文件编号	FIL-C-241227-901410-SWX
上级文件编号	FIL-C-240929-901068-MDX

PPL 算子开发

目录

修订记录

版本号	修订日期	作者	修订内容
1.0	2024.12.27	李胜超	首次制作

第一章 总则

第1条 目的

1） 介绍使用 PPL 编译器进行算子开发及算子性能调优的方法。

第2条 适用范围

1） 所有需要使用 PPL 编译器进行算子开发的员工。

第3条 名词定义

1） PPL：Primitive Programming Language，是一个代码转换工具，它提供了一套 C/C++ 和一套 triton 风格的高级编程接口、负责 TPU 上的内存管理、能够自动进行必要的同步控制和并行计算优化。

第二章 环境配置

目前支持三种获取 PPL 工程或者 release 包的方式，使用任意一种均可进行算子开发。

第4条 从 gerrit 上下载并配置 PPL 工程

```
# 安装 git-lfs
curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
sudo apt-get install git-lfs
# 如何使用 git-lfs
https://wiki.sophgo.com/pages/viewpage.action?pageId=127019078

# clone PPL 工程
git config --global http.sslVerify false

git clone https://shengchao.li@gerrit-ai.sophgo.vip:8443/a/ppl && (cd ppl && mkdir -p .git/hooks && curl -Lo `git rev-parse --git-dir`/hooks/commit-msg https://shengchao.li@gerrit-ai.sophgo.vip:8443/tools/hooks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)

git config --global http.sslVerify true

# 配置 http
cd ppl
git config lfs.https://gerrit-ai.sophgo.vip/ppl.git/info/lfs.locksverify false
git config --global credential.helper store
git config http.sslVerify false

# 获取第三方库
cd ppl/third_party
```

```
scp
guest@172.22.12.22:/data/ppl_third_party/llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release.tar.gz .
(password:123456)

tar -xvhf llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release.tar.gz
mv llvm+mlir-17.0.0-x86_64-linux-gnu-ubuntu-18.04-release/ llvm_release/
```

第5条 从 ftp 服务器下载 PPL release 包

```
# 下载 release 包
# 服务器 ip 地址：172.28.141.89
# 用户名：AI
# 密码：SophgoRelease2022
# 快速拉取指令
wget --ftp-user=AI --ftp-password=SophgoRelease2022 -c ftp://172.28.141.89/sophon-sdk/ppl/daily_build/Master_20241208_053000/ppl_v1.4.97-g8478057a-20241208.tar.gz
```

第6条 从 github sophgo 官方页面获取 PPL release 包

```
# release 包下载地址
https://github.com/sophgo/PPL
# 快速拉取指令
wget
https://github.com/sophgo/PPL/releases/download/v1.4.91/ppl_v1.4.91-gc2fc3685-20241202.tar.gz
```

第7条 编译方式

```
# 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:/workspace --name ppl sophgo/ppl:latest
docker exec -it ppl bash

# 使用 sophgo 通用镜像
docker run --privileged -itd -v $PWD:/workspace --name ppl sophgo/tpuc_dev:latest
docker exec -it ppl bash

# PPL 工程编译
cd /workspace
```

```
source envsetup.sh
# release 包不需要编译
./build.sh (DEBUG)
```

第三章 工程目录介绍

第8条 release 包根目录简介

1) bin/ 目录存放 PPL 编译工具(ppl-compile)和 desc 模式编译工具。
2) doc/ 目录存放《PPL快速入门指南.pdf》和《PPL开发参考手册.pdf》。

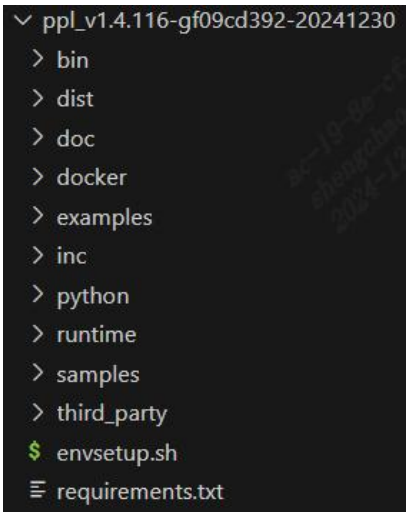

Figure 1.

