PPL 开发参考手册

SOPHGO

2024 年 10 月 24 日

目录

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

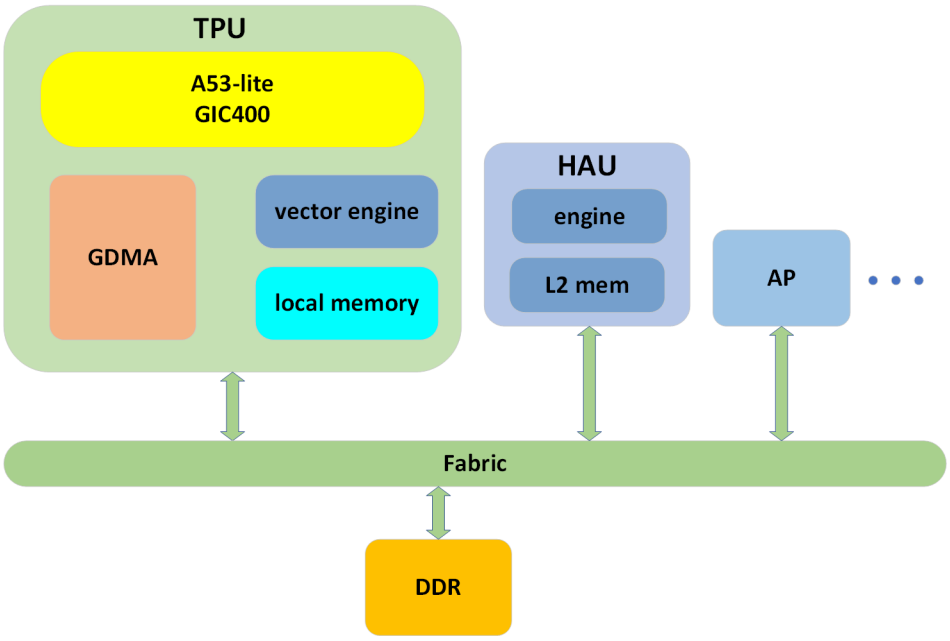
CHAPTER 1

简介

1.1 TPU 简介

1.1.1 TPU 架构

架构概览



TPU(Tensor Processing Unit) 是一种有多个计算核的架构，系统架构如图所示。TPU 系统主要由 TPU core、L2 SRAM(L2 mem)、片外 DDR 存储 (Global Memory)、多个片间通讯 DMA(CDMA) 等组成。

用户在使用 PPL 进行编程时面对的是 TPU Core，TPU 是一个标准的 SIMD 结构。Vector Engine 中共有 LANE_NUM 个 lane，每个 lane 中有 EU_NUM 个 EU。一个 EU 相当于一个独立的计算单元，

每个 lane 中都有一片 local memory，每个 lane 只能访问自己的 local memory。相同时刻 lane 都在执行同一条指令，EU 在做相同类型的计算。

针对不同芯片 TPU 架构的差异性如下表所示：

表 1: 不同 chip 的 TPU 架构

Chip	TPU CORE NUM	L2 MEM	LANE_NUM	EU_BYTES	LMEM per LANE
bm1684x	1	/	64	64	256KB
bm1688	2	/	32	16	128KB
bm1690	8	128MB	64	64	256KB
sg2380	4	/	32	16	128KB

(1)TPU 系统存储单元

在 TPU 架构中，Host Memory(hmem)、Global Memory(gmem)、Local Memory(lmem) 和 Static Memory(smem) 都是用于存储数据的不同类型的内存。它们之间有以下的区别和联系：

- hmem(主机内存) 是计算机系统中的普通内存，用于存储程序和数据。TPU 可以通过 DMA 技术从 hmem 中读取和写入数据，实现与主机的数据交换和通信。
- gmem(全局内存) 用于存储全局的共享数据。gmem 可以由所有的计算核心共享，并通过 CDMA 技术进行访问和通信。
- lmem(本地内存) 是每个计算 Lane 独有的一块内存，用于存储计算核心所需的局部数据。lmem 的访问速度比 gmem 更快，可以在计算核心内部实现高效的数据访问和交换。
- smem(静态内存) 通常用于存储模型的静态数据，包括模型的权重、偏置值、查找表等不会改变的数据。在计算过程中，模型的静态数据将从 hmem 中加载到 smem 中，并在后续的计算过程中使用。

在 TPU 架构中，hmem 通常用于存储输入数据、训练数据、模型参数等，gmem 用于存储全局共享数据，例如共享的缓存区和寄存器等，而 lmem 则用于存储计算核心所需的局部数据和中间结果。为了实现高效的数据交换和通信，TPU 架构通过 GDMA(Global DMA) 实现 gmem 和 lmem 之间的通信，通过 CDMA(Crossbar DMA，跨点 DMA) 实现 gmem 和 hmem 之间的通信。

(2)Lane 计算单元

TPU Core 内的 Lane 采用 SIMD 的模式，由 TIU 来统一控制。每个 Lane 的 lmem 在地址上是连续的。Lane 中的计算单元分 Vector 和 Cube 两种。Vector 负责处理向量运算的部分，可以执行加、乘和其他一些基本的向量操作，主要用于处理非矩阵计算的部分。Cube 负责处理矩阵乘法的部分，被设计为高效地执行大规模的矩阵乘法操作，主要进行 conv 和 mm2 的相关计算。

每个 Lane 有 EU_BYTES 个 EU，每种数据类型 (dtype) 下的 EU_NUM = EU_BYTES / sizeof(dtype) 。

每个 Lane 中的 Cube 计算单元由很多个乘加器 (MAC) 组成，且支持 INT8、FP8、BF16、FP16 的数据类型，每种数据类型对应的 MAC 数量如下表所示。其中每次 Cube 单元运行时，只允许其中一种数据类型的计算。

表 2: 各种数据类型下 Cube 的 MAC 数量

数据类型	MAC_NUM
INT8	256
FP8	256
BF16	128
FP16	128

在 TIU 模块的控制下，所有 Lane 同时执行一条指令。每个 Vector 和 Cube 只能访问自己的 lmem，或者在运算时接受 TIU 模块收集的其他 lmem 的数据。

(3)GDMA

GDMA 支持 lmem、smem、L2 mem、gmem 之间或内部做数据搬运，注意这里的 lmem 和 smem 指的是 GDMA 所在的本地 TPU Core 的 lmem 和 smem。GDMA 也支持源地址是本地 TPU Core 的 lmem/smem，目的地址是其他 TPU Core 的 lmem/smem 的数据搬运，但不支持源地址非本地 TPU Core 的 lmem 的数据搬运。

(4)HAU

HAU (Hardware Acceleration Unit) 是用于数据排序和 top-K 等功能的专用加速器。HAU 能访问 gmem 和 L2 mem 空间. 每个 HAU 都有一个独立的指令流，可相互独立的并行运行。

(5)SDMA

SDMA 支持 L2 mem、gmem 之间或内部的数据搬运。每个 SDMA 有 2 个独立的指令流 Buffer，这两个指令流共同竞争它们所对应的 SDMA 资源。

1.2 TPU 中数据的存储模式

1.2.1 内存与数据排列

在 TPU 中，我们使用 Tensor 来描述数据。

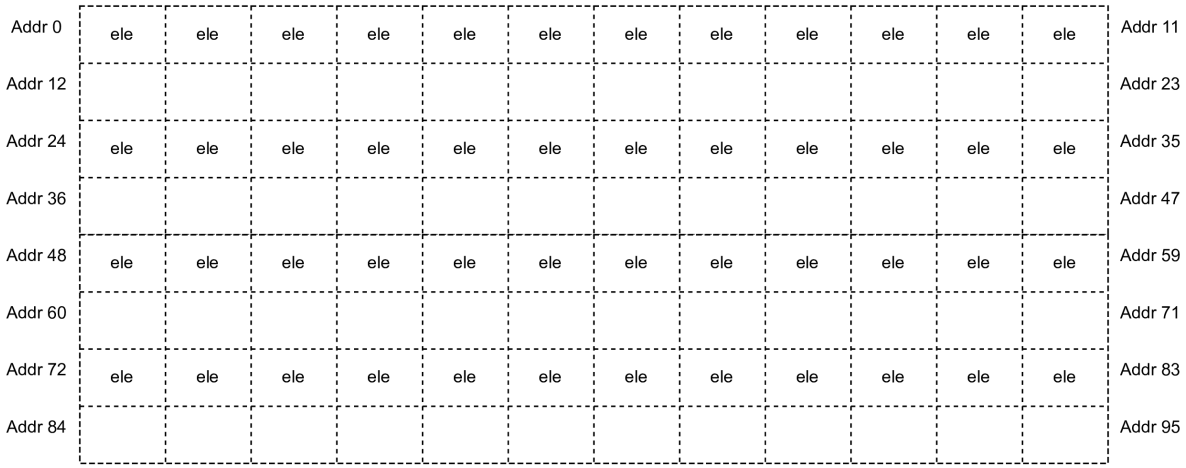
Tensor 是一个 4 维数组，使用 4 元组 (N, C, H, W) 来描述一个 Tensor 的几何尺寸 (**Shape**)。Tensor(n, c, h, w) 表示在 (n, c, h, w) 索引下的数据元素。

Stride 用于描述 Tensor 在实际内存当中是如何摆放，同样使用 4 元组 (N_stride，C_stride，H_stride，W_stride) 来描述，表示 **Tensor** 在内存当中存放时，元素间间隔了多少元素，具体而言：

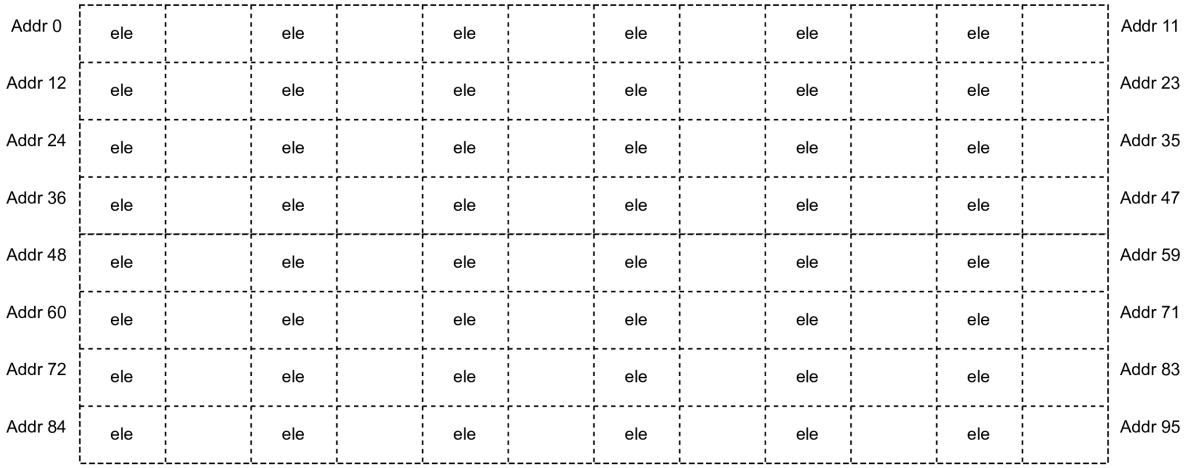
- W_stride 描述的是从 Tensor(n,c,h,w) 到 Tensor(n,c,h,w+1) 两个元素之间，在内存存储时，间隔了多少个元素。
- H_stride 描述的是从 Tensor(n,c,h,w) 到 Tensor(n,c,h+1,w) 两个元素之间，在内存存储时，间隔了多少个元素。
- C_stride 描述的是从 Tensor(n,c,h,w) 到 Tensor(n,c,h+1,w) 两个元素之间，在内存存储时，间隔了多少个元素。
- N_stride 描述的是从 Tensor(n,c,h,w) 到 Tensor(n+1,c,h,w) 两个元素之间，在内存存储时，间隔了多少个元素。

假设现在有一个 Tensor，它的每一个元素都占 1 个 byte，它的 Shape 是 (4，3，2，2)，如果它的存储方式 Stride 是 (12，4，2，1)，在内存当中，它的排列方式就会如下图所示：

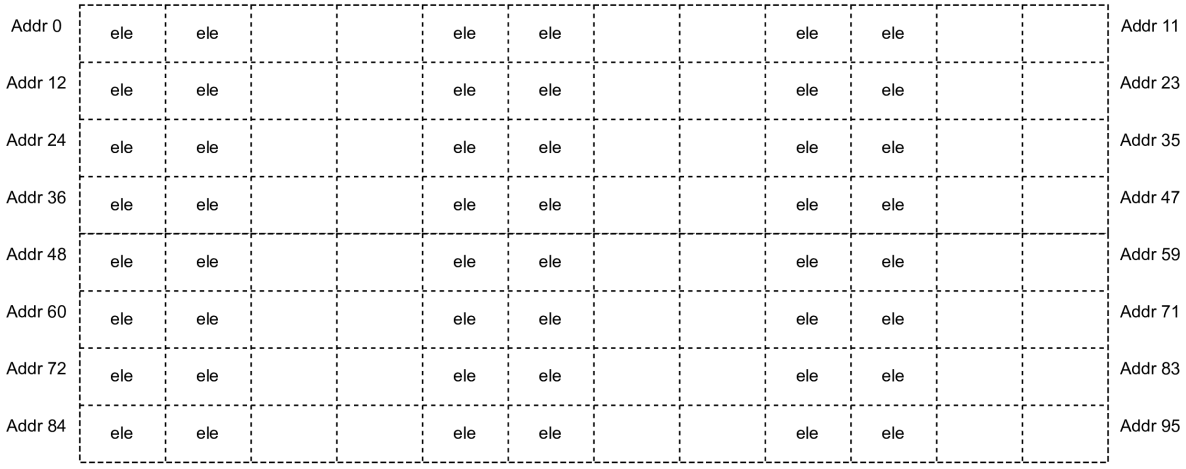
如果它的存储方式 Stride 是 (24，4，2，1)，在内存当中，它的排列方式就会如下所示：



如果它的存储方式 Stride 是 (24，8，4，2)，在内存当中，它的排列方式就会如下所示：



如果它的存储方式 Stride 是 (24，8，4，1)，在内存当中，它的排列方式就会如下所示：



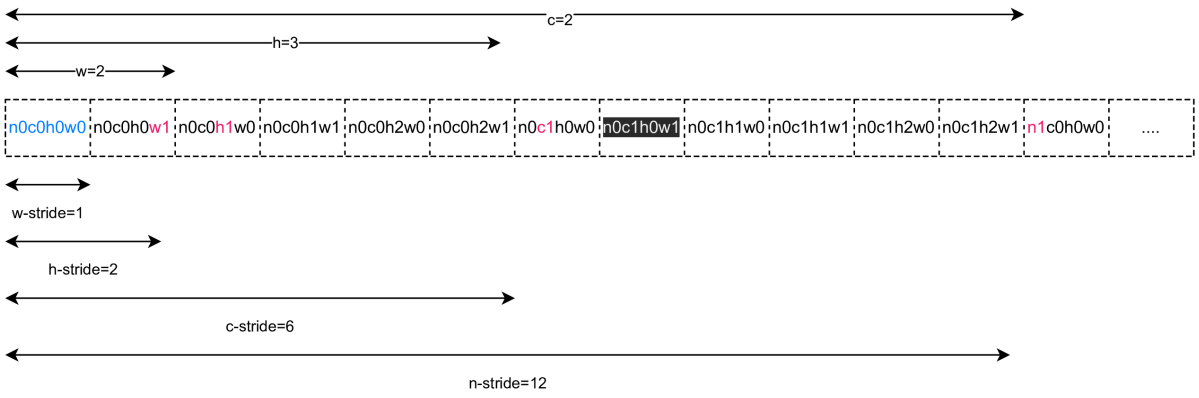
1.2.2 Tensor 在 gloabl memory / L2 memory 的排列方式

gmem 由一块 DDR 内存组成。一个 Shape 为 (N，C，H，W) 的 Tensor，在 **gmem** 排列，对应的 Stride 为:

- W_Stride = 1
- H_Stride = W
- C_Stride = H*W
- N_Stride = C*H*W

这种排列方式被称为 **连续存储方式**。

例：一个 Shape(N=2，C=2，H=3，W=2) 的 Tensor 在 gmem 排列方式



上图中，n0c0h0w0 代表 Tensor(0, 0, 0, 0) 位置的元素。

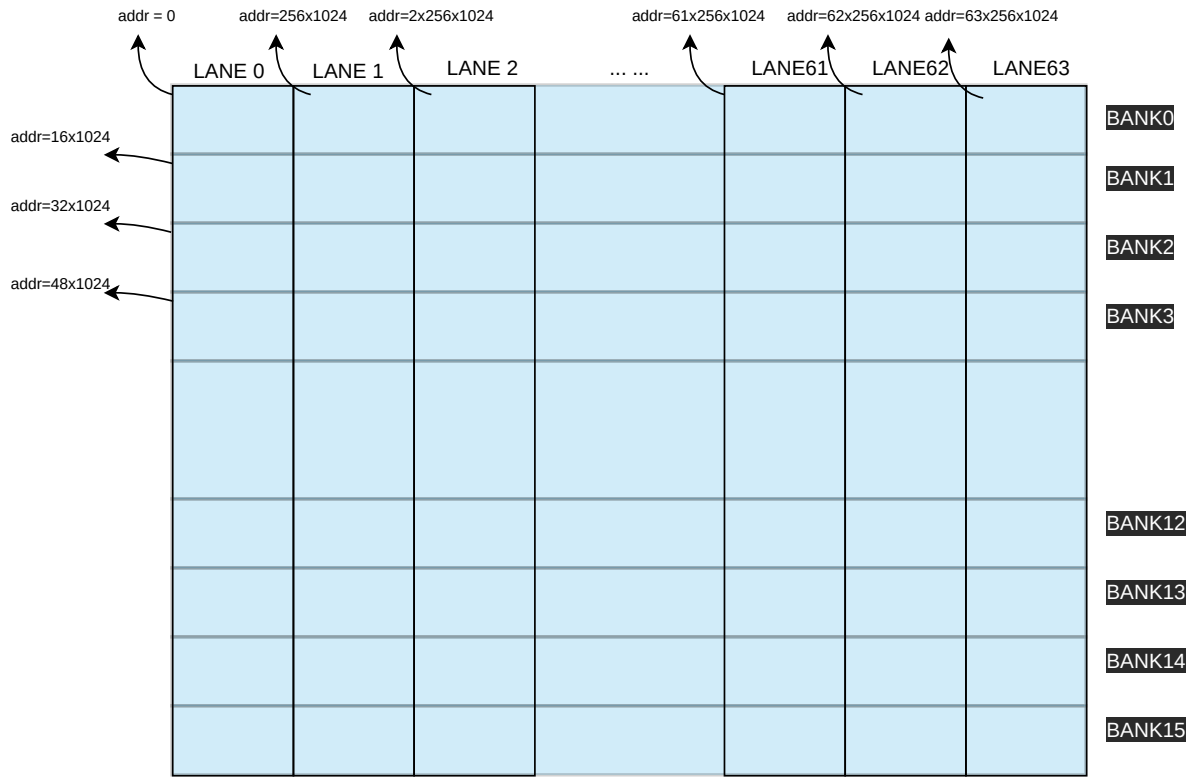
1.2.3 Tensor 在 lmem 的排列方式

lmem 的物理组成和地址分配

lmem 共由多片 SRAM(静态随机存取存储器) 构成，每一片 SRAM 都被称为一个 **Bank**。

每个 TPU core 的内部存储一共由 16 个 Bank 构成。16 个 Bank 组成整个 **lmem**。

整个 lmem 同时被划分为了 N 个 **lane** ，地址分配如下图所示:



lmem 大小为 **256KB * LANE_NUM **，每个 Lane 的 lmem 容量为 256KB，地址分配按照 Lane 进行分配，其中，Lane0 对应 **0 ~ 256*1024-1** 的地址，Lane1 对应 **256*1024 ~ 2*256*1024-1** 的地址，依次类推。

Tensor 在 lmem 上排列的基本规则

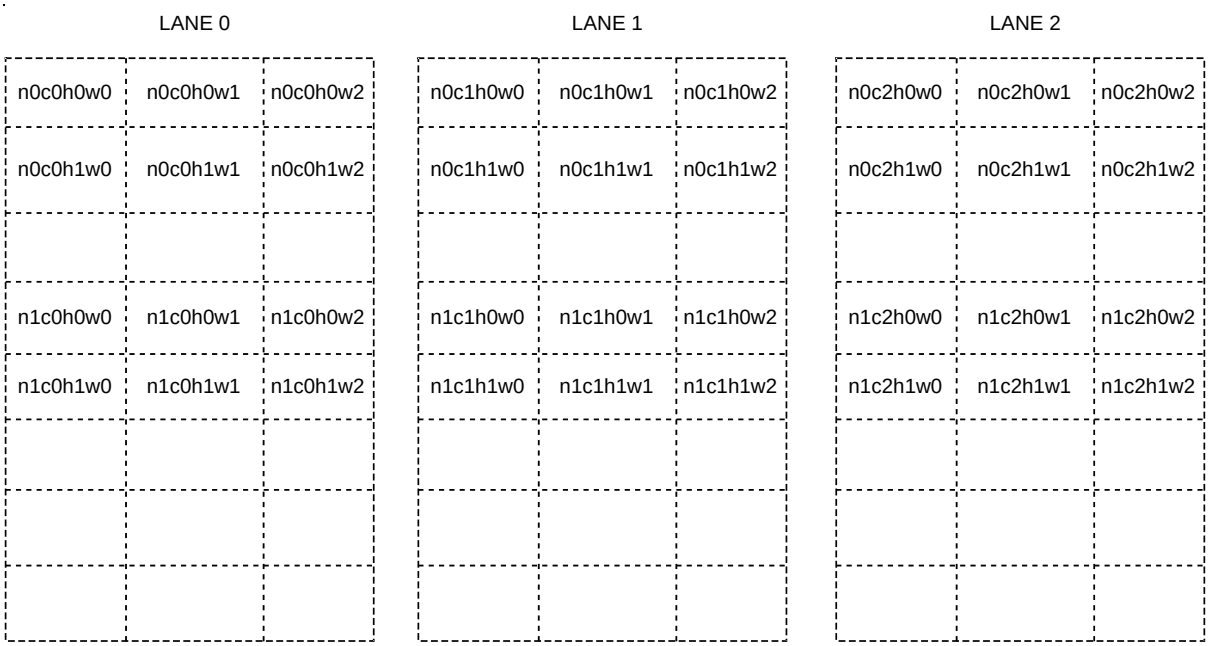
Tensor 在 lmem 上的排布方式与 gmem 的排布方式不同，主要区别在于 **C 维度的数据排布方式**。

一个 Shape 为 (N，C，H，W) 的 Tensor，Tensor(N，c，H，W) 代表：当 C = c 时，Tensor 的数据切片。

对于不同的 c，Tensor(N，c，H，W) 分配在不同的 Lane 上。

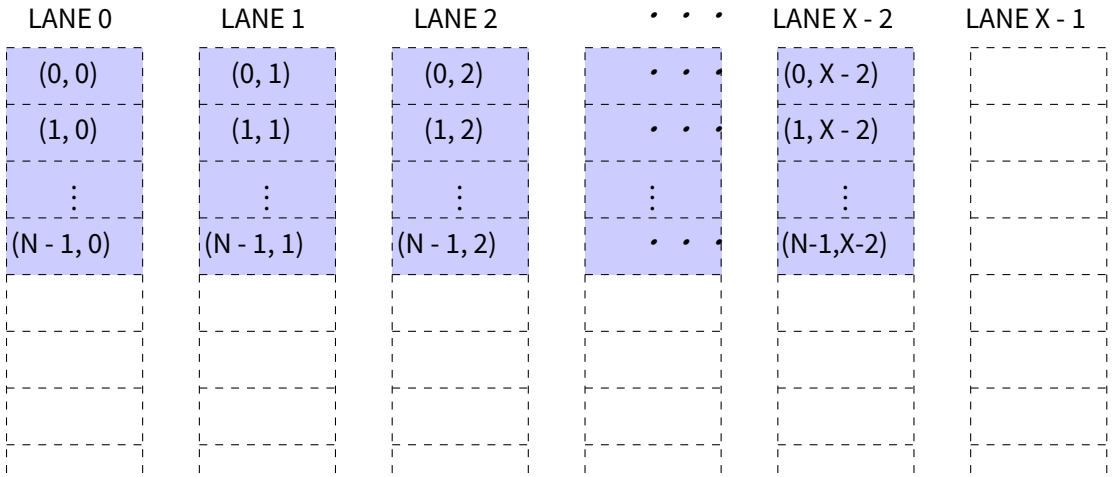
例:Tensor 的 Shape(N=2，C=3，H=2，W=3), Stride(N_stride = 9，C_stride = 9，H_stride = 3，W_stride = 1)

那么 Tensor 在 lmem 上的数据排列方式如下所示。



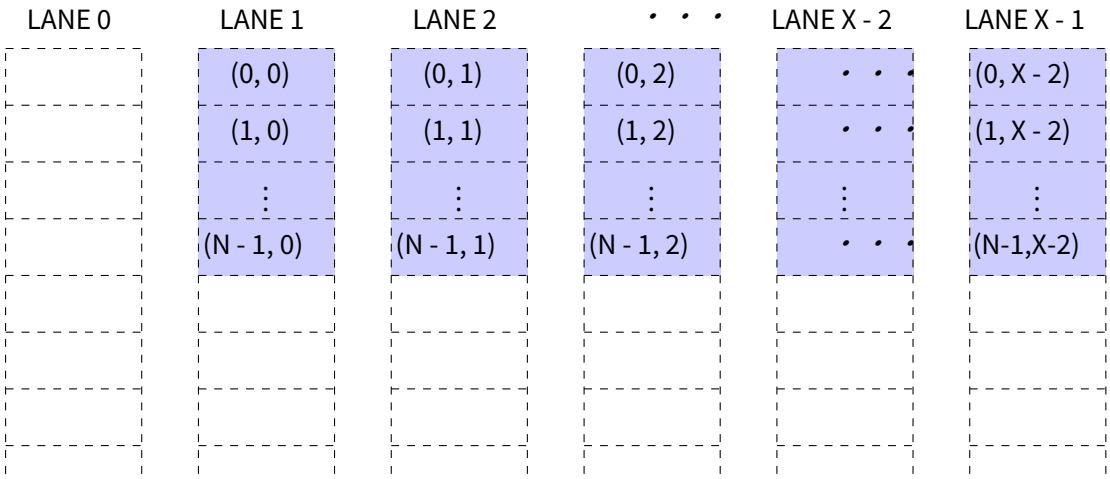
不同大小的 C 影响着实际存储方式，假设现在一共由 X 个 Lane，考虑下面几种存储情形：

情形 1: 当 Tensor 的 Shape 的维度 C = X-1 时，当 Tensor 从 Lane0 开始存储时，那么 Tensor 在 lmem 上的排布方式如下所示。



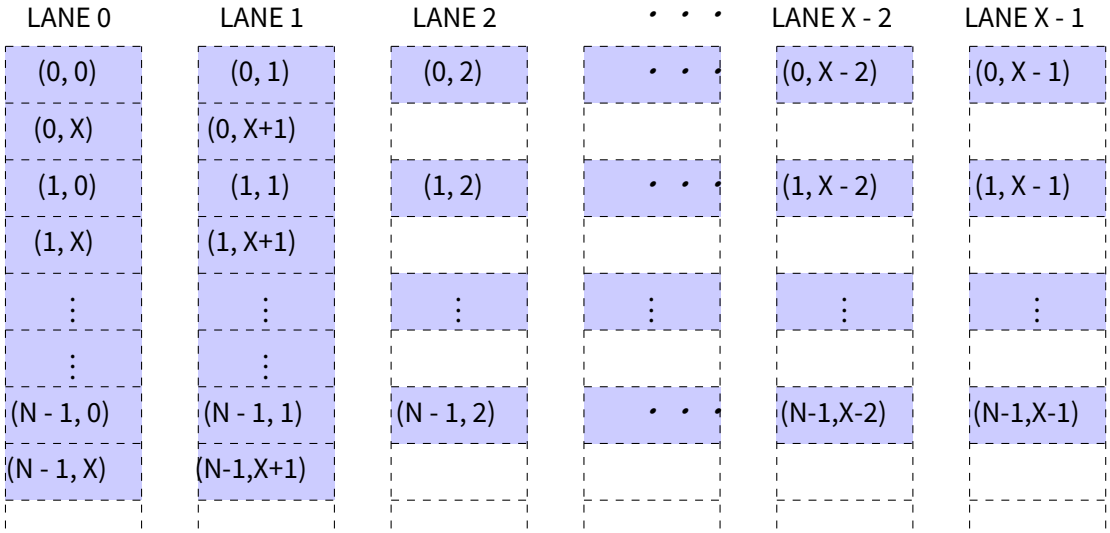
在 Lane X-1 的地址空间，没有数据分布。

情形 2: 当 Tensor 的 Shape 的维度 C = X-1 时，当 Tensor 从 Lane1 开始存储时，那么 Tensor 在 lmem 上的排布方式如下所示。



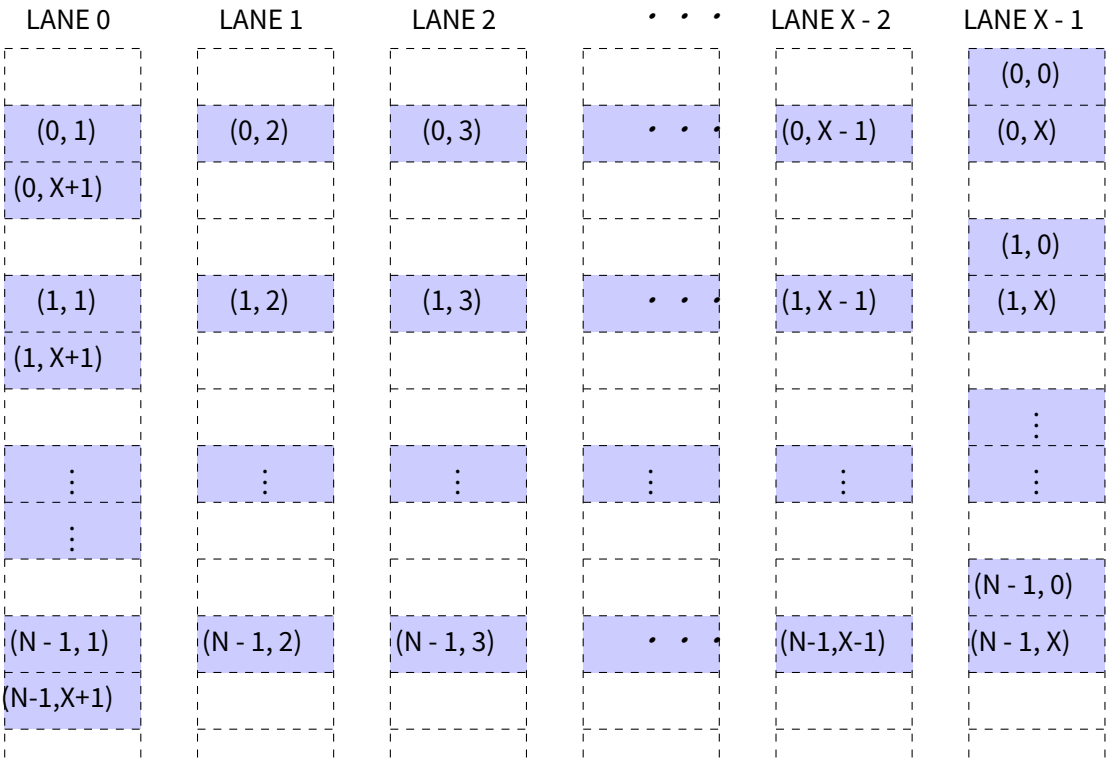
在 Lane0 的地址空间，没有数据分布。

情形 3: 当 Tensor 的 Shape 的维度 C = X + 2，当 Tensor 从 Lane0 开始存储时，那么 Tensor 在 lmem 上的排布方式如下所示。



C=X+1 维度的数据被排布到 Lane0，而 C=X+2 维度的数据被排布到 Lane1。并且注意到，下一个 N 维度仍然从 Lane0 开始排布。

情形 4: 当 Tensor 的 Shape 的维度 C= X + 2，而 Tensor 从 Lane X-1 开始存储时，那么 Tensor 在 lmem 上的排布方式如下所示。



可以看到，C=0 维度的数据被排布到了 Lane X-1, C=1 维度的数据被排布到了 Lane0 上，依次类推，C= X+2 维度的数据被排布到了 Lane0 上。

lmem 上几种常用数据排布方式

上面介绍了 Tensor 在 lmem 上存储的基本原则，现在介绍指令集常用的几种数据排布方式：

1. N-Bytes 对齐存储方式

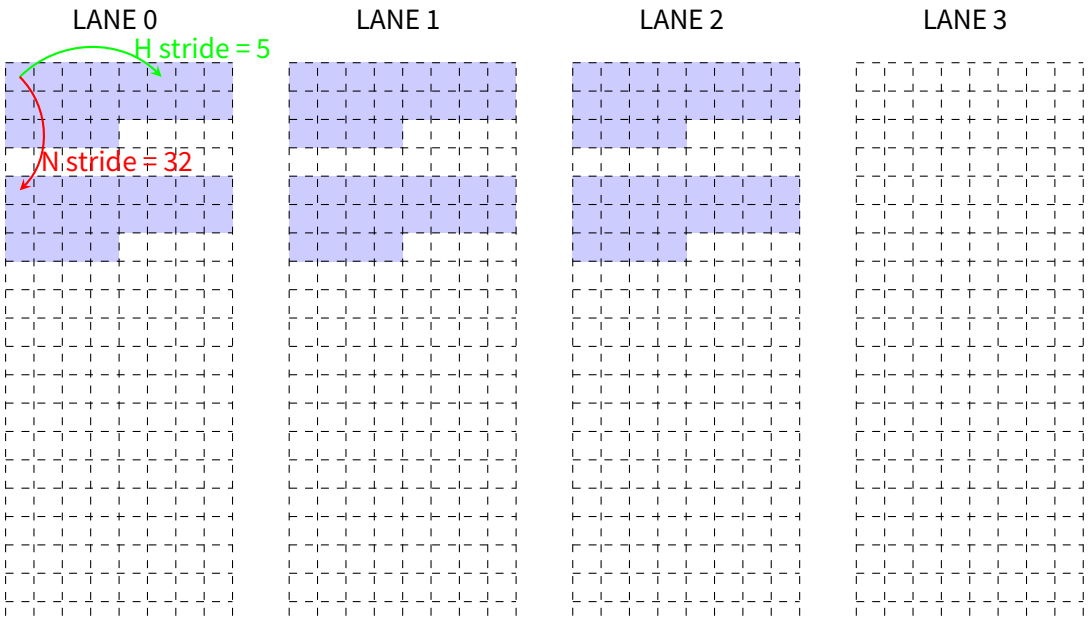
“64-Bytes 对齐存储方式”是 bm1684x 和 bm1690 最常用的 Tensor 存储方式，而“16-Bytes 对齐存储方式”是 1688 最常用的 Tensor 存储方式，它是指 Tensor 排放存储要满足以下几个约束：

- Tensor 的起始地址是 64 的整数倍
- W_stride = 1
- H_stride = W
- C_stride = ceil(H*W, 16) * 16 (数据元素是 **32-bits**)；ceil(H*W, 32) * 32 (数据元素是 **16-bits**)；ceil(H*W, 64) * 64 (数据元素是 **8-bits**)
- N_stride = C_stride * (单个 Lane 上 channel 的个数)

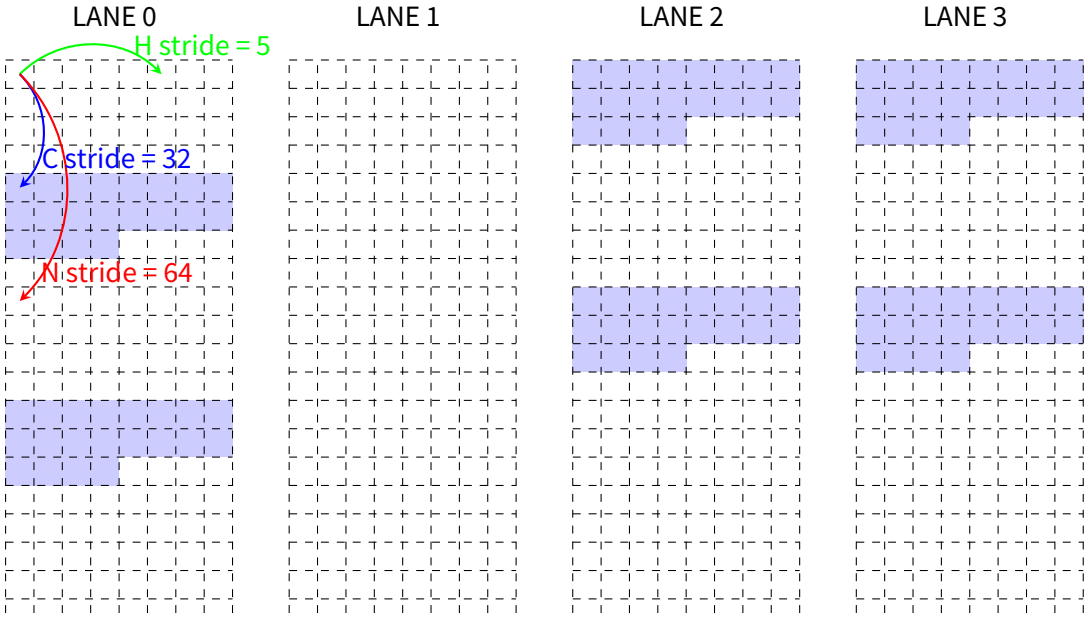
上面，ceil 是向上取整的意思。

为举例简便，假设 Lane 个数 =4

例 1: Tensor 的 Shape(N=2, C=3, H=4, W=5), 数据类型为 float32，Lane0 开始存储：



例 2: Tensor 的 Shape(N=2, C=3, H=4, W=5), 数据类型为 float32，Lane2 开始存储：



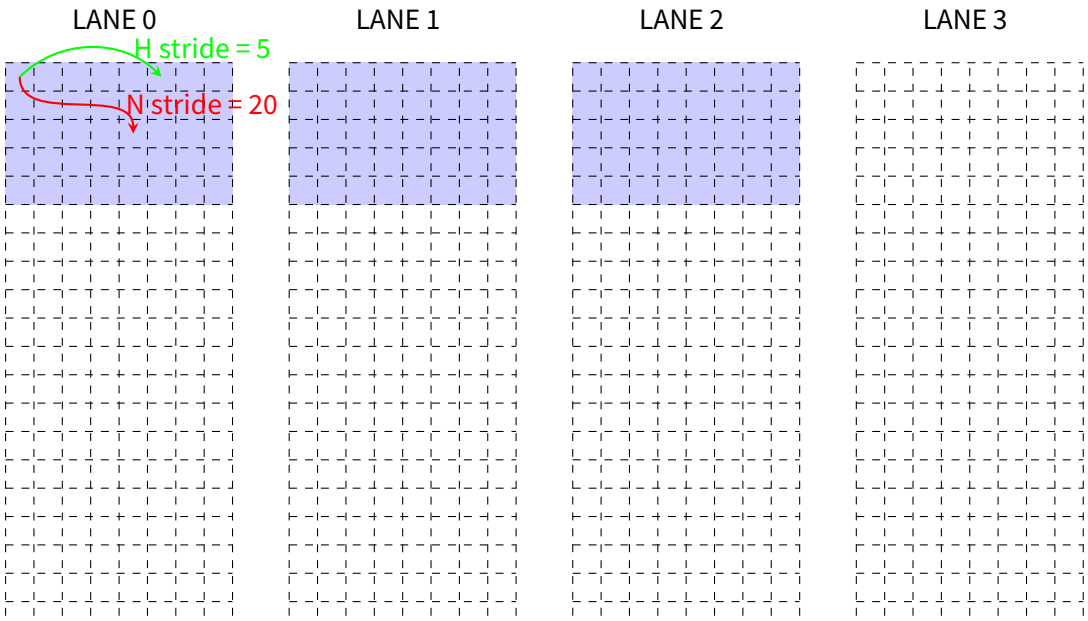
2. Compact 紧凑存储方式

“紧凑存储方式”也是较为常用的 Tensor 存储方式。假设 Tensor 的 Shape 为 (N，C，H，W)，按照“紧凑存储方式存储”要满足以下约束：

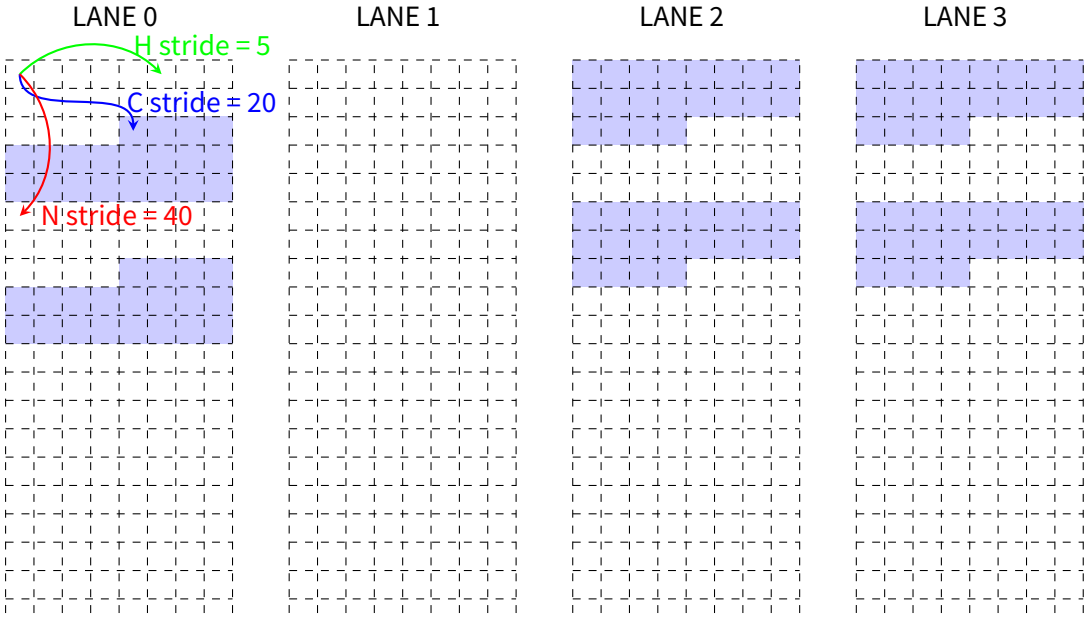
- Tensor 的起始地址是 4 的整数倍。
- W_stride = 1
- H_stride = W
- C_stride = H * W
- N_stride = C_stride * (单个 Lane 上 channel 的个数)

