代码链接使用的底库libbm1684x.a
link_directories(${PPL_TOP}/runtime/bm1684x/lib)

# Set the output file for the shared library
set(SHARED_LIBRARY_OUTPUT_FILE lib${CHIP}_kernel_module)

# Create the shared library
add_library(${SHARED_LIBRARY_OUTPUT_FILE} SHARED ${DEVICE_SRCS} ${CUS_
↪TOP}/src/ppl_helper.c)

# Link the libraries for the shared library
target_link_libraries(${SHARED_LIBRARY_OUTPUT_FILE} -Wl,--whole-archive libbm1684x.
↪a -Wl,--no-whole-archive m)
```

(续下页)

(接上页)

```
if ("${CMAKE_BUILD_TYPE}" STREQUAL "Debug")
  MESSAGE (STATUS "Current is Debug mode")
  SET (FW_DEBUG_FLAGS "-DUSING_FW_DEBUG")
ENDIF ()
# Set the output file properties for the shared library
set_target_properties(${SHARED_LIBRARY_OUTPUT_FILE} PROPERTIES PREFIX ""F
↪SUFFIX ".so"
                COMPILE_FLAGS "-fPIC ${FW_DEBUG_FLAGS}" LINK_FLAGS "-shared")

# Set the path to the input file
set(INPUT_FILE "${CMAKE_BINARY_DIR}/lib${CHIP}_kernel_module.so")

# Set the path to the output file
set(KERNEL_HEADER "${CMAKE_BINARY_DIR}/kernel_module_data.h")
add_custom_command(
    OUTPUT ${KERNEL_HEADER}
    DEPENDS ${SHARED_LIBRARY_OUTPUT_FILE}
    COMMAND echo "const unsigned int kernel_module_data[] = {" > ${KERNEL_HEADER}
    COMMAND hexdump -v -e '1/4 \"0x%08x,\\n\"' ${INPUT_FILE} >> ${KERNEL_HEADER}
    COMMAND echo "};" >> ${KERNEL_HEADER}
)

# Add a custom target that depends on the custom command
add_custom_target(gen_kernel_module_data_target DEPENDS ${KERNEL_HEADER})
# Add a custom target for the shared library
add_custom_target(dynamic_library DEPENDS ${SHARED_LIBRARY_OUTPUT_FILE})
```

1.4.10 生成可执行文件

和 cmodel 仿真模式类似，区别主要在于要考虑额外库文件的链接。

1.4.11 runtime 需要配置的环境变量

sample 中有 run.sh 脚本，用于运行生成的可执行文件。pcie 和 soc 模式，需要自行安装 libsophon 等 runtime 库 cmodel 模式，可以使用 ppl 自带的模拟器，需要手动配置模拟器库路径，可以参考 run.sh 进行配置

使用方法：

```
chmod +x run.sh
./run.sh bm1684x cmodel
#./run.sh bm1688 cmodel
#./run.sh bm1690 cmodel
```

```
set -ex
mode=${1:-cmodel}
if [ "$mode" == "cmodel" ]; then
  export LD_LIBRARY_PATH=${PPL_PROJECT_ROOT}/runtime/bm1684x/lib/:${PPL_
↪PROJECT_ROOT}/runtime/bm1684x/libsophon/bmlib/lib:$PWD/lib:$LD_LIBRARY_PATH
fi
./test_case
```

对于 bm1690 芯片类型，如果没有安装 tpuv7-runtime-emulator 的 deb 包，则需要类似 run_bm1690.sh 中在运行环境添加如下几个环境变量。其中，TPU_EMULATOR_PATH，TPU_SCALAR_EMULATOR_PATH 均为 ${PPL_PROJECT_ROOT}/runtime/bm1690/tpuv7-runtime-emulator/lib/ 下的 runtime 库，名称是固定的；而后就是 cmake 架构中 insall 操作时的 TPU_KERNEL_PATH 路径。最后要将这两个路径添加到 LD_LIBRARY_PATH 环境变量中。