3) docker/ 目录存放 PPL 专用镜像的编译脚本。
4) example/ 目录存放使用 C/C++ 和 python 指令接口开发的示例算子。
5) inc/ 目录存放 PPL 指令接口头文件。
6) python/ 目录存放使用 python 开发的辅助工具。
7) runtime/ 目录存放不同芯片的 cmodel 运行时库和依赖的头文件。
8) samples/ 目录存放 PPL 代码在不同芯片不同模式下运行的示例。
9) third_party/ 目录存放第三方库。
10) envsetup.sh 为环境变量设置脚本。

第9条 PPL 工程根目录简介

1) 参考《PPL编译器开发》COP。

第四章 基本算子开发

第10条 以 add 算子为例，介绍使用 PPL 进行算子开发的全过程

1) PPL device 端代码实现功能的编写逻辑为将 gmem 数据上传到 lmem→

在 lmem 上计算→从 lmem 下载结果数据至 gmem。

```
// 示例代码位于 ppl/examples/cxx/arith/add_pipeline.pl
__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 gmem 上的数据
  auto in_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_inp);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);
  auto in = tensor<fp16>(shape);   // 使用 tensor 申请 lmem 上的空间
  auto res = tensor<fp16>(shape);

  float scalar_c = 0.25;
  dma::load(in, in_gtensor);          // 从 gmem 上 load 数据到 lmem
  tiu::fadd(res, in, scalar_c);       // 进行加法操作
  dma::store(res_gtensor, res);       // 将结果从 lmem 上 store 到 gmem
}
```

2) ppl 使用 gtensor 抽象表示 gmem/l2mem 上的内存及其 shape、stride 和 offset 等信息；使用 tensor 抽象表示 lmem 上的内存及其 shape、stride 和 offset 等信息。

3) 需要将 kernel 函数传入参数中的 gmem 内存地址绑定到 gtensor 并指定为 GLOBAL 类型；使用 shape 申请 lmem 的地址空间。

第11条 算子开发注意事项

1) gmem 和 l2mem 均使用 gtensor 结构体，但 gmem 绑定 gtensor 时必须传入地址；l2mem 绑定 gtensor 时可以缺省地址。

```
// 示例代码位于 ppl/examples/cxx/matmul/mm2_fp16_sync.pl
auto left_gt = gtensor<fp16>(left_global_shape, GLOBAL, ptr_left);
auto right_gt = gtensor<fp16>(right_global_shape, GLOBAL, ptr_right);
auto res_gt = gtensor<fp16>(res_global_shape, GLOBAL, ptr_res);

gtensor<fp16> l2_left(l2_left_max_shape, L2), l2_right(L2);
```

2) 当使用动态 block时，即需要将 block 信息作为参数传入 kernel 函数，必须添加 const 修饰符。

```
// 示例代码位于 ppl/examples/cxx/arith/add_dyn_block.pl
__KERNEL__ void add(fp16 *ptr_res, fp16 *ptr_inp, int W, const int block_w){...}
```

3) block_shape 与 real_shape：

```
// 示例代码位于 ppl/examples/cxx/conv/fconv2d_single_loop.pl
dim4 bias_block_shape = {1, block_oc, 1, 1};
auto bias_tensor = tensor<fp32>(bias_block_shape, TPU_COMPACT);
dim4 bias_real_shape = {1, curr_oc, 1, 1};
dim4 bias_offset = {0, idx_oc, 0, 0};
auto bias = bias_tensor.view(bias_real_shape);

dim4 weight_block_shape = {1, block_oc, block_ic / nic * kh * kw, nic};
dim4 weight_real_shape = {1, curr_oc, div_up(curr_ic, nic) * kh * kw, nic};
auto weight = make_tensor<fp16>(weight_block_shape,weight_real_shape,TPU_COMPACT);
```

a) block_shape 为进行内存分配的形状大小，real_shape 为在循环中计算出的实际进行计算的形状大小。

b) 参与内存分配的 block_shape 必须是常数，或者作为参数传入(参数必须有 const 修饰)。

c) 支持先使用 block_shape 申请 lmem 内存，然后使用 view 取用内存中实际需要计算的大小。

d) 支持使用 make_tensor 同时传入 block_shape 和 real_shape 直接取用内存中实际需要计算的大小。

第12条 精度验证

1) 当确认 tensor 信息与指令调用无误，使用 ppl_compile.py 脚本对比经过 PPL 优化的代码和未经过 PPL 优化的代码，验证 PPL 优化的正确性。

```
ppl_compile.py --src examples/cxx/arith/add_pipeline.pl --chip bm1684x --gen_ref
```