为举例简便，假设 Lane 个数 =4

例 1: Tensor 的 Shape(N=2, C=3, H=4, W=5), 数据类型为 float32, Lane0 开始存储：

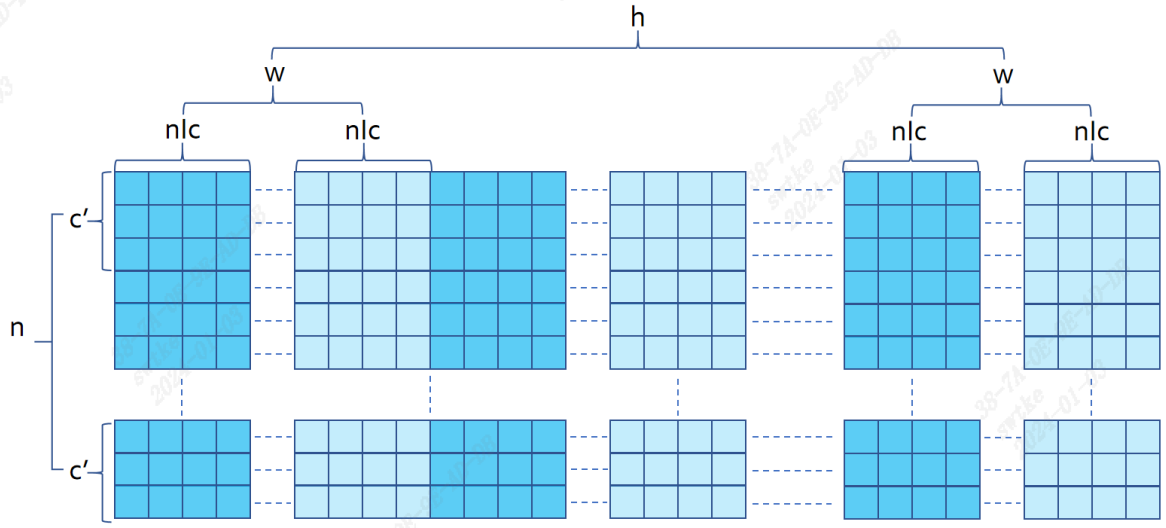


例 2: Tensor 的 Shape(N=2, C=3, H=4, W=5), 数据类型为 float32，Lane2 开始存储：



3. N-IC 存储方式

该存储方式是要将张量的 shape 从 [N, C, H, W] 转换为 [N, Ceil(C/nIC)， H, W, nIC], 并按此 shape 进行存储。在 global momery 上的存储如下图所示：



NIC 存储仅用于卷积计算过程。卷积核需要从 [oc, ic, kh, kw] 转置为 [1, oc, ceil(ic/NIC) * kh * kw, NIC] 存放在 local memory 中。

IC 数：

- BM1684x / BM1690：INT8 kernel 以 64IC 格式存放，FP16/BF16 kernel 以 32IC 格式存放
- BM1688 ： INT8 kernel 以 32IC 格式存放，FP16/BF16 kernel 以 16IC 格式存放
- SG2380 ： INT8 kernel 以 32IC 格式存放，FP16/BF16 kernel 以 32IC 格式存放

假设卷积核的 shape 是 [oc, ic, kh, kw]，分别表示 output channel、input channel、卷积核的高度以及卷积核的宽度。

64IC 存储在 local memory 上满足：

- W_stride = 1
- H_stride = 64
- C_stride = 64 * kw * kh * ceil(ic/64)
- N_stride = 64 * kw * kh * ceil(ic/64) * ceil(oc/ LANE_NUM)

其中，每 64 个 input channel 作为一组进行存储，每组之间的 stride 为 64 * kw * kh。在 local memory 可以等效表示为 Compact 存储，shape 为 [1, oc, ceil(ic / 64) * kw * kh, 64]

32IC 存储在 local memory 上满足：

- W_stride = 1
- H_stride = 32
- C_stride = 32 * kw * kh * ceil(ic/32)
- N_stride = 32 * kw * kh * ceil(ic/32) * ceil(oc/ LANE_NUM)

其中，每 32 个 input channel 作为一组进行存储，每组之间的 stride 为 32 * kw * kh。在 local memory 可以等效表示为 Compact 存储，shape 为 [1, oc, ceil(ic / 32) * kw * kh, 32]

fp32 类型的 kernel 不需要做特殊处理，以 [1, oc, ic, kh * kw] 存储在 local memory 上

CHAPTER 2

PPL 环境配置及运行测试

2.1 开发环境配置

本章介绍开发环境配置, 代码在 docker 中编译和运行。

2.1.1 PPL 环境初始化

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
|    |    ├─── include/    # device 代码依赖的头文件
|    |    ├─── qemu/       # device 代码依赖的头文件和库
```

(续下页)

(接上页)

```
|   |   ├── 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
```

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
```

2.1.2 代码测试

使用 ppl_compile.py –src XXX –chip XXX –gen_ref –out XXX 命令来运行.pl 文件。

命令组成

- **ppl_compile.py**：这是主命令，调用 ppl_compile.py 脚本。该脚本用于自动化运行流程，包括编译、运行测试案例等。
- **–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。
- **–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
- **–chip**：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。
- **–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)，并对比两个函数的结果，结果以 npz 格式存储在生成的 test_xxx/data 目录下。

可选参数

- **–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’。
- **–build_debug**：生成 debug 版本，默认为 False。
- **–gdb**：使用 gdb 运行程序，与 build_debug 配合使用，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。

例如，当在根目录下使用运行位于 ppl/examples/cxx/matmul/ 的 mm2_fp16.pl 文件时，使用以下命令：

```
ppl_compile.py --src ./examples/cxx/matmul/mm2_fp16.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_main 的文件夹，若未使用–out 指定文件夹名称，则自动使用 __TEST__ 修饰的测试函数的名称。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。

2.1.3 编译过程介绍

PPL 提供的 ppl_compile.py 脚本主要通过调用 ppl-compile 二进制文件进行工作。

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。

以下通过一个示例介绍 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_main/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。

2.1.4 sg2380 环境配置

PPL 支持 sg2380 ，但 sg2380 代码是基于 RISC-V 架构的，测试和运行 sg2380 代码需要 Qemu 来对 RISC-V 架构进行模拟和虚拟化支持。用户在 PPL 中配置 sg2380 runtime，需要将一些第三方库下载到 third_party 目录，目录结构如下：

```
├─── 2380/
|    ├─── llvm-project/
|    ├─── riscv64-unknown-elf-toolsuite-17.9.0-2023.10.0/
|    ├─── sifive_x280mc8/
|    ├─── qemu-sifive/
|    ├─── TPU1686/
```

- llvm-project 包含了一个特殊版本的 Clang-18 编译器，负责将 sg2380 代码编译生成 ELF 格式的可执行文件
- riscv64-unknown-elf-toolsuite-17.9.0-2023.10.0 是 Clang-18 的工具链
- sifive_x280mc8 包含了 Clang-18 编译 sg2380 代码时需要链接的 metal 和 metal-gloss 等库文件
- qemu-sifive 包含了 Qemu 的源码
- TPU1686 提供了 Qemu 运行由 sg2380 代码生成的 elf 文件时所需要的 libc-model_backend 库

以上第三方库均可从 gerrit 上获取，riscv64-unknown-elf-toolsuite-17.9.0-2023.10.0 源码在 iree-rv64-baremetal 项目的 third_party 目录下，其他文件可直接在 gerrit 上搜索同名的项目进行 clone。

PPL 提供了构建 2380 runtime 的脚本，在根目录下 source 2380build_helper.sh 后，运行 build_2380_runtime 即可。

```
cd ppl
# 当前已处于ppl目录下
source envsetup.sh
```

(续下页)

(接上页)

```
source 2380build_helper.sh

build_2380_runtime
```

使用 ppl_compile.py –src XXX –chip sg2380 –gen_ref 命令来运行.pl 文件，命令参数与 bm1684x 、bm1688 和 bm1690 一致。此外，PPL 提供了 build_ccode_for_system 和 run_qemu_system 两个命令，前者可以将 PPL 由 pl 源码生成的 sg2380 代码编译为 elf 可执行文件，后者将可执行文件在 Qemu 上运行。

2.1.5 添加 PPL 支持

AddPPL.cmake 是为了方便用户将编写的 pl 文件转换为能在 TPU 上运行的 C 代码的工具。它通过封装 ppl-compile 编译命令，简化了从 PL 文件到可执行代码的转换过程，用户可以将编译得到的 C 代码编译成 so，并利用 runtime 的动态加载接口进行调用。

引入 AddPPL.cmake

首先，确保 AddPPL.cmake 文件位于用户项目中某个可访问的目录下。AddPPL.cmake 默认放置在 ppl/runtime/scripts/目录下，因此最直接地引入 AddPPL.cmake 的命令为:

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

设置 pl 文件编译规则

AddPPL.cmake 中提供了名为 ppl_gen 的函数用于对 pl 文件进行编译，用户可以通过向 ppl_gen 函数传递参数来指定 pl 文件所在路径和设置输出文件路径。ppl_gen 的参数必须按照指定的顺序输入；前三个传入参数为必选参数，之后的参数均为可选参数。

必选参数

- 输入文件位置：传入的第一个参数是一个文件路径，用于指定待编译的.pl 文件所在的位置。
- 芯片类型：传入的第二个参数为芯片类型，可选的类型有 bm1684x、bm1688、bm1690。
- 输出文件位置：传入的第三个参数是一个路径，用于指定编译生成的文件存放的位置。

可选参数

- 头文件搜索路径：支持用户自定义头文件搜索路径。默认情况下 ppl_gen 只会将 ppl/inc 设置为头文件搜索路径，如果用户希望加入其他路径 inc_path，可直接将路径传入 ppl_gen，只要在必选参数之后传入。
- 宏：支持用户在编译时定义宏，使用方法与 cmake 语法一致。例如，传入字符串’-DCORENUM=3’。

AddPPL.cmake 还提供了两个函数用于设置编译环境

- add_ppl_include：设置 pl 代码需要的头文件路径
- add_ppl_def：设置 pl 代码依赖的宏定义

以下用 ppl/samples/add_pipeline 中的 add_pipeline.pl 文件来展示通过 AddPPL.cmake 来将其进行编译的过程:

```
set(SCRIPTS_CMAKE_DIR "${PPL_TOP}/runtime/scripts/")
list(APPEND CMAKE_MODULE_PATH "${SCRIPTS_CMAKE_DIR}")
include(AddPPL)
set(input ${PPL_TOP}/samples/add_pipeline/ppl/add_pipeline.pl)
set(chip bm1684x)
set(output ${CMAKE_CURRENT_SOURCE_DIR})
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")
# add_ppl_include(./include)
# add_ppl_include(../include)
# add_ppl_def("-DTEST")
ppl_gen(${input} ${chip} ${output})
```

构建动态库

编译 pl 文件完成后，会在指定的目录生成一些后端代码，可以进一步将这些代码配置和构建成可执行文件和动态库，以供用户在自己的工程中使用。

上一步编译 pl 文件完成后，可以对生成的 c 文件进一步处理。下面的示例代码展示了将编译得到的 host 和 device 目录内的文件构建为可执行文件和动态库的过程：

```
# 第一部分会将host目录下的源文件构建为可执行文件，然后链接需要的库。
# 如果用户直接使用Kernel函数，可以跳过这一步。
aux_source_directory(host APP_SRC_FILES)
foreach(app_src IN LISTS APP_SRC_FILES)
  get_filename_component(app ${app_src} NAME_WE)
  message(STATUS "add executable: " ${app})
  add_executable(${app} ${app_src})
  target_compile_definitions(${app} PRIVATE -DUSING_CMODEL)
  if(${CHIP} STREQUAL "bm1684x"
    OR ${CHIP} STREQUAL "bm1688")
    target_link_libraries(${app} PRIVATE bmlib pthread)
    add_dependencies(${app} gen_kernel_module_data_target)
  else()
    target_link_libraries(${app} PRIVATE tpuv7_rt cdm_daemon_emulator pthread)
  endif()
  set_target_properties(${app} PROPERTIES OUTPUT_NAME test_case)
  install(TARGETS ${app} DESTINATION ${CMAKE_CURRENT_SOURCE_DIR})
endforeach()

# 第二部分会将device目录下的源文件构建为动态库，然后安装到指定目录中。
aux_source_directory(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_compile_definitions(kernel PRIVATE -DUSING_CMODEL)

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)
```

CHAPTER 3

PPL 编程模型

3.1 PPL 编程模型介绍

PPL（Primitive Programming Language）旨在为 TPU 编程提供一个高效、易用的应用层接口，使得开发者能够通过编写类似 C 语言的代码来直接调用底层的 TPU tensor/vector/scalar primitive 接口完成计算逻辑。PPL 通过将高级抽象的程序代码转换为芯片可识别的指令集，不仅降低了 TPU 编程学习曲线，提升了开发效率，使得编程过程更加直观，还通过负责 pipeline/memory 管理/同步等底层机制极大地提高了程序执行的性能。

核心功能

- 高级编程接口：提供了一套 C/C++ 风格的编程接口，使得开发者可以在不深入了解硬件细节的情况下，通过简单的函数调用来利用底层硬件的强大功能。
- 内存管理：PPL 不仅仅是一个代码转换工具，它还负责 TPU 上的内存管理。PPL 根据 tensor 的生命周期尽量复用内存，降低资源消耗。
- 同步与并行计算优化：PPL 能够自动进行必要的同步控制和并行计算优化，确保程序的正确执行和性能优化。

通过 PPL，用户可以更加专注于算法和应用层面，而无需花费大量时间和精力在底层硬件接口和性能优化上。PPL 旨在提升 TPU 编程易用性，支持了 TPU 编程所需的全部数据类型，并为简化 TPU 编程提供了大量特有的函数和数据结构。PPL 的长远目标是为硬件底层开发提供一种更高效、更友好的编程范式，推动在各种芯片平台上的应用开发和创新实践。

本章将以软件的视角从 TPU 架构、PPL 编程流程、支持的数据类型、数据的存储类型等几个方面详细论述 PPL 的编程模型。

3.1.1 代码结构

由于 TPU 是一个异构的架构设计，由主机端发送指令，设备端接收指令，按照指令执行指定的操作，传统的 TPU-KERNEL 编程驱动 TPU 进行指定的计算需要分别完成主机端和设备端的两部分代码：

- 主机端 (Host): 主机端代码，运行在主机侧，发送控制 TPU 运行的命令。
- 设备端 (Device): 设备端代码，运行在设备侧，通常调用 TPU 的各种指令运行相应的运算。

在进行 PPL 编程时，编写的是设备端（Device）代码。以下是一个使用 TPU 进行两个相同 shape 的 tensor 逐元素除法的代码示例：

```
// ppl.h包含了PPL的核心功能和API
#include "ppl.h"
using namespace ppl;

// 定义常量，表示Tensor的各个维度
const int N = 1;
const int C = 64;
const int H = 32;
const int W = 16;

// fp32_div 函数是PPL编程的KERNEL代码主体，用于执行浮点数Tensor的除法运算。
// 它接受三个指针参数 ptr_dst、ptr_src0 和 ptr_src1，
// 分别表示存储在global memory上的结果张量、被除数张量和除数张量。
// numiter 是提供给后端TPU指令tpu_bdc_fp32_tunable_div_C的参数，表示Newton迭代次数。
__KERNEL__ void fp32_div(float *ptr_dst, float *ptr_src0, float *ptr_src1, int numiter)
{
  // 创建了1个dim4类型对象shape
  // 本例中global memory和local memory上的张量dst、src0和src1的shape相同，
  // 因此创建张量时使用同一个shape，请仔细参考本文档中对各个指令的详细介绍。
  dim4 shape = {N, C, H, W};
  // 创建global memory上的张量
  auto src0_g = gtensor<float>(shape, GLOBAL, ptr_src0);
  // 创建local memory上的张量
  auto src0 = tensor<float>(shape);
  // dma::load接收2个参数，分别是local memory上的tensor和global memory上的gtensor。
  dma::load(src0, src0_g);
  auto src1_g = gtensor<float>(shape, GLOBAL, ptr_src1);
  auto src1 = tensor<float>(shape);
  dma::load(src1, src1_g);
  auto dst = tensor<float>(shape);

  // 在数据载入完成后，就可以调用tiu指令进行具体的计算。
  // 这里执行除法操作的结果保存在local memory上的Tensor对象dst中，
  // 需要将结果传回global memory使HOST代码进行后续的流程。
  tiu::fdiv(dst, src0, src1, numiter);
  // 将数据从local memory载入到global memory需要调用dma::store。
  auto dst_g = gtensor<float>(shape, GLOBAL, ptr_dst);
  // dma::store接收2个参数，分别是global memory上的gtensor和local memory上的tensor。
  dma::store(dst_g, dst);
}
```

给 HOST 端调用的 KERNEL 代码需要通过 __KERNEL__ 修饰符来进行标识。

KERNEL 代码的主要职能是驱动 TPU 执行具体的计算任务。驱动 TPU 进行计算加速通常分为以下几步：

- 将数据从主机端内存搬运到 TPU 的系统内存（global memory）当中，
- 再将数据从系统内存（global memory）再搬运到 local memory 当中，
- 驱动计算单元对 local memory 当中的数据进行计算，并将计算结果储存在 local memory，
- 将计算结果从 local memory 搬运回 global memory，

- 将 global memory 中的计算结果搬运回主机端内存。

PPL 提供 dma、tiu、hau 等多类指令，KERNEL 代码通过这些指令进行 TPU 的驱动。

在上文的示例中，KERNEL 代码的作用为：

- 定义和初始化一些变量，包括 tensor 的 shape。
- 使用 tensor 创建了 3 个 local memory 上 float 类型的 tensor 对象，包括 dst、src0 和 src1。
- 使用 gtensor 创建了 3 个 global memory 上 float 类型的 gtensor 对象，包括 dst_g、src0_g 和 src1_g。
- 使用 dma::load 将数据从 global memory 加载到 tensor 对象。
- 使用 tiu::fdiv 驱动 TPU 对数据加载后的 src0 和 src1 进行计算，将结果保存至 dst。
- 使用 dma::store 将 dst 中的数据传回 global memory (即 dst_g 绑定的 ptr_dst)。

KERNEL 代码的意义除了专注于具体的 TPU 计算，还在于 KENERL 部分可以由 TPU Core 多核并行执行，可以完成高度并行的计算任务。

KERNEL 代码与 HOST 代码协同工作，HOST 可以将 KERNEL 任务提交到加速器后，继续执行其他任务或者准备下一个 KERNEL 任务的数据，这样可以实现主机端和设备端的异步执行，提高资源利用率。

为了方便调试，PPL 中也可以编写生成随机输入的测试代码

```
// test 函数是测试代码的主体。在函数内部，首先定义了global memory上的张量形状shape，
// 然后分别为结果张量dst、被除数张量src0和除数张量src1分配内存，
// 并使用rand函数随机生成数据。最后调用KERNEL代码执行除法运算。
__TEST__ void test() {
  // 创建了1个dim4类型的shape，表示张量dst、src0和src1的形状。
  // global memory上张量的地址分配通过malloc函数进行，
  // 使用malloc函数需要提供所创建张量的类型和形状，
  // 类型需要通过类型模板参数显示地传入，形状需要传入一个dim4类型的地址。
  dim4 shape = {N, C, H, W};
  auto dst = malloc<float>(&shape);
  auto src0 = malloc<float>(&shape);
  // rand函数用于给global memory上张量生成随机数据，
  // 需要传入4个参数，分别是张量地址、dim4类型对象的地址以及随机数的上下界。
  rand(src0, &shape, -1.f, 1.f);
  auto src1 = malloc<float>(&shape);
  rand(src1, &shape, -1.f, 1.f);

  fp32_div(dst, src0, src1, 2);
}
```

测试代码需要通过 __TEST__ 修饰符来进行标识。

测试代码的主要职能是准备数据和管理 TPU 的执行。这通常包括以下步骤：

- 内存分配：在 global memory 的内存中为输入和输出数据分配空间。
- 数据初始化：生成随机输入数据或者读取本地数据。
- KERNEL 调度：将输入数据发送到 KERNEL 代码，进行 TPU 上的计算。

在上文的示例中，TEST 代码完成以下工作：

- 定义一个 shape 变量，它是一个 dim4 类型的结构，代表了一个 4 维张量的形状，具体为 {1, 64, 32, 16}。
- 使用 malloc 申请了 3 个 float 类型的张量的地址空间，dst、src0、src1，大小由 shape 指定。
- 使用 rand 函数初始化 src0 和 src1，填充范围在 -1.0 到 1.0 的随机数。
- 调用 fp32_div 函数，该函数负责在 TPU 上执行 src0 和 src1 进行逐元素相除，并将结果存储在 dst 张量中。

3.2 Tensor 介绍

3.2.1 dim

PPL 中提供 dim 结构体用于表示数据维度。PPL 中的 dim 结构体有 dim2 和 dim4 两种。以 dim4 为例，以下是 dim4 定义中的构造函数部分：

```
struct dim4 {
  ...
  int n, c, h, w;
  dim4(int _n, int _c, int _h, int _w);
  ...
};
```

这里 dim4 封装了四个整型值，分别对应不同的维度：批量大小（n）、通道数（c）、高度（h）和宽度（w）。提供了构造函数允许在创建实例时初始化这些维度值，接收四个整型参数 _n、_c、_h 和 _w，分别用于设置批量大小、通道数、高度和宽度。

其他 dim 结构体与 dim4 类似，不同之处在于 dim2 只封装了两个整型值（h，w）。

注意事项

- 以 dim4 为例，创建 dim 结构体的一般形式为：dim4 shape = {N, C, H, W};
- dim 均支持转换构造和赋值运算符重载，可用 dim 对象来对新的同类型的 dim 对象进行初始化。仍以 dim4 为例，在已创建了一个 dim4 类型的 shape 后，可通过 dim4 shape2 = shape; 或者 dim4 shape2; shape2 = shape; 来创建新的 dim4 对象 shape2。

使用示例

```
const int N = 1;
const int C = 64;
const int H = 32;
const int W = 16;
dim4 src_shape = {N, C, H, W}; //通过常量初始化dim4对象
dim4 dst_shape = src_shape;    //通过赋值运算符重载初始化dim4对象
dim4 bias_shape(src_shape);    //通过拷贝构造初始化dim4对象
```

3.2.2 Tensor

tensor 是对数据在 TPU local memory 上存储方式的一种抽象，包含了数据在 TPU 上存储的地址、shape、stride 等信息。用户可以指定 tensor 的地址，也可以不指定，由 PPL 自动分配地址。如果选择 PPL 自动分配地址，**则创建 tensor 时必须传入编译期为常量的 shape 信息**（对 tensor 进行 view 或 sub_view 操作时，shape 不必是常数，但是要自己保证操作后对 tensor 进行的运算不会超过原始 tensor 的范围），ppl 会在编译期根据 tensor 的 shape 和生命周期来分配 local 上的 memory。推荐使用自动分配地址，并且尽量不要重复使用同一个 tensor，尽量使用新的 tensor 来存放运算结果，这样可以更好的复用内存，并减少 bank conflict。

PPL 中 tensor 结构体的构造函数部分为：

```
template <typename dtype> struct tensor {
  public:
    ...
    template <typename dimT>
    explicit tensor(dimT &_shape, align_mode_t align_mode = TPU_ALIGN,
                    long long address = -1);
    ...
};
```

参数

- _shape: tensor 的几何尺寸，为一个 dim4 类型的结构体。

- align_mode: 描述 tensor 在 Local Memory 上的存储方式 (TPU_ALIGN 表示对齐存储、TPU_COMPACT 表示紧凑存储、TPU_ROW_ALIGN 表示行对齐存储)。
- address: 表示 tensor 在 Memory 上的起始地址，-1 表示由 PPL 自动分配。

注意事项

- 用户创建 tensor 需要提供 tensor 的数据类型 dtype，并将 dtype 通过模板参数传入。
- 创建 tensor 必须提供的参数只有 _shape，并且如果需要 PPL 自动进行内存分配，_shape 的值在编译期必须是常量。tensor 的 stride 可以不提供，PPL 会根据 tensor 的 shape 以及存储方式自动计算，但是用户需要了解数据在 TPU 上的存储方式与 stride 的关系，以避免运算结果不符合预期的问题。
- 创建 tensor 时需要考虑 tensor 在 local memory 上的存储方式，PPL 默认的存储方式为 N-byte 对齐存储，如果需要修改存储方式为 compact 存储，只需将 align_mode 改为 TPU_COMPACT。如果该 tensor 的存储为其他形式，可通过 view 函数指定 stride。
- 创建 tensor 后，PPL 会根据 tensor 的使用场景自动进行地址分配，一般情况下不建议用户指定 address。
- 创建 tensor 的一般形式为：auto myTensor = tensor<dtype>(shape);。
- PPL 中的 tensor 不支持拷贝构造和赋值运算符重载，即不允许将一个 tensor 类型的对象赋值给另一个已存在的 tensor 对象。
- PPL 不会自动修改 tensor 的 shape，也就是一个 tensor 作为 tiu 运算的输出时，不会自动根据输入 shape 修改这个 tensor 的 shape 为输出 shape，当这个 tensor 作为下一个运算的输入时，需要手动使用 view 或 sub_view 修改它的 shape。

使用示例

```
dim4 src_shape = {N, C, H, W};
// 创建一个bf16类型的64-Bytes对齐存储的tensor
auto src0 = tensor<bf16>(src_shape);
// 创建一个fp32类型的compact对齐存储的tensor
auto src1 = tensor<fp32>(src_shape, TPU_COMPACT);
```

view

在 PPL 中，view 函数是用来改变看待 tensor 的方式而不改变其数据的一种方法。在 for 循环中，最后一次循环的数据量通常是小于前面循环的，例如：

```
int N = 10;
int block_n = 4;
for (int i = 0; i < N; i += block_n) {
  int slice = min(block_n, N - i);
}
```

循环前两次 slice 为 4，最后一次为 2。同样的，在 tensor 的运算过程中，最后一次循环的数据量很可能会比前几次小，因此在循环中需要通过 view 来修改 tensor 的 shape，使 tensor 的 shape 为当前实际 shape。调用 view 后并不会改变 tensor 的实际内存，如果 view 的参数只有 shape，那么返回 tensor 的 stride 是根据 view 传入的 shape 重新计算得到的。此功能也可以当做 reshape 使用，但是用户需要非常熟悉数据在 TPU 上的排列方式，否则极易出现问题。

view 还有其他 2 个功能：1. 改变 tensor 的数据类型 2. 改变 tensor 的存储方式。

在已创建 tensor 对象 src （auto src = tensor<type>(shape); ）的情况下：

- 改变类型：代码形式为 auto src1 = src.view<type1>();
- 改变形状，tensor 的 stride 也会改变：代码形式为 auto src2 = src.view(shape2);
- 改变存储方式：代码形式为 auto src3 = src.view(shape3, stride3);

注意事项

- 由于 tensor 对象不支持赋值运算符重载，PPL 不允许对已创建的 tensor 修改类型、形状和存储，即不可以调用 src = src.view(...);
- view 只传入 shape 参数，则会更新 tensor 的 shape 和 stride。stride 是通过新的 shape 以及对齐方式自动计算
- view 传入 shape 和 stride ，则新的存储方式为 Free 存储
- view 的 3 个功能可以组合使用，可以改变 tensor 的类型、形状和存储方式（只修改描述信息，不修改实际数据）。

使用示例

```
dim4 shape = {N, C, H, W};
dim4 stride = {N_stride, C_stride, H_stride, W_stride};
dim4 shape1 = {N1, C1, H1, W1};
dim4 stride1 = {C1*H1*W1, H1*W1, W1, 1};
auto src = tensor<bf16>(shape);          //创建一个bf16类型的64-Bytes对齐存储的tensor
auto src1 = src.view<float>();          //仅改变tensor的数据类型
auto src2 = src.view(shape1);           //仅改变tensor的形状
auto src3 = src.view(shape, stride);    //仅改变tensor的存储方式
auto src4 = src.view<float>(shape1);    //改变tensor的数据类型、形状
auto src5 = src.view<float>(shape1, stride1); //改变tensor的数据类型、形状、存储方式
```

sub_view

PPL 中的 sub_view 的主要作用是获取已有的 tensor 的数据的一部分，类似 crop 操作。

在已创建 tensor 对象 src （auto src = tensor<type>(shape); ）的情况下：具体使用 sub_view 的代码形式为：auto src1 = src.sub_view(shape1, offset);

注意事项

- sub_view 返回的对象也是一个 tensor 对象，它的数据类型和在 local memory 上的存储方式与父 tensor 相同
- 返回 tensor 的地址是 sub_view 参数 offset 与父 tensor 的 stride 计算后得到的
- 返回 tensor 的 stride 为父 tensor 的 stride
- 需要注意数据 C 维度是存储在不同 Lane 上的，如果在 C 维度做 sub_view 需要十分了解数据在 Lane 上的分布，否则容易出错

使用示例

```
dim4 shape = {N, C, H, W};
dim4 shape1 = {N1, C1, H1, W1};
dim4 offset = {n, c, h, w};
auto src = tensor<bf16>(shape); //创建一个bf16类型、64-Bytes对齐存储的tensor
auto src1 = src.sub_view(shape1, offset); //返回src自offset开始，形状为shape1的一个子tensor
```

make_tensor

PPL 中的 make_tensor 的主要作用是创建一个 tensor，然后对该 tensor 进行 view 操作。其具体功能与 auto src = tensor<bf16>(block_shape);，auto src1 = src.view(real_shape); 这两句代码一致。为了提高 PPL 易用性和减少重复代码量，使用 make_tensor 替代上述两句代码。

具体使用 make_tensor 的代码形式为：auto src1 = make_tensor<bf16>(block_shape, real_shape);。配合 PPL 的自动内存分配功能使用，主要用于 for 循环中 block_shape 必须为编译期常量的情况，而最后一次真实的 real_shape 通常小于 block_shape，即需要进行 view 操作，此时，real_shape 不必是常量。

注意事项

- make_tensor 返回的对象也是一个 tensor 对象
- real_shape 必须小于等于 block_shape

使用示例

```
int N = 10;
int block_n = 4;
for (int i = 0; i < N; i += block_n) {
  ...
  int slice_n = min(block_n, N - i);
  dim4 block_shape = {slice_n, C, H, W};
  dim4 real_shape = {slice_n, C, H, W};
  // 申请一个bf16类型、64-Bytes对齐存储、大小为real_shape的tensor
  auto src = make_tensor<bf16>(block_shape, real_shape);

  // 与下面两句功能相同
  // 申请一个bf16类型、64-Bytes对齐存储、大小为block_shape的tensor
  auto src0 = tensor<bf16>(block_shape);
  // view成为一个bf16类型、64-Bytes对齐存储、大小为real_shape的tensor
  auto src1 = src0.view(real_shape);
  ...
}
```

3.2.3 gtensor

gtensor 是对数据在 global memory 或 l2 memory 上存储方式的一种抽象，包含了数据在 DDR 上存储的地址、shape、stride 等信息。创建 global memory 上的 gtensor 必须指定地址。

PPL 中 gtensor 结构体的构造函数部分为：

```
template <typename dtype> struct gtensor {
  public:
    ...
    template <typename dimT>
    explicit gtensor(dimT &_shape, tensor_mode_t mode, dtype *address = nullptr);
    ...
};
```

参数

- _shape: gtensor 的几何尺寸，为一个 dim 类型的结构体。
- mode: tensor 数据存储的位置，包括 global memory、L2 memory。
- address: tensor 在 global memory 上的起始地址，nullptr 表示由 PPL 自动分配。

使用示例

```
dim4 shape = {N, C, H, W};
dim4 shape1 = {N1, C1, H1, W1};
dim4 offset = {n, c, h, w};
// 创建一个位于global memory上bf16类型、紧密存储的gtensor
auto src = gtensor<bf16>(shape, GLOBAL, ptr_src);
auto src1 = src.sub_view(shape1, offset); // 返回src自offset开始，形状为shape1的一个子gtensor
```

make_gtensor

PPL 中的 make_gtensor 的主要作用是创建一个带有设定 stride 的 gtensor。其具体功能与 auto src = gtensor<bf16>(shape, GLOBAL, ptr_global);, auto src1 = src.view(shape, stride); 这两句代码一致。

具体使用 make_gtensor 的代码形式为：auto src1 = make_gtensor<bf16>(shape, GLOBAL, ptr_global, stride);。

注意事项

- make_gtensor 返回的对象是一个 gtensor 对象

使用示例

```
dim4 shape = {N, C, H, W};
dim4 stride = {C*H*W, H*W, W, 1};
// 申请一个bf16类型、64-Bytes对齐存储、大小为shape1的tensor
auto src = make_gtensor<bf16>(shape, GLOBAL, ptr_global, stride);

// 与下面两句功能相同
// 申请一个bf16类型、64-Bytes对齐存储、大小为shape的tensor
auto src0 = gtensor<bf16>(shape, GLOBAL, ptr_global);
// 然后view成为一个bf16类型、64-Bytes对齐存储、大小为shape1的tensor
auto src1 = src0.view(shape, stride);
```

3.2.4 数据类型

PPL 中的 tensor 主要支持以下数据类型，每种算子所支持的数据类型存在差别，具体信息请参考后续章节。

整型：

- INT8 (signed/unsigned), 8 bits
- INT16 (signed/unsigned), 16 bits
- INT32 (signed/unsigned), 32 bits

浮点类型：

- FP8, 8 bits
- FP16, 16 bits
- BF16, 16 bits
- FP32, 32 bits
- TF32, 19 bits

FP16

FP16 遵守 IEEE std 754-2008，支持操作数与结果为 Denormal 的格式。FP16 小数如下表所示，s 表示符号位 (1 bit)，e 表示阶码 (5 bits)，f 表示尾数 (10 bits):

表 1: FP16 数值表示

Expression	Value	Description	Example
$e = 255, f \neq 0$	NaN	非数 (NAN)	0xFFFF, 0x7F81
$s = 0, e = 255, f = 0$	+Inf	正无穷 (+INF)	0x7F80
$s = 1, e = 255, f = 0$	-Inf	负无穷 (-INF)	0xFF80
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times (1.f)$	规格数 (normal)	0x7800, 0x2333
$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times (0.f)$	非规格数 (denormal)	0x807F, 0x0001
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示 -0，s=0 表示 +0	0x8000, 0x0000

BF16

BF16 宽度 16bit 浮点，其小数如下表所示，s 表示符号位 (1 bit)，e 表示阶码 (8 bits)，f 表示尾数 (7 bits):

表 2: BF16 数值表示

Expression	Value	Description	Example
$e = 255, f \neq 0$	NaN	非数 (NAN)	0xFFFF, 0x7F81
$s = 0, e = 255, f = 0$	+Inf	正无穷 (+INF)	0x7F80
$s = 1, e = 255, f = 0$	-Inf	负无穷 (-INF)	0xFF80
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times (1.f)$	规格数 (normal)	0x7800, 0x2333
$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times (0.f)$	非规格数 (denormal)	0x807F, 0x0001
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示 -0，s=0 表示 +0	0x8000, 0x0000

FP32

FP32 遵守 IEEE std 754-2008. FP32 小数如下表所示，s 表示符号位 (1 bit)，e 表示阶码 (8 bits)，f 表示尾数 (23 bits)

表 3: FP32 数值表示

Expression	Value	Description	Example
$e = 255, f \neq 0$	NaN	非数 (NAN)	0xFFFF0000, 0x7F810000
$s = 0, e = 255, f = 0$	+Inf	正无穷 (+INF)	0x7F800000
$s = 1, e = 255, f = 0$	-Inf	负无穷 (-INF)	0xFF800000
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times (1.f)$	规格数 (normal)	0x78000000, 0x23332333
$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times (0.f)$	非规格数 (denormal)	0x807F6666, 0x00000001
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示 -0，s=0 表示 +0	0x80000000, 0x00000000

FP8

FP8 是宽度 8bit 的浮点，编码格式分为 F8E4 和 F8E5. F8E4 即 1bit 符号位、4bits 阶码、3bits 尾数，F8E5 即即 1bit 符号位、5bits 阶码、2bits 尾数。F8E4 编码格式被推荐用于深度学习模型的 Weight 和 Activation，而 F8E5 编码格式被推荐用于深度学习模型的 gradient.

表 4: F8E4 数值表示

Expression	Value	Description	Example
$e = 15, f = 7$	NaN	非数 (NAN)	0xFF, 0x7F
N/A	+Inf	正无穷 (+INF)	N/A
N/A	-Inf	负无穷 (-INF)	N/A
$0 < e <= 15,$	$(-1)^s \times 2^{e-7} \times (1.f)$	规格数 (normal)	0x7E, 0x08
$e = 0, f \neq 0$	$(-1)^s \times 2^{-6} \times (0.f)$	非规格数 (denormal)	0x07, 0x01
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示 -0，s=0 表示 +0	0x80, 0x00

表 5: F8E5 数值表示

Expression	Value	Description	Example
$e = 31, f \neq 0$	NaN	非数 (NAN)	0xFE, 0x7D
$s = 0, e = 31, f = 0$	+Inf	正无穷 (+INF)	0x7C
$s = 1, e = 31, f = 0$	-Inf	负无穷 (-INF)	0xFC
$0 < e < 31$	$(-1)^s 2^{e-15}(1.f)$	规格数 (normal)	0xFB, 0x84
$e = 0, f \neq 0$	$(-1)^s 2^{-14}(0.f)$	非规格数 (denormal)	0x83, 0x81
$e = 0, f = 0$	$(-1)^s 0$	s=1 表示 -0，s=0 表示 +0	0x80, 0x00

FP20

此数据类型为 FP32 的有损压缩格式, 用于减少 CDMA 芯片间通讯的数据带宽。FP20 只在 ARE 中参与 reduce 计算 (MUL/ADD/MAX 等)，其他情况下只作为存储格式存放在 GMEM/L2M 中。FP20 数值如下表所示，s 表示符号位 (1 bit)，e 表示阶码 (8 bits)，f 表示尾数 (11 bits)。

表 6: FP20 数值表示

Expression	Value	Description	Example
$e = 255, f \neq 0$	NaN	非数 (NAN)	0xFFFF0, 0x7F810
$s = 0, e = 255, f = 0$	+Inf	正无穷 (+INF)	0x7F800
$s = 1, e = 255, f = 0$	-Inf	负无穷 (-INF)	0xFF800
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times (1.f)$	规格数 (normal)	0x78000, 0x23332
$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times (0.f)$	非规格数 (denormal)	0x807F6, 0x00001
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示 -0，s=0 表示 +0	0x80000, 0x00000

TF32

TF32 是一种浮点格式，由 1 bit 符号位, 8 bit 指数位, 以及 10 bit 尾数位组成，占 19 bit。TF32 结合了 BF16 和 FP16 等格式的优点，具有较大的表示范围和精度，比 FP32 格式更节省数据/计算位宽。TF32 只在 CUBE 的计算中使用：数据在 LMEM 上仍以 FP32 格式存放，在 CUBE 读取数据时会转为 TF32 格式参与计算；计算结果再转换为 FP32 存回 LMEM.

表 7: TF32 数值表示

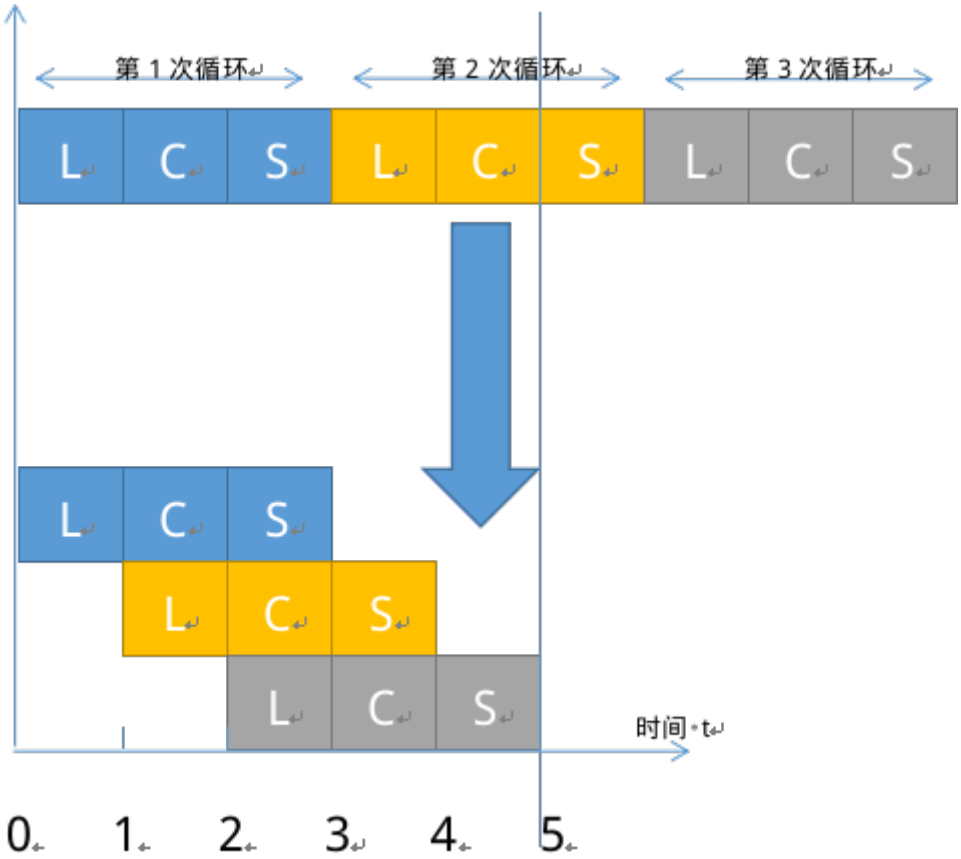
Expression	Value	Description
$e = 255, f \neq 0$	NaN	非数 (NAN)
$s = 0, e = 255, f = 0$	+Inf	正无穷 (+INF)
$s = 1, e = 255, f = 0$	-Inf	负无穷 (-INF)
$0 < e < 255$	$(-1)^s \times 2^{(e-127)} \times (1.f)$	规格数 (normal)
$e = 0, f \neq 0$	$(-1)^s \times 2^{-126} \times (0.f)$	非规格数 (denormal)
$e = 0, f = 0$	$(-1)^s \times 0$	s=1 表示-0，s=0 表示 +0

3.3 自动内存分配

创建 Tensor 时，如果不指定 address 参数，PPL 就会在编译期自动对 Tensor 进行地址分配，因此 Tensor 的 shape 在编译期必须是常量，这样 PPL 才能计算 Tensor 需要的内存大小。用户可以在使用 Tensor 时，可以使用 view 或 sub_view 修改 Tensor 的 shape 为实际数据 shape。

3.4 自动流水线并行

3.4.1 基本原理



上图描述了一个循环 3 次的计算，串行与并行的耗时，L 表示 dma load 操作，C 表示 tiu computer 操作，S 表示 dma store 操作。这个计算用伪代码表示如下：