```
set -ex
export TPU_EMULATOR_PATH=${PPL_PROJECT_ROOT}/runtime/bm1690/tpuv7-runtime-
↪emulator/lib/libtpuv7_emulator.so
export TPU_SCALAR_EMULATOR_PATH=${PPL_PROJECT_ROOT}/runtime/bm1690/
↪tpuv7-runtime-emulator/lib/libtpuv7_scalar_emulator.so
export TPU_KERNEL_PATH=./lib
export PPL_KERNEL_PATH=./lib/libkernel.so
export LD_LIBRARY_PATH=${PPL_PROJECT_ROOT}/runtime/bm1690/tpuv7-runtime-
↪emulator/lib/:$PWD/lib:$LD_LIBRARY_PATH
./test_case
```

CHAPTER 2

PPL 算子开发及性能优化示例

该章节首先介绍使用 PPL 开发算子的整个流程，以具体例子（加法）来详细介绍 PPL 的开发和优化，最后展示每一步优化的性能情况。

2.1 PPL 算子开发概述

PPL device 端代码实现功能的编写逻辑为将 global mem 数据上传到 local mem、在 local mem 上计算和从 local mem 下载结果数据至 global mem。

2.1.1 device 端代码实现

device 端示例代码如下所示：

```
__KERNEL__ void add_kernel_ori(fp16 *ptr_res, fp16 *ptr_inp, const int N,
                               const int C, const int H, const int W) {
  dim4 shape = {N, C, H, W};
  // 使用 gtensor 封装 global memory 上的数据
  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);
  // 申请 tpu local memory 上的内存
  auto in = tensor<fp16>(shape);
  auto res = tensor<fp16>(shape);
  float scalar_c = 0.25;

  // 从 global memory 上 load 数据到 tpu local memory 上
  dma::load(in, in_gtensor);
  // 做加法
  tiu::fadd(res, in, scalar_c);
  // 将数据从 local mem store 回 global memory
  dma::store(res_gtensor, res);
}
```

ppl 使用 gtensor 抽象表示 global/l2 mem 上的内存及其 shape、stride 和 offset 等信息；使用 tensor 抽象表示 local mem 上的内存及其 shape、stride 和 offset 等信息。所以，首先需要将 kernel 函数传入参数中的 global mem 内存地址绑定到 gtensor 并指定为 GLOBAL 类型；然后，使用 shape 申请 local mem 的地址空间。

加法 kernel 函数的运行逻辑为：1. 从 global memory 上 load 数据到 tpu local memory；2. 做加法；3. 将数据从 local memory 上 store 回 global memory。

2.1.2 device 端代码性能优化

在 kernel 函数基础功能实现正确的基础上，可以分三步进行性能优化。

对齐 lane num

tpu 在进行运算时会按照 tensor shape 的 channel 分发到 LANE_NUM 个 lane 上进行并行计算；所以，将 tensor 的 channel 对齐 LANE_NUM 能更好的发挥硬件的并行性能。如果，数据的计算对于 shape 的形状依赖较少 (例如逐元素计算、单维度 softmax 等)，则可以将数据切分到 channel 维度上，并对齐 LANE_NUM，可以实现一定程度的性能优化。

上面加法 kernel 函数为逐元素加法，所以可以改写为：

```
__KERNEL__ void add_kernel_align_lane0(fp16 *ptr_res, fp16 *ptr_inp, const int N,
                            const int C, const int H, const int W) {
  int n = 1, c = LANE_NUM, h = 1;
  int element_num = N * C * H * W;
  int w = div_up(element_num, LANE_NUM);
  dim4 shape = {n, c, h, w};

  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);

  auto in = tensor<fp16>(shape);
  auto res = tensor<fp16>(shape);
  float scalar_c = 0.25;

  dma::load(in, in_gtensor);
  tiu::fadd(res, in, scalar_c);
  dma::store(res_gtensor, res);
}
```

因为示例中的加法计算为逐元素计算，所以对于 shape 的形状不存在依赖，故可以先将 tensor 视为一维：

```
int element_num = N * C * H * W;
```

然后，在将一维的数据按 LANE_NUM 进行划分，并将 channel 维度设置为 LANE_NUM：

```
int w = div_up(element_num, LANE_NUM);
dim4 shape = {1, LANE_NUM, 1, w};
```

对于 shape 形状存在依赖的算子，即不可以随意更改 tensor shape 的算子，例如 matmul，可以将 M、N、K 中的某一维放在 channel 维度上实现加速：

```
// 完整代码见 ppl/example/cxx/matmul/mm2_fp16.pl
dim4 res_global_shape = {1, M, 1, N};
dim4 left_global_shape = {1, M, 1, K};
dim4 right_global_shape = {1, K, 1, N};
```

注意： ** 上述示例中，将整个 tensor 按照 LANE_NUM 划分，** 必须保证 N * C * H * W 的结果是 LANE_NUM 的整数倍，否则会导致访问越界**。或者可以使用下面的实现方式进行划分：