2) 编写 pytorch 版本算子，验证 ppl 算子精度。

a) 从 test_add_pipeline/data 目录获取 tpu 输入和输出（add_pipeline_tar.npz）。文件路径 dir 与 pl 文件同名即可。npz 文件中数据的排序方式，与 kernel 函数参数中地址的排序一致。

```python
import torch
from tool.test_process import test_processor

@test_processor
def compute_add_pipeline_main(tpu_out, shapes, **kwargs):
    left = tpu_out["1"]
    N, C, H, W = shapes
    tensor_left = torch.tensor(left.reshape([N, C, H, W]), dtype=torch.float)
    torch_out = torch.add(tensor_left, 0.25)
    return {"0": torch_out.numpy()}

if __name__ == "__main__":
    compute_add_pipeline_main(dir="add_pipeline", shapes=(8,32,1,4096))
```

Figure 2.

b) 需要用户编写 pytorch 的计算逻辑，然后直接使用 PPL 封装的验证函数，对比 tpu 和 pytorch 的计算结果。

第13条 代码调试

1) 在调用 ppl_compile.py 脚本时，添加 --gdb 选项，即可自动编译并运行 debug 版本的 test_case 程序，并在程序运行时进入调试界面；调试的源码为 ppl 编译生成的 tpu_kernel 代码，代码存放在 test_add_pipeline/device 目录下。

2) 支持 tensor 信息打印，不支持数据打印。

```
97              __ppl_tensor_info v66 = {.shape = v62, .stride = v20, .addr = v21.addr + v65, .dtype = DT_FP16, .mode
 = 2, .align_mode = 4, .size = v61, .offset = v65, .unsigned_flag = 0, .default_stride = false};
(gdb)
98              if (v66.size) {
(gdb) p v66
$3 = {shape = {n = 8, c = 32, h = 1, w = 512}, stride = {n = 131072, c = 4096, h = 4096, w = 1}, addr = 4384096256,
  dtype = DT_FP16, mode = 2, align_mode = 4, size = 512, offset = 0, unsigned_flag = false, default_stride = false}
(gdb)
```

Figure 3.

3) 支持在 pl 文件直接打印 tensor 信息，结果会输出在终端。打印 tensor 信息需要使用 to_string 函数进行数据提取。

```
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
  dma::load(in_tensor, in_gtensor.sub_view(cur_shape, offset));  // 从 ddr 上 load 数据到 tpu 上
  tiu::fadd(res, in_tensor, scalar_c);     // 做加法      liang.chen, 6个月前 • refine shape infer
  ppl::print("in tensor info : %s\n", to_string(in_tensor));
  dma::store(res_gtensor.sub_view(cur_shape, offset), res);    // 将数据从 local mem 到 ddr
}
```

```
in tensor info :
        shape: {8, 32, 1, 512}
        stride: {0, 0, 0, 0}
        addr: 16384
        mode: 0
        unsigned_flag: 0

in tensor info :
        shape: {8, 32, 1, 512}
        stride: {0, 0, 0, 0}
        addr: 16384
        mode: 0
        unsigned_flag: 0
```

Figure 4.

