PPL 快速入门指南

SOPHGO

2024 年 12 月 30 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 是基于 c/c++ 语法扩展的、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
```

(续下页)

(接上页)

```
|   |   └── tpuv7-emulator/  # device 代码依赖的头文件和库
|   ├── sg2380/        # sg2380 runtime 库
|   |   ├── 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 代码示例

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

#ifdef __bm1690__
#define CORENUM0 8
#else
#define CORENUM0 1
#endif

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

__KERNEL__ void add_pipeline(fp32 *ptr_res, fp32 *ptr_inp, int W) {
  // 在TPU上运行的主函数需要加上__KERNEL__关键字
  const int N = 1;
  const int C = 1;
  const int H = 1;
  // 设置KERNEL函数运行在几个核上（用于编译期计算L2 Memory，不会在 deivce 上实际运行）
  ppl::set_core_num(CORENUM0);
  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gtensor封装global memory上的数据
  auto in_gtensor = gtensor<fp32>(global_shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp32>(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 = 16; // 定义单个核上，每次循环处理的W block size。
  dim4 block_shape = {N, C, H, block_w}; // 定义单词循环处理的数据shape
  auto in_tensor = tensor<fp32>(block_shape);
  // 申请tpu local memory上的内存，由于PPL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
  auto res = tensor<fp32>(block_shape);
  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
    auto cur_tensor =
        in_tensor.view(cur_shape); // 修改tensor的shape为此次循环处理的数据shape
    auto cur_res = res.view(cur_shape);

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

1.4 PPL 编译

ppl-compile:

PPL 编译使用的命令是 ppl-compile：

ppl-compile 命令主要用于将 host 端的代码转换成 device 端运行的代码。

命令的一般形式：

```
ppl-compile <path_to_plfile> --print-debug-info --print-ir --gen-test --gen-ref --chip <chip_name>
↪ --O2 --o <path_to_save_result>
```

命令组成

- **path_to_plfile**：指定 pl 文件的位置。
- **–print-debug-info**：打印 MLIR 的 debug 信息。
- **–print-ir**：打印 ir，指 dump ir 至输出文件夹中。
- **–chip**：指定目标芯片类型 <XXX>。
- **–O2**：优化等级 2，输出 ppl-frontend、ppl-opt、final 三层 ir 和 device 代码。
- **–o**：指定生成的结果文件存放的位置。

可选参数

- **–x**：指定 pl 文件中代码的语言类型，在 PPL 编程中，pl 文件使用 C++ 风格，因此该选项一般设置为 --x c++。
- **–function**：指定 pl 文件中哪些函数需要进行前端处理。一般情况下需要将所有函数放入处理范围，所以会设置 --function=*。
- **–O0**：优化等级 0，无 ppl 优化。
- **–O1**：优化等级 1，进行 ppl-canonicalize 优化。
- **–O2**：优化等级 2，进行 ppl-canonicalize 和 ppl-pipeline 优化。
- **–O3**：优化等级 3，进行所有 ppl 优化
- **–gen-test**：进行对照测试。
- **–gen-ref**：生成对照组（即不进行 pipeline 等 ppl 优化的对照组，用于验证 ppl 优化的正确性）。
- **–print-debug-info**：打印 MLIIR 的 debug 信息。
- **–print-ir**：输出 IR。
- **–desc**：生成 Descriptor 模式的 host 代码，与 tpu-mlir 对接时使用。
- **–device**：需要先设置–desc，生成 Descriptor 模式的 device 代码，与 tpu-mlir 对接时使用。

以 samples/add_pipeline/ppl/add_pipeline.pl 源码为例，运行编译如下：

```
ppl-compile samples/add_pipeline/ppl/add_pipeline.pl --print-debug-info --print-ir --chip
↪bm1684x --O2 --o test
```

运行此命令后，会在当前目录生成 test 目录，目录结构如下：

```
├─── device/          # 编译后生成的device端代码存放目录
├─── host/            # 编译后生成的host端代码存放目录
├─── include/         # 编译后生成的头文件存放目录
├─── src/             # 编译后生成的应用端测试代码存放目录
├─── *.mlir           # 如果设置了--print-ir，则会dump编译过程中的ir文件
```

ppl-compile.py:

也可以使用辅助工具 ppl_compile.py 来编译运行，该脚本用于自动化运行流程，包括编译、运行测试案例等。

命令组成

- **ppl_compile.py**：这是主命令，调用 ppl_compile.py 脚本。
- –**src**：此选项后跟一个参数，指定被运行的 pl 文件的路径。pl 文件包含了要执行的测试代码或者测试配置。
- –**chip**：此选项后跟一个参数，指定目标芯片类型。这个参数帮助脚本确定如何编译和执行，以适应不同芯片的特性。可选的参数有 bm1684x、bm1690、bm1688、sg2380。