```
__KERNEL__ void add_kernel_align_lane1(fp16 *ptr_res, fp16 *ptr_inp,
                            const int N, const int C, const int H,
                            const int W) {
```

(续下页)

(接上页)

```
  const int c = N * C, w = H * W;
  dim4 shape = {1, c, 1, w};
  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);

  float scalar_c = 0.25;

  int block_c = LANE_NUM;
  int block_w = 512;
  dim4 in_shape = {1, block_c, 1, block_w};

  for (int idx_c = 0; idx_c < c; idx_c += block_c) {
    int real_c = min(block_c, c - idx_c);
    for (int idx_w = 0; idx_w < w; idx_w += block_w) {
      int real_w = min(block_w, w - idx_w);
      dim4 real_shape = {1, real_c, 1, real_w};
      dim4 offset = {0, idx_c, 0, idx_w};
      auto in = make_tensor<fp16>(in_shape, real_shape);
      auto res = make_tensor<fp16>(in_shape, real_shape);
      dma::load(in, in_gtensor.sub_view(real_shape, offset));
      tiu::fadd(res, in, scalar_c);
      dma::store(res_gtensor.sub_view(real_shape, offset), res);
    }
  }
}
```

数据切分与流水并行

local memory 的大小存在限制，当数据规模过大时，无法一次性将所有数据 load 进入 local memory，即无法经过一轮运算就得到结果，需要分批次将数据传入 tpu，然后排队计算。

tpu 各个 engine 之间的运行是独立且互不干扰的（仅存在数据流动依赖关系），于是可以将数据切分为适当的大小，然后利用 tpu 的硬件特性实现内存传输（dma）时间的隐藏，进而实现性能优化。流水并行具体原理可参考 doc 目录下的《PPL 开发参考手册.pdf》。

上述加法 kernel 可进一步优化为：

```
__KERNEL__ void add_kernel_pipeline(fp16 *ptr_res, fp16 *ptr_inp, const int N,
                                    const int C, const int H, const int W) {
  int n = 1, c = LANE_NUM, h = 1;
  int element_num = N * C * H * W;
  int w = div_up(element_num, LANE_NUM);
  dim4 shape = {n, c, h, w};
  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);

  int block_w = 512;
  dim4 block_shape = {n, c, h, block_w};
  float scalar_c = 0.25;

  for (int w_idx = 0; w_idx < W; w_idx += block_w) {
    enable_pipeline();
    int tile_w = min(block_w, W - w_idx);
    dim4 cur_shape = {n, c, h, tile_w};
    auto in = make_tensor<fp16>(block_shape, cur_shape);
    auto res = make_tensor<fp16>(block_shape, cur_shape);

    dim4 offset = {0, 0, 0, w_idx};
    dma::load(in, in_gtensor.sub_view(cur_shape, offset));
    tiu::fadd(res, in, scalar_c);
```

(续下页)

(接上页)

```
    dma::store(res_gtensor.sub_view(cur_shape, offset), res);
  }
}
```

对 W 维度进行切分，并使用 enable_pipeline 指令开启自动流水并行，即可以将一部分 load 和 store 操作消耗的时间隐藏。

同理，上述示例中，将整个 tensor 按照 LANE_NUM 划分，**必须保证 N * C * H * W 的结果是 LANE_NUM 的整数倍，否则会导致访问越界**。或者可以使用下面的实现方式进行划分：

```
__KERNEL__ void add_kernel_pipeline(fp16 *ptr_res, fp16 *ptr_inp,
                                    const int N, const int C, const int H,
                                    const int W) {
  const int c = N * C, w = H * W;
  dim4 shape = {1, c, 1, w};
  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);

  float scalar_c = 0.25;

  int block_c = LANE_NUM;
  int block_w = 512;
  dim4 in_shape = {1, block_c, 1, block_w};