```
for (int i = 0; i < 3; ++i) {
  load to a;
  computer b = a + 1;
  store b to ddr;
}
```

假如每个操作耗时是 1，那么串行耗时是 9。dma 和 tiu 其实是可以同时运行的，所以这个计算过程是可以优化成并行的，如上图下方所示，纵轴上的计算就是在并行，那么总耗时就是 5 了。

并行计算步骤如下：

- t0 时刻运行第一次循环的 L
- t1 时刻运行第一次循环的 C 和第二次循环的 L

- t2 时刻运行第一次循环的 S，第二次循环的 C，第三次循环的 L
- t3 时刻运行第二次循环的 S，第三次循环的 C
- t4 时刻运行第三次循环的 S

PPL 可以自动将用户的串行代码优化为并行代码，只需要在 for 循环内加上 **ppl::enable_pipeline()** 函数即可开启此优化。但需要注意，**不是所有的 for 循环都可以进行流水并行优化**。

PPL 自动流水并行是通过将计算分成多个 Stage(例如图中的 L、C、S 就是 3 个 Stage)，然后如上图，将 for 循环展开重排，从而实现的。Stage 的切分是根据 Tensor 以及对这个 Tensor 的计算决定的，例如，Tensor t1 上有连续两次操作，记为 op1，op2，如果 op1 与 op2 都是 dma 或者 tiu，那么 op1 与 op2 的 Stage 是一样的，如果一个为 dma，一个为 tiu，那么 op2 的 Stage 为 op1 的 Stage + 1。

Stage 的切分示例:

```
for (int i = 0; i < 3; ++i) {
  load to a;          // Stage = 0
  computer b = a + 1;   // Stage = 1
  store b to ddr;       // Stage = 2
}

for (int i = 0; i < 3; ++i) {
  load to a;          // Stage = 0
  load to b;          // Stage = 0
  computer c = a + b;   // Stage = 1
  store c to ddr;       // Stage = 2

for (int i = 0; i < 3; ++i) {
  load to a;          // Stage = 0
  computer c = a + 1;   // Stage = 1
  load to b;          // Stage = 0
  computer d = c + b;   // Stage = 1
  store d to ddr;       // Stage = 2
```

使用 PPL 自动流水并行需要注意的事项

- for 循环内至少有 2 种 Stage 才能做流水并行。
- 如果一个 Tensor 被 2 种 Stage 使用，那么这个 Tensor 需要复制一份，即 ping-pong 内存。
- 如果需要复制的 Tensor 有初始值，或者 for 结束后还需要使用，那么会出问题，这种情况也无法进行自动流水并行。
- PPL 目前只支持 Tensor 复制一份，因此一个 Tensor 最多只能被 2 个 Stage 使用。

3.5 多 TPU core 编程模型

3.5.1 多 TPU core 编程介绍

PPL 支持多 TPU core 开发。在进行多核开发时用户看到的是多个 TPU Core System，例如：对于 BM1690 芯片，每个 TPU Core System 包含 1 个由 64 个 Lanes 的 Vector、Cube 和 TIU 组成的 TPU 计算引擎、1 个 GDMA、1 块 LMEM、1 块 SMEM、1 个 Scalar Engine、1 个 HAU、1 个 SDMA.

多 TPU Core 编程模型，即用户所写程序是针对一个 TPU Core System 的，执行时程序将被复制到多个 TPU Core System 中并行执行。用户在进行多核的开发时能获取到两个与多 TPU Core 编程有关的变量：CoreNum 和 CoreID。CoreNum 即 TPU Core System Number，意思是执行一个任务时要用到的 TPU Core System 的数量。CoreID 即 Core System Identity，意思是程序所运行的 TPU Core 标识，例如 CoreNum = 4，则 CoreID 可以是 0,1,2,3，CoreID 必须小于 CoreNum。多 TPU Core 编程时，根据 CoreNum 对任务要处理的数据进行多 Core 划分，然后根据 Core ID 选择该 Core 要处理的数据.

用户可以看到的存储空间有 LMEM、SMEM、L2M 和 DDR。LMEM 和 SMEM 是一个 TPU Core System 私有的，L2M 和 DDR 可以是私有的也可以是共享的。当用户利用多个 TPU Core System 来共

同完成一个任务时，此时 L2M 和 DDR 对于多个 TPU Core System 来说是共享的；当用户只利用 1 个 TPU Core System 来完成任务时，此时 L2M 和 DDR 是私有的。例如：用户所写程序想利用 4 个 TPU Core System 来完成一个计算任务，此时 L2M 中的一半空间 (因为 BM1690 芯片总共有 8 个 TPU Core System) 可被 4 个 TPU Core System 共享，共享空间大小为了方便软件调度，原则上按照核数比例分配，同时该程序分配的 DDR 空间也可被这 4 个 TPU Core System 共享。而在一个 TPU Core System 内部，用户需要利用 SDMA 指令和 GDMA 指令来搬运数据，利用 TPU 指令和 HAU 指令来做计算.

3.5.2 使用 PPL 进行多核编程

由于 kernel 函数可使用的 L2 memory 大小是跟 kernel 函数使用的 TPU core 数量有关，因此在编译期需要确定要使用的 TPU core 数据，可以通过以下方式来设置核数量：

使用 set_core_num() 来设置多核

set_core_num() 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 set_core_num 后，还需要调用 get_core_num() 来获取运行时 kernel 函数使用的 core 数量，可以通过宏来设置 core 数量，然后编译的时候将宏传入，以便在编译期修改 core 数量。**推荐使用此方式**。

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

#ifndef CORE_NUM
#define CORE_NUM 2
#endif
__KERNEL__ void add_pipeline(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 = 1;
 const int C = 64;
 const int H = 1;
 ppl::set_core_num(CORE_NUM);    // 设置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_num = 6;
 // ppl::set_core_num(core_num);        // 设置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 = 128;                    // 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 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
 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
```

(续下页)

(接上页)

```
    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
    dma::store(res_gtensor.sub_view(cur_shape, offset), res);   // 将数据从 local mem 到 ddr
  }
}

__TEST__ void add_pipeline_main() {
  const int N = 1;
  const int C = 64;
  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_pipeline(res, inp, W);          // 使用 6 个核运行核函数
}
```

使用 MULTI_CORE 来设置多核

MULTI_CORE 实际是封装了一个模板参数来设置宏数量，如果用户编写了 __TEST__ 函数，则可以在 test 函数中通过模板参数传入 core 数量，如果用户不编写 __TEST__ 函数则不能使用此方式，或者需要自己手动对 kernel 函数使用模板实例化。

```
#include "ppl.h"  // PPL 代码依赖的头文件
using namespace ppl;
MULTI_CORE
__KERNEL__ void add_pipeline(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 = 1;
  const int C = 64;
  const int H = 1;
  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_num = 6;
  // ppl::set_core_num(core_num);        // 设置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 = 128;                      // 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 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
```

(续下页)

(接上页)

```
  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
    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
    dma::store(res_gtensor.sub_view(cur_shape, offset), res);    // 将数据从 local mem 到 ddr
  }
}

__TEST__ void add_pipeline_main() {
  const int N = 1;
  const int C = 64;
  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_pipeline<6>(res, inp, W);          // 使用 6 个核运行核函数
}
```

3.6 动态 BLOCK

由于 PPL 负责自动分配内存，因此一般情况需要用户使用静态的 BLOCK，即 **tensor 初始化时，shape 必须是常量（对 tensor 进行 view 或 sub_view 操作的 shape 不必是常量）**，以便在编译期完成内存分配的计算，但是在一些情况下，静态的 Block size 会导致性能不佳，例如矩阵乘计算中，如果使用静态的 Block ，则所有的维度都必须切分，而切分后运算会导致矩阵重复搬运，因此我们提供了动态 BLOCK 功能，此时内存分配的计算不再是编译期完成，而是在 Host 端封装函数运行时即时计算。Host 封装函数在调用 kernel 函数前，会先调用内存分配函数，确认内存分配成功后再调用 kernel 函数，如果内存分配失败，则返回 -1 ，用户需要改小 Block size，然后再调用 Host 端封装函数，直到返回 0。

注意：使用动态 BLOCK 功能，即 Block size 作为参数传入 kernel 函数时，必须添加 const 关键字修饰。

3.6.1 动态 BLOCK 代码示例

```
#include "ppl.h"
using namespace ppl;

// block_w 必须添加 const 关键字修饰
__KERNEL__ void add(fp16 *ptr_res, fp16 *ptr_inp, int W, const int block_w) {
  const int N = 1;
  const int C = 64;
  const int H = 1;
  ppl::set_core_num(6);
  int core_num = ppl::get_core_num();
  int core_idx = ppl::get_core_index();
  if (core_idx >= core_num) {
    return;
  }
```

(续下页)

(接上页)

```
  int slice = div_up(W, core_num);
  int cur_slice = min(slice, (W - slice * core_idx));
  int slice_offset = core_idx * slice;
  dim4 global_stride = {C * H * W, H * W, W, 1};

  dim4 block_shape = {N, C, H, block_w};
  auto in_tensor = tensor<fp16>(block_shape);
  auto res = tensor<fp16>(block_shape);
  float scalar_c = 0.25;

  for (int w_idx = 0; w_idx < cur_slice; w_idx += block_w) {
    enable_pipeline();
    int tile_w = min(block_w, cur_slice - w_idx);
    dim4 cur_shape = {N, C, H, tile_w};
    auto cur_tensor = in_tensor.view(cur_shape);
    auto cur_res = res.view(cur_shape);

    dim4 offset = {0, 0, 0, slice_offset + w_idx};
    dma::load(cur_tensor, ptr_inp, &offset, &global_stride);
    tiu::fp_add(cur_res, cur_tensor, scalar_c);
    dma::store(ptr_res, cur_res, &offset, &global_stride);
  }
}
```

3.6.2 编译

动态 BLOCK 编译与普通模式是一样的

以 example/cxx/arith/add_dyn_block.pl 源码为例，运行编译如下：

```
ppl-compile examples/cxx/arith/add_dyn_block.pl --print-debug-info --print-ir --chip bm1684x
--O2 --o test
```

运行此命令后，会在当前路径下生成 test 目录，目录结构如下：

```
├─── device/           # 编译后生成的device端代码存放目录，代码名字为：KERNEL函数名字.c
|    └─── add.c        # KERNEL函数代码
├─── host/             # 编译后生成的host端代码存放目录
|    ├─── add.cpp      # HOST端封装调用kernel函数代码
|    └─── add_mem.cpp  # HOST端内存分配函数代码(动态 BLOCK 功能特有)
├─── include/          # 编译后生成的头文件，包括device与host端
|    ├─── add.h        # KERNEL函数头文件
|    └─── add_mem.h    # HOST端内存分配函数头文件(动态 BLOCK 功能特有)
└─── *.mlir/          # 如果设置了--print-ir，则会dump编译过程中的ir文件
```

使用动态 BLOCK 功能会多产生 ${KERNEL_FUNC_NAME}_mem.cpp、${KERNEL_FUNC_NAME}_mem.h 两个文件，这两个文件作用是在 HOST 端进行内存分配工作。另外 KERNEL 函数参数会在末尾多添加一个参数，用来传递 HOST 端实时计算出来的地址。此时 KERNEL 函数参数结构体定义如下：

```
#ifdef __cplusplus
extern "C" {
#endif
typedef struct {
  global_addr_t v1;       // fp16 *ptr_res
  global_addr_t v2;       // fp16 *ptr_inp
  int32_t v3;             // int W
  int32_t v4;             // const int block_w
  int32_t v5[4];          // 自动生成的内存地址数组
```

(续下页)

(接上页)

```
} tpu_kernel_api_add_t;
#ifdef __cplusplus
}
#endif
```

add_mem.cpp 内容如下：

```
#include "add_mem.h"
#include <ppl_mem.h>
#include <mutex>
#include <memory>
int add_check_mem(unsigned long long v1, unsigned long long v2,
                  int32_t v3, int32_t v4, int32_t* v5) {
  int64_t localMemSize = 262144;
  int64_t l2MemSize = 100663296;
  int bankNum = 16;
  int bankSize = 16384;
  static std::map<int, __PplMemNode> localMems;
  static std::map<int, __PplMemNode> l2Mems;
  std::map<int, int64_t> memSizes;
  std::vector<int64_t> addrs;
  std::mutex mtx;
  int tensor_num = 4;
  if (localMems.empty()) {
    std::unique_lock<std::mutex> lock(mtx);
    if (localMems.empty()) {
      localMems.emplace(0, __PplMemNode(0, 1, 4, std::set<int>({3,2})));
      localMems.emplace(1, __PplMemNode(1, 1, 4, std::set<int>({3,2})));
      localMems.emplace(2, __PplMemNode(2, 1, 4, std::set<int>({1,3,0})));
      localMems.emplace(3, __PplMemNode(3, 1, 4, std::set<int>({1,2,0})));
    }
  }
  memSizes[0] = align_up(calc_mem_size({1, 64, 1, v4}, 64, 64, 2, 1), 64);
  memSizes[1] = align_up(calc_mem_size({1, 64, 1, v4}, 64, 64, 2, 1), 64);
  memSizes[2] = align_up(calc_mem_size({1, 64, 1, v4}, 64, 64, 2, 1), 64);
  memSizes[3] = align_up(calc_mem_size({1, 64, 1, v4}, 64, 64, 2, 1), 64);
  addrs.resize(tensor_num);
  int ret = check_mem_valid(localMemSize, l2MemSize, bankSize, bankNum,
                            localMems, l2Mems, memSizes, addrs);
  if (ret != 0) {
    assert(0 && "local memory not enough, please reduce block size");
    return -1;
  }
  for (int i = 0; i < tensor_num; ++i) {
    v5[i] = addrs[i];
  }
  return 0;
}
```

这里面包含一个名字为 ${KERNEL_FUNC_NAME}_check_mem 的函数，参数与 KERNEL 函数一致 (末尾自动添加了内存地址数组 v5)。

3.7 自动 shape 推导

为了减少用户重复设置 tensor 的 shape 信息，我们支持对有输入的指令的 **输出**做自动的 shape 推导，这样用户就不用对每个 tensor 都设置 shape 了。目前只支持静态 Block size；动态 Block size 需要用户手动设置所有 tensor shape。

shape 推导的原则如下：

- 我们只对 tiu/dma 等存在输入的指令的输出进行 shape 推导，如果某个 tensor 不是来源于前面 tiu/dma 的输出，用户也没手动设置 shape，那么将会出错

```
tensor<fp16> input;  // 这里必须手动设置input tensor shape，否则会出错
tiu::zero(input);    // input作为zero的输入，但是shape是未知的，也无法推导
dma::load(input, g_input) // 如果g_input设置了shape，此时input的shape可以推导得出
```

- 我们可以推导输出 tensor 的 block shape；如果 tensor 需要使用 view 我们也会自动设置 view
- 如果创建一大块内存，但通过 sub_view 每次只存入一小块内存，那么这个大块内存需要用户手动设置 shape

```
tensor<fp16> input;  // 这里必须手动设置input tensor shape，否则会出错
for (int i = 0; i < 10; ++i) {
  dim4 offset = {i, 0, 0, 0};
  dma::load(input.sub_view(shape, offset), g_input);
}
```

使用示例

示例中 sub_left、sub_right、res_fp16 等 tensor 的 shape 是通过自动推导得到

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void mm2_fp16(fp16 *ptr_res, fp16 *ptr_left, fp16 *ptr_right, int M,
                         int K, int N) {
  const int block_m = 128;
  const int block_k = 256;
  const int block_n = 256;

  dim4 res_global_shape = {1, M, 1, N};
  dim4 left_global_shape = {1, M, 1, K};
  dim4 right_global_shape = {1, K, 1, N};

  auto res_gtensor = gtensor<fp16>(res_global_shape, GLOBAL, ptr_res);
  auto left_gtensor = gtensor<fp16>(left_global_shape, GLOBAL, ptr_left);
  auto right_gtensor = gtensor<fp16>(right_global_shape, GLOBAL, ptr_right);

  dim4 res_max_shape = {1, block_m, 1, block_n};
  dim4 left_max_shape = {1, block_m, 1, block_k};
  dim4 right_max_shape = {1, block_k, 1, block_n};
  tensor<fp16> res_fp16;
  for (auto idx_m = 0; idx_m < M; idx_m += block_m) {
    for (auto idx_n = 0; idx_n < N; idx_n += block_n) {
      int m = min(block_m, M - idx_m);
      int n = min(block_n, N - idx_n);
      dim4 res_shape = {1, m, 1, n};
      auto sub_res = make_tensor<fp32>(res_max_shape, res_shape);
      tiu::zero(sub_res);
      for (auto idx_k = 0; idx_k < K; idx_k += block_k) {
        enable_pipeline();
        int k = min(block_k, K - idx_k);
        dim4 left_max_shape = {1, block_m, 1, block_k};
        dim4 right_max_shape = {1, block_k, 1, block_n};
```

(续下页)

(接上页)

```
      dim4 left_real_shape = {1, m, 1, k};
      dim4 right_real_shape = {1, k, 1, n};

      tensor<fp16> sub_left, sub_right;
      dim4 left_offset = {0, idx_m, 0, idx_k};
      dim4 right_offset = {0, idx_k, 0, idx_n};
      dma::load(sub_left, left_gtensor.sub_view(left_real_shape, left_offset));
      dma::load(sub_right, right_gtensor.sub_view(right_real_shape, right_offset));
      tiu::fmm2(sub_res, sub_left, sub_right, true, true);
    }
    tiu::cast(res_fp16, sub_res);
    dim4 res_offset = {0, idx_m, 0, idx_n};
    dma::store(res_gtensor.sub_view(res_shape, res_offset), res_fp16);
  }
 }
}
```

3.8 PPL 编程注意事项

- 在循环中一般使用调用 tensor.view 返回的 tensor；或者使用调用 make_tensor 返回的 tensor。
- 尽量不使用数组，PPL 对数组的支持目前不完善，复杂的场景可能出问题，尽量避免使用数组，或者使用数组时，初始化值，然后不再修改值。
- 不对标量做复杂的计算。
- 尽量将 Tensor shape 的 c 维度对齐到 Lane num，这样能达到比较好的性能，因为底层指令会将 c 维度数据平摊到不同的 Lane 上，然后并行计算。

CHAPTER 4

PPL API

4.1 基础定义

- 数据类型

```
typedef enum data_type_t {
  DT_NONE = 0,
  DT_FP32,
  DT_FP16,
  DT_BF16,
  DT_FP8E5M2,
  DT_FP8E4M3,
  DT_FP20,
  DT_TF32,
  DT_INT32,
  DT_UINT32,
  DT_INT16,
  DT_UINT16,
  DT_INT8,
  DT_UINT8,
  DT_INT4,
  DT_UINT4,
  DT_INT64,
  DT_UINT64,
};
```

- 基础数学运算类型枚举

```
enum arith_mode_t {
  ARITH_AND = 0,
  ARITH_OR = 1,
  ARITH_XOR = 2,
  ARITH_MIN = 3,
  ARITH_MAX = 4,
  ARITH_ADD = 5,
  ARITH_SUB = 6,
  ARITH_MUL = 7,
  ARITH_DIV = 8,
```

(续下页)

(接上页)

```
  ARITH_DIFF_ABS = 9,
  ARITH_MAC = 10,
};
```

- 比较运算类型枚举

```
enum comparision_mode_t {
  GREATER = 0,
  LESS = 1,
  EQUAL = 2,
};
```

- 舍入模式枚举

```
enum rounding_mode_t {
  RM_HALF_TO_EVEN = 0,
  RM_HALF_AWAY_FROM_ZERO = 1,
  RM_TOWARDS_ZERO = 2,
  RM_DOWN = 3,
  RM_UP = 4,
  RM_HALF_UP = 5,
  RM_HALF_DOWN = 6,
};
```

- 参数表枚举

```
enum coeff_table_mode_t {
  EXP = 0,
  LOG = 1,
  SIN = 2,
  COS = 3,
  TAN = 4,
  ARCSIN = 5,
  ERF_TAYLOR = 6,
};
```

- 转置模式枚举

```
enum transpose_mode_t {
  NC_TRANS = 0,
  CW_TRANS = 1,
};
```

- 数据对齐模式枚举

```
enum align_mode_t {
  CONTINUOUS = 0,
  TPU_ALIGN = 1,
  TPU_COMPACT = 2,
  TPU_ROW_ALIGN = 3,
  NONE_ALIGN = 4,
};
```

- Tensor 的 Memory 位置

```
enum tensor_mode_t {
  L2 = 1,
  GLOBAL = 2,
};
```

4.2 舍入模式

舍入是指按照一定的规则舍去某些数字后面多余的尾数的过程，以得到更简短、明确的数字表示。给定 x，舍入结果是 y，有下面的舍入模式供选择。

4.2.1 邻近偶数四舍五入（Half to Even）

四舍五入，当小数值为 0.5 时舍入到邻近的偶数，对应的枚举值是 RM_HALF_TO_EVEN。

4.2.2 远离原点四舍五入（Half Away From Zero）

四舍五入，正数接近于正无穷，负数接近于负无穷，对应的枚举值是 RM_HALF_AWAY_FROM_ZERO，公式如下

$$\mathrm{y} = \mathrm{sign(x)}\,\lfloor |\mathrm{x}| + 0.5 \rfloor = -\mathrm{sign(x)}\,\lceil -|\mathrm{x}| - 0.5 \rceil$$

4.2.3 截断取整（Towards Zero）

无条件舍去，接近于原点，对应的枚举值是 RM_TOWARDS_ZERO，公式如下

$$\mathrm{y} = \mathrm{sign(x)}\,\lfloor |\mathrm{x}| \rfloor = -\mathrm{sign(x)}\,\lceil -|\mathrm{x}| \rceil = \begin{cases} \lfloor \mathrm{x} \rfloor & \text{if } \mathrm{x} > 0, \\ \lceil \mathrm{x} \rceil & \text{otherwise.} \end{cases}$$

4.2.4 下取整（Down）

接近于负无穷，对应的枚举值是 RM_DOWN，公式如下

$$\mathrm{y} = \lfloor \mathrm{x} \rfloor = -\lceil -\mathrm{x} \rceil$$

4.2.5 上取整（Up）

接近于正无穷，对应的枚举值是 RM_UP，公式如下

$$\mathrm{y} = \lceil \mathrm{x} \rceil = -\lfloor -\mathrm{x} \rfloor$$

4.2.6 向上四舍五入（Half Up）

四舍五入，接近于正无穷，对应的枚举值是 RM_HALF_UP，公式如下

$$\mathrm{y} = \lceil \mathrm{x} + 0.5 \rceil = -\lfloor -\mathrm{x} - 0.5 \rfloor = \left\lfloor \frac{\lfloor 2\mathrm{x} \rfloor}{2} \right\rfloor$$

4.2.7 向下四舍五入（Half Down）

四舍五入，接近于正无穷，对应的枚举值是 RM_HALF_DOWN，公式如下

$$\mathrm{y} = \lfloor \mathrm{x} - 0.5 \rfloor = -\lceil -\mathrm{x} + 0.5 \rceil = \left\lfloor \frac{\lceil 2\mathrm{x} \rceil}{2} \right\rfloor$$

4.2.8 例子

下表列出不同舍入模式下 x 与 y 的对应关系。

	Half to Even	Half Away From Zero	Towards Zero	Down	Up	Half Up	Half Down
$+1.8$	$+2$	$+2$	$+1$	$+1$	$+2$	$+2$	$+2$
$+1.5$	$+2$	$+2$	$+1$	$+1$	$+2$	$+2$	$+1$
$+1.2$	$+1$	$+1$	$+1$	$+1$	$+2$	$+1$	$+1$
$+0.8$	$+1$	$+1$	0	0	$+1$	$+1$	$+1$
$+0.5$	0	$+1$	0	0	$+1$	$+1$	0
$+0.2$	0	0	0	0	$+1$	0	0
-0.2	0	0	0	-1	0	0	0
-0.5	0	-1	0	-1	0	0	-1
-0.8	-1	-1	0	-1	0	-1	-1
-1.2	-1	-1	-1	-2	-1	-1	-1
-1.5	-2	-2	-1	-2	-1	-1	-2
-1.8	-2	-2	-1	-2	-1	-2	-2

4.3 GDMA 操作

4.3.1 数据搬运

ppl::dma::load

将张量的元素从 global / l2 memory 拷贝到 local memory

```
template <typename DataType>
void load(tensor<DataType> &dst, gtensor<DataType> &src);
```

$$\mathrm{dst}(n, c, h, w) = \mathrm{src}(n, c, h, w)$$

参数

- dst(tensor): local memory 上的张量
- src(gtensor): global / l2 memory 上的张量

注意事项

- dst_shape 和 src_shape 应相等。
- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void dmaload_fp32(float *ptr_rst, float *ptr_inp) {
  dim4 src_shape = {N, C, H, W};

  auto in_gt = gtensor<float>(src_shape, GLOBAL, ptr_inp);
  auto src = tensor<float>(src_shape);
  dma::load(src, in_gt);
}
```

ppl::dma::store

将张量的元素从 local memory 拷贝到 global memory

```
template <typename DataType>
void store(gtensor<DataType> *dst, tensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(gtensor): global memory 上的张量
- src(tensor): local memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 N-byte aligned layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

具体使用方式与load 一致。

ppl::dma::move

将张量的元素从 global 拷贝到 global 。

```
void move(gtensor<DataType> &dst, gtensor<DataType> &src);
```

$$dst(n, c, h, w) = src(n, c, h, w)$$

参数

- dst(gtensor): global memory 上的张量
- src(gtensor): global memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void s2s_fp32(fp32 *ptr_rst, fp32 *ptr_inp) {
  dim4 shape = {1, 2, 16, 16};
  auto in_gt = gtensor<fp32>(shape, GLOBAL, ptr_inp);
  auto out_gt = gtensor<fp32>(shape, GLOBAL, ptr_rst);
  dma::move(out_gt, in_gt);
}
```

ppl::dma::move

将张量的元素从 local memory 拷贝到 local memory。

```
template <typename DataType>
void move(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = src(n, c, h, w)$$

参数

- dst(gtensor): local memory 上的结果张量
- src(tensor): local memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 N-byte aligned layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void l2l_fp32(float *ptr_rst, float *ptr_inp) {

  const int N = 1;
  const int C = 64;
  const int H = 16;
  const int W = 1;

  dim4 src_shape = {N, C, H, W};
  dim4 dst_shape = {N, C, H, W};

  auto src = tensor<float>(src_shape);
  auto dst = tensor<float>(dst_shape);

  auto in_gt = gtensor<fp32>(src_shape, GLOBAL, ptr_inp);
  auto out_gt = gtensor<fp32>(dst_shape, GLOBAL, ptr_rst);

  auto in = tensor<fp32>(src_shape);
  auto out = tensor<fp32>(dst_shape);
  dma::load(in, in_gt);
  dma::move(out, in);
  dma::store(out_gt, out);
}
```

ppl::dma::load_transpose_cw

将张量的元素从 global / l2 memory 拷贝到 local memory，并进行 C 和 W 的维度转置。

```
template <typename DataType>
void load_transpose_cw(tensor<DataType> &dst, gtensor<DataType> &src,
                       transpose_mode_t trans_mode);
```

$$dst(n, c, h, w) = src(n, w, h, c)$$

参数

- dst(tensor): local memory 上的张量
- src(gtensor): global / l2 memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void dmaload_nc_trans_fp32(float *ptr_rst, float *ptr_inp) {
  const int N = 2;
  const int C = 4;
  const int H = 16;
  const int W = 32;

  const int N_ = 2;
  const int C_ = 5;
  const int H_ = 17;
  const int W_ = 32;
  dim4 src_shape = {N_, C_, H_, W_};
  dim4 dst_shape = {C, N, H, W};

  dim4 src_offset = {0, 1, 1, 0};

  auto src = tensor<float>(dst_shape);
  auto src_gt = gtensor<float>(src_shape, GLOBAL, ptr_inp);
  transpose_mode_t trans_mode = NC_TRANS;
  dim4 src_sub_shape = {N, C, H, W};
  dma::load_transpose_cw(src, src_gt.sub_view(src_sub_shape, src_offset));

  auto dst_gt = gtensor<float>(dst_shape, GLOBAL, ptr_rst);
  dma::store(dst_gt, src);
}
```

ppl::dma::load_transpose_nc

将张量的元素从 global / l2 memory 拷贝到 local memory，并进行 N 和 C 的维度转置。

```
template <typename DataType>
void load_transpose_nc(tensor<DataType> &dst, gtensor<DataType> &src,
                       transpose_mode_t trans_mode);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{c},\mathsf{n},\mathsf{h},\mathsf{w})$$

参数

- dst(tensor): local memory 上的张量
- src(gtensor): global / l2 memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

ppl::dma::store_transpose_cw

将张量的元素从 local memory 拷贝到 global memory，并进行 C 和 W 的维度转置。

```
template <typename DataType>
void store_transpose_cw(gtensor<DataType> &dst, tensor<DataType> &src,
                        transpose_mode_t trans_mode);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{w},\mathsf{h},\mathsf{c})$$

参数

- dst(gtensor): global memory 上的张量
- src(tensor): local memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 N-byte aligned layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

具体使用方式与load_transpose_nc 一致。

ppl::dma::store_transpose_nc

将张量的元素从 local memory 拷贝到 global memory，并进行 N 和 C 的维度转置。

```
template <typename DataType>
void store_transpose_nc(gtensor<DataType> &dst, tensor<DataType> &src,
                        transpose_mode_t trans_mode);
```

$$dst(n, c, h, w) = src(c, n, h, w)$$

参数

- dst(gtensor): global memory 上的张量
- src(tensor): local memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 N-byte aligned layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

具体使用方式与load_transpose_nc 一致。

ppl::dma::load_compact

张量的元素从 global / l2 memory 拷贝到 local memory，并按照 compact layout 排列。

```
template <typename DataType>
void load_compact(tensor<DataType> &dst, gtensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(tensor): local memory 上的张量
- src(gtensor): global / l2 memory 上的张量

注意事项

- dst 是 compact layout，src 是 continuous layout。

使用示例

具体使用方式与load 中 load 的使用方式一致。

ppl::dma::store_compact

将 local memory 上按照 compact layout 排列的张量的元素拷贝到 global memory。

```
template <typename DataType>
void store_compact(gtensor<DataType> &dst, tensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(gtensor): global memory 上的张量
- src(tensor): local memory 上的张量

注意事项

- dst 是 continuous layout，src 是 compact layout。

使用示例

ppl::dma::move_cross_lane

跨 LANE 拷贝张量的元素。

```
template <typename DataType>
void move_cross_lane(tensor<DataType> &dst, tensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(tensor): global memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 可以不从同一个 lane 开始，但都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。

使用示例

ppl::dma::reverse

将 tensor 数据沿着 dim 维度做倒置，支持 8/16/32 bit 等数据位宽。

```
template <typename DataType>
void reverse(tensor<DataType> &dst, tensor<DataType> &src, int dim);

template <typename DataType>
void reverse(tensor<DataType> &dst, gtensor<DataType> &src, int dim);

template <typename DataType>
void reverse(gtensor<DataType> &dst, tensor<DataType> &src, int dim);

template <typename DataType>
void reverse(gtensor<DataType> &dst, gtensor<DataType> &src, int dim);
```

参数

- dst(tensor/gtensor): global/local memory 上的 dst 张量
- src(tensor/gtensor): global/local memory 上的 src 张量

注意事项

- 当在 src 为 LMEM 时只支持 c 维的反转。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- **仅支持 SG2380**。

使用示例

ppl::dma::vload

将张量的元素从 global / l2 memory 拷贝到 vector 寄存器。

```
template <typename DataType>
void vload(int dst_v_idx, gtensor<DataType> &src);
```

参数

- dst_v_idx(int): vector 寄存器的索引
- src(gtensor): global memory 上的 src 张量

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- **仅支持 SG2380**。

使用示例

ppl::dma::vstore

将张量的元素从 vector 寄存器拷贝到 global / l2 memory。

```
template <typename DataType>
void vstore(gtensor<DataType> &dst, int v_idx);
```

参数

- src(gtensor): global memory 上的 dst 张量
- v_idx(int): vector 寄存器的索引

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- **仅支持 SG2380**。

使用示例

ppl::dma::move_tv

从 vector 寄存器加载数据到 LMEM/SMEM 的指令。指令将一组 vector 寄存器的数据加载到 LMEM 或 SMEM 上。

```
template <typename DataType>
void move_tv(tensor<DataType> &dst, int v_idx);

void move_tv(int smem_offset, int v_idx);
```

参数

- dst(tensor): local memory 上的 dst 张量
- smem_offset(int): static memory 上的地址偏移量
- v_idx(int): vector 寄存器的索引

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据须为 64bit 的倍数。
- **仅支持 SG2380**。

使用示例

ppl::dma::move_distv

从 vector 寄存器分发数据到 LMEM 的指令。指令将一组 vector 寄存器的数据均分到所有 Lane 上。

```
template <typename DataType>
void move_distv(tensor<DataType> &dst, int v_idx);
```

参数

- dst(tensor): local memory 上的 dst 张量
- v_idx(int): vector 寄存器的索引

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据大小须为 64bit 的倍数，并且数据元素个数须能被 LANE_NUM 整除。
- **仅支持 SG2380**。

使用示例

ppl::dma::move_vv

从 vector 寄存器加载数据到 LMEM/SMEM 的指令。指令将两组 vector 寄存器的数据拼接加载到 LMEM 或 SMEM 上。

```
template <typename DataType>
void move_vv(tensor<DataType> &dst, int v_idx0, int v_idx1);

void move_vv(int smem_offset, int v_idx0, int v_idx1);
```

参数

- dst(tensor): local memory 上的 dst 张量
- smem_offset(int): static memory 上的地址偏移量
- v_idx0(int): vector 寄存器的索引
- v_idx1(int): vector 寄存器的索引

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据大小须为 64bit 的倍数。
- **仅支持 SG2380**。

使用示例

. raw:: latex

newpage

ppl::dma::move_distvv

从 vector 寄存器分发数据到 LMEM 的指令。指令将两组 vector 寄存器的数据均分到所有 Lane 上。

```
template <typename DataType>
void move_distvv(tensor<DataType> &dst, int v_idx0, int v_idx1);
```

参数

- dst(tensor): local memory 上的 dst 张量
- v_idx0(int): vector 寄存器的索引
- v_idx1(int): vector 寄存器的索引

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据大小须为 64bit 的倍数，并且数据元素个数须能被 LANE_NUM 整除。
- **仅支持 SG2380**。

使用示例

ppl::dma::move_vt

从 LMEM/SMEM 存储数据到 vector 寄存器的指令。指令将 LMEM 或 SMEM 中的数据存储到一组 vector 寄存器中。

```
template <typename DataType>
void move_vt(int dst_v_idx, tensor<DataType> &src);

void move_vt(int dst_v_idx, int smem_offset);
```

参数

- dst_v_idx(int): vector 寄存器的索引
- src(tensor): local memory 上的 src 张量
- smem_offset(int): static memory 上的地址偏移量

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据大小须为 64bit 的倍数。
- **仅支持 SG2380**。

使用示例

. raw:: latex

newpage

ppl::dma::move_vcoll

从 LMEM 收集数据到 vector 寄存器的指令。指令从 LMEM 各个 LANE 相同位置收集数据并存储到一组 vector 寄存器上。

```
template <typename DataType>
void move_vcoll(int dst_v_idx, tensor<DataType> &src);
```

参数

- dst_v_idx(int): vector 寄存器的索引
- src(tensor): local memory 上的 src 张量

注意事项

- 必须配合 vset 使用，即在使用 vector 寄存器之前，必须先设置寄存器状态；若 vector 寄存器被多个 api 使用，则仅需第一次使用前设置一次。
- v_idx 必须是 vset 所设置的 lmul 数值的整数倍。
- 该指令搬运的数据大小须为 64bit 的倍数，并且数据元素个数须能被 LANE_NUM 整除。
- **仅支持 SG2380**。

使用示例

4.3.2 内存填充

ppl::dma::fill

将 global memory 中的张量的元素置成常数。

```
template <typename DataType0, typename DataType1>
void fill(gtensor<DataType0> &dst, DataType1 C);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{C}$$

参数

- dst(gtensor): global memory 上的张量
- C: 常数

注意事项

- 如果 stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- stride->w 小于等于 128 / sizeof(DataType)。

使用示例

ppl::dma::fill

将 local memory 中的张量的元素置成常数。

```
template <typename DataType0, typename DataType1>
void fill(tensor<DataType0> &dst, DataType1 C);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{C}$$

参数

- dst(tensor): local memory 上的张量
- C: 常数

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- dst_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

4.3.3 数据广播

ppl::dma::load_broadcast

将张量的元素从 global memory 拷贝到 local memory，并在 C 维度进行广播。

```
template <typename DataType>
void load_broadcast(tensor<DataType> &dst, gtensor<DataType> &src,
                    int num = LANE_NUM);
```

$$dst(n, c, h, w) = src(n, 1, h, w)$$

参数

- dst(gtensor): global memory 上的张量
- src(tensor): local memory 上的张量
- num(int): C 维度广播数量，默认为 LANE_NUM

注意事项

- src 的 shape 是 [shape->n, 1, shape->h, shape->w]。
- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- 如果 dst 从 lane X 开始，X 的取值范围是 [0, LANE_NUM - 1]，且 shape->c 小于等于 LANE_NUM - X。
- dst_stride->w 和 src_stride->w 只能是 1。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void dmaload_bc_fp32(float *ptr_rst, float *ptr_inp) {
  const int N = 1;
  const int C = 1;
  const int H = 16;
  const int W = 1;

  dim4 src_shape = {N, C, H, W};
  dim4 dst_shape = {N, LANE_NUM, H, W};

  auto src_gt = gtensor<float>(src_shape, GLOBAL, ptr_inp);
  auto dst_gt = gtensor<float>(dst_shape, GLOBAL, ptr_rst);

  auto src = tensor<float>(dst_shape);
  dma::load_broadcast(src, src_gt);
  dma::store(dst_gt, src);
}
```

ppl::dma::broadcast

将张量的元素从 local memory 拷贝到 local memory，并在 C 维度进行广播。

```
template <typename DataType>
void broadcast(tensor<DataType> &dst, tensor<DataType> &src,
               int num = LANE_NUM);
```

$$dst(n, c, h, w) = src(n, 1, h, w)$$

参数

- dst(tensor): local memory 上的结果张量
- src(tensor): local memory 上的张量
- num(int): C 维度广播数量，默认为 LANE_NUM

注意事项

- src 的 shape 是 [shape->n, 1, shape->h, shape->w]。
- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 N-byte aligned layout，否则是 free layout。
- 如果 dst 从 lane X 开始，X 的取值范围是 [0, LANE_NUM - 1]，且 shape->c 小于等于 LANE_NUM - X。
- dst_stride->w 和 src_stride->w 只能是 1。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void dmaload_bc_L2L_fp32(float *ptr_rst, float *ptr_inp) {

  const int N = 22;
  const int C = 62;
  const int H = 23;
  const int W = 14;
  dim4 src_shape = {N, 1, H, W};
  dim4 dst_shape = {N, C, H, W};
  auto src = tensor<float>(src_shape);
  auto dst = tensor<float>(dst_shape);

  auto in_gt = gtensor<float>(src_shape, GLOBAL, ptr_inp);
  auto rst_gt = gtensor<float>(dst_shape, GLOBAL, ptr_rst);
  dma::load(src, in_gt);
  dma::broadcast(dst, src);
  dma::store(rst_gt, dst);
}
```

4.3.4 scatter 和 gather 操作

ppl::dma::gather_h

通过 h 维度的索引取值得到输出张量，即 dst = param[index]。

```
template <typename DataType0, typename DataType1, typename DataType2,
        typename DataType3>
void gather_h([tensor|gtensor]<DataType0> &dst,
              [tensor|gtensor]<DataType1> &param,
              [tensor|gtensor]<DataType2> &index, DataType3 C);
```

$$\mathsf{dst}(1,\mathsf{c},\mathsf{h},\mathsf{w}) = \begin{cases} \mathsf{param}(1,\mathsf{c},\mathsf{index}(1,\mathsf{c},\mathsf{h},1),\mathsf{w}) & \text{如果 } \mathsf{index}(1,\mathsf{c},\mathsf{h},1) \text{ 有效} \\ \mathsf{C} & \text{其他情况} \end{cases}$$

参数

- dst(gtensor/tensor): dst 在 global memory 或 local memory 中的张量
- param(gtensor/tensor): param 在 global memory 或 local memory 中的张量
- index(gtensor/tensor): index 在 global memory 或 local memory 中的张量
- C: 常数

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout (dst 数据类型是 tensor) 或 continuous layout (dst 数据类型是 gtensor)，否则是 free layout 。
- 如果 param_stride 是默认值，则 param 是 N-byte aligned layout (param 数据类型是 tensor) 或 continuous layout (param 数据类型是 gtensor)，否则是 free layout。
- 如果 index_stride 是默认值，则 index 是 N-byte aligned layout (index 数据类型是 tensor) 或 continuous layout (index 数据类型是 gtensor)，否则是 free layout。
- 如果 index 的地址能被 512 整除，则性能更优。
- shape->n 只能是 1，param 的 shape 是 [1, shape->c, param_h, shape->w]，index 的 shape 是 [1, shape->c, shape->h, 1]。
- dst_stride->w、param_stride->w 和 index_stride->h 只能是 1。
- index 的元素的数据类型是 DT_UINT32，有效取值范围是 [0, param_h - 1]。

使用示例

```
// dst 数据类型是 tensor, param 数据类型是 gtensor, index 数据类型是 gtensor
#include "ppl.h"

using namespace ppl;
__KERNEL__ void gather_h_l2l_index_local_int16(int16 *ptr_output,
                                               int16 *ptr_param,
                                               uint32 *ptr_index) {
  const int N = 1;
  const int C = 64;
  const int H = 64;
  const int W = 64;
  const int param_h = 128;
  dim4 output_shape = {N, C, H, W};
  dim4 param_shape = {N, C, param_h, W};
  dim4 index_shape = {N, C, H, 1};

  auto output = tensor<int16>(output_shape);
```

(续下页)