可选参数

- –**out**：此选项后跟一个参数，指定运行以及 ppl 编译生成文件的目录；如果未指定此参数，则默认生成使用 pl 文件名称且添加 test_ 前缀的文件目录。
- –**gen_ref**：需要代码中存在 __TEST__ 修饰的测试函数。添加此选项后，会生成未经过 PPL 优化的 kernel 函数 (ref) 和优化后的 kernel 函数 (tar)，并对比两个函数的运行结果，结果以 npz 格式存储在生成的 test_xxx/data 目录下。
- –**disable_print_ir**：表示禁用在优化阶段打印所有 pass 前后的 IR，action=’store_true’。
- –**disable_pipeline**：表示禁用 ppl-pipeline 优化，action=’store_true’。
- –**disable_canonicalize**：表示禁用 ppl-canonicalize 优化，action=’store_true’。
- –**gdb**：使用 gdb 运行 test_case 程序，默认为 False。
- –**profiling**：生成 profiling 文件，需要用户自行安装 PerfAI 工具，默认为 False。
- –**desc**：生成 Descriptor 模式下的代码，与 tpu-mlir 对接时使用，默认为 False。
- –**core_num**：设置多核代码运行时所需的核数量，默认数量为 1。
- –**bank_conflict**：控制生成 device 端代码时，是否考虑 bank 冲突，默认为 True。
- –**autotune**：控制自动 profiling，默认为 False。
- –**mode**：此选项后跟一个参数，指定程序运行的环境，可选的参数有 cmodel、pcie、soc，默认为 cmodel。

```
ppl_compile.py --src samples/add_pipeline/ppl/add_pipeline.pl --chip bm1684x --gen_ref
```

运行此命令后，会在当前目录下生成名称为 test_mm2_fp16 的文件夹，若未使用–out 指定文件夹名称，则自动使用 xxx.pl 文件中的 xxx，并加上 test_ 前缀，生成 test_xxx 文件夹。data 目录下 xxx_input.npz 为 __TEST__ 函数生成的输入数据，xxx_ref.npz 为未经 PPL 优化的 kernel 函数计算结果，xxx_tar.npz 为经过 PPL 优化后的 kernel 函数计算结果。用户可以使用 torch 编写自己的测试代码，并与 xxx_tar.npz 对比结果，确认 kernel 运算结果是否正确。目录结构如下：

```
├─── data/              # 代码运行后的数据结果存放目录
├─── device/            # ppl 编译后生成的device端代码存放目录
├─── host/              # ppl 编译后生成的host端代码存放目录
├─── include/           # ppl 编译后生成的头文件存放目录
├─── lib/               # 编译test_add_pipeline目录后生成的库文件存放目录
├─── profiling/         # 运行后生成的profiling文件存放目录
├─── src/               # ppl 编译后生成的应用端测试代码存放目录
├─── *.mlir             # 如果设置了--print-ir，则会dump编译过程中的ir文件
├─── CMakeLists.txt/    # 设定test_add_pipeline目录编译选项
├─── test_case          # 编译test_add_pipeline目录得到的二进制文件
```

ppl_compile.py 是用来辅助编译测试的，它与 ppl_compile 二进制文件的区别在于：

- 当 ppl_compile.py 脚本不指定使用–gen_ref 选项时，与 ppl_compile 二进制文件功能一致
- 当指定使用–gen_ref 选项时，ppl_compile.py 脚本会生成未经过 ppl 优化的对照组 ref，然后分别运行对照组和目标组 (tar) 代码，并对比两者的计算结果。

以下通过一个示例简要介绍 ppl_compile.py 工作的完整流程：

```
# 基于 host 端代码 mm2_fp16.pl 生成，ppl-frontend、opt、final 三层ir文件
# 基于 final.mlir 生成 device 端运行的代码
ppl-compile ./examples/cxx/matmul/mm2_fp16.pl --print-debug-info
--print-ir --gen-test --gen-ref --chip bm1690 --O2 --o ./test_mm2_fp16

# 编译 test_XXX 目录下的 device 端代码，生成可执行文件 test_case
cmake .. -DDEBUG=False -DCHIP=bm1690 -DDEV_MODE=cmodel
make install

# 执行 test_case，检验 ppl 优化的正确性
./test_mm2_fp16/test_case

# 最后调用 npz_help.py 脚本来执行比较操作，具体而言，npz_help.py 中包含多种选项，
# 使用 compare 选项表示比较操作。比较的对象是两个文件，
# 一个是通过TPU-KERNEL编程优化得到的输出文件，
# 另一个是增加了PPL特有优化得到的输出文件。
# 对两个文件中包含两种优化计算的结果张量进行了三种比较，
# 得到最小余弦相似度、最小欧几里得相似度和最小信噪比，
# 并在最小余弦相似度和最小欧几里得相似度大于 0.99，
# 且最小信噪比为0时才认定两种优化得到的结果一致，对比通过。
npz_help.py compare ./test_mm2_fp16/data/mm2_fp16_ref.npz
./test_mm2_fp16/data/mm2_fp16_tar.npz -vv --tolerance 0.99,0.99
```

在执行到任意一步出现问题时，ppl_compile.py 都会中断进程并返回当前步骤的错误信息，用户可根据返回结果方便地进行调试。

PPL 也提供了 AddPPL.cmake，方便将 PPL 编译集成到 cmake 中，用法如下：

1.5 Samples 使用说明

1.5.1 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/      #
```

1.5.2 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
```

(续下页)

(接上页)

```
./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
```

1.5.3 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
```

1.5.4 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。

# 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
```

1.5.5 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
```

1.5.6 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"
```

（续下页）

(接上页)

```
#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>
#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) {
```

(续下页)

(接上页)

```
    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;
    ((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");
```

(续下页)

(接上页)

```
  // 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;
  // 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
```

(续下页)

(接上页)

```
  return 0;
}
```

1.5.7 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"

// 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;
}
```

1.5.8 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})
```

（续下页）

(接上页)

```
# 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)
  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)
```

1.5.9 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})
```

1.5.10 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。

1.5.11 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。