4) 当存在精度问题时，同步截断 ppl 算子和 pytorch 版本算子，进行逐层结果对比。具体操作为：

a) 在 kernel 函数中添加测试参数用于 dump 中间层数据（示例代码位于ppl/examples/cxx/llm/tgi/w4a16_matmul.pl）。

```
// __KERNEL__ void matmul_w4a16_rtrans_mc(fp16 *ptr_res, fp16 *ptr_left,
//                                        uint8 *ptr_right, uint8 *ptr_scale_zp,
//                                        fp16 *ptr_test, int8 *ptr_test_u8, int
//                                        M, int K, int N, int group_size, const
//                                        int m_slice, const int n_slice) {
__KERNEL__ void matmul_w4a16_rtrans_mc(fp16 *ptr_res, fp16 *ptr_left,
                                       uint8 *ptr_right, uint8 *ptr_scale_zp,
                                       int M, const int K, const int N,
                                       const int group_size, const int m_slice,
                                       const int n_slice) {
```

Figure 5.

b) 在 pytorch 算子和 PPL 算子中截断输出，进行数据比对，定位精度损失出现的位置（示例代码位于 ppl/examples/cxx/llm/tgi/w4a16_matmul.py）。

c) PPL 算子可以在任意一步操作后进行 store 操作，将计算结果 dump 下来。

```
// sub_res = right_i8 - zp_i8
dim4 zp_shape_ = {1, cur_n, groups, 1};
dim4 right_group_shape = {1, cur_n, groups, group_size};
auto sub_res_u8 = make_tensor<int8>(right_block_shape, right_real_shape);
tiu::sub(sub_res_u8.view(right_group_shape),
         right_local_u8.view(right_group_shape),
         zp_local_u8_.view(zp_shape_), 0, RM_HALF_AWAY_FROM_ZERO, true);
// // 对齐
// dma::store(test_u8_gtensor.sub_view(right_real_shape, right_offset),
//            sub_res_u8);
```

Figure 6.

```
    sub_res_tensor = right_int8_tensor.reshape([N, group, group_size]) - zp_int8_tensor.reshape([N, group, 1])
    sub_res_f16_tensor = sub_res_tensor.to(torch.float16)
    mul_res_f16 = sub_res_f16_tensor.reshape([N, group, group_size]) * scale_fp16_tensor.reshape([N, group, 1])
    torch_out = F.linear(tensor_left, mul_res_f16.reshape([N, K]).to(torch.float))

    return {"5": sub_res_tensor.numpy()}      You, 现在 • Uncommitted changes

matmul(dir="w4a16_matmul", M=1240, K=4096, N=4096, group_size=128)
```

Figure 7.

d) pytorch 算子 return 时进行截断，直接使用与 PPL 截断相同计算的结果进行结果对比。数字 5 表示使用 sub_res_tensor 与 tpu 计算结果 npz 文件中的第 5 个数据集进行比对。

第五章 多核算子开发

目前 PPL 支持针对 BM1684x、BM1688、BM1690 和 SG2380 芯片编写算子，BM1684x 仅有一个 core，BM1688 有两个 core，BM1690 有八个 core，SG2380 有四个 core。以 add 算子为例，介绍多核算子开发的全流程。

第14条 使用 MULTI_CORE 关键字

1) MULTI_CORE 是封装了一个模板参数来设置宏数量，如果编写了 __TEST__ 函数，则可以在 test 函数中通过模板参数传入 core 数量，如果未编写 __TEST__ 函数则不能使用此方式，或者需要手动将 kernel 函数实例化。

```
MULTI_CORE
__KERNEL__ void add_kernel_multi_core(fp16 *ptr_res, fp16 *ptr_inp, int W) {

__TEST__ void add() {
  const int N = 8;
  const int C = 32;
  const int H = 1;
  int W = 4096;
  dim4 shape = {N, C, H, W};
  fp16 *res = rand<fp16>(&shape);
  fp16 *inp = rand<fp16>(&shape, -32.21f, 32.32f);
  add_kernel_multi_core<8>(res, inp, W); // 使用 6 个核运行核函数
}
```

Figure 8.

2) MULTI_CORE 关键字是在编译期生效，kernel 函数运行的时候是没有这条指令的。因此，需要调用 get_core_num() 来获取运行时 kernel 函数使用的 core 数量，并且需要调用 get_core_index() 来获取运行时 kernel 函数使用的 core 索引。

第15条 调用 set_core_num()

1) set_core_num() 也是在编译期生效，同样需要调用 get_core_num() 和 get_core_index() 来获取运行时 kernel 函数使用的 core 数量和索引。

2) 可以通过宏来设置 core 数量，然后编译的时候将宏传入；也可以直接使用 core_num 作为 kernel 函数的参数传入(作为参数时必须加上 const 修饰符，可参考 ppl/examples/cxx/llm/mlp_multicore.pl 文件)，以便在编译期修改 core 数量。

```
__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
  // 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;
  }
```

Figure 9.