(接上页)

```
  auto param = tensor<int16>(param_shape);
  auto index = tensor<uint32>(index_shape);
  auto output_g = gtensor<int16>(output_shape, GLOBAL, ptr_output);
  auto param_g = gtensor<int16>(param_shape, GLOBAL, ptr_param);
  auto index_g = gtensor<uint32>(index_shape, GLOBAL, ptr_index);

  dma::load(param, param_g);
  dma::load(index, index_g);
  dma::gather_h(output, param, index_g, 1);
  dma::store(output_g, output);
}

// dst 数据类型是 gtensor, param 数据类型是 tensor, index 数据类型是 tensor
#include "ppl.h"

using namespace ppl;
MULTI_CORE
__KERNEL__ void gather_h_l2s_index_local_fp16(fp16 *ptr_output,
                                              fp16 *ptr_param,
                                              uint32 *ptr_index) {
  const int N = 1;
  const int C = 64;
  const int H = 64;
  const int W = 64;
  const int param_h = 128;

  int core_num = get_core_num();
  int core_idx = get_core_index();
  if (core_idx >= core_num) {
    return;
  }
  int c_slice = div_up(C, core_num);
  int c_offset = core_idx * c_slice;
  c_slice = max(c_slice, (C - c_slice * (core_num - 1)));

  dim4 output_shape0 = {N, C, H, W};
  dim4 output_shape1 = {N, c_slice, H, W};
  dim4 param_shape0 = {N, C, param_h, W};
  dim4 param_shape1 = {N, c_slice, param_h, W};
  dim4 index_shape0 = {N, C, H, 1};
  dim4 index_shape1 = {N, c_slice, H, 1};
  dim4 offset = {0, c_offset, 0, 0};

  auto output_g = gtensor<fp16>(output_shape0, GLOBAL, ptr_output);
  auto param = tensor<fp16>(param_shape1);
  auto param_g = gtensor<fp16>(param_shape0, GLOBAL, ptr_param);
  auto index = tensor<uint32>(index_shape1);
  auto index_g = gtensor<uint32>(index_shape0, GLOBAL, ptr_index);

  dma::load(param, param_g.sub_view(param_shape1, offset));
  dma::load(index, index_g.sub_view(index_shape1, offset));
  dma::gather_h(output_g.sub_view(output_shape1, offset), param, index, 1);
}
```

ppl::dma::scatter_h

通过 h 维度的索引取值得到输出张量，即 dst = param[index]。

```
template <typename DataType0, typename DataType1, typename DataType2>
void scatter_h([tensor|gtensor]<DataType0> &dst,
               [tensor|gtensor]<DataType1> &param,
               [tensor|gtensor]<DataType2> &index);
```

$$\mathsf{dst}(1, c, \mathsf{index}(1, c, h, 1), w) = \mathsf{param}(1, c, h, w)$$

参数

- dst(gtensor/tensor): dst 在 global memory 或 local memory 中的张量
- param(gtensor/tensor): param 在 global memory 或 local memory 中的张量
- index(gtensor/tensor): index 在 global memory 或 local memory 中的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 N-byte aligned layout (dst 数据类型是 tensor) 或 continuous layout (dst 数据类型是 gtensor)，否则是 free layout。
- 如果 param_stride 是默认值，则 param 是:ref: N-byte aligned layout (param 数据类型是 tensor) 或 continuous layout (param 数据类型是 gtensor)，否则是 free layout。
- 如果 index_stride 是默认值，则 index 是 N-byte aligned layout (index 数据类型是 tensor) 或 continuous layout (index 数据类型是 gtensor)，否则是 free layout。
- 如果 index 的地址能被 512 整除，则性能更优。
- shape->n 只能是 1，param 的 shape 是 [1, shape->c, param_h, shape->w]，index 的 shape 是 [1, shape->c, shape->h, 1]。
- dst_stride->w、param_stride->w 和 index_stride->h 只能是 1。
- index 的元素的数据类型是 DT_UINT32，有效取值范围是 [0, param_h - 1]。

使用示例

```
// dst 数据类型是 gtensor, param 数据类型是 tensor, index 数据类型是 gtensor
#include "ppl.h"

using namespace ppl;
__KERNEL__ void scatter_h_l2s_index_global_int16(int16 *ptr_output,
                                 int16 *ptr_param,
                                 uint32 *ptr_index) {
  const int N = 1;
  const int C = 64;
  const int H = 128;
  const int W = 64;
  const int param_h = 20;
  dim4 output_shape = {N, C, H, W};
  dim4 param_shape = {N, C, param_h, W};
  dim4 index_shape = {N, C, param_h, 1};

  auto output_g = gtensor<int16>(output_shape, GLOBAL, ptr_output);
  auto param = tensor<int16>(param_shape);
  auto param_g = gtensor<int16>(param_shape, GLOBAL, ptr_param);
  auto index_g = gtensor<uint32>(index_shape, GLOBAL, ptr_index);

  dma::load(param, param_g);
  dma::scatter_h(output_g, param, index_g);
}
```

4.3.5 select 操作

ppl::dma::mask_select

将 local memory 中存储的张量按照 mask 筛选后拷贝到 global memory 中。

```
template <typename DataType0, typename DataType1>
unsigned int mask_select(gtensor<DataType0> &dst,
                         [tensor|gtensor]<DataType0> &src,
                         [tensor|gtensor]<DataType1> &mask);
```

参数

- dst(gtensor): dst 在 global memory 上的张量
- src(tensor/gtensor): src 在 local memory 或 global memory 上的张量
- mask(tensor/gtensor): mask 在 global memory 上的张量或在 local memory 中的张量

返回

- 返回值为 dst index 的个数，类型为 DT_UINT32。

注意事项

使用示例

```
// mask 是 local mem 上的张量
#include "ppl.h"
using namespace ppl;

const int N = 16;
const int C = 65;
const int H = 31;
const int W = 32;

__KERNEL__ void MaskSelectL2S_bf16(bf16 *ptr_res, bf16 *ptr_inp,
                                   int *ptr_mask) {
  dim4 inp_shape = {N, C, H, W};
  auto g_inp = gtensor<bf16>(inp_shape, GLOBAL, ptr_inp);
  auto g_res = gtensor<bf16>(inp_shape, GLOBAL, ptr_res);
  auto g_mask = gtensor<int>(inp_shape, GLOBAL, ptr_mask);
  auto inp = tensor<bf16>(inp_shape);
  dma::load(inp, g_inp);
  auto mask = tensor<int>(inp_shape);
  dma::load(mask, g_mask);
  dma::mask(g_mask, inp, mask);
}

// mask 是 global mem 上的张量
#include "ppl.h"
using namespace ppl;

const int N = 16;
const int C = 64;
const int H = 32;
const int W = 32;

__KERNEL__ void MaskSelectL2S_bf16(bf16 *ptr_res, bf16 *ptr_inp,
                                   int *ptr_mask) {
  dim4 inp_shape = {N, C, H, W};
  auto g_inp = gtensor<bf16>(inp_shape, GLOBAL, ptr_inp);
  auto g_res = gtensor<bf16>(inp_shape, GLOBAL, ptr_res);
  auto g_mask = gtensor<int>(inp_shape, GLOBAL, ptr_mask);
```

(续下页)

(接上页)

```
  auto inp = tensor<bf16>(inp_shape);
  dma::load(inp, g_inp);

  dma::mask(g_res, inp, g_mask);
}
```

ppl::dma::mask_batch_bcast

通过 w 维度的蒙版取值得到输出张量，即 output = src[mask]，并统计蒙版中的非零值数量，src 的 batch 被广播。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void mask_batch_bcast(tensor<DataType0> &dst, tensor<DataType1> &count,
                      tensor<DataType2> &src, tensor<DataType3> &mask,
                      bool is_repeat);
```

对于固定的 n 和 c 有 R 个非零值: $\mathsf{mask}(n,c,0,w_0), \mathsf{mask}(n,c,0,w_1), \cdots, \mathsf{mask}(n,c,0,w_{R-1})$

$$\mathsf{output}(n,c,0,r) = \begin{cases} \mathsf{src}(0,c \mod \mathsf{LANE\_NUM},0,w_r) & \text{如果 src 重复} \\ \mathsf{src}(0,c,0,w_r) & \text{其他情况} \end{cases}$$

$$\mathsf{count}(n,c,0,0) = R$$

参数

- dst(tensor): local memory 上的 dst 张量
- count(tensor): local memory 上的 count 张量
- src(tensor): local memory 上的 src 张量
- mask(tensor): local memory 上的 mask 张量
- is_repeat(bool): src 重复的标志

注意事项

- dst、count、src 和 mask 从同一个 LANE 开始，output、src 和 mask 是 N-byte aligned layout，count 是 compact layout。
- shape->h 只能是 1，count 的 shape 是 [shape->n, shape->c, 1, 1]，src 的 shape 是 [1, shape->c, 1, shape->w]。
- shape->n、shape->c、shape->w 的取值范围是 [1, 65535]。
- DataType3 的有效取值是 DT_UINT32、DT_UINT16 和 DT_UINT8，count 的元素的数据类型是 UINT16，输出范围是 [0, shape->w]。
- 如果 output、count、src 和 mask 两两之间没有 bank conflicting，则性能更优，如果 is_repeat 是 true，则判断是否冲突时，src 的 size 按照 shape 是 [1, 1, 1, shape->w] 计算。

使用示例

```
#include "ppl.h"
using namespace ppl;
MULTI_CORE
__KERNEL__ void
batch_bcast_mask_select_multi_core(bf16 *ptr_out,
                                   bf16 *ptr_inp /*, uint16 *ptr_count*/,
                                   unsigned int *ptr_mask) {
  const int N = 2;
  const int C = 1;
  const int H = 1;
  const int W = 8;
  const int flag = 1;

  int num = get_core_num();
```

(续下页)

(接上页)

```
  int index = get_core_index();

  if (index >= num)
    return;
  int slice = div_up(C, num);
  int index_offset = index * slice;
  slice = MAX(slice, (C - slice * (num - 1)));
  dim4 inp_global_shape = {1, C, H, W};
  dim4 inp_shape = {1, slice, H, W};
  dim4 inp_offset = {0, index_offset, 0, 0};
  tensor<bf16> inp;
  tensor<uint32> mask;
  tensor<uint16> count;


  auto inp_gt = gtensor<bf16>(inp_global_shape, GLOBAL, ptr_inp);
  dma::load(inp, inp_gt.sub_view(inp_shape, inp_offset));

  dim4 out_global_shape = {N, C, H, W};
  dim4 out_shape = {N, slice, H, W};
  auto mask_gt = gtensor<uint32>(out_global_shape, GLOBAL, ptr_mask);
  /*ppl::zeros(mask, &out_shape);*/
  dma::load(mask, mask_gt.sub_view(out_shape, inp_offset));
  auto out = tensor<bf16>(out_shape);
  tiu::fill(out, 0);

  dim4 count_shape = {N, slice, 1, 1};
  dim4 count_stride = {C, 1, 1, 1};
  dim4 count_offset = {0, index_offset, 0, 0};

  auto out_gt = gtensor<bf16>(out_global_shape, GLOBAL, ptr_out);
  dma::mask_batch_bcast(out, count, inp, mask, false);
  dma::store(out_gt.sub_view(out_shape, inp_offset), out);
  /*ppl::dmastore(ptr_count, count, &count_shape, &count_stride,
   * &count_offset);*/
}
```

ppl::dma::nonzero

将 local memory 的输入张量中不为 0 的元素的 index 输出到 global memory 中。

```
template <typename DataType>
uint nonzero(gtensor<DataType> &dst, tensor<DataType> &src);
```

参数

- dst(gtensor): global memory 中的张量
- src(tensor): local memory 中的张量

返回

- 返回值为 dst index 的个数，类型为 DT_UINT32。

注意事项

- dst 是 N-byte aligned layout, src 是 compact layout。
- dst 数据类型的有效取值是 DT_INT32、DT_INT16、DT_INT8。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void dma_nonzero_l2s_int8(int *ptr_dst, int8 *ptr_src) {
  int N = 1;
  int C = 1;
  int H = 11;
  int W = 8;
  dim4 shape = {N, C, H, W};
  dim4 offset = {0, 0, 0, 0};
  auto src = tensor<int8>(shape);

  unsigned int base_idx = 0;
  auto src_gt = gtensor<int8>(shape, GLOBAL, ptr_src);
  auto dst_gt = gtensor<int>(shape, GLOBAL, ptr_dst);
  dma::load(src, src_gt);
  unsigned int idx_num = dma::nonzero(dst_gt, src, base_idx);
}
```

ppl::dma::nonzero

将 global memory 的输入张量中不为 0 的元素的 index 输出到 global memory 中。

```
template <typename DataType0, typename DataType1>
uint nonzero(gtensor<DataType0> &dst, gtensor<DataType1> &src);
```

参数

- dst(gtensor): global memory 上的张量
- src(gtensor): global memory 上的张量

返回

- 返回值为 dst index 的个数，类型为 DT_UINT32。

注意事项

- dst 是 compact layout, src 是 compact layout。
- DataType0 数据类型的有效取值是 DT_INT32、DT_INT16、DT_INT8。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void dma_nonzero_s2s_int16(int *ptr_dst, int16 *ptr_src) {
  int N = 1;
  int C = 1;
  int H = 11;
  int W = 8;
  dim4 shape = {N, C, H, W};

  unsigned int base_idx = 0;
  auto src_gt = gtensor<int16>(shape, GLOBAL, ptr_src);
  auto dst_gt = gtensor<int>(shape, GLOBAL, ptr_dst);
  unsigned int idx_num = dma::nonzero(dst_gt, src_gt, base_idx);
}
```

4.4 SDMA 操作

4.4.1 数据搬运

ppl::sdma::move

将张量的元素在 global memory 与 l2 memory 之间搬运。

```
template <typename DataType>
void move(gtensor<DataType> &dst, gtensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(gtensor): l2 memory 或 global memory 上的张量
- src(gtensor): l2 memory 或 global memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

ppl::sdma::transpose_cw

将张量的元素在 global memory 与 l2 memory 之间拷贝，并进行 C 和 W 的维度转置。

```
template <typename DataType>
void transpose_cw(gtensor<DataType> &dst, gtensor<DataType> *src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{w},\mathsf{h},\mathsf{c})$$

参数

- dst(gtensor): l2 memory 或 global memory 上的张量
- src(gtensor): l2 memory 或 global memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

略

ppl::sdma::transpose_cw

将张量的元素在 global memory 与 l2 memory 之间拷贝，并进行 C 和 N 的维度转置。

```
template <typename DataType>
void transpose_nc(gtensor<DataType> &dst, gtensor<DataType> *src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{c},\mathsf{n},\mathsf{h},\mathsf{w})$$

参数

- dst(gtensor): l2 memory 或 global memory 上的张量
- src(gtensor): l2 memory 或 global memory 上的张量

注意事项

- 如果 dst_stride 是默认值，则 dst 是 continuous layout，否则是 free layout。
- 如果 src_stride 是默认值，则 src 是 continuous layout，否则是 free layout。
- dst_stride->w 和 src_stride->w 小于等于 128 / sizeof(DataType)。

使用示例

略

4.5 HAU 操作

4.5.1 Topk

ppl::hau::topk

按升序或降序排序前 K 个最小或最大的数。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<DataType> &src, bool is2D, int K,
          bool descended);
```

$$dst(k) = src(i_k)$$

如果升序，则

$$src(i_0) \le src(i_1) \le \cdots \le src(i_{K-1}) \le \cdots \le src(i_{len-1})$$

如果降序，则

$$src(i_0) \ge src(i_1) \ge \cdots \ge src(i_{K-1}) \ge \cdots \ge src(i_{len-1})$$

其中，$i_0, i_1, \ldots, i_{len-1}$ 互不相同，是 $0, 1, \ldots, len-1$ 的重排。

参数

- dst(gtensor): global memory 上的 dst 张量
- src(gtensor): global memory 上的 src 张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst 和 src 的地址都被 4 整除。
- src 的 shape 的 W 维度代表了输入的长度 len，推荐将 src reshape 为 [1,1,1,len] 的形式。
- dst 和 src 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- DataType 的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32。
- is2D 决定了是否将 dst 和 src 视为二维数据，此处必须为 false。
- src 的长度是 len，dst 的地址的前 K 个数是排序后的结果，K 小于等于 len。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void topk_int32(int32 *ptr_dst, int32 *ptr_src) {
  const int len = 320;
  const int K = 16;
  const bool descended = true;
  dim4 dst_shape = {1, 1, 1, K};
  dim4 src_shape = {1, 1, 1, len};
  auto gtensor_dst = gtensor<int32>(dst_shape, GLOBAL, ptr_dst);
  auto gtensor_src = gtensor<int32>(src_shape, GLOBAL, ptr_src);
  hau::topk(gtensor_dst, gtensor_src, false, K, descended);
}
```

ppl::hau::topk

2 维数组排序，每一行按升序或降序排序前 K 个最小或最大的数。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<DataType> &src, bool is2D, int K,
          bool descended);
```

$$dst(row, k) = src(row, i_k)$$

如果升序每一行数据按照如下规则计算：

$$src(i_0) \le src(i_1) \le \cdots \le src(i_{K-1}) \le \cdots \le src(i_{len-1})$$

如果降序，每一行数据则：

$$src(i_0) \ge src(i_1) \ge \cdots \ge src(i_{K-1}) \ge \cdots \ge src(i_{len-1})$$

其中，$i_0, i_1, \ldots, i_{len-1}$ 互不相同，是 $0, 1, \ldots, len-1$ 的重排。

参数

- dst(gtensor): global memory 上的 dst 张量
- src(gtensor): global memory 上的 src 张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst 和 src 的地址都被 4 整除。
- src 的 shape 的 N/C/H 维度的乘积代表了输入的行数 row，W 维度代表了每行的长度 len，推荐将 src reshape 为 [1,1,row,len] 的形式。
- dst 和 src 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- DataType 的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32。
- is2D 决定了是否将 dst 和 src 视为二维数据，此处必须为 true。
- dst 的地址的前 row * K 个数是排序后的结果，K 小于等于 len。
- 仅支持 **BM1690**。

使用示例

```
#include "ppl.h"
using namespace ppl;

MULTI_CORE
__KERNEL__ void topk_int32(int32 *ptr_dst, int32 *ptr_src) {
  const int row = 66;
  const int len = 320;
  const int K = 16;
  dim4 dst_shape = {1, 1, row, K};
  dim4 src_shape = {1, 1, row, len};
  auto gt_dst = gtensor<int32>(dst_shape, GLOBAL, ptr_dst);
  auto gt_src = gtensor<int32>(src_shape, GLOBAL, ptr_src);
```

(续下页)

(接上页)

```
  int core_num = ppl::get_core_num();
  int core_idx = ppl::get_core_index();
  if (core_idx >= core_num) {
    return;
  }

  int row_slice = div_up(row, core_num);
  int row_offset = core_idx * row_slice;
  int real_row_slice = min(row_slice, (row - row_offset));

  const bool descended = true;
  dim4 split_dst_shape = {1, 1, real_row_slice, K};
  dim4 split_src_shape = {1, 1, real_row_slice, len};
  dim4 split_offset = {0, 0, row_offset, 0};

  hau::topk(gt_dst.sub_view(split_dst_shape, split_offset),
            gt_src.sub_view(split_src_shape, split_offset),
            true, K, descended);
}
```

ppl::hau::topk

按升序或降序稳定排序前 K 个最小或最大的数，并输出排序后的索引，排序前的索引是自然索引。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<int> &dst_idx, gtensor<DataType> &src,
          bool is2D, int K, bool descended);
```

$$dst\_data(k) = src(i_k) \quad dst\_idx(k) = i_k$$

如果 $src(i_k) = src(i_{k+1})$，则 $i_k < i_{k+1}$

如果升序，则

$$src(i_0) \le src(i_1) \le \cdots \le src(i_{K-1}) \le \cdots \le src(i_{len-1})$$

如果降序，则

$$src(i_0) \ge src(i_1) \ge \cdots \ge src(i_{K-1}) \ge \cdots \ge src(i_{len-1})$$

其中，$i_0, i_1, \ldots, i_{len-1}$ 互不相同，是 $0, 1, \ldots, len-1$ 的重排。

参数

- dst_data(gtensor): global memory 上的 dst 数据张量
- dst_idx(gtensor): global memory 上的 dst 索引张量
- src_data(gtensor): global memory 上的 src 数据张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst_data、dst_idx 和 src_data 都被 4 整除。
- src 的 shape 的 W 维度代表了输入的长度 len，推荐将 src reshape 为 [1,1,1,len] 的形式。
- dst、dst_idx 和 src 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- dst 和 dst_idx 相关，当对 dst 使用 view 或者 sub_view 进行调整时，注意 dst_idx 要保持一致。
- dst 和 src 的元素的数据类型的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32，dst_idx 的元素的数据类型的有效值是 DT_INT32。
- is2D 决定了是否将 dst 和 src 视为二维数据，此处必须为 false。
- src 的长度是 len，dst 的地址的前 K 个数是排序后的结果，dst_idx 的地址的前 K 个数是对应 dst 中前 K 个数的自然索引，K 小于等于 len。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void topK_f32(float *ptr_res, int *ptr_idx,
                         float *ptr_inp, const int K) {
  const int len = 64;
  const bool descended = false;
  dim4 res_shape = {1, 1, 1, 200};
```

(续下页)

(接上页)

```
  dim4 inp_shape = {1, 1, 1, 128};
  auto gt_res = gtensor<float>(res_shape, GLOBAL, ptr_res);
  auto gt_idx = gtensor<int>(res_shape, GLOBAL, ptr_idx);
  auto gt_inp = gtensor<float>(inp_shape, GLOBAL, ptr_inp);

  dim4 real_inp_shape = {1, 1, 1, len};
  hau::topk(gt_res, gt_idx, gt_inp.view(real_inp_shape),
            false, K, descended);
}
```

ppl::hau::topk

2 维数组排序，每一行按升序或降序稳定排序前 K 个最小或最大的数，并输出排序后的索引，排序前的索引是自然索引。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<int> &dst_idx, gtensor<DataType> &src,
          bool is2D, int K, bool descended);
```

$$\text{dst\_data}(row, k) = \text{src}(row, i_k) \quad \text{dst\_idx}(row, k) = i_k$$

$$\text{如果 } \text{src}(row, i_k) = \text{src}(row, i_{k+1}), \text{ 则 } i_k < i_{k+1}$$

如果升序，则每一行

$$\text{src}(i_0) \le \text{src}(i_1) \le \cdots \le \text{src}(i_{K-1}) \le \cdots \le \text{src}(i_{len-1})$$

如果降序，则每一行

$$\text{src}(i_0) \ge \text{src}(i_1) \ge \cdots \ge \text{src}(i_{K-1}) \ge \cdots \ge \text{src}(i_{len-1})$$

其中，$i_0, i_1, \ldots, i_{len-1}$ 互不相同，是 $0, 1, \ldots, len-1$ 的重排。

参数

- dst_data(gtensor): global memory 上的 dst 数据张量
- dst_idx(gtensor): global memory 上的 dst 索引张量
- src_data(gtensor): global memory 上的 src 数据张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst_data、dst_idx 和 src_data 都被 4 整除。
- src 的 shape 的 N/C/H 维度的乘积代表了输入的行数 row，W 维度代表了每行的长度 len，推荐将 src reshape 为 [1,1,row,len] 的形式。
- dst、dst_idx 和 src 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- dst 和 dst_idx 相关，当对 dst 使用 view 或者 sub_view 进行调整时，注意 dst_idx 要保持一致。
- dst 和 src 的元素的数据类型的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32，dst_idx 的元素的数据类型的有效值是 DT_INT32。
- is2D 决定了是否将 dst 和 src 视为二维数据，此处必须为 true。
- dst 的地址的前 row * K 个数是排序后的结果，dst_idx 的地址的前 row * K 个数是相应的自然索引，K 小于等于 len。
- 仅支持 **BM1690**。

使用示例

```
#include "ppl.h"
using namespace ppl;

MULTI_CORE
__KERNEL__ void topk_f32(float *ptr_value, int *ptr_idx,
                         float *ptr_data, const int K) {
  const int row = 80;
  int core_num = ppl::tpu_core_num();
  int core_idx = ppl::tpu_core_index();
  if (core_idx >= core_num) {
    return;
  }

  int row_slice = div_up(row, core_num);
  int row_offset = core_idx * row_slice;
  int real_row_slice = min(row_slice, (row - row_offset));
  const int len = 320;
  const bool descended = false;

  dim4 res_shape = {1, 1, 80, 80};
  dim4 inp_shape = {1, 1, 80, 320};
  auto gt_res = gtensor<float>(res_shape, GLOBAL, ptr_value);
  auto gt_idx = gtensor<int>(res_shape, GLOBAL, ptr_idx);
  auto gt_inp = gtensor<float>(inp_shape, GLOBAL, ptr_data);

  dim4 real_inp_shape = {1, 1, real_row_slice, len};
  dim4 offset = {0, 0, row_offset, 0};
  dim4 real_res_shape = {1, 1, real_row_slice, K};
  hau::topk(gt_res.sub_view(real_res_shape, offset),
            gt_idx.sub_view(real_res_shape, offset),
            gt_inp.sub_view(real_inp_shape, offset),
            true, K, descended);
}
```

ppl::hau::topk

按升序或降序稳定排序前 K 个最小或最大的数，并输出排序后的索引，排序前的索引是指定索引。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<int> &dst_idx, gtensor<DataType> &src,
          gtensor<int> &src_idx, bool is2D, int K, bool descended);
```

$$\text{dst\_data}(k) = \text{src\_data}(i_k) \quad \text{dst\_idx}(k) = \text{src\_idx}(i_k)$$

如果 $\text{src\_data}(i_k) = \text{src\_data}(i_{k+1})$，则 $\text{src\_idx}(i_k) \le \text{src\_idx}(i_{k+1})$

如果升序，则

$$\text{src\_data}(i_0) \le \text{src\_data}(i_1) \le \cdots \le \text{src\_data}(i_{K-1}) \le \cdots \le \text{src\_data}(i_{len-1})$$

如果降序，则

$$\text{src\_data}(i_0) \ge \text{src\_data}(i_1) \ge \cdots \ge \text{src\_data}(i_{K-1}) \ge \cdots \ge \text{src\_data}(i_{len-1})$$

其中，$i_0, i_1, \ldots, i_{len-1}$ 互不相同，是 $0, 1, \ldots, len-1$ 的重排。

参数

- dst_data(gtensor): global memory 上的 dst 数据张量
- dst_idx(gtensor): global memory 上的 dst 索引张量
- src_data(gtensor): global memory 上的 src 数据张量
- src_idx(gtensor): global memory 上的 src 索引张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst_data、dst_idx、src_data 和 src_idx 都被 4 整除。
- src 的 shape 的 W 维度代表了输入的长度 len，推荐将 src reshape 为 [1,1,1,len] 的形式。
- dst、dst_idx 和 src、src_idx 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- dst 和 dst_idx 相关，当对 dst 使用 view 或者 sub_view 进行调整时，注意 dst_idx 要保持一致。src 和 src_idx 同理。
- dst 和 src 的元素的数据类型的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32，dst_idx 和 src_idx 的元素的数据类型的有效值是 DT_INT32。
- dst_data 和 dst_idx 的长度是 len，前 K 个数是排序后的结果和对应的索引，K 小于等于 len。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void topK_f32(float *ptr_res, int *ptr_res_idx,
                         float *ptr_inp, int *ptr_inp_idx,
                         const int K) {
  const int len = 64;
  const bool descended = false;
```

(续下页)

(接上页)

```
  dim4 res_shape = {1, 1, 1, 200};
  dim4 inp_shape = {1, 1, 1, 128};
  auto gt_res = gtensor<float>(res_shape, GLOBAL, ptr_res);
  auto gt_res_idx = gtensor<int>(res_shape, GLOBAL, ptr_res_idx);
  auto gt_inp = gtensor<float>(inp_shape, GLOBAL, ptr_inp);
  auto gt_inp_idx = gtensor<int>(res_shape, GLOBAL, ptr_inp_idx);

  dim4 real_inp_shape = {1, 1, 1, len};
  hau::topk(gt_res, gt_res_idx, gt_inp.view(real_inp_shape),
          gt_inp_idx.view(real_inp_shape), true, K, descended);
}
```

ppl::hau::topk

2 维数组排序，每行按升序或降序稳定排序前 K 个最小或最大的数，并输出排序后的索引，排序前的索引是指定索引。

```
template <typename DataType>
void topk(gtensor<DataType> &dst, gtensor<int> &dst_idx, gtensor<DataType> &src,
          gtensor<int> &src_idx, bool is2D, int K, bool descended);
```

$$\text{dst\_data}(\text{row}, k) = \text{src\_data}(\text{row}, i_k) \quad \text{dst\_idx}(\text{row}, k) = \text{src\_idx}(i_k)$$

如果 $\text{src\_data}(\text{row}, i_k) = \text{src\_data}(\text{row}, i_{k+1})$，则 $\text{src\_idx}(\text{row}, i_k) \leq \text{src\_idx}(\text{row}, i_{k+1})$

如果升序，则每行

$$\text{src\_data}(i_0) \leq \text{src\_data}(i_1) \leq \cdots \leq \text{src\_data}(i_{K-1}) \leq \cdots \leq \text{src\_data}(i_{\text{len}-1})$$

如果降序，则每行

$$\text{src\_data}(i_0) \geq \text{src\_data}(i_1) \geq \cdots \geq \text{src\_data}(i_{K-1}) \geq \cdots \geq \text{src\_data}(i_{\text{len}-1})$$

其中，$i_0, i_1, \ldots, i_{\text{len}-1}$ 互不相同，是 $0, 1, \ldots, \text{len} - 1$ 的重排。

参数

- dst_data(gtensor): global memory 上的 dst 数据张量
- dst_idx(gtensor): global memory 上的 dst 索引张量
- src_data(gtensor): global memory 上的 src 数据张量
- src_idx(gtensor): global memory 上的 src 索引张量
- is2D(bool): 区分 dst 和 src 是一维或二维的标志
- K(int): 排序长度
- descended(bool): 降序的标志

注意事项

- dst_data、dst_idx、src_data 和 src_idx 都被 4 整除。
- src 的 shape 的 N/C/H 维度的乘积代表了输入的行数 row，W 维度代表了每行的长度 len，推荐将 src reshape 为 [1,1,row,len] 的形式。
- dst、dst_idx 和 src、src_idx 的地址偏移可通过 sub_view 进行调整：new_dst = dst.sub_view(new_shape, offset)。
- dst 和 dst_idx 相关，当对 dst 使用 view 或者 sub_view 进行调整时，注意 dst_idx 要保持一致。src 和 src_idx 同理。
- dst 和 src 的元素的数据类型的有效取值是 DT_FP32、DT_INT32 和 DT_UINT32，dst_idx 和 src_idx 的元素的数据类型的有效值是 DT_INT32。
- is2D 决定了是否将 dst 和 src 视为二维数据，此处必须为 true。
- dst 的地址的前 row * K 个数是排序后的结果，dst_idx 的地址的前 row * K 个数是相应的指定索引，K 小于等于 len。
- 仅支持 **BM1690**。

使用示例

```
#include "ppl.h"
using namespace ppl;

MULTI_CORE
__KERNEL__ void topk_f32(float *ptr_value, int *ptr_idx,
                         float *ptr_data, int *ptr_src_idx,
                         const int K) {
  const int row = 80;
  int core_num = ppl::tpu_core_num();
  int core_idx = ppl::tpu_core_index();
  if (core_idx >= core_num) {
    return;
  }

  int row_slice = div_up(row, core_num);
  int row_offset = core_idx * row_slice;
  int real_row_slice = min(row_slice, (row - row_offset));
  const int len = 320;
  const bool descended = false;

  dim4 res_shape = {1, 1, 80, 80};
  dim4 inp_shape = {1, 1, 80, 320};
  auto gt_res = gtensor<float>(res_shape, GLOBAL, ptr_value);
  auto gt_res_idx = gtensor<int>(res_shape, GLOBAL, ptr_idx);
  auto gt_inp = gtensor<float>(inp_shape, GLOBAL, ptr_data);
  auto gt_inp_idx = gtensor<int>(inp_shape, GLOBAL, ptr_src_idx);

  dim4 real_inp_shape = {1, 1, real_row_slice, len};
  dim4 offset = {0, 0, row_offset, 0};
  dim4 real_res_shape = {1, 1, real_row_slice, K};
  hau::topk(gt_res.sub_view(real_res_shape, offset),
            gt_res_idx.sub_view(real_res_shape, offset),
            gt_inp.sub_view(real_inp_shape, offset),
            gt_inp_idx.sub_view(real_inp_shape, offset),
            true, K, descended);
}
```

4.5.2 Gather

ppl::hau::gather_line

通过 line 的索引取值得到输出张量，即 dst = param[index]。

```
template <typename DataType0, typename DataType1, typename DataType2>
void gather_line(gtensor<DataType0> &dst, gtensor<DataType0> &param,
                 gtensor<DataType1> &index, DataType2 C, int start, int end,
                 bool fill_const);
```

$$\mathrm{dst}(h,w) = \begin{cases} \mathrm{param}(\mathrm{index}(h) - \mathrm{start}, w) & \text{如果 index(h) 是有效索引} \\ C & \text{如果 index(h) 是无效索引，fill\_const 是 true} \end{cases}$$

参数

- dst(gtensor): global memory 上的 dst 数据张量
- param(gtensor): global memory 上的 param 张量
- index(gtensor): global memory 上的 index 张量
- C: 常数
- start(int): 有效索引的起始值
- end(int): 有效索引的结束值
- fill_const(bool): dst 在无效索引处填 C 的标志

注意事项

- dst、param 和 index 的起始地址都被 64 整除。
- dst 的 shape 是 [1, 1, index_len, line_len]，param 的 shape 是 [1, 1, line_num, line_len]，index 的 shape 是 [1, 1, 1, index_len]，都是 continuous layout。
- index 的元素的数据类型是 UINT32，有效索引的范围是 [start, end)，start 大于等于 0，start 小于等于 end，end 小于 line_num。
- 如果索引无效，fill_const 是 false，则 dst 的对应元素不会被填。
- 仅支持 **BM1684X**，与 h_gather 可以达到同等效果。

使用示例

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void hau_line_gather_bf16(bf16 *ptr_output, bf16 *ptr_param,
                                     uint32 *ptr_index) {
  const int line_num = 128;
  const int line_len = 64;
  const int index_len = 32;
  const int start = 0;
  const int end = 64;

  dim4 output_shape = {1, 1, index_len, line_len};
  auto gtensor_output = gtensor<bf16>(output_shape, GLOBAL, ptr_output);
  dim4 param_shape = {1, 1, line_num, line_len};
  auto gtensor_input = gtensor<bf16>(param_shape, GLOBAL, ptr_param);
  dim4 index_shape = {1, 1, 1, index_len};
  auto gtensor_index = gtensor<uint32>(index_shape, GLOBAL, ptr_index);
```

(续下页)

(接上页)

```
  hau::gather_line(gtensor_output, gtensor_input, gtensor_index, 1, start, end, true);
}
```

4.6 TIU 运算

4.6.1 Pointwise

ppl::tiu::abs

对张量的元素取绝对值。

```
template <typename DataType>
void abs(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = |src(n, c, h, w)|$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。

使用示例

```
#include "ppl.h"

using namespace ppl;

__KERNEL__ void abs_bf16(bf16 *ptr_res, bf16 *ptr_inp) {
  const int N = 1;
  const int C = 64;
  const int H = 32;
  const int W = 16;
  dim4 inp_shape = {N, C, H, W};
  auto g_inp = gtensor<bf16>(inp_shape, GLOBAL, ptr_inp);
  auto g_res = gtensor<bf16>(inp_shape, GLOBAL, ptr_res);
  auto inp = tensor<bf16>(inp_shape);
  auto res = tensor<bf16>(inp_shape);

  dma::load(inp, g_inp);
  tiu::abs(res, inp);
  dma::store(g_res, res);
}
```

ppl::tiu::fabs

对张量的元素取绝对值。

```
template <typename DataType>
void fabs(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = |src(n, c, h, w)|$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。

使用示例

具体使用方式与abs 一致。

ppl::tiu::add

1. 两个张量的元素相加，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void add(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, char shift, int round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src0}(n,c,h,w) + \mathrm{src1}(n,c,h,w)) \ \textbf{左移} \ \mathrm{shift} & \text{如果 } \mathrm{shift} > 0 \\ (\mathrm{src0}(n,c,h,w) + \mathrm{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathrm{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

2. 张量的元素和常数相加，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void add(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C, char shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) + C) \ \textbf{左移} \ \mathrm{shift} & \text{如果 } \mathrm{shift} > 0 \\ (\mathrm{src}(n,c,h,w) + C) \ \textbf{右移} \ -\mathrm{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- shift 的取值范围是 [-32, 31]。
- DataType0、DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。
- DataType0 是无符号的当且仅当 DataType1 和 DataType2 都是无符号的。

- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

3. 两个张量的元素相加，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1>
void add(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, tensor<int8> &shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src0}(n,c,h,w) + \mathsf{src1}(n,c,h,w)) \ \textbf{左移} \ \mathsf{shift}(1,c,1,1) & \text{如果 } \mathsf{shift}(1,c,1,1) > 0 \\ (\mathsf{src0}(n,c,h,w) + \mathsf{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathsf{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

4. 张量的元素和常数相加，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void add(tensor<DataType0> &dst, tensor<DataType1> &src, tensor<int8> shift,
         DataType2 C, rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src}(n,c,h,w) + C) \ \textbf{左移} \ \mathsf{shift}(1,c,1,1) & \text{如果 } \mathsf{shift}(1,c,1,1) > 0 \\ (\mathsf{src}(n,c,h,w) + C) \ \textbf{右移} \ -\mathsf{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- shift(tensor): shift 张量
- C: 常数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0、src1 和 shift 从同一个 lane 开始。
- dst、src0、src1 和 shift 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- shift 的 shape 是 [1, c, 1, 1]，compact layout，取值范围是 [-32, 31]。

- DataType0、DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。
- DataType0 是无符号的当且仅当 DataType1 和 DataType2 都是无符号的。

使用示例

```
// 两个张量的元素相加
#include "ppl.h"

using namespace ppl;

const int N = 1;
const int C = 64;
const int H = 16;
const int W = 1;
__KERNEL__ void arith_int(int16 *rst_and, int *rst_sft, int16 *inp_1,
                          int16 *inp_2, int *inp, int *sft) {
  dim4 shape = {N, C, H, W};
  auto inp_gtensor_1 = gtensor<int16>(shape, GLOBAL, inp_1);
  auto inp_gtensor_2 = gtensor<int16>(shape, GLOBAL, inp_2);
  auto inp_gtensor = gtensor<int>(shape, GLOBAL, inp);
  auto sft_gtensor = gtensor<int>(shape, GLOBAL, sft);
  auto rst_and_gtensor = gtensor<int16>(shape, GLOBAL, rst_and);
  auto rst_sft_gtensor = gtensor<int>(shape, GLOBAL, rst_sft);

  auto inp_1_l = tensor<int16>(shape);
  auto inp_2_l = tensor<int16>(shape);
  auto rst_and_l = tensor<int16>(shape);

  auto inp_l = tensor<int>(shape);
  auto sft_l = tensor<int>(shape);
  auto rst_sft_l = tensor<int>(shape);

  auto rst_sub_l = tensor<int16>(shape);

  dma::load(inp_1_l, inp_gtensor_1);
  dma::load(inp_2_l, inp_gtensor_2);
  dma::load(inp_l, inp_gtensor);
  dma::load(sft_l, sft_gtensor);

  // tiu::bitwise_and(rst_and_l, inp_1_l, inp_2_l);
  tiu::add(rst_and_l, inp_1_l, inp_1_l, 0, RM_DOWN, true);
  tiu::add(rst_and_l, rst_and_l, 1, 0, RM_DOWN, true);
  tiu::sub(rst_and_l, rst_and_l, inp_2_l, 0, RM_DOWN, true);
  tiu::sub(rst_and_l, 1, rst_and_l, 0, RM_DOWN, true);
  tiu::sub(rst_and_l, rst_and_l, 1, 0, RM_DOWN, true);
  tiu::mul(rst_and_l, rst_and_l, 2, 0, RM_DOWN, true);
  tiu::mul(rst_and_l, rst_and_l, inp_2_l, 0, RM_DOWN, true);
  tiu::shift(rst_sft_l, inp_l, sft_l, RM_HALF_TO_EVEN);
  dma::store(rst_and_gtensor, rst_and_l);

  dma::store(rst_sft_gtensor, rst_sft_l);
}
```

ppl::tiu::fadd

1. 两个张量的元素相加。

```
template <typename DataType>
void fadd(tensor<DataType> &dst, tensor<DataType> &src0,
          tensor<DataType> &src1);
```

$$dst(n, c, h, w) = src0(n, c, h, w) + src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素和常数相加。

```
template <typename DataType>
void fadd(tensor<DataType> &dst, tensor<DataType> &src, float C);
```

$$dst(n, c, h, w) = src(n, c, h, w) + C$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16、DT_BFP16。
- 支持在 N/H/W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

```
#include "ppl.h"
using namespace ppl;

MULTI_CORE
__KERNEL__ void add_c_fp16(fp16 *ptr_res, fp16 *ptr_inp) {
  const int N = 1;
  const int C = 64;
  const int H = 32;
  const int W = 16;
  int core_num = ppl::get_core_num();
  int core_idx = ppl::get_core_index();
```

(续下页)

(接上页)

```
  if (core_idx >= core_num) {
    return;
  }

  int slice = div_up(C, core_num);
  int c_offset = core_idx * slice;
  int real_slice = min(slice, C - c_offset);
  dim4 res_max_shape = {N, slice, H, W};
  dim4 shape = {N, real_slice, H, W};
  dim4 stride = {C * H * W, H * W, W, 1};
  dim4 offset = {0, c_offset, 0, 0};
  auto inp = make_tensor<fp16>(res_max_shape, shape);
  auto res = make_tensor<fp16>(res_max_shape, shape);
  float scalar_c = 0.25;

  dma::load(inp, ptr_inp, &offset, &stride);
  tiu::fadd(res, inp, scalar_c);
  dma::store(ptr_res, res, &offset, &stride);
}
```

ppl::tiu::bitwise_not

对张量的元素按位取反。

```
template <typename DataType>
void bitwise_not(tensor<DataType> &dst, tensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathbf{NOT}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}))$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的取值范围是 DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。

使用示例

具体使用方式与abs 一致。

ppl::tiu::bitwise_and

1. 两个张量的元素做按位与运算。

```
template <typename DataType>
void bitwise_and(tensor<DataType> &dst, tensor<DataType> &src0,
                 tensor<DataType> &src1);
```

$$dst(n, c, h, w) = src0(n, c, h, w) \quad \textbf{AND} \quad src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做按位与运算。

```
template <typename DataType>
void bitwise_and(tensor<DataType> &dst, tensor<DataType> &src, DataType C);
```

$$dst(n, c, h, w) = src(n, c, h, w) \quad \textbf{AND} \quad C$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

```
#include "ppl.h"

using namespace ppl;

__KERNEL__ void bitwise_and_c_int8(int8 *ptr_res1, int8 *ptr_res2,
                                   int8 *ptr_left, int8 *ptr_right,
                                   int8 scalar_c) {
  const int N = 1;
  const int C = 64;
  const int H = 32;
```

(续下页)