  for (int idx_c = 0; idx_c < c; idx_c += block_c) {
    enable_pipeline();
    int real_c = min(block_c, c - idx_c);
    for (int idx_w = 0; idx_w < w; idx_w += block_w) {
      int real_w = min(block_w, w - idx_w);
      dim4 real_shape = {1, real_c, 1, real_w};
      dim4 offset = {0, idx_c, 0, idx_w};
      auto in = make_tensor<fp16>(in_shape, real_shape);
      auto res = make_tensor<fp16>(in_shape, real_shape);
      dma::load(in, in_gtensor.sub_view(real_shape, offset));
      tiu::fadd(res, in, scalar_c);
      dma::store(res_gtensor.sub_view(real_shape, offset), res);
    }
  }
}
```

多核并行

目前 PPL 支持针对 BM1684x、BM1688、BM1690 和 SG2380 芯片编写算子，其中 BM1684x 仅有一个 core，BM1688 有两个 core，BM1690 有八个 core，SG2380 有四个 core；而 core 与 core 之间是并行运算。于是，可以将数据按照 core 的数目进行切分，每个 core 上进行相同的运算，但涉及的数据块不一样，理论上可以提升 CORE_NUM 倍数的性能。

上述加法 kernel 可进一步优化为：

```
#ifdef __bm1690__
#define CORE_NUM 8
#elif __bm1688__
#define CORE_NUM 2
#else
#define CORE_NUM 1
#endif

__KERNEL__ void add_kernel_multi_core(fp16 *ptr_res, fp16 *ptr_inp, int W) {
  // 在TPU上运行的主函数需要加上 __KERNEL__ 关键字
  // 在多核（bm1690等）上运行的主函数需要添加 MULTI_CORE 关键字
```

(续下页)

(接上页)

```
  // 或者不使用 MULTI_CORE 关键字，直接可以调用 ppl::set_core_num
  const int N = 8;
  const int C = 32;
  const int H = 1;
  ppl::set_core_num(CORE_NUM);    // 获取当前程序运行使用的总的核数量
  int core_num = ppl::get_core_num();    // 获取当前程序运行使用的总的核数量
  int core_idx = ppl::get_core_index();  // 获取当前是在哪个核上运行
  if (core_idx >= core_num) {
    return;
  }

  assert(W > 0);
  dim4 global_shape = {N, C, H, W};
  // 使用tensor封装global memory上的数据
  auto in_gtensor = gtensor<fp16>(global_shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(global_shape, GLOBAL, ptr_res);

  int slice = div_up(W, core_num);        // 计算每个核上处理的 W size
  int cur_slice = min(slice, (W - slice*core_idx)); // 计算当前核上处理的 W size
  int slice_offset = core_idx * slice;    // 计算当前核处理的数据在 ddr 上的偏移

  int block_w = 512;                      // 定义单个核上，每次循环处理的 W block size
  dim4 block_shape = {N, C, H, block_w};  // 定义单次循环处理的数据 shape
  // 申请 tpu local memory 上的内存，由于 PP L是在编译期计算 local memory 大小，
  // 所以 tensor 初始化的 shape 的值在编译期必须是常量
  tensor<fp16> in_tensor, res;
  float scalar_c = 0.25;

  for (int w_idx = 0; w_idx < cur_slice; w_idx += block_w) {
    enable_pipeline();                    // 开启 PPL 自动流水并行优化
    int tile_w = min(block_w, cur_slice - w_idx);  // 当前循环需要处理的 W 尺寸
    dim4 cur_shape = {N, C, H, tile_w};   // 当前循环的输入数据 shape

    dim4 offset = {0, 0, 0, slice_offset + w_idx};  // 当前需要计算的数据在 ddr 上的偏移
    // 从 ddr 上 load 数据到 tpu 上
    dma::load(in_tensor, in_gtensor.sub_view(cur_shape, offset));
    tiu::fadd(res, in_tensor, scalar_c);    // 做加法
    // 将数据从 local mem 到 ddr
    dma::store(res_gtensor.sub_view(cur_shape, offset), res);
  }
}
```

首先，设置计算所需 core 的数目；然后，将 W 维度按照 core 的数目进行切分；再对每个 core 上的数据进行切分并开启流水并行。

2.1.3 性能优化统计

上述加法代码详见 ppl/example/cxx/arith/add_pipeline.pl，性能测试数据规模为 shape = {8, 32, 1, 4096}，具体消耗时间如下表：

表 1: 不同优化方式耗时统计

实现方式	耗时 (us)	提升幅度
原始实现	62.952	/
对齐 lane	62.44	0.82%
流水并行	19.141	226.21%
多核并行	8.296	130.73%

性能统计皆以对齐 LANE_NUM 的代码形式为标准。