第六章 性能测试

与 pytorch 版本算子精度对比通过后，进行性能测试的方式分为两种。以 add 算子为例介绍两种性能测试方式。

第16条 使用 ppl_compile.py 脚本进行单测

1) 设置好切分策略后，直接调用 ppl_compile.py 脚本，添加 --profiling 选项进行单测。

```
ppl_compile.py --src examples/cxx/arith/add_pipeline.pl --chip bm1684x --gen_ref --profiling
```

```
==============================
profiling result:

==============================
================================================================================
CoreId  Parallelism(%) totalTime(us) TiuWorkingRatio totalTiuCycle totalGdmaCycle GdmaDdrAvgBandwidth(GB/s) GdmaL2AvgBandwidth(GB/s)
0       201.69%        8.163         1.79%           146           8159           64.29                    0
1       201.69%        8.163         1.79%           146           8159           64.29                    0
2       201.69%        8.163         1.79%           146           8159           64.29                    0
3       201.69%        8.163         1.79%           146           8159           64.29                    0
4       201.69%        8.163         1.79%           146           8159           64.29                    0
5       201.69%        8.163         1.79%           146           8159           64.29                    0
6       201.69%        8.163         1.79%           146           8159           64.29                    0
7       201.69%        8.163         1.79%           146           8159           64.29                    0
Overall 201.69%        8.163         1.79%           0.15us        8.16us         64.29                    0
================================================================================
```

Figure 10.

2) 结果会以表格的形式打印在终端，test_add_pipeline/profiling 目录下会生成对应的 profiling 文件。

第17条 使用 ppl_compile.py 脚本进行批量测试

1) 需要在 .pl 文件中添加 __AUTOTUNE__ 函数，编写所需测试的切分策略，运行调用 ppl_compile.py 脚本，添加 --autotune 选项进行批量测试。

```
ppl_compile.py --src examples/cxx/arith/add_pipeline.pl --chip bm1684x --gen_ref --autotune
```

```
__AUTOTUNE__ void add_profile() {
  for (int w = 1024; w < 4096; w+=1024) {
    add_kernel_multi_core(nullptr, nullptr, w);
  }
}
```

Figure 11.

2) 结果会以表格的形式打印在终端，test_add_pipeline/profiling 目录下会生成对应的 profiling 文件。

```
================================
add_kernel_multi_core with args: (1024) profiling result:

================================
================================================================================
CoreId  Parallelism(%) totalTime(us) TiuWorkingRatio totalTiuCycle totalGdmaCycle GdmaDdrAvgBandwidth(GB/s) GdmaL2AvgBandwidth(GB/s)
0       201.75%        2.403         2.08%          50            2399           54.73                    0
1       201.75%        2.403         2.08%          50            2399           54.73                    0
2       201.75%        2.403         2.08%          50            2399           54.73                    0
3       201.75%        2.403         2.08%          50            2399           54.73                    0
4       201.75%        2.403         2.08%          50            2399           54.73                    0
5       201.75%        2.403         2.08%          50            2399           54.73                    0
6       201.75%        2.403         2.08%          50            2399           54.73                    0
7       201.75%        2.403         2.08%          50            2399           54.73                    0
Overall 201.75%        2.403         2.08%          0.05us        2.40us         54.73                    0
================================================================================
================================
add_kernel_multi_core with args: (2048) profiling result:

================================
================================================================================
CoreId  Parallelism(%) totalTime(us) TiuWorkingRatio totalTiuCycle totalGdmaCycle GdmaDdrAvgBandwidth(GB/s) GdmaL2AvgBandwidth(GB/s)
0       201.71%        4.323         1.90%          82            4319           60.75                    0
1       201.71%        4.323         1.90%          82            4319           60.75                    0
2       201.71%        4.323         1.90%          82            4319           60.75                    0
3       201.71%        4.323         1.90%          82            4319           60.75                    0
4       201.71%        4.323         1.90%          82            4319           60.75                    0
5       201.71%        4.323         1.90%          82            4319           60.75                    0
6       201.71%        4.323         1.90%          82            4319           60.75                    0
7       201.71%        4.323         1.90%          82            4319           60.75                    0
Overall 201.71%        4.323         1.90%          0.08us        4.32us         60.75                    0
================================================================================
================================
add_kernel_multi_core with args: (3072) profiling result:

================================
================================================================================
CoreId  Parallelism(%) totalTime(us) TiuWorkingRatio totalTiuCycle totalGdmaCycle GdmaDdrAvgBandwidth(GB/s) GdmaL2AvgBandwidth(GB/s)
0       201.70%        6.243         1.83%          114           6239           63.07                    0
1       201.70%        6.243         1.83%          114           6239           63.07                    0
2       201.70%        6.243         1.83%          114           6239           63.07                    0
3       201.70%        6.243         1.83%          114           6239           63.07                    0
4       201.70%        6.243         1.83%          114           6239           63.07                    0
5       201.70%        6.243         1.83%          114           6239           63.07                    0
6       201.70%        6.243         1.83%          114           6239           63.07                    0
7       201.70%        6.243         1.83%          114           6239           63.07                    0
Overall 201.70%        6.243         1.83%          0.11us        6.24us         63.07                    0
================================================================================
```