(接上页)

```
  const int W = 16;
  dim4 inp_shape = {N, C, H, W};
  auto left = tensor<int8>(inp_shape);
  auto right = tensor<int8>(inp_shape);
  auto res1 = tensor<int8>(inp_shape);
  auto res2 = tensor<int8>(inp_shape);

  auto g_left =  gtensor<int8>(inp_shape, GLOBAL, ptr_left);
  auto g_right =  gtensor<int8>(inp_shape, GLOBAL, ptr_right);
  auto g_res1 =  gtensor<int8>(inp_shape, GLOBAL, ptr_res1);
  auto g_res2 =  gtensor<int8>(inp_shape, GLOBAL, ptr_res2);

  dma::load(left, g_left);
  dma::load(right, g_left);
  tiu::bitwise_and(res1, left, right);
  tiu::bitwise_and(res2, left, scalar_c);
  dma::store(g_res1, res1);
  dma::store(g_res2, res2);
}

__TEST__ void __test__() {
  dim4 shape = {1, 64, 32, 16};
  auto left = ppl::rand<int8>(&shape, -32, 32);
  auto right = ppl::rand<int8>(&shape, -32, 32);
  auto res1 = ppl::malloc<int8>(&shape);
  auto res2 = ppl::malloc<int8>(&shape);
  int8 scalar = 25;
  bitwise_and_c_int8(res1, res2, left, right, scalar);
}
```

ppl::tiu::bitwise_or

1. 两个张量的元素做按位或运算。

```
template <typename DataType>
void bitwise_or(tensor<DataType> &dst, tensor<DataType> &src0,
                tensor<DataType> &src1);
```

$$dst(n, c, h, w) = src0(n, c, h, w) \quad \textbf{OR} \quad src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做按位或运算。

```
template <typename DataType>
void bitwise_or(tensor<DataType> &dst, tensor<DataType> &src, T2 C);
```

$$dst(n, c, h, w) = src(n, c, h, w) \quad \textbf{OR} \quad C$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与bitwise_and 一致。

ppl::tiu::bitwise_xor

1. 两个张量的元素做按位异或运算。

```
template <typename DataType>
void bitwise_xor(tensor<DataType> &dst, tensor<DataType> &src0,
                 tensor<DataType> &src1);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src0}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) \ \ \textbf{XOR} \ \ \mathsf{src1}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w})$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做按位异或运算。

```
template <typename DataType>
void bitwise_xor(tensor<DataType> &dst, tensor<DataType> &src, DataType C);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) \ \ \textbf{XOR} \ \ \mathsf{C}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与bitwise_and 一致。

cast

转换张量的元素的数据类型，对结果做 saturation。

```
template <typename DataType0, typename DataType1>
void cast(tensor<DataType0> &dst, tensor<DataType1> &src,
          rounding_mode_t round_mode = RM_HALF_TO_EVEN);
```

$$\mathrm{dst}(n, c, h, w) = \mathbf{cast}(\mathrm{src}(n, c, h, w))$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- round_mode(rounding_mode_t): 舍入模式

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 不支持 DT_FP16 和 DT_BFP16 的数据类型互转，如果 dst_dtype 与 src_dtype 相同，则 dst 与 src 的元素完全相同。
- 下表中列出数据类型转换时 mode 是否有效，⋆ 代表有效，空白代表无效。

src - dst	FP32	FP16	BFP16	INT32	UINT32	INT16	UINT16	INT8	UINT8
FP32		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
FP16				⋆	⋆	⋆	⋆	⋆	⋆
BFP16				⋆	⋆	⋆	⋆	⋆	⋆
INT32	⋆	⋆	⋆						
UINT32	⋆	⋆	⋆						
INT16		⋆	⋆						
UINT16		⋆	⋆						
INT8									
UINT8									

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void cast_to_f16(fp16 *ptr_rst, fp32 *ptr_inp) {
  const int N = 2;
  const int C = 8;
  const int H = 64;
  const int W = 32;

  dim4 src_shape = {N, C, H, W};
  auto inp = tensor<fp32>(src_shape);
  auto rst = tensor<fp16>(src_shape);

  auto g_inp = gtensor<fp32>(src_shape, GLOBAL, ptr_inp);
  auto g_rst = gtensor<fp16>(src_shape, GLOBAL, ptr_rst);
  dma::load(inp, g_inp);
  tiu::cast(rst, inp, RM_HALF_TO_EVEN);
  dma::store(g_rst, rst);
}
```

ppl::tiu::ceiling

浮点类型的张量的元素向正无穷舍入到附近的同类型的整数。

```
template <typename DataType>
void ceiling(tensor<DataType> &out, tensor<DataType> &in);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{ceiling}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}))$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。

使用示例

ppl::tiu::fdiv

1. 两个张量的元素相除。

```
template <typename DataType0, typename DataType1, typename DataType2>
void fdiv(tensor<DataType0> &dst, tensor<DataType1> &src0,
          tensor<DataType2> &src1, int num_iter);
```

$$\mathrm{dst}(n,c,h,w) = \frac{\mathrm{src0}(n,c,h,w)}{\mathrm{src1}(n,c,h,w)}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- num_iter(int): Newton 迭代次数

2. 张量作为被除数，常量作为除数。

```
template <typename DataType0, typename DataType1>
void fdiv(tensor<DataType0> &dst, tensor<DataType1> &src, float C, int num_iter);
```

$$\mathrm{dst}(n,c,h,w) = \frac{\mathrm{src}(n,c,h,w)}{C}$$

参数

- dst(tensor): 运算结果张量
- C: 常数
- src(tensor): src 张量
- num_iter(int): Newton 迭代次数

3. 张量作为除数，常量作为被除数。

```
template <typename DataType0, typename DataType1>
void fdiv(tensor<DataType0> &dst, float C, tensor<DataType1> &src, int num_iter);
```

$$\mathrm{dst}(n,c,h,w) = \frac{C}{\mathrm{src}(n,c,h,w)}$$

参数

- dst(tensor): 运算结果张量
- C: 常数
- src(tensor): src 张量
- num_iter(int): Newton 迭代次数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- num_iter 是性能参数，取值范围是 [1, 4]。
- 支持在 N/H/W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

```
#include "ppl.h"
using namespace ppl;
// 定义常量，表示Tensor的各个维度
const int N = 1;
const int C = 64;
const int H = 32;
const int W = 16;
// fp32_tunable_div 函数是 PPL 编程的 KERNEL 代码主体
// 用于执行浮点数 Tensor 的除法运算。
// 它接受三个指针参数 ptr_dst、ptr_src0 和 ptr_src1，
// 分别表示存储在 Global Memory 上的结果张量、被除数张量和除数张量。
// num_iter 是提供给后端TPU指令 tpu_bdc_fp32_tunable_div_C 的参数，
// 表示 Newton 迭代次数。
__KERNEL__ void fp32_tunable_div(float *ptr_dst, float *ptr_src0, float *ptr_src1,
                                 int numiter) {
  // 创建 dim4 类型对象 src_shape，表示 Tensor 的形状
  dim4 src_shape = {N, C, H, W};

  // 创建3个 gtensor 对象，对应DDR地址 ptr_dst、ptr_src0、ptr_src1上的数据
  auto g_src0 = gtensor<float>(src_shape, GLOBAL, ptr_src0);
  auto g_src1 = gtensor<float>(src_shape, GLOBAL, ptr_src1);
  auto g_dst = gtensor<float>(src_shape, GLOBAL, ptr_dst);

  // 创建3个 Tensor 对象，分别是 src0、src1、dst，这三个 Tensor 的 shape 相同。
  auto src0 = tensor<float>(src_shape);
  auto src1 = tensor<float>(src_shape);
  auto dst = tensor<int>(src_shape);

  // 使用 dma::load 将 Global Memory 上的被除数和除数张量数据载入到
  // Local Memory 上的 Tensor 中
  dma::load(src0, g_src0);
  dma::load(src1, g_src1);
 // 使用 tiu::fdiv 进行张量元素间除法计算，
  // 将计算结果通过 dma::store 传入 Global Memory 上的结果张量中
  tiu::fdiv(dst, src0, src1, numiter);
  dma::store(g_dst, dst);
}
```

ppl::tiu::fexp

张量的元素为指数，自然常数 e 为底数的指数运算。

```
template <typename DataType>
void fexp(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = e^{src(n,c,h,w)}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。
- n、c、h、w 的取值范围是 [1, 65535]，h * w 小于等于 65535。
- 如果 $src(n, c, h, w)$ 小于 -103 或大于 88，则 $e^{src(n,c,h,w)}$ 分别是 e^{-103} 和 e^{88}。

使用示例

略

ppl::tiu::flog

张量的元素为对数，自然常数 e 为底数的对数运算。

```
template <typename DataType>
void flog(tensor<DataType> &dst, tensor<DataType> &src);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \log_{\mathsf{e}}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}))$$

注意事项

- dst(tensor): 运算结果张量
- src(tensor): src 张量

注意事项

- dst、src 从同一个 lane 开始。
- DataType 的取值范围是 DT_FP32、DT_FP16、DT_BFP16。
- dst、src 的 shape 每一个维度的取值范围是 [1, 65535]，并且 shape->h * shape->w 小于等于 65535。
- log 的计算是通过泰勒展开以及查表进行运算的，如果 src 值太大可能会出现不收敛的现象。

使用示例

略

ppl::tiu::flogx

以 x 为底数对张量的元素做对数运算。

```
template <typename DataType>
void flogx(tensor<DataType> &dst, tensor<DataType> &src,
           tensor<DataType> &work, tensor<DataType> &coeff, float x);
```

$$dst(n, c, h, w) = \log_x(src(n, c, h, w))$$

注意事项

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- work(tensor): work 张量
- coeff(tensor): coeff 张量

注意事项

- dst、src 从同一个 lane 开始。
- DataType 的取值范围是 DT_FP32、DT_FP16、DT_BFP16。
- dst、src 的 shape 每一个维度的取值范围是 [1, 65535]，并且 shape->h * shape->w 小于等于 65535。
- log 的计算是通过泰勒展开以及查表进行运算的，如果 src 值太大可能会出现不收敛的现象。

使用示例

略

ppl::tiu::floor

浮点类型的张量的元素向负无穷舍入到附近的同类型的整数。

```
template <typename DataType>
void floor(tensor<DataType> &out, tensor<DataType> &in);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{floor}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}))$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。

使用示例

略

ppl::tiu::mac

1. 两个张量的元素相乘，再对结果做累加，在累加之前会对结果的原数据算数左移，在累加之后对结果算数右移。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mac(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, int l_shift, int r_shift,
         rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = (\mathrm{dst}(n,c,h,w) \ \textbf{左移}\ \mathrm{lshift} + \mathrm{src0}(n,c,h,w) \times \mathrm{src1}(n,c,h,w)) \ \textbf{右移}\ \mathrm{rshift}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- lshift(int): 左移位数
- rshift(int): 右移位数
- round_mode(rounding_mode_t): 右移舍入模式

2. 张量的元素与常数相乘，再对结果做累加，在累加之前会对结果的原数据算数左移，在累加之后对结果算数右移。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mac(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C,
         int l_shift, int r_shift, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = (\mathrm{dst}(n,c,h,w) \ \textbf{左移}\ \mathrm{lshift} + \mathrm{src}(n,c,h,w) \times C) \ \textbf{右移}\ \mathrm{rshift}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数
- lshift(int): 左移位数
- rshift(int): 右移位数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src0、和 src1 从同一个 lane 开始。
- dst、src0、和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- lshift 和 rshift 的取值范围是 [0, 31]。
- DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8。

- DataType1 和 DataType2 都是 DT_UINT8 则 DataType0 的元素的数据类型是 DT_UINT16, 累加前的元素的数据类型也被认定为 DT_UINT16, 否则是 DT_INT16。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与add 一致。

ppl::tiu::fmac

1. 两个张量的元素相乘，再对结果做累加。

```
void fmac(tensor<float> &dst, tensor<float> &src0, tensor<float> &src1);
```

$$dst(n, c, h, w) = dst(n, c, h, w) + src0(n, c, h, w) \times src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素和常数相乘，再对结果做累加。

```
void fmac(tensor<float> &dst, tensor<float> &src0, float C);
```

$$dst(n, c, h, w) = dst(n, c, h, w) + src(n, c, h, w) \times C$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N/H/W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与fadd 一致。

ppl::tiu::mul

1. 两个张量的元素相乘，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mul(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, char shift, rounding_mode_t round_mode,
         bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src0}(n,c,h,w) \times \mathsf{src1}(n,c,h,w)) \ \textbf{左移} \ \mathsf{shift} & \text{如果 } \mathsf{shift} > 0 \\ (\mathsf{src0}(n,c,h,w) \times \mathsf{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathsf{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

2. 张量的元素和常数相乘，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mul(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C, char shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src}(n,c,h,w) \times C) \ \textbf{左移} \ \mathsf{shift} & \text{如果 } \mathsf{shift} > 0 \\ (\mathsf{src}(n,c,h,w) \times C) \ \textbf{右移} \ -\mathsf{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- shift 的取值范围是 [-64, 31]。
- DataType0、DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。

- DataType0 是无符号的当且仅当 DataType1 和 DataType2 都是无符号的。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

3. 两个张量的元素相乘，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mul(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, tensor<int8> &shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src0}(n,c,h,w) \times \mathsf{src1}(n,c,h,w)) \ \textbf{左移} \ \mathsf{shift}(1,c,1,1) & \text{如果 } \mathsf{shift}(1,c,1,1) > 0 \\ (\mathsf{src0}(n,c,h,w) \times \mathsf{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathsf{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

4. 张量的元素和常数相乘，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void mul(tensor<DataType0> &dst, tensor<DataType1> &src0, tensor<int8> shift,
         DataType2 C, rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src}(n,c,h,w) \times C) \ \textbf{左移} \ \mathsf{shift}(1,c,1,1) & \text{如果 } \mathsf{shift}(1,c,1,1) > 0 \\ (\mathsf{src}(n,c,h,w) \times C) \ \textbf{右移} \ -\mathsf{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- shift(tensor): shift 张量
- C: 常数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0、src1 和 shift 从同一个 lane 开始。
- dst、src0、src1 和 shift 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。

- shift 的 shape 是 [1, c, 1, 1]，compact layout，取值范围是 [-64, 31]。
- DataType0、DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。
- DataType0 是无符号的当且仅当 DataType1 和 DataType2 都是无符号的。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与add 一致。

ppl::tiu::fmul

1. 两个张量的元素相乘。

```
template <typename DataType>
void fmul(tensor<DataType> &dst, tensor<DataType> &src0,
          tensor<DataType> &src1);
```

$$dst(n, c, h, w) = src0(n, c, h, w) \times src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素和常数相乘。

```
template <typename DataType>
void fmul(tensor<DataType> &dst, tensor<DataType> &src0, float C);
```

$$dst(n, c, h, w) = src(n, c, h, w) \times C$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16、DT_BFP16。
- 支持在 N/H/W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与fadd 一致。

ppl::tiu::round

浮点类型的张量的元素舍入到附近的同类型的整数。

```
template <typename DataType>
void round(tensor<DataType> &out, tensor<DataType> &in);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathsf{round}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}))$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。

使用示例

略

ppl::tiu::sub

1. 两个张量的元素相减，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, char shift, rounding_mode_t round_mode,
         bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src0}(n,c,h,w) - \mathsf{src1}(n,c,h,w)) \ \textbf{左移} \ \mathsf{shift} & \text{如果 } \mathsf{shift} > 0 \\ (\mathsf{src0}(n,c,h,w) - \mathsf{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathsf{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

2. 张量的元素减常数，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C, char shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathsf{dst}(n,c,h,w) = \begin{cases} (\mathsf{src}(n,c,h,w) - C) \ \textbf{左移} \ \mathsf{shift} & \text{如果 } \mathsf{shift} > 0 \\ (\mathsf{src}(n,c,h,w) - C) \ \textbf{右移} \ -\mathsf{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

3. 常数减张量的元素，对结果做算术移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, DataType1 C, tensor<DataType2> &src, char shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (C - \mathrm{src}(n,c,h,w)) \ \textbf{左移} \ \mathrm{shift} & \text{如果 } \mathrm{shift} > 0 \\ (C - \mathrm{src}(n,c,h,w)) \ \textbf{右移} \ -\mathrm{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- C: 常数
- src(tensor): src 张量
- shift(char): 移位数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- shift 的取值范围是 [-32, 31]。
- DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。
- DataType0 的有效取值是 DT_INT32、DT_INT16、和 DT_INT8。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

4. 两个张量的元素相加，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1, tensor<int8> &shift,
         rounding_mode_t round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src0}(n,c,h,w) - \mathrm{src1}(n,c,h,w)) \ \textbf{左移} \ \mathrm{shift}(1,c,1,1) & \text{如果 } \mathrm{shift}(1,c,1,1) > 0 \\ (\mathrm{src0}(n,c,h,w) - \mathrm{src1}(n,c,h,w)) \ \textbf{右移} \ -\mathrm{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

5 张量的元素减常数，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, tensor<DataType1> &src, tensor<int8> shift,
         DataType2 C, rounding_mode_t round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) - C) \ \textbf{左移} \ \mathrm{shift}(1,c,1,1) & \text{如果 } \mathrm{shift}(1,c,1,1) > 0 \\ (\mathrm{src}(n,c,h,w) - C) \ \textbf{右移} \ -\mathrm{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- shift(tensor): shift 张量
- C: 常数
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

6. 常数减张量的元素，对结果按 channel 做算数移位，再对结果做 saturation（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void sub(tensor<DataType0> &dst, DataType1 C, tensor<DataType2> &src,
         tensor<int8> &shift, rounding_mode_t round_mode, bool saturation);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (C - \mathrm{src}(n,c,h,w)) \ \textbf{左移} \ \mathrm{shift}(1,c,1,1) & \text{如果 } \mathrm{shift}(1,c,1,1) > 0 \\ (C - \mathrm{src}(n,c,h,w)) \ \textbf{右移} \ -\mathrm{shift}(1,c,1,1) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- C: 常数
- src(tensor): src 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式
- saturation(bool): 饱和处理标志

注意事项

- dst、src0、src1 和 shift 从同一个 lane 开始。
- dst、src0、src1 和 shift 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- shift 的 shape 是 [1, c, 1, 1]，compact layout，取值范围是 [-32, 31]。
- DataType1 和 DataType2 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。
- DataType0 的有效取值是 DT_INT32、DT_INT16、和 DT_INT8。

使用示例

具体使用方式与add 一致。

ppl::tiu::fsub

1. 两个张量的元素相减。

```
template <typename DataType>
void fsub(tensor<DataType> &dst, tensor<DataType> &src0,
          tensor<DataType> &src1);
```

$$dst(n, c, h, w) = src0(n, c, h, w) - src1(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素和常数相减。

```
template <typename DataType>
void fsub(tensor<DataType> &dst, tensor<DataType> &src0, float C);
```

$$dst(n, c, h, w) = src(n, c, h, w) - C$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

3. 常数减张量的元素。

```
template <typename DataType>
void fsub(tensor<DataType> &dst, float C, tensor<DataType> &src1);
```

$$dst(n, c, h, w) = C - src(n, c, h, w)$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。

- DataType 的有效取值是 DT_FP32、DT_FP16、DT_BFP16。
- 支持在 N/H/W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与fadd 一致。

ppl::tiu::fsub_abs

1. 计算两个张量的元素的差的绝对值。

```
template <typename DataType>
void fsub_abs(tensor<DataType> &dst, tensor<DataType> &src0,
              tensor<DataType> &src1);
```

$$dst(n, c, h, w) = |src0(n, c, h, w) - src1(n, c, h, w)|$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 计算张量的元素与常数的差的绝对值。

```
template <typename DataType>
void fsub_abs(tensor<DataType> &dst, tensor<DataType> &src, float C);
```

$$dst(n, c, h, w) = |src(n, c, h, w) - C|$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- DataType 的有效取值是 DT_FP32、DT_FP16、DT_BFP16。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与fadd 一致。

ppl::tiu::shift

张量的元素做算术移位运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void shift(tensor<DataType0> &dst, tensor<DataType1> &src,
           tensor<DataType2> &shift, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src}(n,c,h,w) \ \textbf{左移} \ \mathrm{shift}(n,c,h,w) & \text{如果 } \mathrm{shift}(n,c,h,w) > 0 \\ \mathrm{src}(n,c,h,w) \ \textbf{右移} \ -\mathrm{shift}(n,c,h,w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式

```
template <typename DataType0, typename DataType1>
void shift(tensor<DataType0> &dst, tensor<DataType1> &src, char shift_c,
           rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src}(n,c,h,w) \ \textbf{左移} \ \mathrm{shift}_c & \text{如果 } \mathrm{shift}_c > 0 \\ \mathrm{src}(n,c,h,w) \ \textbf{右移} \ -\mathrm{shift}_c & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- shift_c: shift 常数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 shift 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType0 和 DataType1 数据类型的有效取值是：DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- DataType2 数据类型的有效取值是：DT_INT32、DT_INT16、DT_INT8；shift 的元素的取值范围是 [-32, 31]。
- DataType1 的位宽大于等于 DataType2 位宽。
- dst 和 src 的 layout 可以为任意 layout。

使用示例

```
// shift 为张量
#include "ppl.h"

using namespace ppl;

const int N = 1;
const int C = 64;
const int H = 16;
```

(续下页)

(接上页)

```
const int W = 1;
__KERNEL__ void arith_shift_int(int *ptr_rst, int *ptr_inp, int *ptr_sft) {
  dim4 shape = {N, C, H, W};
  auto dst = tensor<int>(shape);
  auto inp = tensor<int>(shape);
  auto sft = tensor<int>(shape);
  auto g_dst =  gtensor<int>(shape, GLOBAL, ptr_rst);
  auto g_inp =  gtensor<int>(shape, GLOBAL, ptr_inp);
  auto g_sft =  gtensor<int>(shape, GLOBAL, ptr_sft);

  dma::load(inp, g_inp);
  dma::load(sft, g_sft);
  tiu::shift(dst, inp, sft, RM_HALF_TO_EVEN);
  dma::store(g_dst, dst);
}

// shift 为常数
#include "ppl.h"

using namespace ppl;

const int N = 1;
const int C = 64;
const int H = 16;
const int W = 1;
__KERNEL__ void arith_shift_C_int(int *ptr_rst, int *ptr_inp, int ch) {
  dim4 shape = {N, C, H, W};
  auto dst = tensor<int>(shape);
  auto src = tensor<int>(shape);
  auto g_dst =  gtensor<int>(shape, GLOBAL, ptr_rst);
  auto g_inp =  gtensor<int>(shape, GLOBAL, ptr_inp);

  dma::load(src, g_inp);
  tiu::shift(dst, src, ch, RM_HALF_TO_EVEN);
  dma::store(g_dst, dst);
}
```

ppl::tiu::logical_shift

张量的元素做逻辑移位运算。

```
template <typename DataType0 , typename DataType1, typename DataType2>
void logical_shift(tensor<DataType0> &dst, tensor<DataType1> &src,
                   tensor<DataType2> &shift, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src}(n,c,h,w) \ \textbf{左移} \ \mathrm{shift}(n,c,h,w) & \text{如果 } \mathrm{shift}(n,c,h,w) > 0 \\ \mathrm{src}(n,c,h,w) \ \textbf{右移} \ -\mathrm{shift}(n,c,h,w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- shift(tensor): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式

```
template <typename DataType0 , typename DataType1>
void logical_shift(tensor<DataType0> &dst, tensor<DataType1> &src, char shift_c,
                   rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src}(n,c,h,w) \ \textbf{左移} \ \mathrm{shift}_c & \text{如果 } \mathrm{shift}_c > 0 \\ \mathrm{src}(n,c,h,w) \ \textbf{右移} \ -\mathrm{shift}_c & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- shift_c: shift 常数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 shift 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType0 和 DataType1 数据类型的有效取值是：DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- DataType2 数据类型的有效取值是：DT_INT32、DT_INT16、DT_INT8；shift 的元素的取值范围是 [-32, 31]。
- DataType1 的位宽大于等于 DataType2 位宽。
- dst 和 src 的 layout 可以为任意 layout。

使用示例

具体使用方式与shift 一致。

ppl::tiu::circular_shift

张量的元素做绕回移位运算。

```
template <typename DataType0 , typename DataType1>
void circular_shift(tensor<DataType0> &dst, tensor<DataType1>/DataType1 &src,
                    tensor<char>/char &shift, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src}(n,c,h,w) \quad \textbf{左移} \quad \mathrm{shift}(n,c,h,w) & \text{如果 } \mathrm{shift}(n,c,h,w) > 0 \\ \mathrm{src}(n,c,h,w) \quad \textbf{右移} \quad -\mathrm{shift}(n,c,h,w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor/scalar): src 张量
- shift(tensor/scalar): shift 张量
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 shift 从同一个 lane 开始，任意排列。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType0 和 DataType1 数据类型的有效取值是：DT_INT32、DT_INT16、DT_INT8、DT_UINT32、DT_UINT16、DT_UINT8。
- shift 的元素的取值范围小于 DataType1 的位宽。
- **目前仅支持 SG2380**。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void shift_int(uint32 *ptr_rst0, uint32 *ptr_rst1, uint32 *ptr_rst2,
                          uint32 *ptr_inp, int8 *ptr_sft0, int8 *ptr_sft1) {
  const int N = 1;
  const int C = 64;
  const int H = 16;
  const int W = 1;
  dim4 shape = {N, C, H, W};

  tensor<uint32> dst0, dst1, dst2, inp;
  tensor<int8> sft0, sft1;
  auto dst0_g = gtensor<uint32>(shape, GLOBAL, ptr_rst0);
  auto dst1_g = gtensor<uint32>(shape, GLOBAL, ptr_rst1);
  auto dst2_g = gtensor<uint32>(shape, GLOBAL, ptr_rst2);
  auto inp_g = gtensor<uint32>(shape, GLOBAL, ptr_inp);
  auto sft0_g = gtensor<int8>(shape, GLOBAL, ptr_sft0);
  auto sft1_g = gtensor<int8>(shape, GLOBAL, ptr_sft1);

  dma::load(inp, inp_g);
  dma::load(sft0, sft0_g);
  dma::load(sft1, sft1_g);
  tiu::shift(dst0, inp, sft0, RM_HALF_TO_EVEN);
  tiu::logical_shift(dst1, inp, 2, RM_HALF_TO_EVEN);
  tiu::circular_shift(dst2, inp, sft1, RM_DOWN);
  dma::store(dst0_g, dst0);
```

(续下页)

(接上页)

```
  dma::store(dst1_g, dst1);
  dma::store(dst2_g, dst2);
}
```

4.6.2 Comparison

ppl::tiu::min

1. 两个张量的元素做取小运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void min(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1);
```

$$dst(n, c, h, w) = min(src0(n, c, h, w), src1(n, c, h, w))$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做取小运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void min(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C);
```

$$dst(n, c, h, w) = min(src(n, c, h, w), C)$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

ppl::tiu::fmin

1. 两个张量的元素做取小运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void fmin(tensor<DataType0> &dst, tensor<DataType1> &src0,
          tensor<DataType2> &src1);
```

$$dst(n, c, h, w) = min(src0(n, c, h, w), src1(n, c, h, w))$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做取小运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void fmax(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C);
```

$$dst(n, c, h, w) = min(src(n, c, h, w), C)$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

ppl::tiu::max

1. 两个张量的元素做取大运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void max(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1);
```

$$dst(n, c, h, w) = max(src0(n, c, h, w), src1(n, c, h, w))$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做取大运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void max(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C);
```

$$dst(n, c, h, w) = max(src(n, c, h, w), C)$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

ppl::tiu::fmax

1. 两个张量的元素做取大运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void max(tensor<DataType0> &dst, tensor<DataType1> &src0,
         tensor<DataType2> &src1);
```

$$dst(n, c, h, w) = max(src0(n, c, h, w), src1(n, c, h, w))$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

2. 张量的元素与常数做取大运算。

```
template <typename DataType0, typename DataType1, typename DataType2>
void max(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C);
```

$$dst(n, c, h, w) = max(src(n, c, h, w), C)$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- C: 常数

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

ppl::tiu::vc_min

两个向量的元素交叉做取小运算。

```
template <typename DataType>
void vc_min(tensor<DataType> &dst, tensor<DataType> &src0,
            tensor<DataType> &src1);
```

$$dst(m, n) = min(src0(m), src1(n))$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

注意事项

- dst 和 src1 从同一个 lane 开始。
- src0 的 shape 类型为 dim4，大小为 [1, src0_len / src0_len_per_channel, 1, src0_len_per_channel]，src1 的 shape 类型为 dim4，大小为 [1, src1_len / src1_len_per_channel, 1, src1_len_per_channel]，dst 的 shape 类型为 dim4，大小为 [src0_len, src1_len / src1_len_per_channel, 1, src1_len_per_channel]。
- src0_len 和 src1_len 的取值范围是 [1, 65535]。
- src0_len_per_channel 和 src1_len_per_channel 是 src0 和 src1 在每个 channel 的长度，len 必须能被 len_per_channel 整除。
- dst 是 matrix layout，dst 的行数是 src0_len，列数是 src1_len。
- src0 和 src1 是 vector layout。

使用示例

ppl::tiu::fvc_min

两个向量的元素交叉做取小运算。

```
template <typename DataType>
void vc_min(tensor<DataType> &dst, tensor<DataType> &src0,
            tensor<DataType> &src1);
```

$$dst(m, n) = min(src0(m), src1(n))$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

注意事项

- dst 和 src1 从同一个 lane 开始。
- src0 的 shape 类型为 dim4，大小为 [1, src0_len / src0_len_per_channel, 1, src0_len_per_channel]，src1 的 shape 类型为 dim4，大小为 [1, src1_len / src1_len_per_channel, 1, src1_len_per_channel]，dst 的 shape 类型为 dim4，大小为 [src0_len, src1_len / src1_len_per_channel, 1, src1_len_per_channel]。
- src0_len 和 src1_len 的取值范围是 [1, 65535]。
- src0_len_per_channel 和 src1_len_per_channel 是 src0 和 src1 在每个 channel 的长度，len 必须能被 len_per_channel 整除。
- dst 是 matrix layout，dst 的行数是 src0_len，列数是 src1_len。
- src0 和 src1 是 vector layout。

使用示例

ppl::tiu::vc_max

两个向量的元素交叉做取大运算。

```
template <typename DataType>
void vc_max(tensor<DataType> &dst, tensor<DataType> &src0,
            tensor<DataType> &src1);
```

$$dst(m, n) = max(src0(m), src1(n))$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

注意事项

- dst 和 src1 从同一个 lane 开始。
- src0 的 shape 类型为 dim4，大小为 [1, src0_len / src0_len_per_channel, 1, src0_len_per_channel]，src1 的 shape 类型为 dim4，大小为 [1, src1_len / src1_len_per_channel, 1, src1_len_per_channel]，dst 的 shape 类型为 dim4，大小为 [src0_len, src1_len / src1_len_per_channel, 1, src1_len_per_channel]。
- src0_len 和 src1_len 的取值范围是 [1, 65535]。
- src0_len_per_channel 和 src1_len_per_channel 是 src0 和 src1 在每个 channel 的长度，len 必须能被 len_per_channel 整除。
- dst 是 matrix layout，dst 的行数是 src0_len，列数是 src1_len。
- src0 和 src1 是 vector layout。

使用示例

ppl::tiu::fvc_max

两个向量的元素交叉做取大运算。

```
template <typename DataType>
void vc_max(tensor<DataType> &dst, tensor<DataType> &src0,
            tensor<DataType> &src1);
```

$$dst(m, n) = max(src0(m), src1(n))$$

参数

- dst(tensor): 运算结果张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量

注意事项

- dst 和 src1 从同一个 lane 开始。
- src0 的 shape 类型为 dim4，大小为 [1, src0_len / src0_len_per_channel, 1, src0_len_per_channel]，src1 的 shape 类型为 dim4，大小为 [1, src1_len / src1_len_per_channel, 1, src1_len_per_channel]，dst 的 shape 类型为 dim4，大小为 [src0_len, src1_len / src1_len_per_channel, 1, src1_len_per_channel]。
- src0_len 和 src1_len 的取值范围是 [1, 65535]。
- src0_len_per_channel 和 src1_len_per_channel 是 src0 和 src1 在每个 channel 的长度，len 必须能被 len_per_channel 整除。
- dst 是 matrix layout，dst 的行数是 src0_len，列数是 src1_len。
- src0 和 src1 是 vector layout。

使用示例

ppl::tiu::gt

1. 比较张量的元素是否大于另一个张量的元素，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2>
void gt(tensor<DataType0> &dst, tensor<DataType1> &src0,
        tensor<DataType1> &src1, DataType2 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src0}(n,c,h,w) > \mathrm{src1}(n,c,h,w) \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- true_val: 真值

2. 比较张量的元素是否大于常数，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void gt(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C,
        DataType3 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src}(n,c,h,w) > C \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src 张量
- C: 常数
- true_val: 真值

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的位宽大于等于 DataType0 位宽。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

```
// 比较张量的元素是否大于另一个张量的元素
#include "ppl.h"

using namespace ppl;

__KERNEL__ void greater_bf16(fp16 *ptr_res, fp16 *ptr_left, fp16 *ptr_right,
                             int W) {
  const int block_w = 512;
  float true_val = 1;

  dim4 shape = {1,1,1,W};
  auto l_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_left);
  auto r_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_right);
  auto res_gtensor = gtensor<fp16>(shape, GLOBAL, ptr_res);

  for (auto idx_w = 0; idx_w < W; idx_w += block_w) {
    enable_pipeline();
    dim4 tensor_shape = {1, 1, 1, block_w};
    auto left = tensor<fp16>(tensor_shape);
    auto right = tensor<fp16>(tensor_shape);
    auto res = tensor<fp16>(tensor_shape);

    int w = min(block_w, W - idx_w);
    dim4 local_shape = {1, 1, 1, w};
    dim4 stride = {1, 1, W, 1};
    dim4 offset = {0, 0, 0, idx_w};

    dma::load(left, l_gtensor.sub_view(local_shape, offset));
    dma::load(right, r_gtensor.sub_view(local_shape, offset));
    tiu::gt(res, left, right, true_val);
    dma::store(res_gtensor.sub_view(local_shape, offset), res);
  }
}
```

ppl::tiu::lt

1. 比较张量的元素是否小于另一个张量的元素，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2>
void lt(tensor<DataType0> &dst, tensor<DataType1> &src0,
        tensor<DataType1> &src1, DataType2 true_val);
```

$$\text{dst}(n,c,h,w) = \begin{cases} \text{true\_val} & \text{如果 src0}(n,c,h,w) < \text{src1}(n,c,h,w) \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- true_val: 真值

2. 比较张量的元素是否小于常数，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void lt(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C,
        DataType3 true_val);
```

$$\text{dst}(n,c,h,w) = \begin{cases} \text{true\_val} & \text{如果 src}(n,c,h,w) < C \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- C: 常数
- true_val: 真值

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的位宽大于等于 DataType0 位宽。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与gt 一致。

ppl::tiu::eq

1. 比较张量的元素是否小于另一个张量的元素，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2>
void eq(tensor<DataType0> &dst, tensor<DataType1> &src0,
        tensor<DataType1> &src1, DataType2 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src0}(n,c,h,w) = \mathrm{src1}(n,c,h,w) \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- true_val: 真值

2. 比较张量的元素是否小于常数，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void eq(tensor<DataType0> &dst, tensor<DataType1> &src0, DataType2 C,
        DataType3 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src}(n,c,h,w) = C \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- C: 常数
- true_val: 真值

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的位宽大于等于 DataType0 位宽。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与gt 一致。

ppl::tiu::lt

1. 比较张量的元素是否小于另一个张量的元素，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2>
void lt(tensor<DataType0> &dst, tensor<DataType1> &src0,
        tensor<DataType1> &src1, DataType2 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src0}(n,c,h,w) < \mathrm{src1}(n,c,h,w) \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor): src0 张量
- src1(tensor): src1 张量
- true_val: 真值

2. 比较张量的元素是否小于常数，可自定义真值。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void lt(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 C,
        DataType3 true_val);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{true\_val} & \text{如果 } \mathrm{src}(n,c,h,w) < C \\ 0 & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src(tensor): src 张量
- C: 常数
- true_val: 真值

注意事项

- dst、src0 和 src1 从同一个 lane 开始。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的位宽大于等于 DataType0 位宽。
- dst 和 src 的 layout 可以为任意 layout。
- 支持在 N / H / W 维度自动进行 broadcast，例如左操作数 shape 为 [2, 64, 64, 64]，右操作数 shape 为 [1, 64, 1, 1] 是可以直接运算的。

使用示例

具体使用方式与gt 一致。

ppl::tiu::gt_select

两个张量的元素比较大小，选取另外两个张量的元素作为结果。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4>
void gt_select(tensor<DataType0> &dst, DataType1 &src0, DataType2 &src1,
               DataType3 &src2, DataType4 &src3);
```

$$\text{dst}(n, c, h, w) = \begin{cases} \text{src2} & \text{如果 src0} > \text{src1} \\ \text{src3} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor/const): src0 张量或常量
- src1(tensor/const): src1 张量或常量
- src2(tensor/const): src2 张量或常量
- src3(tensor/const): src3 张量或常量

注意事项

- 对于上式中的 src0、src1、src2 和 src3，如果 src 的类型是 tensor，则 src 的 shape =[n, c, h, w]；如果 src 的类型是 vector，则 src 的 shape = [0, c~~mod~~LANE_NUM, 0, 0]；src2 和 src3 的类型是 tensor 或常数。
- dst 和 src 从同一个 lane 开始，dst 是 N-byte aligned layout。。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 src 的类型是 tensor，则 src 与 dst 的 shape 相同，N-byte aligned layout，如果 src 的类型是 vector，则 src 的 shape 是 [1, c, 1, 1], 且必须满足 c <= LANE_NUM。
- src0 和 src1 数据类型的位宽小于等于 dst 数据类型的位宽。

使用示例

ppl::tiu::lt_select

两个张量的元素比较大小，选取另外两个张量的元素作为结果。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4>
void lt_select(tensor<DataType0> &dst, DataType1 &src0, DataType2 &src1,
               DataType3 &src2, DataType4 &src3);
```

$$\text{dst}(n,c,h,w) = \begin{cases} \text{src2} & \text{如果 src0} < \text{src1} \\ \text{src3} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor/const): src0 张量或常量
- src1(tensor/const): src1 张量或常量
- src2(tensor/const): src2 张量或常量
- src3(tensor/const): src3 张量或常量

注意事项

- 对于上式中的 src0、src1、src2 和 src3，如果 src 的类型是 tensor，则 src 的 shape =[n, c, h, w]；如果 src 的类型是 vector，则 src 的 shape = [0, c~~mod~~LANE_NUM, 0, 0]；src2 和 src3 的类型是 tensor 或常数。
- dst 和 src 从同一个 lane 开始，dst 是 N-byte aligned layout。。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 src 的类型是 tensor，则 src 与 dst 的 shape 相同，N-byte aligned layout，如果 src 的类型是 vector，则 src 的 shape 是 [1, c, 1, 1], 且必须满足 c <= LANE_NUM。
- src0 和 src1 数据类型的位宽小于等于 dst 数据类型的位宽。

使用示例

ppl::tiu::eq_select

两个张量的元素比较大小，选取另外两个张量的元素作为结果。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4>
void eq_select(tensor<DataType0> &dst, DataType1 &src0, DataType2 &src1,
               DataType3 &src2, DataType4 &src3);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{src2} & \text{如果 } \mathrm{src0} = \mathrm{src1} \\ \mathrm{src3} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): 运算结果 dst 张量
- src0(tensor/const): src0 张量或常量
- src1(tensor/const): src1 张量或常量
- src2(tensor/const): src2 张量或常量
- src3(tensor/const): src3 张量或常量

注意事项

- 对于上式中的 src0、src1、src2 和 src3，如果 src 的类型是 tensor，则 src 的 shape =[n, c, h, w]；如果 src 的类型是 vector，则 src 的 shape = [0, c~~mod~~LANE_NUM, 0, 0]；src2 和 src3 的类型是 tensor 或常数。
- dst 和 src 从同一个 lane 开始，dst 是 N-byte aligned layout。。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 src 的类型是 tensor，则 src 与 dst 的 shape 相同，N-byte aligned layout，如果 src 的类型是 vector，则 src 的 shape 是 [1, c, 1, 1], 且必须满足 c <= LANE_NUM。
- src0 和 src1 数据类型的位宽小于等于 dst 数据类型的位宽。

使用示例

ppl::tiu::max_gt_select

两个张量的元素比较大小，选取其中较大的和另外两个张量的元素作为结果。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4, typename DataType5>
void max_gt_select(tensor<DataType0> &dst0, tensor<DataType1> &dst1,
                   DataType2 &src0, DataType3 &src1, DataType4 &src2,
                   DataType5 &src3);
```

$$\text{dst0}(n, c, h, w) = \max(\text{src0}, \text{src1})$$

$$\text{dst1}(n, c, h, w) = \begin{cases} \text{src2} & \text{如果 src0} > \text{src1} \\ \text{src3} & \text{其他情况} \end{cases}$$

参数

- dst0(tensor): 运算结果 dst0 张量
- dst1(tensor): 运算结果 dst1 张量
- src0(tensor/const): src0 张量或常量
- src1(tensor/const): src1 张量或常量
- src2(tensor/const): src2 张量或常量
- src3(tensor/const): src3 张量或常量

注意事项

- 对于上式中的 src0、src1、src2 和 src3，如果 src 的类型是 tensor，则 src 的 shape =[n, c, h, w]；如果 src 的类型是 vector，则 src 的 shape = [0, c~~mod~~LANE_NUM, 0, 0]；src2 和 src3 的类型是 tensor 或常数。
- dst 和 src 从同一个 lane 开始，dst 是 N-byte aligned layout。。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 src 的类型是 tensor，则 src 与 dst 的 shape 相同，N-byte aligned layout，如果 src 的类型是 vector，则 src 的 shape 是 [1, c, 1, 1], 且必须满足 c <= LANE_NUM。
- src0 和 src1 数据类型的位宽小于等于 dst 数据类型的位宽。

使用示例

ppl::tiu::min_lt_select

两个张量的元素比较大小，选取其中较小的和另外两个张量的元素作为结果。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4, typename DataType5>
void min_lt_select(tensor<DataType0> &dst0, tensor<DataType1> &dst1,
                   DataType2 &src0, DataType3 &src1, DataType4 &src2,
                   DataType5 &src3);
```

$$\text{dst0}(n, c, h, w) = \max(\text{src0}, \text{src1})$$

$$\text{dst1}(n, c, h, w) = \begin{cases} \text{src2} & \text{如果 src0} > \text{src1} \\ \text{src3} & \text{其他情况} \end{cases}$$

参数

- dst0(tensor): 运算结果 dst0 张量
- dst1(tensor): 运算结果 dst1 张量
- src0(tensor/const): src0 张量或常量
- src1(tensor/const): src1 张量或常量
- src2(tensor/const): src2 张量或常量
- src3(tensor/const): src3 张量或常量

注意事项

- 对于上式中的 src0、src1、src2 和 src3，如果 src 的类型是 tensor，则 src 的 shape =[n, c, h, w]；如果 src 的类型是 vector，则 src 的 shape = [0, c~~mod~~LANE_NUM, 0, 0]；src2 和 src3 的类型是 tensor 或常数。
- dst 和 src 从同一个 lane 开始，dst 是 N-byte aligned layout。
- dst、src0 和 src1 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 src 的类型是 tensor，则 src 与 dst 的 shape 相同，N-byte aligned layout，如果 src 的类型是 vector，则 src 的 shape 是 [1, c, 1, 1], 且必须满足 c <= LANE_NUM。
- src0 和 src1 数据类型的位宽小于等于 dst 数据类型的位宽。

使用示例

4.6.3 Indexing & Select & Gather & Scatter

ppl::tiu::gather_w

通过 w 维度的索引取值得到输出张量，即 dst = param[index]。

```
template <typename DataType0, typename DataType1>
void gather_w(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index);
```

$$\mathsf{dst}(n, c, 1, w) = \mathsf{param}(n, c, 1, \mathsf{index}(1, w, 1, 1))$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- shape->h 只能是 1，param 的 shape 是 [shape->n, shape->c, 1, param_w]，index 的 shape 是 [1, shape->w, 1, 1]。
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_w - 1]。

使用示例

ppl::tiu::gather_w

通过 w 维度的索引取值得到输出张量，即 dst = param[index]，索引的最大值特殊处理。

```
template <typename DataType0, typename DataType1>
void gather_w(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index, DataType2 C, bool fill_const);
```

$$\mathrm{dst}(n,c,0,w) = \begin{cases} \mathrm{param}(n,c,0,\mathrm{index}(0,w,0,0)) & \text{如果 } \mathrm{index}(0,w,0,0) \text{ 不是最大值} \\ C & \text{如果 } \mathrm{index}(0,w,0,0) \text{ 是最大值，fill\_const 是 true} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- C: 常数
- fill_const(bool): dst 在索引最大值处填 C 的标志

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- shape->h 只能是 1，param 的 shape 是 [shape->n, shape->c, 1, param_w]，index 的 shape 是 [1, shape->w, 1, 1]。
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_w - 1]，如果 DataType1 是 DT_UINT16，则索引最大值为 65535，如果 DataType1 是 DT_UINT8，则索引最大值为 255。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。

使用示例

ppl::tiu::scatter_w

通过 w 维度的索引改变输出张量的对应元素，即 dst[index] = param。

```
template <typename DataType0, typename DataType1>
void scatter_w(tensor<DataType0> &dst, tensor<DataType0> &param,
               tensor<DataType1> &index);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},0,\mathsf{index}(0,\mathsf{w},0,0)) = \mathsf{param}(\mathsf{n},\mathsf{c},0,\mathsf{w})$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- shape->h 只能是 1，param 的 shape 是 [shape->n, shape->c, 1, param_w]，index 的 shape 是 [1, shape->w, 1, 1]。
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_w - 1]。

使用示例

ppl::tiu::gather_hw

通过 h 和 w 维度的索引取值得到输出张量，即 dst = param[index]。

```
template <typename DataType0, typename DataType1>
void hw_gather(tensor<DataType0> &dst, tensor<DataType0> &param,
               tensor<DataType1> &index);
```

$$dst(n, c, h, w) = param(n, c, h_{param}, w_{param})$$

$$h_{param} = index(0, h \times W_{dst} + w, 0, 1)$$

$$w_{param} = index(0, h \times W_{dst} + w, 0, 0)$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- param 的 shape 是 [shape->n, shape->c, param_h, param_w]，index 的 shape 是 [1, shape->h * shape->w, 1, 2]。
- shape->n、shape->c、shape->h、shape->w、param_h 和 param_w 的取值范围是 [1, 65535]，shape->h * shape->w 小于等于 65535。
- DataType1 的数据类型是 UINT16，index(0, k, 0, 0) 存放 param 的 w 维度的坐标，取值范围是 [0, param_w - 1]，index(0, k, 0, 1) 存放 param 的 h 维度的坐标，取值范围是 [0, param_h - 1]。

使用示例

```
#include "ppl.h"

using namespace ppl;
MULTI_CORE
__KERNEL__ void hgather_w_bf16(bf16 *ptr_output, bf16 *ptr_param,
                               uint16 *ptr_index) {
  const int N = 2;
  const int C = 2;
  const int H = 64;
  const int W = 64;
  const int param_h = 128;
  const int param_w = 256;

  int core_num = get_core_num();
  int core_idx = get_core_index();
  if (core_idx >= core_num) {
    return;
  }
  int c_slice = div_up(C, core_num);
  int c_offset = core_idx * c_slice;
  c_slice = max(c_slice, (C - c_slice * (core_num - 1)));

  dim4 output_global_shape = {N, C, H, W};
  dim4 output_shape = {N, c_slice, H, W};
```

(续下页)

(接上页)

```
  dim4 param_global_shape = {N, C, param_h, param_w};
  dim4 param_shape = {N, c_slice, param_h, param_w};
  dim4 index_shape = {1, H * W, 1, 2};
  dim4 offset = {0, c_offset, 0, 0};

  auto param_gt = gtensor<bf16>(param_global_shape, GLOBAL, ptr_param);
  auto in_gt = gtensor<uint16>(index_shape, GLOBAL, ptr_index);
  auto out_gt = gtensor<bf16>(output_global_shape, GLOBAL, ptr_output);

  auto output = tensor<bf16>(output_shape);
  auto param = tensor<bf16>(param_shape);
  auto index = tensor<uint16>(index_shape, false);

  dma::load(param, param_gt.sub_view(param_shape, offset));
  dma::load_compact(index, in_gt);
  tiu::gather_hw(output, param, index, param_h, param_w);
  dma::store(out_gt.sub_view(output_shape, offset), output);
}
```

ppl::tiu::gather_hw

通过 h 和 w 维度的索引取值得到输出张量，即 dst = param[index]，索引的最大值特殊处理。

```
template <typename DataType0, typename DataType1>
void gather_hw(tensor<DataType0> &dst, tensor<DataType0> &param,
               tensor<DataType1> &index, DataType2 C, bool fill_const);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{param}(n,c,h_{\mathrm{param}},w_{\mathrm{param}}) & \text{如果 } h_{\mathrm{param}} \text{ 和 } w_{\mathrm{param}} \text{ 都不是最大值} \\ C & \text{如果 } h_{\mathrm{param}} \text{ 或 } w_{\mathrm{param}} \text{ 是最大值，fill\_const 是 true} \end{cases}$$

$$h_{\mathrm{param}} = \mathrm{index}(0, h \times W_{\mathrm{dst}} + w, 0, 1)$$

$$w_{\mathrm{param}} = \mathrm{index}(0, h \times W_{\mathrm{dst}} + w, 0, 0)$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- C: 常数
- fill_const(bool): dst 在索引最大值处填 C 的标志

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- param 的 shape 是 [shape->n, shape->c, param_h, param_w]，index 的 shape 是 [1, shape->h * shape->w, 1, 2]。
- shape->n、shape->c、shape->h、shape->w、param_h 和 param_w 的取值范围是 [1, 65535]，shape->h * shape->w 小于等于 65535。
- DataType1 的数据类型是 UINT16，索引最大值为 65535，index(0, k, 0, 0) 存放 param 的 w 维度的坐标，取值范围是 [0, param_w - 1] 和索引最大值，index(0, k, 0, 1) 存放 param 的 h 维度的坐标，取值范围是 [0, param_h - 1] 和索引最大值。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。

使用示例

ppl::tiu::scatter_hw

通过 h 和 w 维度的索引取值得到输出张量，即 dst = param[index]，索引的最大值特殊处理。

```
template <typename DataType0, typename DataType1>
void scatter_hw(tensor<DataType0> &dst, tensor<DataType0> &param,
                tensor<DataType1> &index);
```

$$dst(n, c, h, w) = param(n, c, h_{param}, w_{param})$$

$$h = index(0, h_{param} \times W_{param} + w_{param}, 0, 1)$$

$$w = index(0, h_{param} \times W_{param} + w_{param}, 0, 0)$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量

注意事项

- dst 和 param 从同一个 lane 开始，都是 N-byte aligned layout，index 从 lane 0 开始，compact layout。
- param 的 shape 是 [shape->n, shape->c, param_h, param_w]，index 的 shape 是 [1, shape->h * shape->w, 1, 2]。
- shape->n、shape->c、shape->h、shape->w、param_h 和 param_w 的取值范围是 [1, 65535]，shape->h * shape->w 小于等于 65535。
- DataType1 的数据类型是 UINT16，索引最大值为 65535，index(0, k, 0, 0) 存放 param 的 w 维度的坐标，取值范围是 [0, param_w - 1] 和索引最大值，index(0, k, 0, 1) 存放 param 的 h 维度的坐标，取值范围是 [0, param_h - 1] 和索引最大值。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。

使用示例

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void scatter_hw_bf16(bf16 *ptr_output, bf16 *ptr_param,
                                uint16 *ptr_index) {
  const int N = 1;
  const int C = 1;
  const int H = 128;
  const int W = 128;
  const int param_h = 32;
  const int param_w = 64;
  dim4 output_shape = {N, C, H, W};
  dim4 param_shape = {N, C, param_h, param_w};
  dim4 index_shape = {1, param_h * param_w, 1, 2};

  auto param_gt = gtensor<bf16>(param_shape, GLOBAL, ptr_param);
  auto in_gt = gtensor<uint16>(index_shape, GLOBAL, ptr_index);
  auto out_gt = gtensor<bf16>(output_shape, GLOBAL, ptr_output);

  auto output = tensor<bf16>(output_shape);
  auto param = tensor<bf16>(param_shape);
  auto index = tensor<uint16>(index_shape);
```

(续下页)

(接上页)

```
  dma::load(output, out_gt);
  dma::load(param, param_gt);
  dma::load_compact(index, in_gt);
  tiu::scatter_hw(output, param, index);
  dma::store(out_gt, output);
}
```

ppl::tiu::gather_w

通过 w 维度的索引取值得到输出张量，即 dst = param[index]，param 的 batch 被广播。

```
template <typename DataType0, typename DataType1>
void gather_w(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index, bool is_param_repeated);
```

$$\mathrm{dst}(n,c,0,w) = \begin{cases} \mathrm{param}(0, c \mod \mathrm{LANE\_NUM}, 0, \mathrm{index}(n,c,0,w)) & \text{如果 param 重复} \\ \mathrm{param}(0,c,0,\mathrm{index}(n,c,0,w)) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- is_param_repeated(bool): param 重复的标志

注意事项

- dst、param 和 index 从同一个 lane 开始，都是 N-byte aligned layout。
- shape->h 只能是 1，param 的 shape 是 [1, shape->c, 1, param_w]。
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_w - 1]。
- 如果 dst、param 和 index 两两之间没有 bank conflicting，则性能更优，如果 is_param_repeated 是 true，则判断是否冲突时，param 的 size 按照 shape 是 [1, 1, 1, param_w] 计算。

使用示例

ppl::tiu::gather_w

通过 w 维度的索引取值得到输出张量，即 dst = param[index]，param 的 batch 被广播，索引的最大值特殊处理。

```
template <typename DataType0, typename DataType1, typename DataType2>
void gather_w(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index, DataType2 C,
              bool is_param_repeated, bool fill_const);
```

$$dst(n,c,0,w) = \begin{cases} param(0,s,0,index(n,c,0,w)) & \text{如果 } index(n,c,0,w) \text{ 不是最大值} \\ C & \text{如果 } index(n,c,0,w) \text{ 是最大值，fill\_const 是 true} \end{cases}$$

$$s = \begin{cases} c \mod LANE\_NUM & \text{如果 param 重复} \\ c & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- C: 常数
- is_param_repeated(bool): param 重复的标志
- fill_const(bool): dst 在索引最大值处填 C 的标志

注意事项

- dst、param 和 index 从同一个 lane 开始，都是 N-byte aligned layout。
- shape->h 只能是 1，param 的 shape 是 [1, shape->c, 1, param_w]。
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_w - 1] 和索引最大值，如果 DataType1 是 DT_UINT16，则索引最大值为 65535，如果 DataType1 是 DT_UINT8，则索引最大值为 255。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。
- 如果 dst、param 和 index 两两之间没有 bank conflicting，则性能更优，如果 is_param_repeated 是 true，则判断是否冲突时，param 的 size 按照 shape 是 [1, 1, 1, param_w] 计算。

使用示例

ppl::tiu::scatter_w

通过 w 维度的索引改变输出张量的对应元素，即 dst[index] = param，param 的 batch 被广播。

```
template <typename DataType0, typename DataType1>
void scatter_w(tensor<DataType0> &dst, tensor<DataType0> &param,
               tensor<DataType1> &index, bool is_param_repeated);
```

$$\text{dst}(n,c,0,\text{index}(n,c,0,w)) = \begin{cases} \text{param}(0,c \mod \text{LANE\_NUM},0,w) & \text{如果 param 重复} \\ \text{param}(0,c,0,w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- is_param_repeated(bool): param 重复的标志

注意事项

- dst、param 和 index 从同一个 lane 开始，都是 N-byte aligned layout。
- shape->h 只能是 1，param 的 shape 是 [1, shape->c, 1, param_w]，index 的 shape 是 [shape->n, shape->c, 1, param_w]
- shape->n、shape->c、shape->w 和 param_w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, shape->w - 1]。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。
- 如果 dst、param 和 index 两两之间没有 bank conflicting，则性能更优，如果 is_param_repeated 是 true，则判断是否冲突时，param 的 size 按照 shape 是 [1, 1, 1, param_w] 计算。

使用示例

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void scatter_w_bf16(bf16 *ptr_output, bf16 *ptr_param,
                   uint8 *ptr_index) {
  const int N = 64;
  const int C = 64;
  const int H = 1;
  const int W = 256;
  const int param_w = 64;
  dim4 output_shape = {N, C, H, W};
  dim4 output_stride = {C * H * W, H * W, W, 1};
  dim4 param_shape = {N, C, H, param_w};
  dim4 param_stride = {C * H * param_w, H * param_w, param_w, 1};
  dim4 index_shape = {1, param_w, 1, 1};
  dim4 index_stride = {param_w, 1, 1, 1};
  dim4 offset = {0, 0, 0, 0};

  //Note: the w dim of output is decided by the max value of index,
  //      can't static inference
  auto output = tensor<bf16>(output_shape);
  tensor<bf16> param;
```

(续下页)

(接上页)

```
  tensor<uint8> index;
  auto o_gt = gtensor<bf16>(output_shape, GLOBAL, ptr_output);
  auto param_gt = gtensor<bf16>(param_shape, GLOBAL, ptr_param);
  auto index_gt = gtensor<uint8>(index_shape, GLOBAL, ptr_index);
  dma::load(output, o_gt);
  dma::load(param, param_gt);
  dma::load_compact(index, index_gt);
  tiu::scatter_w(output, param, index);
  dma::store(o_gt, output);
}
```

ppl::tiu::gather_h

通过 h 维度的索引取值得到输出张量，即 dst = param[index]，param 的 batch 被广播。

```
template <typename DataType0, typename DataType1>
void gather_h(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index, bool is_param_repeated);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} \mathrm{param}(0, c \mod \mathrm{LANE\_NUM}, \mathrm{index}(n,c,h,0), w) & \text{如果 param 重复} \\ \mathrm{param}(0, c, \mathrm{index}(n,c,h,0), w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- is_param_repeated(bool): param 重复的标志

注意事项

- dst、param 和 index 从同一个 lane 开始, dst 和 param 是 line N-byte aligned layout; index 是:ref:N-byte aligned layout。
- param 的 shape 是 [1, shape->c, param_h, shape->w]，index 的 shape 是 [shape->n, shape->c, shape->h, 1]。
- shape->n、shape->c、shape->h、shape->w 和 param_h 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_h - 1]。

使用示例

ppl::tiu::gather_h

通过 h 维度的索引取值得到输出张量，即 dst = param[index]，param 的 batch 被广播，索引的最大值特殊处理。

```
template <typename DataType0, typename DataType1, typename DataType2>
void gather_h(tensor<DataType0> &dst, tensor<DataType0> &param,
              tensor<DataType1> &index, DataType2 C,
              bool is_param_repeated, bool fill_const);
```

$$dst(n,c,h,w) = \begin{cases} param(0,s,index(n,c,h,0),w) & \text{如果 } index(n,c,h,0) \text{ 不是最大值} \\ C & \text{如果 } index(n,c,h,0) \text{ 是最大值，fill\_const 是 true} \end{cases}$$

$$s = \begin{cases} c \mod LANE\_NUM & \text{如果 param 重复} \\ c & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- C: 常数
- is_param_repeated(bool): param 重复的标志
- fill_const(bool): dst 在索引最大值处填 C 的标志

注意事项

- dst、param 和 index 从同一个 lane 开始，都是 N-byte aligned layout。
- param 的 shape 是 [1, shape->c, param_h, shape->w]，index 的 shape 是 [shape->n, shape->c, shape->h, 1]。
- shape->n、shape->c、shape->h、shape->w 和 param_h 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_h - 1] 和索引最大值，如果 DataType1 是 DT_UINT16，则索引最大值为 65535，如果 DataType1 是 DT_UINT8，则索引最大值为 255。
- 如果索引是最大值，fill_const 是 false，则 dst 的对应元素不会被填。

使用示例

ppl::tiu::scatter_h

通过 h 维度的索引改变输出张量的对应元素，即 dst[index] = param，param 的 batch 被广播。

```
template <typename DataType0, typename DataType1>
void scatter_h(tensor<DataType0> &dst, tensor<DataType0> &param,
               tensor<DataType1> &index, bool is_param_repeated);
```

$$\mathrm{dst}(n,c,\mathrm{index}(n,c,h,0),w) = \begin{cases} \mathrm{param}(0,c \mod \mathrm{LANE\_NUM},h,w) & \text{如果 param 重复} \\ \mathrm{param}(0,c,h,w) & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- param(tensor): local memory 上的 param 张量
- index(tensor): local memory 上的 index 张量
- is_param_repeated(bool): param 重复的标志

注意事项

- **dst、param 和 index 从同一个 lane 开始，**
  dst 和 param 是 line N-byte aligned layout; index 是:ref:N-byte aligned layout。
- param 的 shape 是 [1, shape->c, param_h, shape->w]，index 的 shape 是 [shape->n, shape->c, shape->h, 1]。
- shape->n、shape->c、shape->h、shape->w 和 param_h 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_UINT16 和 DT_UINT8，index 的元素的取值范围是 [0, param_h - 1] 和索引最大值。

使用示例

ppl::tiu::nonzero

生成非 0 元素索引的指令。指令 Tensor src 的 W 维非 0 元素的索引写到 Tensor dst_idx 中，返回每个 channel 的非 0 元素的个数并写到 Tensor dst_cnt 中。

```
template <typename DataType>
void nonzero(tensor<DataType> &dst_idx, tensor<uint16> &dst_cnt,
             tensor<DataType> &src);
```

参数

- dst_idx(tensor): local memory 上的 dst_idx 张量
- dst_cnt(tensor): local memory 上的 dst_cnt 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst_idx、dst_cnt 和 src 从同一个 lane 开始；dst_idx 和 src 是 N-byte aligned layout；dst_cnt 是 compact layout。
- dst_idx 的 shape 是 [n, c, 1, w]，dst_cnt 的 shape 是 [n, c, 1, 1]，src 的 shape 是 [n, c, 1, w]。
- n、c、和 w 的取值范围是 [1, 65535]。
- 目前仅支持 Chip == SG2380 。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void sfu_kernel(bf16 *ptr_rsqrt_res, int16 *ptr_norm_res,
                    int16 *ptr_idx_res, uint16 *ptr_cnt_res,
                    uint16 *ptr_clz_res, uint16 *ptr_clo_res,
                    bf16 *ptr_inp) {
  const int N = 32;
  const int C = 64;
  const int H = 1;
  const int W = 16;
  dim4 shape = {N, C, H, W};
  dim4 shape_cnt = {N, C, H, 1};
  tensor<bf16> rsqrt_res;
  tensor<int16> norm_res, sub_res, idx_res;
  tensor<uint16> cnt_res, clz_res, clo_res;
  tensor<bf16> inp;
  auto inp_g = gtensor<bf16>(shape, GLOBAL, ptr_inp);
  auto rsqrt_res_g = gtensor<bf16>(shape, GLOBAL, ptr_rsqrt_res);
  auto norm_res_g = gtensor<int16>(shape, GLOBAL, ptr_norm_res);
  auto idx_res_g = gtensor<int16>(shape, GLOBAL, ptr_idx_res);
  auto cnt_res_g = gtensor<uint16>(shape_cnt, GLOBAL, ptr_cnt_res);
  auto clz_res_g = gtensor<uint16>(shape, GLOBAL, ptr_clz_res);
  auto clo_res_g = gtensor<uint16>(shape, GLOBAL, ptr_clo_res);

  dma::load(inp, inp_g);
  tiu::frsqrt(rsqrt_res, inp, 3);
  dma::store(rsqrt_res_g, rsqrt_res);

  tiu::norm(norm_res, rsqrt_res);
  dma::store(norm_res_g, norm_res);

  tiu::sub(sub_res, norm_res, 131, 0, RM_HALF_TO_EVEN, false);
  tiu::nonzero(idx_res, cnt_res, sub_res);
```

(续下页)

(接上页)

```
  dma::store(idx_res_g, idx_res);
  dma::store(cnt_res_g, cnt_res);

  tiu::clz(clz_res, norm_res);
  tiu::clo(clo_res, norm_res);
  dma::store(clz_res_g, clz_res);
  dma::store(clo_res_g, clo_res);
}
```

ppl::tiu::mask_select

将 mask 为 1 相同位置上的 src 的 W 维元素写到 dst 中，并返回每个 Channel 的元素的个数并写到 dst_cnt 中。

```
template <typename DataType0, typename DataType1>
void mask_select(tensor<DataType0> &dst, tensor<uint16> &dst_cnt,
                 tensor<DataType0> &src, tensor<DataType1> &mask);
```

参数

- dst(tensor): dst 张量
- dst_cnt(tensor): dst_cnt 张量
- src(tensor): src 张量
- mask(tensor): mask 张量

注意事项

- dst、dst_cnt、src 和 mask 从同一个 lane 开始。
- dst 的 shape 是 [n, c, 1, w]，layout 为 N-byte aligned layout；dst_cnt 的 shape 是 [n, c, 1, 1]，layout 为 compact layout；如果 src 的 shape 是 [1, c, 1, w]，则其 layout 为 N-byte aligned layout；如果 src 的 shape 是 [1, 1, 1, w]，则其 layout 为 compact layout；mask 的 shape 是 [n, c, 1, w]，则其 layout 为 N-byte aligned layout。
- DataType1 的有效取值是 DT_UINT32、DT_UINT16、DT_UINT16。
- n、c、h 和 w 的取值范围是 [1, 65535]。
- **目前仅支持 SG2380**。

使用示例

4.6.4 Convolution

ppl::tiu::fconv

1. 2D 卷积，结果按 channel 加 bias（可选），再对结果做累加（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void fconv(tensor<DataType0> &dst, tensor<DataType1> &src,
           tensor<DataType1> &weight, DataType2 &bias, int oc, dim2 *k_shape,
           dim2 *stride, dim2 *dilation, padding_t *pad, bool result_add,
           data_type_t out_dtype, bool has_bias = true);
```

2. 核为常数的 2D 卷积，结果按 channel 加 bias（可选），再对结果做累加（可选）。

```
template <typename DataType0, typename DataType1>
void fconv(tensor<DataType0> &dst, tensor<DataType1> &src, float weight_C,
           int oc, dim2 *k_shape, padding_t *pad, dim2 *stride, dim2 *dilation,
           bool result_add, data_type_t out_dtype);
```

3. 其他接口：

```
template <typename DataType0, typename DataType1>
void fconv(tensor<DataType0> &dst, tensor<DataType1> &src,
           tensor<DataType1> &weight, int oc, dim2 *k_shape, dim2 *stride,
           dim2 *dilation, padding_t *pad, bool result_add,
           data_type_t out_dtype);

template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void fconv(tensor<DataType0> &dst, tensor<DataType1> &src, DataType2 &weight,
           DataType3 &bias, dim2 *k_shape, padding_t *pad, dim2 *stride,
           dim2 *dilation, dim2 *ins, bool result_add, bool saturate);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 数
- k_shape(dim2*): 指向 kernel 的 size 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 padding 的指针
- result_add(bool): 对结果做累加的标志，如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_FP32，而不是 DT_FP16 或 DT_BFP16。
- out_dtype(data_type_t): 结果数据类型，为 0 表示与输入一致；如果结果在下一次运算时需要累加，则必须为 DT_FP32
- has_bias(bool): 是否有 bias
- **saturate(bool): 对结果做饱和;**

当结果为 INT4/INT8/INT16 时：

true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时：MAX_VAL）
false：普通饱和（有符时，MIN_VAL / MAX_VAL，无符时：MAX_VAL）

当结果为 BF16/FP16 时:

true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan

注意事项

- dst、weight 和 bias 从同一个 LANE 开始，src 从 LANE 0 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16，DataType0 与 DataType1 相同，或是 DT_FP32。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- 如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_FP32，而不是 DT_FP16 或 DT_BFP16。
- 当 bias 为空时，has_bias 无效，当 bias 为 tensor 时，根据 has_bias 值决定是否对 bias 进行累加。
- saturate 仅对 **SG2380** 芯片生效

使用示例

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void conv_fp16(fp16 *ptr_res, fp16 *ptr_in, fp16 *ptr_w,
                          float *ptr_b) {
  int n = 1;
  int c = 16;
  int h = 48;
  int w = 48;
  int oc = 128;
  int kh = 3;
  int kw = 3;
  int stride_h = 1;
  int stride_w = 1;
  int pad_t = 1;
  int pad_b = 1;
  int pad_l = 1;
  int pad_r = 1;
  int oh = (h - kh + pad_t + pad_b) / stride_h + 1; // 计算输出特征图的高度
  int ow = (w - kw + pad_l + pad_r) / stride_w + 1; // 计算输出特征图的宽度

  int data_size = sizeof(fp16);
  int ws_w = LANE_NUM / data_size;
  int ws_h = div_up(c, LANE_NUM / data_size) * kh * kw;
  int ws_c = div_up(oc, LANE_NUM);
```

(续下页)

(接上页)

```
  dim4 in_shape = {n, c, h, w};
  dim4 weight_shape = {1, oc, ws_h, ws_w};
  dim4 bias_shape = {1, oc, 1, 1};
  dim4 out_shape = {n, oc, oh, ow};
  padding_t pad = {pad_t, pad_b, pad_l, pad_r};
  dim2 stride = {stride_h, stride_w};
  dim2 dilation = {1, 1};
  dim2 kernel = {kh, kw};

  tensor<fp16> in, weight, out;
  tensor<float> bias;
  auto in_gt = gtensor<fp16>(in_shape, GLOBAL, ptr_in);
  auto w_gt = gtensor<fp16>(weight_shape, GLOBAL, ptr_w);
  auto b_gt = gtensor<float>(bias_shape, GLOBAL, ptr_b);
  auto o_gt = gtensor<fp16>(out_shape, GLOBAL, ptr_res);
  dma::load(in, in_gt);
  // weight
  dma::load(weight, w_gt);
  dma::load_compact(bias, b_gt);
  tiu::fconv(out, in, weight, bias, oc, &kernel, &stride, &dilation, &pad,
             false, 0);
  dma::store(o_gt, out);
}
```

ppl::tiu::fdeconv

做 2D 反卷积的 2D 卷积（input 可以插零，kernel 做旋转），结果按 channel 加 bias（可选），再对结果做累加（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2>
void fdeconv(tensor<DataType0> &dst, tensor<DataType1> &src,
             tensor<DataType1> &weight, tensor<DataType2> &bias, int oc,
             dim2 *k_shape, dim2 *insert, padding_t *pad, dim2 *dilation,
             bool result_add, bool out32);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 数
- k_shape(dim2*): 指向 kernel 的 size 的指针
- insert(dim2*): 指向 insert 的指针
- pad(padding_t*): 指向 padding 的指针
- dilation(dim2*): 指向 dilation 的指针
- result_add(bool): 对结果做累加的标志
- out32(bool): 结果数据类型是 DT_FP32 的标志

注意事项

- dst、weight 和 bias 从同一个 LANE 开始，src 从 LANE 0 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = ((src_h - 1) * (insert_h + 1) + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = ((src_w - 1) * (insert_w + 1) + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16，DataType0 与 DataType1 相同，或是 DT_FP32。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::fdw_conv

2D depthwise 卷积，结果按 channel 加 bias（可选），再对结果做累加（可选）。

```
template <typename DataType>
void fdw_conv(tensor<DataType> &output, tensor<DataType> &input,
              tensor<DataType> &weight, tensor<DataType> &bias,
              dim2 *k_shape, padding_t *pad, dim2 *stride, dim2 *dilation);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 padding 的指针
- stride(dim2*): 指向 stride 的的指针
- dilation(dim2*): 指向 dilation 的指针

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, src_c, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16，DataType0 与 DataType1 相同，或是 DT_FP32。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::fdw_deconv

2D depthwise 反卷积（input 可以插零，kernel 做旋转），结果按 channel 加 bias（可选）。

```
template <typename DataType0, typename DataType1>
void fdw_deconv(tensor<DataType0> &dst, tensor<DataType1> &src,
                tensor<DataType1> &weight, tensor<DataType1> &bias,
                dim2 *k_shape, dim2 *insert, padding_t *pad, dim2 *dilation,
                bool kernel_is_const);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- k_shape(dim2*): 指向 kernel 的 size 的指针
- insert(dim2*): 指向 insert 的指针
- pad(padding_t*): 指向 padding 的指针
- dilation(dim2*): 指向 dilation 的指针
- kernel_is_const(bool): kernel 是否为常数的标志

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, src_c, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16，DataType0 与 DataType1 相同。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::dw_conv

2D depthwise 卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做算数右移，支持 pad 填充常数，支持 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void dw_conv(tensor<DataType0> &dst, tensor<DataType1> &src,
             tensor<DataType2> &weight, tensor<DataType3> &bias,
             int pad_val, dim2 *k_shape, padding_t *pad, dim2 *stride,
```

(续下页)

(接上页)

```
                dim2 *dilation, uint8 rshift, bool result_relu,
                rounding_mode_t round_mode);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor or none): local memory 上的 bias 张量
- pad_val(int): pad 填充的常数值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 pad 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- rshift(uint8): 右移位数
- result_relu(bool): 对结果做 ReLU 的标志
- round_mode(rounding_mode_t): 舍入模式

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, dst, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1、DataType2 和 DataType3 的有效取值是 DT_INT8 和 DT_UINT8，DataType0 的有效取值是 DT_INT32、DT_UINT32、DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- rshift 的取值范围是 [0, 31]。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

```
#include "ppl.h"
using namespace ppl;
const int n = 10;
const int c = 10;
const int h = 33;
```

(续下页)

(接上页)

```
const int w = 33;
const int kh = 3;
const int kw = 3;
const int stride_h = 1;
const int stride_w = 1;
const int pad_t = 1;    // pad must be [1, c, 1, 2]
const int pad_b = c;
const int pad_l = 1;
const int pad_r = 2;
const int dh = 1;
const int dw = 1;
const int oh = (h + pad_t + pad_b - ((kh - 1) * dh + 1)) / stride_h + 1;
const int ow = (w + pad_l + pad_r - ((kw - 1) * dw + 1)) / stride_w + 1;

__KERNEL__ void depthwise2d_uint(uint32 *ptr_res, uint8 *ptr_in,
                          uint8 *ptr_w, uint32 *ptr_b) {
                            dim4 in_shape = {n, c, h, w};
  dim4 weight_shape = {1, c, kh, kw};
  dim4 bias_shape = {1, c, 1, 1};
  dim4 out_shape = {n, c, oh, ow};
  padding_t pad = {pad_t, pad_b, pad_l, pad_r};
  dim2 stride = {stride_h, stride_w};
  dim2 dilation = {1, 1};
  dim4 offset = {0, 0, 0, 0};
  dim2 kernel = {kh, kw};

  auto in = tensor<uint8>(in_shape);
  auto weight = tensor<uint8>(weight_shape);
  // if have bias, it's bitwidth must be 32
  auto bias = tensor<uint32>(bias_shape);
  auto out = tensor<uint32>(out_shape);

  int pad_val = 0;
  uint8 rshift = 2;
  bool result_relu = false;
  rounding_mode_t round_mode = RM_TOWARDS_ZERO;

  auto in_g = gtensor<uint8>(in_shape, GLOBAL, ptr_in);
  auto in_w = gtensor<uint8>(weight_shape, GLOBAL, ptr_w);
  auto in_b = gtensor<uint32>(bias_shape, GLOBAL, ptr_b);
  auto out_g = gtensor<uint32>(out_shape, GLOBAL, ptr_res);

  dma::load(in, in_g);
  dma::load_compact(weight, in_w);
  dma::load_compact(bias, in_b);
  tiu::dw_conv(out, in, weight, bias, pad_val, &kernel, &pad, &stride,
               &dilation, rshift, result_relu, round_mode);
  dma::store(out_g, out);
}
```

ppl::tiu::dw_conv

核为常数的 2D depthwise 卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做算数右移，支持 insert，支持 pad 填充常数，支持 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2>
void dw_conv(tensor<DataType0> &dst, tensor<DataType1> &src,
             int weight_C, tensor<DataType2> &bias, int pad_val,
             dim2 *k_shape, padding_t *pad, dim2 *stride, dim2 *dilation,
             uint8 rshift, bool result_relu, rounding_mode_t round_mode);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight_C(int): weight 的常数值
- bias(tensor or none): local memory 上的 bias 张量
- pad_val(int): pad 填充的常数值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 pad 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- rshift(uint8): 右移位数
- result_relu(bool): 对结果做 ReLU 的标志
- round_mode(rounding_mode_t): 舍入模式

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_INT8 和 DT_UINT8，DataType2 的有效取值是 DT_INT32 和 DT_UINT32，DataType0 的有效取值是 DT_INT32、DT_UINT32、DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。
- 如果 DataType1 和 DataType2 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- rshift 的取值范围是 [0, 31]。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::dw_deconv

2D depthwise 反卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做算数右移，支持 insert，支持 pad 填充常数，支持 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void dw_deconv(tensor<DataType0> &dst, tensor<DataType1> &src,
               tensor<DataType2> &weight, tensor<DataType3> &bias,
               int pad_val, dim2 *k_shape, padding_t *pad, dim2 *insert,
               dim2 *dilation, uint8 rshift, bool result_relu,
               rounding_mode_t round_mode);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor or none): local memory 上的 bias 张量
- pad_val(int): pad 填充的常数值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 pad 的指针
- insert(dim2*): 指向 insert 的指针
- dilation(dim2*): 指向 dilation 的指针
- rshift(uint8): 右移位数
- result_relu(bool): 对结果做 ReLU 的标志
- round_mode(rounding_mode_t): 舍入模式

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout，
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8，DataType3 的有效取值是 DT_INT32 和 DT_UINT32，DataType0 的有效取值是 DT_INT32、DT_UINT32、DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- rshift 的取值范围是 [0, 31]。

- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::conv_sym

对称量化 2D 卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做算数右移。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void conv_sym(tensor<DataType0> &dst, tensor<DataType1> &src,
              tensor<DataType2> &weight, DataType3 &bias, int oc,
              dim2 *k_shape, dim2 *stride, dim2 *dilation, padding_t *pad,
              bool result_relu, int8 rshift)
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- result_relu(bool): 对结果做 ReLU 的标志
- rshift(int8): 右移位数

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1] compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8。
- 当芯片类型是 **BM1684X** 时，DataType3 的有效取值是 DT_INT16、DT_UINT16。
- 当芯片类型是 **BM1690** 时，DataType3 的有效取值是 DT_INT32、DT_UINT32。
- 不支持 **SG2380** 芯片
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- rshift 的取值范围是 [0, 31]。

- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::conv_sym_rq

对称量化 2D 卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做 requant。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void conv_sym_rq(tensor<DataType0> &dst, tensor<DataType1> &src,
                 tensor<DataType2> &weight, DataType3 &bias, int oc,
                 dim2 *k_shape, dim2 *stride, dim2 *dilation, padding_t *pad,
                 bool result_relu, int multiplier, int8 rshift, short out_zp,
                 bool saturate)
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- result_relu(bool): 对结果做 ReLU 的标志
- multiplier(int): 乘子常数
- rshift(int8): 右移位数
- out_zp(short): 结果偏置
- **saturate(bool): 对结果做饱和;**

  当结果为 INT4/INT8/INT16 时:
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时：MAX_VAL）
  false：普通饱和（有符时，MIN_VAL / MAX_VAL，无符时：MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8。

- 当芯片类型是 **BM1684X** 时，DataType3 的有效取值是 DT_INT16、DT_UINT16。
- 当芯片类型是 **BM1690** 时，DataType3 的有效取值是 DT_INT32、DT_UINT32。
- 不支持 **SG2380** 芯片
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- rshift 的取值范围是 [0, 31]。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::conv_sym_rq

对称量化 2D 卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做 requant。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void conv_sym_rq(tensor<DataType0> &dst, tensor<DataType1> &src,
                 tensor<DataType2> &weight, DataType3 &bias, int oc,
                 dim2 *k_shape, dim2 *stride, dim2 *dilation, padding_t *pad,
                 bool result_relu, DataType4 &requant, bool saturate,
                 rounding_mode_t round, bool kernel_rotate);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- result_relu(bool): 对结果做 ReLU 的标志
- requant(uint64/tensor): 量化参数
- **saturate(bool): 对结果做饱和;**

  当结果为 INT4/INT8/INT16 时:
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时: MAX_VAL）
  false: 普通饱和（有符时，MIN_VAL / MAX_VAL，无符时: MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan
- round(rounding_mode_t): round 模式，仅 **SG2380** 芯片生效
- kernel_rotate(bool): 对 kernel 进行旋转，仅 **SG2380** 芯片生效

特殊参数

表 1: 不同 chip 参数属性

	chip	dtype	shape	layout	note
requant	bm1684x bm1688 bm1690	int32	(1, oc, 1, 2)	compact	(1,c,1,0[31:0]) 为 multiplier; (1,c,1,1[7:0]) 为 shift; (1,c,1,1[31:16]) 为 offset;
	sg2380	int32 uint64	(1, oc, 1, 2)	compact 或常数	参数为 Tensor: dtype 为 int32 (1,c,1,0[31:0]) 为 multiplier; (1,c,1,1[7:0]) 为 shift; (1,c,1,1[31:16]) 为 offset; 为常数时: dtype 为 uint64 val[31:0] 为 multiplier; val[39:32] 为 shift; val[55:40] 为 offset;

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8。DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8。
- 当芯片类型是 **BM1684X** 时，DataType3 的有效取值是 DT_INT16、DT_UINT16。
- 当芯片类型是 **BM1690** 时，DataType3 的有效取值是 DT_INT32、DT_UINT32。
- 不支持 **SG2380** 芯片
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- rshift 的取值范围是 [0, 31]。

- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::conv_asym

非对称量化 2D 卷积，支持对结果进行累加（可选），支持 pad，支持 weight 减去 kzp。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4, typename DataType5>
void conv_asym(tensor<DataType0> &dst, tensor<DataType1> &src,
               DataType2 &weight, DataType3 &bias, int oc, dim2 *k_shape,
               dim2 *stride, dim2 *dilation, padding_t *pad, DataType4 &pad_val,
               bool result_add, data_type_t out_dtype, bool has_bias,
               DataType5 &kzp, bool saturate, rounding_mode_t round,
               bool kernel_rotate);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量, 仅 **SG2380** 支持
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- pad_val(常数或 tensor): pad 填充值，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- result_add(bool): 对结果是否做累加
- out_dtype(data_type_t): 结果数据类型，为 0 表示与输入一致；如果结果在下一次运算时需要累加，则必须为 DT_INT32 或 DT_UINT32
- has_bias(bool): 是否有 bias
- kzp(常数或 tensor): kernel 偏置，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- **saturate(bool): 对结果做饱和;**

  当结果为 INT4/INT8/INT16 时:
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时: MAX_VAL）
  false: 普通饱和（有符时，MIN_VAL / MAX_VAL，无符时: MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan
- round(rounding_mode_t): round 模式，仅 **SG2380** 芯片生效
- kernel_rotate(bool): 对 kernel 进行旋转，仅 **SG2380** 芯片生效

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。

- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32 、DT_UINT32、DT_INT16 和 DT_UINT16。DataType1、DataType2、DataType3 和 DataType4 的有效取值是 DT_INT8 和 DT_UINT8。
- 如果 DataType1、DataType2、DataType3、DataType5 都是无符号的，并且 kzp_val != 0，则 dst 累加前的元素的数据类型 DataType0 被认定为无符号的。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- 当 bias 为空时，has_bias 无效，当 bias 为 tensor 时，根据 has_bias 值决定是否对 bias 进行累加。
- 如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_INT32 或 DT_UINT32。

使用示例

ppl::tiu::conv_asym_rq

非对称量化 2D 卷积，支持对结果进行累加（可选），支持 pad，支持 weight 减去 kzp，支持对结果做 requant。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4, typename DataType5>
void conv_asym_rq(tensor<DataType0> &dst, tensor<DataType1> &src,
                  tensor<DataType2> &weight, DataType3 &bias, int oc,
                  dim2 *k_shape, dim2 *stride, dim2 *dilation, padding_t *pad,
                  DataType4 &pad_val, bool result_relu, bool result_add,
                  data_type_t out_dtype, bool has_bias, DataType5 &kzp,
                  int multiplier, int8 rshift, short out_zp, bool saturate,
                  rounding_mode_t round)
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 pad 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad_val(常数或 tensor): pad 填充值，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- rshift(int8): 右移位数
- result_add(bool): 对结果是否做累加
- out_dtype(data_type_t): 结果数据类型，为 0 表示与输入一致；如果结果在下一次运算时需要累加，则必须为 DT_INT32 或 DT_UINT32。
- has_bias(bool): 是否有 bias
- kzp(常数或 tensor): kernel 偏置，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- multiplier(int): 乘子常数
- rshift(int8): 右移位数
- out_zp(short): 结果偏置
- **saturate(bool): 对结果做饱和;**

  当结果为 INT4/INT8/INT16 时:
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时: MAX_VAL）
  false: 普通饱和（有符时，MIN_VAL / MAX_VAL，无符时: MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan
- round(rounding_mode_t): round 模式
- 不支持 **SG2380** 芯片

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。

- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32 、DT_UINT32、DT_INT16 和 DT_UINT16。DataType1、DataType2、DataType3 和 DataType4 的有效取值是 DT_INT8 和 DT_UINT8。
- 如果 DataType1、DataType2、DataType3、DataType5 都是无符号的，并且 kzp_val != 0，则 dst 累加前的元素的数据类型 DataType0 被认定为无符号的。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- 当 bias 为空时，has_bias 无效，当 bias 为 tensor 时，根据 has_bias 值决定是否对 bias 进行累加。
- 如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_INT32 或 DT_UINT32

使用示例

ppl::tiu::conv_asym_rq

非对称量化 2D 卷积，支持对结果进行累加（可选），支持 pad，支持 weight 减去 kzp，支持对结果做 requant。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4, typename DataType5,
          typename DataType6>
void conv_asym_rq(tensor<DataType0> &dst, tensor<DataType1> &src,
                  tensor<DataType2> &weight, DataType3 &bias, int oc,
                  dim2 *k_shape, dim2 *stride, dim2 *dilation, padding_t *pad,
                  DataType4 &pad_val, bool result_add, data_type_t out_dtype,
                  bool has_bias, DataType5 &kzp, tensor<DataType6> &requant,
                  bool saturate, rounding_mode_t round, bool kernel_rotate);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- pad(padding_t*): 指向 pad 的指针
- stride(dim2*): 指向 stride 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad_val(常数或 tensor): pad 填充值，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- rshift(int8): 右移位数
- result_add(bool): 对结果是否做累加
- out_dtype(data_type_t): 结果数据类型，为 0 表示与输入一致；如果结果在下一次运算时需要累加，则必须为 DT_INT32 或 DT_UINT32
- has_bias(bool): 是否有 bias
- kzp(常数或 tensor): kernel 偏置，当为 tensor 时，shape 为 [1, oc, 1, 1], compact layout
- requant(uint64/tensor): 量化参数
- **saturate(bool): 对结果做饱和；**

  当结果为 INT4/INT8/INT16 时：
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时：MAX_VAL）
  false：普通饱和（有符时，MIN_VAL / MAX_VAL，无符时：MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan
- round(rounding_mode_t): round 模式
- kernel_rotate(bool): 对 kernel 进行旋转，仅 **SG2380** 芯片生效

特殊参数

表 2: 不同 chip 参数属性

	chip	dtype	shape	layout	note
requant	bm1684x bm1688 bm1690	int32	(1, oc, 1, 2)	compact	(1,c,1,0[31:0]) 为 multiplier; (1,c,1,1[7:0]) 为 shift; (1,c,1,1[31:16]) 为 offset;
	sg2380	int32 uint64	(1, oc, 1, 2)	compact 或常数	参数为 Tensor: dtype 为 int32 (1,c,1,0[31:0]) 为 multiplier; (1,c,1,1[7:0]) 为 shift; (1,c,1,1[31:16]) 为 offset; 为常数时: dtype 为 uint64 val[31:0] 为 multiplier; val[39:32] 为 shift; val[55:40] 为 offset;

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = (src_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = (src_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout。
- weight 的 shape 为 [1, oc, ceil(ic / NIC) * kh * kw, NIC] compact layout, 具体参考 **TPU 中数据的存储模式章节**，N-IC 存储部分。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32 、DT_UINT32、DT_INT16 和 DT_UINT16。DataType1、DataType2、DataType3 和 DataType4 的有效取值是 DT_INT8 和 DT_UINT8。
- 如果 DataType1、DataType2、DataType3、DataType5 都是无符号的，并且 kzp_val != 0，则 dst 累加前的元素的数据类型 DataType0 被认定为无符号的。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- 当 bias 为空时，has_bias 无效，当 bias 为 tensor 时，根据 has_bias 值决定是否对 bias 进行累加。
- 如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_INT32 或 DT_UINT32

使用示例

ppl::tiu::deconv_sym

对称量化 2D 反卷积，结果按 channel 加 bias（可选），再对结果做 ReLU（可选），最后对结果做算数右移，支持 pad，支持 insert，支持指定输出类型（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void deconv_sym(tensor<DataType0> &dst, tensor<DataType1> &src,
                tensor<DataType2> &weight, tensor<DataType3> &bias, int oc,
                dim2 *k_shape, dim2 *dilation, padding_t *pad, dim2 *ins,
                bool result_relu, data_type_t out_dtype, bool has_bias, uint8 rsift);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- bias(tensor): local memory 上的 bias 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- ins(dim2*): 指向 insert 的指针
- result_relu(bool): 对结果做 ReLU 的标志
- out_dtype(data_type_t): 结果的数据类型
- rshift(uint8): 右移位数

注意事项

- dst、src、weight 和 bias 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = ((src_h - 1) * (insert_h + 1) + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = ((src_w - 1) * (insert_w + 1) + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- src 是 free layout，bias 的 shape 是 [1, oc, 1, 1]，compact layout，weight 的 shape 是 [1, oc, div_up(src_c, NIC) * kernel->h * kernel->w, NIC]，NIC layout。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16 、DT_UINT16、DT_INT8 和 DT_UINT8，支持通过 out_dtype 参数进行指定。DataType1 和 DataType2 的有效取值是 DT_INT8 和 DT_UINT8。
- 当芯片类型是 **BM1684X** 时，DataType3 的有效取值是 DT_INT16、DT_UINT16。
- 当芯片类型是 **BM1690** 时，DataType3 的有效取值是 DT_INT32、DT_UINT32。
- 如果 DataType1、DataType2 和 DataType3 中有一个是有符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为有符号的，否则是无符号的。
- 如果 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- rshift 的取值范围是 [0, 31]。

- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::deconv_asym

非对称量化 2D 反卷积，支持对结果进行累加（可选），支持 pad，支持 weight 减去 kzp 常数，支持常数 insert，支持指定输出类型（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void deconv_asym(tensor<DataType0> &dst, tensor<DataType1> &src,
                 tensor<DataType2> &weight, int oc, dim2 *k_shape,
                 dim2 *dilation, padding_t *pad, dim2 *ins, int pad_val,
                 int insert_val, bool result_add, data_type_t out_dtype,
                 DataType3 kzp_val);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- weight(tensor): local memory 上的 weight 张量
- oc(int): dst 的 channel 值
- k_shape(dim2*): 指向 kernel 的 size 的指针
- dilation(dim2*): 指向 dilation 的指针
- pad(padding_t*): 指向 pad 的指针
- insert(dim2*): 指向 insert 的指针
- pad_val(int): pad 的值
- insert_val(int): insert 的值
- result_add(bool): 对结果做累加的标志
- out_dtype(data_type_t): 结果的数据类型
- kzp_val: kzp 的值

注意事项

- dst、src 和 weight 从同一个 LANE 开始。
- dst 的 shape 是 [src_n, oc, dst_h, dst_w]，N-byte aligned layout，

  dst_h = ((src_h - 1) * (insert_h + 1) + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  dst_w = ((src_w - 1) * (insert_w + 1) + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- weight 的 shape 是 [1, oc, div_up(src_c, NIC) * kernel_h * kernel_w, NIC]，NIC layout，src 是 free layout。
- src_n、src_c、src_h、src_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32 、DT_UINT32、DT_INT16 和 DT_UINT16，支持通过 out_dtype 参数进行指定。DataType1、DataType2、DataType3 和 DataType4 的有效取值是 DT_INT8 和 DT_UINT8。
- 如果 DataType1、DataType2 和 DataType3 都是无符号的，并且 kzp_val = 0，则 dst 累加前的元素的数据类型 DataType0 被认定为无符号的。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

4.6.5 Matmul

ppl::tiu::mm

两个矩阵相乘。