Figure 12.

第七章 性能调优

当算子性能较差时，推荐分三步进行性能优化。以 add 算子为例介绍性能优化流程。

第18条 对齐 LANE_NUM

1) 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。
2) 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。
3) 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);
}
```

4) 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;
```

5) 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};
```

6) 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};
```

7) 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) {
  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);
    }
  }
}
```

第19条 数据切分与流水并行

1) lmem 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 lmem，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。
2) 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。
3) 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);
        dma::store(res_gtensor.sub_view(cur_shape, offset), res);
    }
}
```

4) 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。

5) 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);
      }
    }
}
```

第20条 多核并行

1) 当算子针对的芯片有多个 core 时，由于 core 与 core 之间是并行运算，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。

2) 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
  // 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 gmem 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 lmem 上的内存，由于 PPL 是在编译期计算 lmem 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 lmem 到 gmem
    dma::store(res_gtensor.sub_view(cur_shape, offset), res);
  }
}
```

3) 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。

第八章 checklist

Cat.	编号	Check 项目	Check 方法与参考值	Check 结果	PR & 时间
环境配置	1.	PPL 工程配置 git-lfs	使用 PPL 工程开发算子时是否配		

			置 git-lfs。 参考值：是。		
	2.	PPL 工程配置 http 环境	使用 PPL 工程开发算子时是否配置 http 环境。 参考值：是。		
	3.	设置环境变量	是否设置环境变量。 参考值：是。		
算子开发	4.	gtensor 结构体	是否注意到 gmem 和 l2mem 绑定 gtensor 结构体的区别。 参考值：是。		
	5.	动态 block	是否注意到动态 block 参数需要添加 const 关键字修饰。 参考值：是。		
	6.	block_shape 与 real_shape	是否注意到 block_shape 与 real_shape 的区别。 参考值：是。		
	7.	设置 core 数量	是否正确使用 MULTI_CORE 关键字或者 set_core_num 函数设置 core 数目。 参考值：是。		
	8.	获取 core 数目和索引	是否使用了 get_core_num 和 get_core_index 函数获取 core 数目和索引。		

			参考值：是。		
正确性验证	9.	验证 PPL 优化正确性	是否使用 --gen_ref 选项验证 PPL 优化的正确性。 参考值：是。		
	10.	验证 PPL 算子精度	是否编写 pytorch 算子验证 PPL 算子的精度。 参考值：是。		
性能测试及调优	11.	批量性能测试	是否编写 __AUTOTUNE__ 函数设置切分策略。 参考值：是。		
	12.	性能调优	channel 是否对齐 LANE_NUM。 参考值：是。		
	13.	性能调优	是否切分适当并开启流水并行。 参考值：是。		
	14.	性能调优	多核芯片是否使用了多个 core。 参考值：是。		