```
template <typename DataType0, typename DataType1>
void mm(tensor<DataType0> &dst, tensor<DataType1> &left,
        tensor<DataType1> &right, bool ltrans = false, bool rtrans = false);
```

$$dst(m, n) = \sum_{k} left(m, k) \times right(k, n)$$

参数

- dst(tensor): local memory 上的 dst 矩阵
- left(tensor): local memory 上的 left 矩阵
- right(tensor): local memory 上的 right 矩阵
- ltrans(bool): 左矩阵转置标志
- rtrans(bool): 右矩阵转置标志

注意事项

- dst 和 right 从同一个 lane 开始。
- left 的 shape 类型为 dim4，大小为 [M, K / l_cols_perchannel, 1, l_cols_perchannel]，right 的 shape 类型为 dim4，大小为 [K, N / r_cols_perchannel, 1, r_cols_perchannel]，dst 的 shape 类型为 dim4，大小为 [M, N / r_cols_perchannel, 1, r_cols_perchannel]。
- M、N 和 K 的取值范围是 [1, 65535]。
- l_cols_perchannel 和 r_cols_perchannel 是 left 和 right 在每个 channel 的长度，K/N 必须能被 cols_perchannel 整除。
- dst、left 和 right 是 matrix layout，dst 的行数是 M，列数是 N，right 的行数是 K。
- DataType1 有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，如果 DataType1 是无符号的，则 DataType0 的元素的数据类型是 DT_UINT32，否则是 DT_INT32。
- 该指令不支持矩阵转置，ltrans 和 rtrans 仅标志左右矩阵在计算前是否经过转置。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void mm_int32(int32 *ptr_res, int32 *ptr_left, int32 *ptr_right) {
  const int M = 64;
  const int K = 32;
  const int N = 96;
  const int l_cols_perchannel = 8;
  const int r_cols_perchannel = 8;
  dim4 left_tensor_shape = {M, K / l_cols_perchannel, 1, l_cols_perchannel};
  dim4 right_tensor_shape = {K, N / r_cols_perchannel, 1, r_cols_perchannel};
  dim4 res_tensor_shape = {M, N / r_cols_perchannel, 1, r_cols_perchannel};
  auto g_left = gtensor<int32>(left_tensor_shape, GLOBAL, ptr_left);
  auto g_right = gtensor<int32>(right_tensor_shape, GLOBAL, ptr_right);
  auto g_res = gtensor<int32>(res_tensor_shape, GLOBAL, ptr_res);
```

(续下页)

(接上页)

```
  auto left = tensor<int32>(left_tensor_shape);
  auto right = tensor<int32>(right_tensor_shape);
  auto res = tensor<int32>(res_tensor_shape);

  dma::load(left, g_left);
  dma::load(right, g_right);
  tiu::mm(res, left, right);
  dma::store(g_res, res);
}
```

ppl::tiu::mm

两个矩阵相乘，对结果做累加（可选，在累加之前可对结果的原数据算数左移），再对结果做 ReLU（可选），最后对结果做算数右移，结果有 saturation。

```
template <typename DataType0, typename DataType1>
void mm(tensor<DataType0> &dst, tensor<DataType1> &left,
        tensor<DataType1> &right, bool ltrans, bool rtrans, bool result_add,
        int lshift, int rshift, bool do_relu);
```

$$\mathsf{dst(m,n)} = \mathsf{ReLU}((\mathsf{dst(m,n)} \ \textbf{左移} \ \mathsf{lshift}) + (\sum_{\mathsf{k}} \mathsf{left(m,k)} \times \mathsf{right(k,n)})) \ \textbf{右移} \ \mathsf{rshift}$$

参数

- dst(tensor): local memory 上的 dst 矩阵
- left(tensor): local memory 上的 left 矩阵
- right(tensor): local memory 上的 right 矩阵
- result_add(bool): 对结果做累加的标志
- lshift(int): 左移位数
- rshift(int): 右移位数
- do_relu(bool): 对结果做 ReLU 的标志
- ltrans(bool): 左矩阵转置标志
- rtrans(bool): 右矩阵转置标志

注意事项

- dst 和 right 从同一个 lane 开始。
- left 的 shape 类型为 dim4，大小为 [M, K / l_cols_perchannel, 1, l_cols_perchannel]，right 的 shape 类型为 dim4，大小为 [K, N / r_cols_perchannel, 1, r_cols_perchannel]，dst 的 shape 类型为 dim4，大小为 [M, N / r_cols_perchannel, 1, r_cols_perchannel]。
- M、N 和 K 的取值范围是 [1, 65535]。
- l_cols_perchannel 和 r_cols_perchannel 是 left 和 right 在每个 channel 的长度，K/N 必须能被 cols_perchannel 整除。
- dst、left 和 right 是 matrix layout，dst 的行数是 M，列数是 N，right 的行数是 K。
- DataType0 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，DataType1 的有效取值是 DT_INT8 和 DT_UINT8。
- 如果 DataType1 是无符号的，则 dst 累加前的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- 如果 DataType1 是无符号的，或 do_relu 是 true，则 dst 的元素的数据类型 DataType0 被认定为无符号的，否则是有符号的。
- 如果 DataType0 是 DT_INT8 或 DT_UINT8，对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_INT16 或 DT_UINT16，而不是 DT_INT8 或 DT_UINT8。
- lshift 和 rshift 的取值范围是 [0, 31]。
- **BM1690** 不支持左移、右移和累加，即 lshift == 0、rshift == result_add == false。
- 该指令不支持矩阵转置，ltrans 和 rtrans 仅标志左右矩阵在计算前是否经过转置。

使用示例

ppl::tiu::fmm

两个矩阵相乘，再对结果做累加（可选）。

```
template <typename DataType0, typename DataType1>
void fmm(tensor<DataType0> &dst, tensor<DataType1> &left,
         tensor<DataType1> &right, bool ltrans, bool rtrans, bool result_add);
```

$$dst(m,n) = dst(m,n) + (\sum_{k} left(m,k) \times right(k,n))$$

参数

- dst(tensor): local memory 上的 dst 矩阵
- left(tensor): local memory 上的 left 矩阵
- right(tensor): local memory 上的 right 矩阵
- result_add(bool): 对结果做累加的标志
- ltrans(bool): 左矩阵转置标志
- rtrans(bool): 右矩阵转置标志

注意事项

- dst、left 和 right 的地址都被 4 整除。
- left 的 shape 类型为 dim4，大小为 [M, K / l_cols_perchannel, 1, l_cols_perchannel]，right 的 shape 类型为 dim4，大小为 [K, N / r_cols_perchannel, 1, r_cols_perchannel]，dst 的 shape 类型为 dim4，大小为 [M, N / r_cols_perchannel, 1, r_cols_perchannel]。
- M、N 和 K 的取值范围是 [1, 65535]。
- l_cols_perchannel 和 r_cols_perchannel 是 left 和 right 在每个 channel 的长度，K/N 必须能被 cols_perchannel 整除。
- dst、left 和 right 是 matrix layout，dst 的行数是 M，列数是 N，right 的行数是 K。
- 该指令不支持矩阵转置，ltrans 和 rtrans 仅标志左右矩阵在计算前是否经过转置。
- 优先使用 fmm2 指令

使用示例

```
#include "ppl.h"
using namespace ppl;

const int M = 400;
const int K = 100;
const int N = 200;

__KERNEL__ void mm_fp32(float *ptr_res, float *ptr_left, float *ptr_right,
                        int M, int K, int N) {
                          const int block_m = 64;
  const int block_k = 32;
  const int block_n = 96;
  const int l_cols_perchannel = 8;
  const int r_cols_perchannel = 8;

  dim4 left_global_shape = {M, K / l_cols_perchannel, 1, l_cols_perchannel};
  dim4 right_global_shape = {K, N / r_cols_perchannel, 1, r_cols_perchannel};
  dim4 res_global_shape = {M, N / r_cols_perchannel, 1, r_cols_perchannel};
```

(续下页)

(接上页)

```
  auto left_gt = gtensor<float>(left_global_shape, GLOBAL, ptr_left);
  auto right_gt = gtensor<float>(right_global_shape, GLOBAL, ptr_right);
  auto res_gt = gtensor<float>(res_global_shape, GLOBAL, ptr_res);

  for (auto idx_m = 0; idx_m < M; idx_m += block_m) {
    for (auto idx_n = 0; idx_n < N; idx_n += block_n) {
      dim4 sum_shape = {block_m, block_n / r_cols_perchannel, 1,
                        r_cols_perchannel};
      auto psum = tensor<fp32>(sum_shape);
      int m = min(block_m, M - idx_m);
      int n = min(block_n, N - idx_n);
      for (auto idx_k = 0; idx_k < K; idx_k += block_k) {
        bool result_add = true;
        if (idx_k == 0) {
          result_add = false;
        }
        ppl::enable_pipeline();
        dim4 left_tensor_shape = {block_m, block_k / l_cols_perchannel, 1,
                                  l_cols_perchannel};
        dim4 right_tensor_shape = {block_k, block_n / r_cols_perchannel, 1,
                                   r_cols_perchannel};
        auto left = tensor<fp32>(left_tensor_shape);
        auto right = tensor<fp32>(right_tensor_shape);
        dim4 left_offset = {idx_m, idx_k / l_cols_perchannel, 0, 0};
        dim4 right_offset = {idx_k, idx_n / r_cols_perchannel, 0, 0};

        dma::load(left, left_gt.sub_view(left_tensor_shape, left_offset));
        dma::load(right, right_gt.sub_view(right_tensor_shape, right_offset));
        tiu::fmm(psum, left, right, result_add);
      }
      int res_row_stride = N;
      dim4 res_offset = {idx_m, idx_n / r_cols_perchannel, 0, 0};
      dma::store(res_gt.sub_view(sum_shape, res_offset), psum);
    }
  }
}
```

ppl::tiu::mm2

两个矩阵相乘/左矩阵乘以右矩阵的转置/两个矩阵的转置相乘，结果也转置，其中右矩阵的元素减 zero-point，再对结果做累加（可选），结果没有 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3, typename DataType4>
void mm2(tensor<DataType0> &rst, DataType1 &left, DataType2 &right,
         DataType3 &bias, DataType4 &r_zp, bool ltrans, bool rtrans,
         bool rst_trans, bool result_add, data_type_t out_dtype,
         bool has_bias);

template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void mm2(tensor<DataType0> &rst, DataType1 &left,
         DataType2 &right, DataType3 &r_zp, bool ltrans, bool rtrans,
         bool rst_trans, bool result_add, data_type_t out_dtype);

template <typename DataType0, typename DataType1, typename DataType2>
void mm2(tensor<DataType0> &rst, DataType1 &left,
         DataType2 &right, bool ltrans, bool rtrans, bool rst_trans);

template <typename DataType0, typename DataType1, typename DataType2>
void mm2(tensor<DataType0> &rst, DataType1 &left, DataType2 &right);

template <typename DataType0, typename DataType1, typename DataType2,
          typename DataType3>
void mm2(tensor<DataType0> &rst, DataType1 &left, DataType2 &right,
 DataType3 r_zp, bool result_add, data_type_t out_dtype);
```

$$\mathrm{dst}_{M\times N} = \mathrm{dst}_{M\times N} + \mathrm{left}_{M\times K} \times (\mathrm{right}_{K\times N} - \mathrm{zp})$$

$$\mathrm{dst}_{M\times N} = \mathrm{left}_{M\times K} \times (\mathrm{right}^{T}_{N\times K} - \mathrm{zp})$$

$$\mathrm{dst}^{T}_{N\times M} = \mathrm{dst}^{T}_{N\times M} + \mathrm{left}^{T}_{K\times M} \times (\mathrm{right}^{T}_{N\times K} - \mathrm{zp})$$

$$\mathrm{dst}(m,n) = \mathrm{dst}(m,n) + \sum_{k} \mathrm{left}(m,k) \times (\mathrm{right}(k,n) - \mathrm{zp})$$

参数

- dst(tensor): local memory 上的 dst 矩阵
- left(tensor|scalar): local memory 上的 left 矩阵或常数
- right(tensor|scalar): local memory 上的 right 矩阵或常数
- bias(tensor): local memory 上的 bias 矩阵或 nullptr
- r_zp(tensor|scalar): zero-point 的 tensor 或常数，若为 tensor，则右矩阵的元素按 column 减 zero-point
- ltrans(bool): 左矩阵转置标志
- rtrans(bool): 右矩阵转置标志
- rst_trans(bool): 结果矩阵转置标志，只有当 ltrans 和 rtrans 都为 true 时，rst_trans 为 true，其他情况都为 false
- result_add(bool): 对结果做累加的标志
- out_dtype(data_type_t): 结果矩阵的数据类型，如果为 DT_NONE 则与输出 tensor 类型一致，仅 **SG2380** 有效
- has_bias(bool): 是否累加 bias，当 bias 为 nullptr 时，无作用

注意事项

- dst、left 和 right 从 lane0 开始，都是 N-byte aligned layout。
- 如果 ltrans == rtrans == true，那么 dst 的 shape 必须是 [1, N, 1, M]，left 的 shape 必须是 [1, K, 1, M]，right 的 shape 必须是 [1, N, 1, K]。
- 如果 ltrans == false 且 rtrans == true，那么 dst 的 shape 必须是 [1, M, 1, N]，left 的 shape 必须是 [1, M, 1, K]，right 的 shape 必须是 [1, N, 1, K]。
- 如果 ltrans == false 且 rtrans == false dst 的 shape 必须是 [1, M, 1, N]，left 的 shape 必须是 [1, M, 1, K]，right 的 shape 必须是 [1, K, 1, N]。
- 当 zr_p 为 tensor 时，其 shape 是 [1, LANE_NUM, 1, N]，而不是 [1, 1, 1, N]，每个 lane 中都有一份相同的。
- M、K 和 N 的取值范围是 [1, 65535]。
- 输出矩阵的数据类型是 DT_INT32`和:cpp:enumerator:`DT_UINT32，左右矩阵的有效取值是 DT_INT8 和 DT_UINT8。
- 不支持单独转置左矩阵。

使用示例

```
#include "ppl.h"
using namespace ppl;
// zero-point 为常数
__KERNEL__ void mm2_int8_all_trans(int32 *ptr_res, int8 *ptr_left,
                                   int8 *ptr_right) {
  const int M = 64;
  const int K = 32;
  const int N = 96;
  int zp = 0;
  dim4 left_tensor_shape = {1, K, 1, M};
  dim4 left_stride = {M * K, M, M, 1};
  dim4 left_offset = {0, 0, 0, 0};
  dim4 right_tensor_shape = {1, N, 1, K};
  dim4 right_stride = {K * N, K, K, 1};
  dim4 right_offset = {0, 0, 0, 0};
  dim4 res_tensor_shape = {1, N, 1, M};
  dim4 res_stride = {M *N, M, M, 1};
  dim4 res_offset = {0, 0, 0, 0};

  auto left_gt = gtensor<int8>(left_tensor_shape, GLOBAL, ptr_left);
  auto right_gt = gtensor<int8>(right_tensor_shape, GLOBAL, ptr_right);
  auto res_gt = gtensor<int32>(res_tensor_shape, GLOBAL, ptr_res);

  auto left = tensor<int8>(left_tensor_shape);
  auto right = tensor<int8>(right_tensor_shape);
  auto res = tensor<int32>(res_tensor_shape);
  dma::load(left, left_gt);
  dma::load(right, right_gt);
  tiu::mm2(res, left, right, zp, true, true, false);
  dma::store(res_gt, res);
}
```

ppl::tiu::fmm2

两个矩阵相乘/左矩阵乘以右矩阵的转置/两个矩阵的转置相乘，结果也转置，再对结果做累加（可选），可对结果做 Relu（可选）。

```
template <typename DataType0, typename DataType1, typename DataType2,
        typename DataType3>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        DataType3 &bias, bool ltrans, bool rtrans, bool rst_trans,
        bool do_relu, bool result_add, data_type_t out_dtype, bool has_bias,
        bool saturate = false);

template <typename DataType0, typename DataType1, typename DataType2,
        typename DataType3>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        DataType3 &bias, bool result_add, data_type_t out_dtype,
        bool has_bias, bool saturate = false);

template <typename DataType0, typename DataType1, typename DataType2,
        typename DataType3>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        DataType3 &bias);

template <typename DataType0, typename DataType1, typename DataType2>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        bool ltrans, bool rtrans, bool rst_trans, bool do_relu,
        bool result_add, data_type_t out_dtype, bool saturate = false);

template <typename DataType0, typename DataType1, typename DataType2>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        bool ltrans, bool rtrans, bool rst_trans);

template <typename DataType0, typename DataType1, typename DataType2>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right,
        bool result_add, data_type_t out_dtype);

template <typename DataType0, typename DataType1, typename DataType2>
void fmm2(tensor<DataType0> &dst, DataType1 &left, DataType2 &right);
```

$$\mathrm{dst}_{M\times N} = \mathrm{dst}_{M\times N} + \mathrm{left}_{M\times K} \times \mathrm{right}_{K\times N}$$

$$\mathrm{dst}_{M\times N} = \mathrm{left}_{M\times K} \times \mathrm{right}^{T}_{N\times K}$$

$$\mathrm{dst}^{T}_{N\times M} = \mathrm{dst}^{T}_{N\times M} + \mathrm{left}^{T}_{K\times M} \times \mathrm{right}^{T}_{N\times K}$$

$$\mathrm{dst}(n,m) = \mathrm{dst}(n,m) + \sum_{k} \mathrm{left}(k,m) \times \mathrm{right}(n,k)$$

参数

- dst(tensor): local memory 上的 dst 矩阵
- left(tensor|scalar): local memory 上的 left 矩阵
- right(tensor|scalar): local memory 上的 right 矩阵
- bias(tensor): local memory 上的 bias
- ltrans(bool): 左矩阵转置标志
- rtrans(bool): 右矩阵转置标志
- rst_trans(bool): 结果矩阵转置标志，只有当 ltrans 和 rtrans 都为 true 时，rst_trans 为 true，其他情况都为 false

- doRelu(bool): 对结果做 ReLU 的标志
- result_add(bool): 对结果做累加的标志，如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_FP32，而不是 DT_FP16 或 DT_BFP16。
- out_dtype(data_type_t): 结果数据类型，为 0 表示与输入一致；如果结果在下一次运算时需要累加，则必须为 DT_FP32
- has_bias(bool): 是否有 bias
- **saturate(bool): 对结果做饱和;**

  当结果为 INT4/INT8/INT16 时:
  true：对称饱和（有符时: MIN_VAL+1 / MAX_VAL，无符时：MAX_VAL）
  false：普通饱和（有符时，MIN_VAL / MAX_VAL，无符时：MAX_VAL）

  当结果为 BF16/FP16 时:
  true: ±Inf 取最大/最小 normal 值，Nan 取 0 false: ±Inf、Nan

注意事项

- dst、left 和 right 从 LANE 0 开始，都是 N-byte aligned layout。
- 如果 ltrans == rtrans == true，那么 dst 的 shape 必须是 [1, N, 1, M]，left 的 shape 必须是 [1, K, 1, M]，right 的 shape 必须是 [1, N, 1, K]。
- 如果 ltrans == false 且 rtrans == true，那么 dst 的 shape 必须是 [1, M, 1, N]，left 的 shape 必须是 [1, M, 1, K]，right 的 shape 必须是 [1, N, 1, K]。
- 如果 ltrans == false 且 rtrans == false dst 的 shape 必须是 [1, M, 1, N]，left 的 shape 必须是 [1, M, 1, K]，right 的 shape 必须是 [1, K, 1, N]。
- M、K 和 N 的取值范围是 [1, 65535]。
- 输出矩阵的有效数据类型是 DT_FP32 或与输入矩阵相同，输入矩阵的有效取值是 DT_FP16 和 DT_BFP16。
- 如果对结果做累加，则累加前的 dst 的元素的数据类型会被认定为 DT_FP32，而不是 DT_FP16 或 DT_BFP16。
- 不支持单独转置左矩阵。
- 当 bias 为空时，has_bias 无效，当 bias 为 tensor 时，根据 has_bias 值决定是否对 bias 进行累加。
- saturate 仅对 **SG2380** 芯片生效

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void mm2_fp16(fp16 *ptr_res, fp16 *ptr_left, fp16 *ptr_right, int M,
              int K, int N) {
  const int block_m = 128;
  const int block_k = 256;
  const int block_n = 256;

  dim4 res_global_shape = {1, M, 1, N};
  dim4 left_global_shape = {1, M, 1, K};
  dim4 right_global_shape = {1, K, 1, N};

  auto res_gtensor = gtensor<fp16>(res_global_shape, GLOBAL, ptr_res);
  auto left_gtensor = gtensor<fp16>(left_global_shape, GLOBAL, ptr_left);
  auto right_gtensor = gtensor<fp16>(right_global_shape, GLOBAL, ptr_right);

  dim4 res_max_shape = {1, block_m, 1, block_n};
  dim4 left_max_shape = {1, block_m, 1, block_k};
```

(续下页)

(接上页)

```
  dim4 right_max_shape = {1, block_k, 1, block_n};
  tensor<fp16> res_fp16;
  for (auto idx_m = 0; idx_m < M; idx_m += block_m) {
    for (auto idx_n = 0; idx_n < N; idx_n += block_n) {
      int m = min(block_m, M - idx_m);
      int n = min(block_n, N - idx_n);
      dim4 res_shape = {1, m, 1, n};
      auto sub_res = make_tensor<fp32>(res_max_shape, res_shape);
      tiu::zero(sub_res);
      for (auto idx_k = 0; idx_k < K; idx_k += block_k) {
        enable_pipeline();
        int k = min(block_k, K - idx_k);
        dim4 left_max_shape = {1, block_m, 1, block_k};
        dim4 right_max_shape = {1, block_k, 1, block_n};
        dim4 left_real_shape = {1, m, 1, k};
        dim4 right_real_shape = {1, k, 1, n};

        tensor<fp16> sub_left, sub_right;
        dim4 left_offset = {0, idx_m, 0, idx_k};
        dim4 right_offset = {0, idx_k, 0, idx_n};
        dma::load(sub_left, left_gtensor.sub_view(left_real_shape, left_offset));
        dma::load(sub_right, right_gtensor.sub_view(right_real_shape, right_offset));
        tiu::fmm2(sub_res, sub_left, sub_right, true, DT_FP32);
      }
      tiu::cast(res_fp16, sub_res);
      dim4 res_offset = {0, idx_m, 0, idx_n};
      dma::store(res_gtensor.sub_view(res_shape, res_offset), res_fp16);
    }
  }
}
```

4.6.6 Pooling

ppl::tiu::pool_max

2D 最大池化。

```
template <typename DataType>
void pool_max(tensor<DataType> &dst, tensor<DataType> &src, dim2 *kernel,
              padding_t *pad, dim2 *stride, dim2 *dilation);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w],

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1,

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_INT8 和 DT_UINT8。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::fpool_max

2D 最大池化。

```
template <typename DataType>
void fpool_max(tensor<DataType> &dst, tensor<DataType> &src, dim2 *kernel,
               padding_t *pad, dim2 *stride, dim2 *dilation);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w]，

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_INT8 和 DT_UINT8。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。

使用示例

ppl::tiu::pool_min

2D 最小池化。

```
template <typename DataType>
void pool_min(tensor<DataType> &dst, tensor<DataType> &src, dim2 *kernel,
              padding_t *pad, dim2 *stride, dim2 *dilation);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w]，

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_INT8 和 DT_UINT8。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- **不支持 BM1684X**

使用示例

ppl::tiu::fpool_min

2D 最小池化。

```
template <typename DataType>
void fpool_min(tensor<DataType> &dst, tensor<DataType> &src, dim2 *kernel,
               padding_t *pad, dim2 *stride, dim2 *dilation);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w]，

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1，

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_INT8 和 DT_UINT8。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- **不支持 BM1684X**

使用示例

ppl::tiu::pool_avg

2D 均值池化，可自定义均值 scale 值，对结果做算术移位，结果有 saturation。

```
template <typename DataType0, typename DataType1>
void pool_avg(tensor<DataType0> &dst, tensor<DataType1> &src, dim2 *kernel,
              padding_t *pad, dim2 *stride, dim2 *dilation, dim2 *ins,
              int scale, int rshift);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小
- ins(dim2*): insert 大小
- scale(int): scale 值
- rshift(int): 右移位数

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w],

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1,

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8; DataType1 的有效取值是 DT_INT8 和 DT_UINT8 ; DataType0 和 DataType1 的符号相同。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- rshift 的取值范围是 [0, 31]。

使用示例

ppl::tiu::fpool_avg

2D 均值池化，可自定义均值 scale 值来代替传统的 1 / (kernel->h * kernel->w)。

```
template <typename DataType>
void fpool_avg(tensor<DataType> &dst, tensor<DataType> &src, dim2 *kernel,
               padding_t *pad, dim2 *stride, dim2 *dilation, dim2 *ins,
               float scale);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- kernel(dim2*): kernel 大小
- pad(padding_t*): pad 大小
- stride(dim2*): stride 大小
- dilation(dim2*): dilation 大小
- ins(dim2*): insert 大小
- scale(int): scale 值

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- output 的 shape 是 [input_n, input_c, output_h, output_w],

  output_h = (input_h + pad->top + pad->bottom - ((kernel->h - 1) * dilation->h + 1)) / stride->h + 1,

  output_w = (input_w + pad->left + pad->right - ((kernel->w - 1) * dilation->w + 1)) / stride->w + 1。
- input_n、input_c、input_h、input_w、kernel->h 和 kernel->w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。
- pad->top、pad->bottom、pad->left 和 pad->right 的取值范围是 [0, 15]，stride->h、stride->w、dilation->h 和 dilation->w 的取值范围是 [1, 15]。
- ins->h、ins->w 的取值范围是 [0, 8)。

使用示例

4.6.7 Dequant & Requant

ppl::tiu::dq0

反量化张量的元素。

```
template <typename DataType0, typename DataType1>
void dq0(tensor<DataType0> &dst, tensor<DataType1> &src, int offset,
         float scale, rounding_mode_t round_mode);
```

$$\mathsf{dst}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) = \mathbf{FP32}(\mathsf{src}(\mathsf{n},\mathsf{c},\mathsf{h},\mathsf{w}) - \mathsf{offset}) \times \mathsf{scale}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- offset(int): 补偿常数
- scale(float): 乘子常数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。

使用示例

ppl::tiu::dq0

按 channel 反量化张量的元素。

```
template <typename DataType0, typename DataType1, typename DataType2>
void dq0(tensor<DataType0> &dst, tensor<DataType1> &src,
         tensor<DataType2> &quant, rounding_mode_t round_mode);
```

$$\mathsf{dst}(n,c,h,w) = \textbf{FP32}(\mathsf{src}(n,c,h,w) - \mathsf{quant}(0,c,0,0)) \times \mathsf{quant}(0,c,0,1)$$

注意事项

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- quant(tensor): local memory 上的 quant 张量
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 quant 从同一个 LANE 开始，都是 N-byte aligned layout。
- quant 的 shape 是 [1, shape->c, 1, 2]，元素的数据类型是 DT_INT32 、DT_FP32，quant(0, c, 0, 0) 是补偿，数据类型是 DT_INT32 ，取值范围与 DataType1 的取值范围相同，quant(0, c, 0, 1) 是乘子，数据类型是 FP32。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。

使用示例

ppl::tiu::dq1

反量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1>
void dq1(tensor<DataType0> &dst, tensor<DataType1> &src, int offset,
         int multiplier, int shift, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) - \mathrm{offset}) \times \mathrm{multiplier} \quad \textbf{左移} \quad \mathrm{shift} & \text{如果 } \mathrm{shift} > 0 \\ (\mathrm{src}(n,c,h,w) - \mathrm{offset}) \times \mathrm{multiplier} \quad \textbf{右移} \quad -\mathrm{shift} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- offset(int): 补偿常数
- multiplier(int): 乘子常数
- shift(int): 移位数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16，DataType1 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，如果 DataType1 是有符号的，则 offset 的数据类型是 DT_INT16，否则是 DT_UINT16。
- dst_rounding_mode 的有效取值是 RM_HALF_TO_EVEN、RM_HALF_AWAY_FROM_ZERO、RM_TOWARDS_ZERO、RM_DOWN 和 RM_UP。
- shift 的取值范围是 [-64, 31]。

使用示例

ppl::tiu::dq1

按 channel 反量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2>
void dq1(tensor<DataType0> &dst, tensor<DataType1> &src,
         tensor<DataType2> &quant, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) - \mathrm{quant}(0,c,0,0)) \times \mathrm{quant}(0,c,0,1) & \textbf{左移} \quad \mathrm{quant}(0,c,0,2) \\ \text{如果 } \mathrm{quant}(0,c,0,2) > 0 & \\ (\mathrm{src}(n,c,h,w) - \mathrm{quant}(0,c,0,0)) \times \mathrm{quant}(0,c,0,1) & \textbf{右移} \quad -\mathrm{quant}(0,c,0,2) \quad \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- quant(tensor): local memory 上的 quant 张量
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 quant 从同一个 LANE 开始，都是 N-byte aligned layout。
- 当芯片类型是 **BM1684X** 时，quant 的 shape 是 [1, shape->c, 1, 3]，元素的数据类型是 DT_INT32，quant(0, c, 0, 0) 是补偿，如果 DataType1 是有符号的，则 quant(0, c, 0, 0) 的取值范围是 [-32768, 32767]，否则是 [0, 65535]，quant(0, c, 0, 1) 是乘子，quant(0, c, 0, 2) 是移位数，取值范围是 [-64, 31]。
- 当芯片类型是 **BM1690** 时，quant 的 shape 是 [1, shape->c, 1, 2]，元素的数据类型是 DT_INT32，quant(0, c, 0, 0) 是补偿，如果 DataType1 是有符号的，则 quant(0, c, 0, 0) 的取值范围是 [-32768, 32767]，否则是 [0, 65535]，quant(0, c, 0, 1) 的低位的 8 个 bit 是移位数，元素的数据类型是 DT_INT8，取值范围是 [-64, 31]，quant(0, c, 0, 1) 的高位的 16 个 bit 是乘子，元素的数据类型是 DT_INT16。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16，DataType1 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8。

使用示例

```
#include "ppl.h"

using namespace ppl;
MULTI_CORE
__KERNEL__ void dqpc_int_int16_int16(int16 *ptr_output, int16 *ptr_input,
                                     int32 *ptr_quant) {
  const int N = 4;
  const int C = 64;
  const int H = 64;
  const int W = 64;
  int core_num = get_core_num();
  int core_idx = get_core_index();
  if (core_idx >= core_num) {
    return;
  }
  int c_slice = div_up(C, core_num);
  int c_offset = core_idx * c_slice;
  c_slice = max(c_slice, (C - c_slice * (core_num - 1)));
```

(续下页)

(接上页)

```
  dim4 inp_shape = {N, c_slice, H, W};
  dim4 inp_stride = {C * H * W, H * W, W, 1};
  dim4 inp_offset = {0, c_offset, 0, 0};
  dim4 quant_shape = {1, c_slice, 1, 2};
  dim4 quant_stride = {2 * C, 2, 2, 1};
  rounding_mode_t round_mode = RM_DOWN;
  auto input = tensor<int16>(inp_shape);
  auto quant = tensor<int32>(quant_shape);
  auto output = tensor<int16>(inp_shape);

  auto g_in = gtensor<int16>(inp_shape, GLOBAL, ptr_input);
  auto g_quant = gtensor<int32>(quant_shape, GLOBAL, ptr_quant);
  auto g_out = gtensor<int16>(inp_shape, GLOBAL, ptr_output);

  dma::load(input, g_in);
  dma::load(quant, g_quant);
  tiu::dq1(output, input, quant, round_mode);
  dma::store(g_out, output);
}
// bm1684x
__TEST__ void __test__(){
  dim4 shape = {4, 64, 64, 64};
  auto dst = ppl::malloc<int16>(&shape);
  auto src = ppl::malloc<int16>(&shape);
  ppl::rand(src, &shape, 0, 32);

  dim4 quant_shape = {1, 64, 1, 3};
  dim4 rand_shape = {1, 64, 1, 1};
  dim4 quant_stride = {3 * 64, 3, 3, 1};
  auto quant = ppl::malloc<int32>(&quant_shape);

  ppl::rand(quant, &rand_shape, &quant_stride, -64, 64);
  dim4 offset1 = {0, 0, 0, 1};
  ppl::rand(quant, &rand_shape, &quant_stride, &offset1, -64, 64);
  dim4 offset2 = {0, 0, 0, 2};
  ppl::rand(quant, &rand_shape, &quant_stride, &offset2, -64, 32);

  dqpc_int_int16_int16(dst,src,quant);
}
// bm1690
__TEST__ void __test__() {
  dim4 shape = {4, 64, 64, 64};
  auto dst = ppl::malloc<int16>(&shape);
  auto src = ppl::malloc<int16>(&shape);
  ppl::rand(src, &shape, -32, 32);

  dim4 quant_shape = {1, 64, 1, 2};
  dim4 rand_shape = {1, 64, 1, 1};
  dim4 quant0_stride = {2 * 64, 2, 1, 1};
  auto quant = ppl::malloc<int32>(&quant_shape);

  ppl::rand(quant, &rand_shape, &quant0_stride, -32, 32);
  dim4 offset1 = {0, 0, 0, 3};
  dim4 quant1_stride = {64 * 4, 4, 1, 1 };

  dim4 offset2 = {0, 0, 0, 4};
  dim4 quant2_stride = { 64 * 8, 8, 1, 1 };
  ppl::rand((int16*)quant, &rand_shape, &quant1_stride,&offset1 ,-64, 64);
  ppl::rand((int8 *)quant, &rand_shape, &quant2_stride, &offset2, -16, 16);
```

(续下页)

(接上页)

```
  dqpc_int_int16_int16<8>(dst, src, quant);
}
```

ppl::tiu::dq2

反量化张量的元素，并将结果转为半精度浮点数；支持按照 gsize 进行反量化；支持使用反量化结果减去 offset，然后在使用 scale 做缩放。

```
template <typename DataType0, typename DataType1>
void dq2(tensor<DataType0> &dst, tensor<DataType1> &src,
         tensor<uint32> &offset_scale, int gsize);
```

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- offset_scale(tensor): offset 和 scale 张量
- gsize(int): group size 常数

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_FP16 和 DT_BFP16，DataType1 的有效取值是 DT_INT4、DT_UINT4、DT_INT8 和 DT_UINT8。
- offset_scale 数据类型为 DT_UINT32，由 offset 和 scale 组成；offset_scale 的前十六位为 offset，后十六位为 scale，两者的数据类型有效取值为 DT_FP16 和 DT_BFP16；offset_scale 的 layout 为 compact layout。

使用示例

ppl::tiu::rq0

重量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1>
void rq0(tensor<DataType0> &dst, tensor<DataType1> &src, float scale, float offset,
         rounding_mode_t dst_round_mode, rounding_mode_t src_round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \mathbf{INT}(\mathbf{FP32}(\mathrm{src}(n,c,h,w)) \times \mathrm{scale} + \mathrm{offset})$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- scale(float): 乘子常数
- offset(float): 补偿常数
- dst_round_mode(rounding_mode_t): 浮点数转化到 dst 的元素的舍入模式
- src_round_mode(rounding_mode_t): src 的元素转化到浮点数的舍入模式

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，DataType1 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16。
- dst_rounding_mode 的有效取值是 RM_HALF_TO_EVEN、RM_HALF_AWAY_FROM_ZERO、RM_TOWARDS_ZERO、RM_DOWN 和 RM_UP。

使用示例

```
#include "ppl.h"

using namespace ppl;
__KERNEL__ void rq_fp_int8_uint16(int8 *ptr_output, uint16 *ptr_input) {
  const int N = 20;
  const int C = 20;
  const int H = 20;
  const int W = 20;
  dim4 inp_shape = {N, C, H, W};
  float scale = 0.5;
  float offset = 0.5;
  rounding_mode_t output_round_mode = RM_TOWARDS_ZERO;
  rounding_mode_t input_round_mode = RM_TOWARDS_ZERO;
  tensor<uint16> input;
  tensor<int8> output;
  auto in_gt = gtensor<uint16>(inp_shape, GLOBAL, ptr_input);
  auto out_gt = gtensor<int8>(inp_shape, GLOBAL, ptr_output);
  dma::load(input, in_gt);
  tiu::rq0(output,input,scale,offset,output_round_mode,input_round_mode);
  dma::store(out_gt, output);
}
```

ppl::tiu::rq0

按 channel 重量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2>
void rq0(tensor<DataType0> &dst, tensor<DataType1> &src,
         tensor<DataType2> &quant, rounding_mode_t dst_round_mode,
         rounding_mode_t src_round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \mathbf{INT}(\mathbf{FP32}(\mathrm{src}(n,c,h,w)) \times \mathrm{quant}(0,c,0,0) + \mathrm{quant}(0,c,0,1))$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- quant(tensor): local memory 上的 quant 张量
- dst_rounding_mode(rounding_mode_t): 浮点数转化到 dst 的元素的舍入模式
- src_rounding_mode(rounding_mode_t): src 的元素转化到浮点数的舍入模式

注意事项

- dst、src 和 quant 从同一个 LANE 开始，都是 N-byte aligned layout。
- quant 的 shape 是 [1, shape->c, 1, 2]，元素的数据类型是 FP32，quant(0, c, 0, 0) 是乘子，quant(0, c, 0, 1) 是补偿。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，DataType1 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16。
- dst_rounding_mode 的有效取值是 RM_HALF_TO_EVEN、RM_HALF_AWAY_FROM_ZERO、RM_TOWARDS_ZERO、RM_DOWN 和 RM_UP。

使用示例

ppl::tiu::rq1

重量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1>
void rq1(tensor<DataType0> &dst, tensor<DataType1> &src, int multiplier,
         int shift, int offset, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) \times \mathrm{multiplier} \ \textbf{左移} \ \mathrm{shift}) + \mathrm{offset} & \text{如果 shift} > 0 \\ (\mathrm{src}(n,c,h,w) \times \mathrm{multiplier} \ \textbf{右移} \ -\mathrm{shift}) + \mathrm{offset} & \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- multiplier(int): 乘子常数
- shift(int): 移位数
- offset(int): 补偿常数
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst 和 src 从同一个 LANE 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，DataType1 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16，如果 DataType0 是有符号的，则 offset 的数据类型是 DT_INT16，否则是 DT_UINT16。
- shift 的取值范围是 [-64, 31]。

使用示例

```
#include "ppl.h"
using namespace ppl;

const int N = 200;
const int C = 20;
const int H = 20;
const int W = 20;

MULTI_CORE
__KERNEL__ void rq_int_int16_int16(int16 *ptr_output, int16 *ptr_input) {
  int core_num = get_core_num();
  int core_idx = get_core_index();
  if (core_idx >= core_num) {
    return;
  }
  int n_slice = div_up(N, core_num);
  int n_offset = core_idx * n_slice;
  int real_n_slice = min(n_slice, (N - n_offset));

  int multiplier = 10;
  int shift = 5;
  int offset = 10;
  dim4 max_inp_shape = {n_slice, C, H, W};
  dim4 inp_shape = {real_n_slice, C, H, W};
  dim4 inp_stride = {C*H*W, H*W, W, 1};
```

(续下页)

(接上页)

```
  dim4 inp_offset = {n_offset, 0, 0, 0};
  rounding_mode_t round_mode = RM_TOWARDS_ZERO;
  auto input = make_tensor<int16>(max_inp_shape, inp_shape);
  auto output = make_tensor<int16>(max_inp_shape, inp_shape);
  dma::load(input, ptr_input, &inp_offset, &inp_stride);
  tiu::rq1(output, input, multiplier, shift, offset, round_mode);
  dma::store(ptr_output, output, &inp_offset, &inp_stride);
}
```

ppl::tiu::rq1

按 channel 重量化张量的元素，结果有 saturation。

```
template <typename DataType0, typename DataType1, typename DataType2>
void rq1(tensor<DataType0> &dst, tensor<DataType1> &src,
         tensor<DataType2> &quant, rounding_mode_t round_mode);
```

$$\mathrm{dst}(n,c,h,w) = \begin{cases} (\mathrm{src}(n,c,h,w) \times \mathrm{quant}(0,c,0,0) \quad \textbf{左移} \quad \mathrm{quant}(0,c,0,1)) + \mathrm{quant}(0,c,0,2) \\ \text{如果 } \mathrm{quant}(0,c,0,1) > 0 \\ (\mathrm{src}(n,c,h,w) \times \mathrm{quant}(0,c,0,0) \quad \textbf{右移} \quad -\mathrm{quant}(0,c,0,1)) + \mathrm{quant}(0,c,0,2) \quad \text{其他情况} \end{cases}$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- quant(tensor): local memory 上的 quant 张量
- round_mode(rounding_mode_t): 右移舍入模式

注意事项

- dst、src 和 quant 从同一个 LANE 开始，都是 N-byte aligned layout。
- 当芯片类型是 **BM1684X** 时，quant 的 shape 是 [1, shape->c, 1, 3]，元素的数据类型是:cpp:enumerator:DT_INT32 , quant(0, c, 0, 0) 是乘子，quant(0, c, 0, 1) 是移位数，取值范围是 [-64, 31]，quant(0, c, 0, 2) 是补偿，如果 DataType0 是有符号的，则 quant(0, c, 0, 2) 的取值范围是 [-32768, 32767]，否则是 [0, 65535]。
- 当芯片类型是 **BM1690** 时，quant 的 shape 是 [1, shape->c, 1, 2]，元素的数据类型是 DT_INT32 ，quant(0, c, 0, 0) 是乘子，quant(0, c, 0, 1) 的低位的 8 个 bit 是移位数，元素的数据类型是 DT_INT8，取值范围是 [-64, 31]，quant(0, c, 0, 1) 的高位的 16 个 bit 是补偿，元素的数据类型是 DT_INT16。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType0 的有效取值是 DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8，DataType1 的有效取值是 DT_INT32、DT_INT16 和 DT_UINT16。

使用示例

```
#include "ppl.h"

using namespace ppl;
MULTI_CORE
__KERNEL__ void rqpc_int_int16_int16(int16 *ptr_output, int16 *ptr_input,
                                     int32 *ptr_quant) {
  const int N = 4;
  const int C = 64;
  const int H = 64;
  const int W = 64;

  int core_num = get_core_num();
  int core_idx = get_core_index();
  if (core_idx >= core_num) {
    return;
  }
  int c_slice = div_up(C, core_num);
  int c_offset = core_idx * c_slice;
  c_slice = max(c_slice, (C - c_slice * (core_num - 1)));
```

(续下页)

(接上页)

```
  dim4 inp_shape = {N, c_slice, H, W};
  dim4 inp_stride = {C * H * W, H * W, W, 1};
  dim4 inp_offset = {0, c_offset, 0, 0};
  dim4 quant_shape = {1, c_slice, 1, 2};
  dim4 quant_stride = {2 * C, 2, 2, 1};
  rounding_mode_t round_mode = RM_DOWN;
  auto input = tensor<int16>(inp_shape);
  auto quant = tensor<int>(quant_shape);
  auto output = tensor<int16>(inp_shape);
  dma::load(input, ptr_input, &inp_offset, &inp_stride);
  dma::load(quant, ptr_quant, &inp_offset, &quant_stride);
  tiu::rq1(output, input, quant, round_mode);
  dma::store(ptr_output, output, &inp_offset, &inp_stride);
}
// bm1684x
__TEST__ void __test__() {
  dim4 shape = {4, 64, 64, 64};
  auto dst = ppl::malloc<int16>(&shape);
  auto src = ppl::malloc<int16>(&shape);
  ppl::rand(src, &shape, 1, 2);

  dim4 quant_shape = {1, 64, 1, 3};
  dim4 rand_shape = {1, 64, 1, 1};
  dim4 quant_stride = {3 * 64, 3, 3, 1};
  auto quant = ppl::malloc<int32>(&quant_shape);

  ppl::rand(quant, &rand_shape, &quant_stride, 2, 3);
  dim4 offset1 = {0, 0, 0, 1};
  ppl::rand(quant, &rand_shape, &quant_stride, &offset1, 1, 2);
  dim4 offset2 = {0, 0, 0, 2};
  ppl::rand(quant, &rand_shape, &quant_stride, &offset2, 1, 2);
  rqpc_int_int16_int16(dst, src, quant);
}
// bm1690
__TEST__ void __test__() {
  dim4 shape = {4, 64, 64, 64};
  auto dst = ppl::malloc<int16>(&shape);
  auto src = ppl::malloc<int16>(&shape);
  ppl::rand(src, &shape, -32, 32);

  dim4 quant_shape = {1, 64, 1, 2};
  dim4 rand_shape = {1, 64, 1, 1};
  dim4 quant0_stride = {2 * 64, 2, 1, 1};
  auto quant = ppl::malloc<int32>(&quant_shape);

  ppl::rand(quant, &rand_shape, &quant0_stride, -32, 32);
  dim4 offset1 = {0, 0, 0, 3};
  dim4 quant1_stride = {64 * 4, 4, 1, 1 };

  dim4 offset2 = {0, 0, 0, 4};
  dim4 quant2_stride = { 64 * 8, 8, 1, 1 };
  ppl::rand((int16*)quant, &rand_shape, &quant1_stride,&offset1 ,-64, 64);
  ppl::rand((int8 *)quant, &rand_shape, &quant2_stride, &offset2, -16, 16);

  rqpc_int_int16_int16<8>(dst, src, quant);
}
```

4.6.8 NN

ppl::tiu::fadd_sqr

1. 张量的元素按 channel 加 bias，结果再平方。

```
template <typename DataType>
void fadd_sqr(tensor<DataType> &dst, tensor<DataType> &src,
              tensor<DataType> &bias);
```

$$dst(n, c, h, w) = (src(n, c, h, w) + bias(0, c, 0, 0))^2$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- bias(tensor): bias 张量

2. 张量的元素加常数，结果再平方。

```
template <typename DataType>
void fadd_sqr(tensor<DataType> &dst, tensor<DataType> &src, float bias_c);
```

$$dst(n, c, h, w) = (src(n, c, h, w) + bias_c)^2$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- bias_c(float): bias 常数

3. 常数加 bias，结果再平方。

```
template <typename DataType>
void fadd_sqr(tensor<DataType> &dst, float src_c, tensor<DataType> &bias);
```

$$dst(n, c, h, w) = (src_c + bias(0, c, 0, 0))^2$$

参数

- dst(tensor): 运算结果张量
- src_c(float): src 常数
- bias(tensor): bias 张量

4. 常数加常数，结果再平方。

```
template <typename DataType>
void fadd_sqr(tensor<DataType> &dst, float src_c, float bias_c);
```

$$dst(n, c, h, w) = (src_c + bias_c)^2$$

参数

- dst(tensor): 运算结果张量
- src_c(float): src 常数
- bias_c(float): bias 常数

注意事项

- dst、src 和 bias 从同一个 lane 开始。
- dst 和 src 是 N-byte aligned layout，bias 的 shape 是 [1, c, 1, 1]，compact layout。
- n、c、h 和 w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。
- **3 和 4 仅适用于 SG2380**。

使用示例

ppl::tiu::fscale

两个张量/常数相乘，结果再加第三个张量/常数。

```
template <typename DataType>
void fscale(tensor<DataType> &dst, tensor<DataType>/float &src,
            tensor<DataType>/float &scale, tensor<DataType>/float &bias);
```

$$\mathsf{dst}(n,c,h,w) = \mathsf{src}(n,c,h,w)/\mathsf{src} \times \mathsf{scale}(n,c,h,w)/\mathsf{scale} + \mathsf{bias}(n,c,h,w)/\mathsf{bias}$$

参数

- dst(tensor): 运算结果张量
- src(tensor/float): src 张量或常数
- scale(tensor/float): scale 张量或常数
- bias(tensor/float): bias 张量或常数

注意事项

- dst、src、scale 和 bias 从同一个 lane 开始。
- dst、src、scale 和 bias 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 dst 和 src 是张量，则此张量是 N-byte aligned layout；如果 scale 和 bias 是张量，则此张量是 compact layout。
- DataType 的有效取值是 DT_FP32、DT_FP16、DT_BFP16。

使用示例

ppl::tiu::fsub_sqr

1. 张量的元素按 channel 减 bias，结果再平方。

```
template <typename DataType>
void fsub_sqr(tensor<DataType> &dst, tensor<DataType> &src,
              tensor<DataType> &bias);
```

$$dst(n, c, h, w) = (src(n, c, h, w) - bias(0, c, 0, 0))^2$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- bias(tensor): bias 张量

2. 张量的元素减常数，结果再平方。

```
template <typename DataType>
void fsub_sqr(tensor<DataType> &dst, tensor<DataType> &src, float bias_c);
```

$$dst(n, c, h, w) = (src(n, c, h, w) - bias_c)^2$$

参数

- dst(tensor): 运算结果张量
- src(tensor): src 张量
- bias_c(float): bias 常数

3. 常数减 bias，结果再平方。

```
template <typename DataType>
void fsub_sqr(tensor<DataType> &dst, float src_c, tensor<DataType> &bias);
```

$$dst(n, c, h, w) = (src_c - bias(0, c, 0, 0))^2$$

参数

- dst(tensor): 运算结果张量
- src_c(float): src 常数
- bias(tensor): bias 张量

4. 常数减常数，结果再平方。

```
template <typename DataType>
void fsub_sqr(tensor<DataType> &dst, float src_c, float bias_c);
```

$$dst(n, c, h, w) = (src_c - bias_c)^2$$

参数

- dst(tensor): 运算结果张量
- src_c(float): src 常数
- bias_c(float): bias 常数

注意事项

- dst、src 和 bias 从同一个 lane 开始。
- dst 和 src 是 N-byte aligned layout，bias 的 shape 是 [1, c, 1, 1]，compact layout。
- n、c、h 和 w 的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16。
- **3 和 4 仅适用于 SG2380**。

使用示例

4.6.9 SFU

ppl::tiu::taylor

提取浮点数指数部分指令。指令对输入浮点 Tensor src 提取指数部分并转化为整型输出到 Tensor dst 中。

```
template <typename DataType>
void taylor(tensor<DataType> &dst, tensor<DataType> &src,
            tensor<DataType> &coeff, int len);
```

$$dst(n,c,h,w) = \sum_{i=0}^{num-1} coeff(0,c \mod NPU\_NUM,0,i) \times src(n,c,h,w)^i$$

参数

- dst(tensor): dst 张量
- src(tensor): src 张量
- coeff(tensor): coeff 张量
- len(int): taylor 级数的项数

注意事项

- dst 和 src 从同一个 lane 开始，都是 N-byte aligned layout。
- n、c、h 和 w 的取值范围是 [1, 65535]。
- len 大于等于 2。
- coeff 从 lane 0 开始，shape 是 [1, LANE_NUM, 1, len]，N-byte aligned layout。
- DataType 的有效取值是 DT_FP16、DT_BFP16、和 DT_FP32。

使用示例

ppl::tiu::norm

提取浮点数指数部分指令。指令对输入浮点 Tensor src 提取指数部分并转化为整型输出到 Tensor dst 中。

```
template <typename DataType0, typename DataType1>
void norm(tensor<DataType0> &dst, tensor<DataType1> &src);
```

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始，都是 N-byte aligned layout。
- shape->n、shape->c、shape->h 和 shape->w 的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_FP16、DT_BFP16、和 DT_FP32；当 DataType1 为 DT_FP16 或 DT_BFP16 时，DataType0 必须为 DT_INT16；当 DataType1 为 DT_FP32 时，DataType0 必须为 DT_INT32 。
- **目前仅支持 SG2380**。

使用示例

```
#include "ppl.h"
using namespace ppl;

__KERNEL__ void sfu_kernel(bf16 *ptr_rsqrt_res, int16 *ptr_norm_res,
                    int16 *ptr_idx_res, uint16 *ptr_cnt_res,
                    uint16 *ptr_clz_res, uint16 *ptr_clo_res,
                    bf16 *ptr_inp) {
 const int N = 32;
 const int C = 64;
 const int H = 1;
 const int W = 16;
 dim4 shape = {N, C, H, W};
 dim4 shape_cnt = {N, C, H, 1};
 tensor<bf16> rsqrt_res;
 tensor<int16> norm_res, sub_res, idx_res;
 tensor<uint16> cnt_res, clz_res, clo_res;
 tensor<bf16> inp;
 auto inp_g = gtensor<bf16>(shape, GLOBAL, ptr_inp);
 auto rsqrt_res_g = gtensor<bf16>(shape, GLOBAL, ptr_rsqrt_res);
 auto norm_res_g = gtensor<int16>(shape, GLOBAL, ptr_norm_res);
 auto idx_res_g = gtensor<int16>(shape, GLOBAL, ptr_idx_res);
 auto cnt_res_g = gtensor<uint16>(shape_cnt, GLOBAL, ptr_cnt_res);
 auto clz_res_g = gtensor<uint16>(shape, GLOBAL, ptr_clz_res);
 auto clo_res_g = gtensor<uint16>(shape, GLOBAL, ptr_clo_res);

 dma::load(inp, inp_g);
 tiu::frsqrt(rsqrt_res, inp, 3);
 dma::store(rsqrt_res_g, rsqrt_res);

 tiu::norm(norm_res, rsqrt_res);
 dma::store(norm_res_g, norm_res);

 tiu::sub(sub_res, norm_res, 131, 0, RM_HALF_TO_EVEN, false);
 tiu::nonzero(idx_res, cnt_res, sub_res);
 dma::store(idx_res_g, idx_res);
 dma::store(cnt_res_g, cnt_res);
```

(续下页)

(接上页)

```
  tiu::clz(clz_res, norm_res);
  tiu::clo(clo_res, norm_res);
  dma::store(clz_res_g, clz_res);
  dma::store(clo_res_g, clo_res);
}
```

ppl::tiu::frsqrt

浮点倒数平方根指令。指令对输入浮点 Tensor src 求倒数平方根计算，并输出结果到 Tensor dst 中。

```
template <typename DataType>
void frsqrt(tensor<DataType> &dst, tensor<DataType> &src, int num_iter);
```

参数

- dst(tensor): local memory 上的运算结果张量
- src(tensor): local memory 上的 src 张量
- num_iter(int): 牛顿迭代次数

注意事项

- dst 和 src 从同一个 lane 开始，都是 N-byte aligned layout。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType 的有效取值是 DT_FP32、DT_FP16 和 DT_BFP16 。
- **目前仅支持 SG2380**。

使用示例

具体使用方式见norm 示例。

ppl::tiu::clz

指令用于计算 Tensor src 元素从最高位到低位遇到第一个 1 所间隔的位数。并且支持当操作数 src 的 n、h 或 w 维大小为 1 时，广播计算。

```
template <typename DataType0, typename DataType1>
void clz(tensor<DataType0> &dst, tensor<DataType1>/DataType1 &src);
```

参数

- dst(tensor): local memory 上的运算结果张量
- src(tensor/scalar): local memory 上的 src 张量/常数

注意事项

- dst 和 src 从同一个 lane 开始，任意排列。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8；DataType1 的有效取值是 DT_UINT32 和 DT_UINT16、DT_UINT8 。
- **目前仅支持 SG2380**。

使用示例

具体使用方式见norm 示例。

ppl::tiu::clo

指令用于计算 Tensor src 元素从最高位到低位遇到第一个 0 所间隔的位数。并且支持当操作数 src 的 n、h 或 w 维大小为 1 时，广播计算。

```
template <typename DataType0, typename DataType1>
void clo(tensor<DataType0> &dst, tensor<DataType1>/DataType1 &src);
```

参数

- dst(tensor): local memory 上的运算结果张量
- src(tensor/scalar): local memory 上的 src 张量/常数

注意事项

- dst 和 src 从同一个 lane 开始，任意排列。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- DataType1 的有效取值是 DT_INT32、DT_UINT32、DT_INT16、DT_UINT16、DT_INT8 和 DT_UINT8；DataType1 的有效取值是 DT_UINT32 和 DT_UINT16、DT_UINT8 。
- **目前仅支持 SG2380**。

使用示例

具体使用方式见norm 示例。

ppl::tiu::smem_bcast

指令将 static_mem_mode 指向的 SMEM 上的数据复制到 dst 的每个 LANE 上。

```
template <typename DataType>
void smem_bcast(tensor<DataType> &dst, coeff_table_mode_t static_mem_mode);
```

参数

- dst(tensor): dst 张量
- static_mem_mode(coeff_table_mode_t): SMEM 上的数据 mode

注意事项

- dst 是 N-byte aligned layout，shape = [1, c, 1, w]，其中 c <= LANE_NUM。
- **目前仅支持 SG2380**。

使用示例

ppl::tiu::smem_dist

指令将 static_mem_mode 指向的 SMEM 上的数据分散到 dst 的每个 LANE 上。

```
template <typename DataType>
void smem_dist(tensor<DataType> &dst, coeff_table_mode_t static_mem_mode);
```

参数

- dst(tensor): dst 张量
- static_mem_mode(coeff_table_mode_t): SMEM 上的数据 mode

注意事项

- dst 是 N-byte aligned layout，shape = [1, c, 1, 1]。
- **目前仅支持 SG2380**。

使用示例

4.6.10 Other api

ppl::tiu::move

拷贝张量的元素。

```
template <typename DataType>
void move(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = src(n, c, h, w)$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- dst 和 src 从同一个 lane 开始。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- dst 和 src 的 layout 可以为任意 layout。

使用示例

略

ppl::tiu::broadcast

广播张量的元素到其他 lane。

```
template <typename DataType>
void broadcast(tensor<DataType> &dst, tensor<DataType> &src,
               int num = LANE_NUM);
```

$$dst(n, c, h, w) = src(n, 1, h, w)$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量
- num(int): C 维度广播数量，默认为 LANE_NUM

注意事项

- dst 和 src 可以不从同一个 lane 开始，都是 N-byte aligned layout。
- src 的 shape 是 [shape->n, 1, shape->h, shape->w]。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果 dst 从 lane X 开始，X 的取值范围是 [0, LANE_NUM - 1]，则 shape->c 小于等于 LANE_NUM - X。

使用示例

ppl::tiu::fill

将张量的元素置成常数。

```
template <typename DataType0, typename DataType1>
void fill(tensor<DataType0> &dst, DataType1 scalar);
```

$$\mathrm{dst}(n, c, h, w) = C$$

参数

- dst(tensor): local memory 上的 dst 张量
- C: 常数

注意事项

- dst 的 shape 每一个维度的取值范围是 [1, 65535]。
- 如果张量的 stride 指针是默认值，则此张量是 N-byte aligned layout，否则是 free layout。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void tiuSetC_int8(int8 *ptr_res, int8 *ptr_src) {
  const int N = 2;
  const int C = 8;
  const int H = 64;
  const int W = 128;

  dim4 shape = {N, C, H, W};
  dim4 set_shape = {1, 4, 16, 32};
  dim4 src_stride = {C*H*W, H*W, W, 1};
  auto g_src = gtensor<int8>(shape, GLOBAL, ptr_src);
  auto g_res = gtensor<int8>(shape, GLOBAL, ptr_res);
  auto src = tensor<int8>(shape);
  auto set_sub_src = src.view(set_shape, src_stride);

  dma::load(src, g_src);
  tiu::fill(set_sub_src, 66);
  dma::store(g_res, src);
}
```

ppl::tiu::transpose_cw

张量的 C 与 W 两个维度转置，推荐在 tensor C > W 时使用。

```
template <typename DataType>
void transpose_cw(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = src(n, w, h, c)$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- src 从 lane 0 开始，N-byte aligned layout，dst 是 line N-byte aligned layout。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。

使用示例

```
#include "ppl.h"

using namespace ppl;

__KERNEL__ void cw_trans_int16(int16 *ptr_res, int16 *ptr_inp) {
  const int N = 8;
  const int C = 32;
  const int H = 16;
  const int W = 64;
  dim4 res_shape = {N, C, H, W};
  dim4 inp_shape = {N, W, H, C};
  auto inp = tensor<int16>(inp_shape);
  auto res = tensor<int16>(res_shape);

  dma::load(inp, ptr_inp);
  dma::load(res, ptr_res);
  tiu::transpose_cw(res, inp);
  dma::store(ptr_res, res);
}
```

ppl::tiu::transpose_wc

张量的 W 与 C 两个维度转置，推荐在 tensor W > C 时使用。

```
template <typename DataType>
void transpose_wc(tensor<DataType> &dst, tensor<DataType> &src);
```

$$dst(n, c, h, w) = src(n, w, h, c)$$

参数

- dst(tensor): local memory 上的 dst 张量
- src(tensor): local memory 上的 src 张量

注意事项

- src 从 lane 0 开始，N-byte aligned layout，dst 是 line N-byte aligned layout。
- dst 和 src 的 shape 每一个维度的取值范围是 [1, 65535]。

使用示例

使用方式与transpose_cw 一致。

ppl::tiu::arange_broadcast

生成等差数列，C 维度广播到 LANE_NUM。

```
template <typename DataType>
void arange_broadcast(tensor<DataType> &dst, int start, int step, int num);
```

$$dst(0, c, 0, w) = start + step \times w$$

参数

- dst(tensor): local memory 上的 dst 张量
- start: 常数, 等差数列的首项
- step: 常数, 等差数列的步长
- num: 常数, 等差数列的项数

注意事项 - dst 的 shape 是 [1, LANE_NUM, 1, num]，line N-byte aligned layout，元素的数据类型是 INT32。- dst 从 LANE 0 开始。

使用示例

```
#include "ppl.h"
using namespace ppl;
__KERNEL__ void function_arange_broadcast(int32* result) {
  int start = 0;
  int end   = 100;
  int step  = 1;

  int num = end - start;//[start,end)
  dim4 shape_local_A = {1, LANE_NUM, 1, num};
  auto local_A = tensor<int32>(shape_local_A);
  tiu::arange_broadcast(local_A, start, step, num);

  dim4 shape_result = {1, 1, 1, num};
  dim4 offset_sub_local_A = {0,0,0,0};
  auto sub_local_A = local_A.sub_view(shape_result, offset_sub_local_A);

  auto r_gtensor = gtensor<int32>(shape_result, GLOBAL, result);
  dma::store(r_gtensor, sub_local_A);
}
```

4.7 Wrapper Func

4.7.1 sigmoid_fp32

sigmoid 激活函数，仅支持 float 类型。

```
void sigmoid_fp32(tensor<fp32> &local_output, tensor<fp32> &local_input,
                  dim4 *shape);
```

参数

- local_output(tensor): output 张量
- local_input(tensor): input 张量
- shape(dim4*): 张量 shape 大小

4.7.2 softsign_fp32

Softsign 激活函数，仅支持 float 类型。

```
template <typename DataType>
void softsign_fp32(tensor<DataType> &out, tensor<DataType> &in,
                   dim4 *block_shape, dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.3 softplus_f32

Softplus 激活函数，仅支持 float 类型。

```
template <typename DataType>
void softplus_f32(tensor<fp32> &out, tensor<fp32> &in, float beta,
                  dim4 *shape, dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- beta(float): beta 参数
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.4 prelu

PRelu 激活函数。

```
template <typename DataType>
void prelu(tensor<DataType> &out, tensor<DataType> &in, float alpha);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- alpha(float): alpha 参数

4.7.5 relu

Relu 激活函数。

```
template <typename DataType>
void relu(tensor<DataType> &out, tensor<DataType> &in);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量

4.7.6 exp_no_overflow

无数据溢出的 exp 指数计算。

```
template <typename DataType>
void exp_no_overflow(tensor<DataType> &out, tensor<DataType> &in, dim4 *shape,
                     dim4 *real_shape);

template <typename DataType>
void exp_no_overflow(tensor<DataType> &out, tensor<DataType> &in, dim4 *shape);
```

参数

- outout(tensor): output 张量
- in(tensor): input 张量
- shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 shape。

4.7.7 quick_pooling

高性能池化计算。

```
template <typename DataType>
void quick_pooling(tensor<DataType> &out_tensor, tensor<DataType> &in_tensor,
                   dim4 *in_block_shape, dim4 *in_real_shape, float fill,
                   int mode, float scale = 1.0);
```

参数

- out_tensor(tensor): output 张量
- in_tensor(tensor): input 张量
- in_block_shape(dim4*): 张量空间大小
- in_real_shape(dim4*): 真实的张量 shape 大小
- fill(float): pad 填充值
- mode(int): 池化类型
- scale(float): 平均池化的除数

注意事项

- in_real_shape 在所有维度上必须小于或等于 in_block_shape

4.7.8 fsqrt

平方根计算，仅支持浮点运算。

```
template <typename DataType>
void fsqrt(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
           dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.9 flog

自然对数计算，仅支持浮点运算。

```
template <typename DataType>
void flog(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
          dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.10 fsin

正弦计算，仅支持浮点运算。

```
template <typename DataType>
void fsin(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
          dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.11 farcsin

反正弦计算，仅支持浮点运算。

```
template <typename DataType>
void farcsin(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
             dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.12 sinh_fp32

双曲正弦计算，仅支持 float 类型。

```
template <typename DataType>
void sinh_fp32(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
               dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.13 arcsinh_fp32

反双曲正弦计算，仅支持 float 类型。

```
template <typename DataType>
void arcsinh_fp32(tensor<DataType> &out, tensor<DataType> &in,
                  dim4 *block_shape, dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.14 fcos

余弦计算，仅支持浮点运算。

```
template <typename DataType>
void fcos(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
          dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.15 farccos

反余弦计算，仅支持浮点运算。

```
template <typename DataType>
void farccos(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
             dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.16 cosh_fp32

双曲余弦计算，仅支持 float 类型。

```
template <typename DataType>
void cosh_fp32(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
               dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.17 arccosh_fp32

反双曲余弦计算，仅支持 float 类型。

```
template <typename DataType>
void arccosh_fp32(tensor<DataType> &out, tensor<DataType> &in,
                  dim4 *block_shape, dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.18 ftan

正切计算，仅支持浮点运算。

```
template <typename DataType>
void ftan(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
          dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.19 tanh_fp32

双曲正切计算，仅支持 float 类型。

```
template <typename DataType>
void tanh_fp32(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
               dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.20 arctanh_fp32

反双曲正切计算，仅支持 float 类型。

```
template <typename DataType>
void arctanh_fp32(tensor<DataType> &out, tensor<DataType> &in,
                  dim4 *block_shape, dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.7.21 fcot

余切计算，仅支持浮点运算。

```
template <typename DataType>
void fcot(tensor<DataType> &out, tensor<DataType> &in, dim4 *block_shape,
          dim4 *real_shape);
```

参数

- out(tensor): output 张量
- in(tensor): input 张量
- block_shape(dim4*): 张量空间大小
- real_shape(dim4*): 真实的张量 shape 大小

注意事项

- real_shape 在所有维度上必须小于或等于 block_shape。

4.8 Kernel Func Utils

4.8.1 基础操作指令

ppl::div_up

向上取整。

```
template <typename U, typename V>
int div_up(U numerator, V denominator) {
  return (numerator + denominator - 1) / denominator * denominator;
}
```

ppl::align

对齐操作。

```
template <typename U, typename V>
int align(U numerator, V denominator) {
  return (numerator + denominator - 1) / denominator;
}
```

ppl::swap

交换操作。

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

ppl::max/min

取大取小操作。

```
int min(int, int);
int max(int, int);
```

ppl::convert_dtype

将 host 端的数据类型转换成对应的 device 端数据类型。

```
template<typename T>
data_type_t convert_dtype();
```

具体对应关系如下表：

表 3: 数据类型对应关系

host	int32	uint32	int16	uint16	int8
device	DT_INT32	DT_UINT32	DT_INT16	DT_UINT16	DT_INT8
host	uint8	int64	uint64	float	bf16
device	DT_UINT8	DT_INT64	DT_UINT64	DT_FP32	DT_BF16
host	fp16	fp8e4m3	fp8e5m2	fp20	
device	DT_FP16	DT_FP8E4M3	DT_FP8E5M2	DT_FP20	

ppl::assert

安全性检查。

```
template <typename DataType>
void assert(DataType condition);
```

ppl::enable_pipeline

开启 ppl 流水优化的指令，写在需要进行流水优化的 for 循环内部。

```
void enable_pipeline();
```

ppl::print

打印 tensor 和变量的信息，打印 tensor 信息时需要配合 to_string 使用。

```
template <typename... Args>
void print(const char *format, Args... args);
```

使用示例

```
#include "ppl.h"
using namespace ppl;

int real_m = min(block_m, qm - _m);
dim4 qi_real_local_shape = {real_h, real_m, 1, cur_d};
dim4 qi_offset = {_h, _m, _b, 0};
tensor<T> qi_tensor;
dma::load(qi_tensor,
          q_global_tensor.sub_view(qi_real_local_shape, qi_offset));
ppl::print("_b: %d, _h: %d, _m %d, qi_tensor: %s\n",
           _b, _h, _m, to_string(qi_tensor));
```

ppl::to_string

将 tensor 的信息转换为 char* 类型的数据，以供打印。

```
template <typename DataType>
char *to_string(tensor<DataType> &src);

template <typename DataType>
char *to_string(gtensor<DataType> &src);
```

ppl::lane_num

获取当前芯片的 lane 个数。

```
int lane_num();
```

ppl::get_eu_num

获取当前数据类型 (DataType) 所对应的 eu 数。

```
template <typename DataType>
int get_eu_num();
```

ppl::get_nic

获取当前数据类型 (DataType) 应该对齐的 ic 数。

```
template <typename DataType>
int get_nic();
```

ppl::vset

设置 vector 寄存器的使用状态；**该指令仅支持 SG2380**。

```
void vset(int ew, int lmul, int v_len);
```

参数

- ew(int): vector 寄存器中储存的数据的位数
- lmul(int): 绑定使用的 vector 寄存器个数
- v_len(bool): 每个 vector 寄存器上存储的元素个数

注意事项

- ew 的可取值为 8、16、32。
- 使用 vector 寄存器时，其索引必须是 vset 设置中 lmul 的整数倍。

ppl::lane_mask

TIU LANE 写屏蔽指令，mask 每个 bit 代表 1 个 Lane，当 bit 为 0 时则屏蔽 TIU 写对应的 Lane; long_valid 为 1 则，mask 将影响后续所有的 TIU 指令；如果为 0 则只影响后续的第一条 TIU 指令；**该指令仅支持 SG2380**。

```
void lane_mask(int mask, bool long_valid);
```

4.8.2 tensor 相关指令

ppl::get_stride

按照不同的 layout 计算 tensor 的 stride。

```
void get_stride(dim4 *stride, dim4 *shape, align_mode_t mode, int eu_num = 0,
                int start_idx = 0);
```

参数

- stride(dim4*): stride 结果
- shape(dim4*): shape 大小
- mode(align_mode_t): layout 模式
- eu_num(int): eu_num 数量
- start_idx(int): 开始计算的 lane 索引

ppl::dim4_index

按照 index 获取 dim4 对象中的数据。

```
int dim4_index(const dim4* d, int index);
```

ppl::make_gtensor_permute

根据 order 的设定顺序，修改 mem_shape，并根据修改后的 shape 生成 global/L2 mem 上的 tensor；order 默认 mem_shape 不变。

```
template <typename dtype>
gtensor<dtype> &make_gtensor_permute(dim4 &mem_shape, tensor_mode_t mode,
                                     dtype *addr, int order[4]);
template <typename dtype>
gtensor<dtype> &make_gtensor_permute(dim4 &mem_shape, tensor_mode_t mode,
                                     dtype *addr);
```

ppl::get_gmem_addr

获取 gtensor 的地址。

```
template <typename DataType>
int64 get_gmem_addr(gtensor<DataType> &src);
```

ppl::get_value

获取当前地址空间的一个元素，配合 get_gmem_addr 使用。

```
template <typename DataType>
DataType get_value(int64 gaddr);
```

4.8.3 多核相关指令

ppl::set_core_num

设置在 num 个核上并行运行 kernel 函数，通常与 get_core_num 和 get_core_index 两个指令配合使用。

```
void set_core_num(int num);
```

ppl::get_core_num

在 kernel 函数中获取使用了多少个核，通常与 set_core_num 和 get_core_index 两个指令配合使用。

```
int get_core_num();
```

ppl::get_core_index

在 kernel 函数中获取当前程序所运行核的索引，通常与 get_core_num 和 get_core_index 两个指令配合使用。

```
int get_core_index();
```

ppl::sync

同步所有设备。

```
void sync();
```

ppl::hau_poll

阻塞设备，直至该语句之前的 hau 操作结束。

```
void hau_poll();
```

ppl::tpu_poll

阻塞设备，直至该语句之前的所有操作结束。

```
void tpu_poll();
```

ppl::msg_send

通过消息同步机制进行多核间的同步操作，**该指令仅支持 SG2380**。

```
void msg_send(int msg_idx, int wait_cnt, bool is_dma);
```

参数

- msg_idx(int): message index
- wait_cnt(int): 等待该 message index 的操作个数
- is_dma(bool): 是否应用于 dma

ppl::msg_wait

通过消息同步机制进行多核间的同步操作，**该指令仅支持 SG2380**。

```
void msg_wait(int msg_idx, int send_cnt, bool is_dma);
```

参数

- msg_idx(int): message index
- send_cnt(int): 需要等待该 message index 发送的次数
- is_dma(bool): 是否应用于 dma

ppl::fence

用于保证该指令前序所有 MOV 读写指令 (load/store/cache) 比该指令后序所有 MOV 读写指令更早被观察到，**该指令仅支持 SG2380**。

```
void fence();
```

4.9 Test Func Utils

4.9.1 内存分配

ppl::malloc

按照 shape 大小，分配 global mem 上的连续地址空间。

```
template <typename DataType>
DataType *malloc(dim4 *shape);
```

4.9.2 随机填充

ppl::rand

1. 给已分配的 global mem 上的连续地址空间填充随机数。

```
template <typename DataType0, typename DataType1>
void rand(DataType0 *ptr, dim4 *shape, dim4 *stride, dim4 *offset,
          DataType1 min_val, DataType1 max_val);

template <typename DataType0, typename DataType1>
void rand(DataType0 *ptr, dim4 *shape, dim4 *stride, DataType1 min_val,
          DataType1 max_val);

template <typename DataType0, typename DataType1>
void rand(DataType0 *ptr, dim4 *shape, DataType1 min_val, DataType1 max_val);
```

参数

- ptr(DataType0*): global mem 上的连续地址空间
- shape(dim4*): shape 大小
- stride(dim4*): stride 大小
- offset(dim4*): offset 大小
- min_val(DataType1): 随机数下限
- max_val(DataType1): 随机数上限

2. 分配 global mem 上的连续地址空间，并填充随机数。

```
template <typename DataType0, typename DataType1>
DataType0 *rand(dim4 *shape, dim4 *stride, dim4 *offset, DataType1 min_val,
                DataType1 max_val);

template <typename DataType0, typename DataType1>
DataType0 *rand(dim4 *shape, dim4 *stride, DataType1 min_val, DataType1 max_val);

template <typename DataType0, typename DataType1>
DataType0 *rand(dim4 *shape, DataType1 min_val, DataType1 max_val);

template <typename DataType> DataType *rand(dim4 *shape);
```

参数

- shape(dim4*): shape 大小
- stride(dim4*): stride 大小
- offset(dim4*): offset 大小
- min_val(DataType1): 随机数下限
- max_val(DataType1): 随机数上限

注意事项

- 第 1 种 rand 指令需要配合 malloc 使用；第二种可以直接使用。
- 如果仅使用 malloc 不使用 rand 进行随机数填充，则内存地址中的元素全为 0 。
- DataType0 可以是任意 PPL 已支持的数据类型；**DataType1 不能设置为 fp16 和 bf16 类型**。

· 随机数的取值范围为 [min_val, max_val) 。

使用示例

```
__TEST__ void fconv2d_main() {
  int N = 1;
  int IC = 3;
  int H = 224;
  int W = 224;
  int OC = 64;
  int kh = 7;
  int kw = 7;
  int pad_t = 3;
  int pad_b = 3;
  int pad_l = 3;
  int pad_r = 3;
  int sh = 2;
  int sw = 2;
  int dh = 1;
  int dw = 1;

  int oh = 112;
  int ow = 112;
  int nic = get_nic<fp16>();
  dim4 res_shape = {N, OC, oh, ow};
  auto res = ppl::malloc<fp16>(&res_shape);
  // 等价于
  // auto res = ppl::rand<fp16>(&in_shape);

  dim4 in_shape = {N, IC, H, W};
  auto in = ppl::malloc<fp16>(&in_shape);
  ppl::rand(in, &in_shape, -1.f, 1.f);
  // 等价于
  // auto in = ppl::rand<fp16>(&in_shape, -1.f, 1.f);

  dim4 w_shape = {1, OC, div_up(IC, nic) * kh * kw, nic};
  auto w = ppl::malloc<fp16>(&w_shape);
  ppl::rand(w, &w_shape, -1.f, 1.f);
  // 等价于
  // auto w = ppl::rand<fp16>(&w_shape, -1.f, 1.f);

  dim4 b_shape = {1, OC, 1, 1};
  auto b = ppl::malloc<fp32>(&b_shape);
  ppl::rand(b, &b_shape, -1.f, 1.f);
  // 等价于
  // auto b = ppl::rand<fp16>(&b_shape, -1.f, 1.f);

  fconv2d(res, in, w, b, N, IC, H, W, OC, true, false, kh, kw,
          sh, sw, dh, dw, pad_t, pad_b, pad_l, pad_r);
}
```

4.9.3 数据读取

ppl::read_npy

读取 npy 类型文件中的数据，并存入 global mem 中。

```
template <typename DataType>
void read_npy(DataType *dst, const char *file_path);
```

参数

- dst(DataType*): global mem 上的地址空间
- file_path(const char*): npy 文件所在路径

注意事项

- file_path 可以使用相对路径和绝对路径。
- DataType 可以与 npy 文件中存储的数据类型不一致，ppl 将自动进行数据类型转换。

使用示例

```
__TEST__ void read_float() {
  dim4 shape = {N, C, H, W};
  auto dst0 = ppl::malloc<fp16>(&shape);
  auto src0 = ppl::malloc<fp16>(&shape);
  auto src1 = ppl::malloc<fp16>(&shape);
  auto src2 = ppl::malloc<fp16>(&shape);
#if 0
  read<fp16, fp32>(
    src0, "<path to ppl>/ppl/regression/unittest/fileload/data/read_float_fp32_1.in");
  read<fp16, fp32>(
    src1, "<path to ppl>/ppl/regression/unittest/fileload/data/read_float_fp32_2.in");
  read<fp16, fp32>(
    src2, "<path to ppl>/ppl/regression/unittest/fileload/data/read_float_fp32_3.in");
#else
  read_npy(src0, "./data/data1.npy");
  read_npz(src1, "./data/data2.npz", "xyz");
  read_bin<fp16, fp32>(src2, "./data/read_float_fp32_1.in");
#endif

  add_float(dst0, src0, src1, src2);
}
```

ppl::read_npz

读取 npz 类型文件中的数据，并存入 global mem 中。

```
template <typename DataType>
void read_npz(DataType *dst, const char *file_path, const char *tensor_name);
```

参数

- dst(DataType*): global mem 上的地址空间
- file_path(const char*): npz 文件所在路径
- tensor_name(const char*): npz 文件中所需读取的数据块名称

注意事项

- file_path 可以使用相对路径和绝对路径。
- DataType 可以与 npz 文件中存储的数据类型不一致，ppl 将自动进行数据类型转换。

使用示例

具体使用方式如read_npy 所示。

ppl::read_bin

读取二进制 (binary) 文件中的数据，并存入 global mem 中。

```
template <typename DstType, typename FileType>
void read_bin(DstType *dst, const char *file_path, FileType file_dtype);
```

参数

- dst(DataType*): global mem 上的地址空间
- file_path(const char*): binary 文件所在路径
- file_dtype(FileType): binary 文件中所需读取的数据类型

注意事项

- file_path 可以使用相对路径和绝对路径。
- DataType 可以与 file_dtype 不一致，ppl 将自动进行数据类型转换。

使用示例

具体使用方式如read_npy 所示。