OPLib kernel list

Blas

L1

dot

tile_dot_f16

介绍：

两个输入向量点乘输出一个元素，输入输出均为float16数据，累加精度为float32。

函数声明：

void dot_f16(SiDeviceHandle dev, void *A, void *B, void *C, int size);

dev为device handle. A，B为输入向量指针， C为输出指针。 size为输入向量的长度，单位为1.（此函数底层由RVV实现，因此size没有颗粒度要求，按1对齐）

L2

gemv

gemv_f32_f16_f16

介绍：

Y = alpha * OP(A) * V + beta * Y

函数声明：

```
void gemv_f32_f16_f16(SiDeviceHandle dev,
        uint8_t trans, // not used for now)
        int N,
        int K,
        void *alpha,
        void *pA,
        int lda, // not used for now
        void *pV,
```

int incV, // not used for now

void *beta,

void *pY,

int incY);

此函数为输入float16，输出float32的gemv实现，输入数据直接读取自global memory，其中pA为矩阵，shape为(N,K)，tile format；pV是向量，shape为(K, 1)，linear format；结果为pY，shape为(N, 1)，linear format

数据对齐要求：

N要求32对齐，K方向上128B对齐，对于输入为f16而言，也就是K要求64对齐

Data layout要求：

pA为矩阵，tile format，shape为((32,N/32),(16,K/16)) ，stride为((16,32*K),(1,32*16))

pV和pY是linear format

gemv_f32_bf16_bf16

同上，为bf16的版本

L3

mma

mma_f16_m32:

介绍：

此函数为float16数据格式的mma实现，输入数据直接读取自global memory，输入输出数据排布为32x32B的tile格式。根据用户输入的数据size底层会自动分配不同的线程数，目前支持一个PEG 32个线程。

函数声明：

void mma_f16_m32(void*A, void *B, void *C, int m, int n, int k);

A， B， C分别为输入与输出矩阵的指针，m，n，k为矩阵size， A矩阵size为mxk，B矩阵size为kxn。

数据对齐要求：

m is 32 aligned, >=32; n is 32 aligned, >=32; K is 16 aligned, >=16

Data layout要求：

A's shape is ((32,m/32),(16,k/16)); A's stride is ((16,32*k),(1,32*16))

B's shape is ((32,n/32),(16,k/16)); B's stride is ((16,32*k),(1,32*16))

注意，B矩阵的size为kxn，但是矩阵乘法单元计算式默认对B矩阵进行了转置，因此B矩阵的输入排布为转置后的样子。这样在经过矩阵乘法单元的默认转置操作后，执行计算才为逻辑上的矩阵AxB。

mma_bf16_m32:

介绍：同上，精度为bf16的版本

mma_f16_acc16_m32:

介绍：

同上，区别是关闭了reuseD功能，即累加数据不使用f32存储而是截成f16的数据精度，每次做mma都从D寄存器读取累加结果并写回到D寄存器。

mma_bf16_acc16_m32:

介绍：同上

tile_mma_32x32_f16_f16_f16_tiled_tensor_smem_simple_demo:

介绍：

此函数仅用于验证shared memory的软件支持情况，函数仅调用一个线程，做矩阵乘法时，先将B矩阵从global memory中搬运到shared memory中，再从shared memory中load到寄存器进行计算。

mma_f16_m16:

介绍：

此函数为float16数据格式的mma实现，输入数据直接读取自global memory，输入输出数据排布为16x64B的tile格式。根据用户输入的数据size底层会自动分配不同的线程数，目前最支持一个PEG 32个线程。

函数声明：

void mma_f16_m16(void*A, void *B, void *C, int m, int n, int k);

A， B， C分别为输入与输出矩阵的指针，m，n，k为矩阵size， A矩阵size为mxk，B矩阵size为kxn。

数据对齐要求：

m =16; n is 32 aligned, >=32; K is 32 aligned, >=32

Data layout要求：

A's shape is ((16,1),(32,k/32)); A's stride is ((16,16*k),(1,16*32))

B's shape is ((32,n/32),(16,k/16)); B's stride is ((16,32*k),(1,32*16))

注意，B矩阵的size为kxn，但是矩阵乘法单元计算式默认对B矩阵进行了转置，因此B矩阵的输入排布为转置后的样子。这样在经过矩阵乘法单元的默认转置操作后，执行计算才为逻辑上的矩阵AxB

mma_bf16_m16:

介绍：同上，输入类型为fp16

mma_f32_f16_m32:

介绍：mma_f16_m32，输入类型为fp16, 输出类型为f32

mma_f32_bf16_m16:

介绍：同mma_bf16_m16，输入类型为bf16, 输出类型为f32

mma_f32_f16_m16:

介绍：同mma_bf16_m16，输入类型为f16, 输出类型为f32

mma_f32_bf16_m32:

介绍：同mma_f16_m32，输入类型为bf16, 输出类型为f32

mma_f32_bf16_m8:

介绍：

此函数为bfloat16数据格式的mma实现，输入数据直接读取自global memory，输入输出数据排布为8x128B的tile格式，输出类型为f32。根据用户输入的数据size底层会自动分配不同的线程数，目前最支持一个PEG 32个线程。

函数声明：

void mma_f32_bf16_m8(void*A, void *B, void *C, int m, int n, int k);

A， B， C分别为输入与输出矩阵的指针，m，n，k为矩阵size， A矩阵size为mxk，B矩阵size为kxn。

数据对齐要求：

m =8; n is 32 aligned, >=32; K is 64 aligned, >=64

Data layout要求：

A's shape is ((8,1),(64,k/32)); A's stride is ((16,8*k),(1,8*64))

B's shape is ((32,n/32),(16,k/16)); B's stride is ((16,32*k),(1,32*16))

注意，B矩阵的size为kxn，但是矩阵乘法单元计算式默认对B矩阵进行了转置，因此B矩阵的输入排布为转置后的样子。这样在经过矩阵乘法单元的默认转置操作后，执行计算才为逻辑上的矩阵AxB

mma_f32_f16_m8:

介绍：同上，输入类型为f16, 输出类型为f32

mma_dte:

介绍：支持的输入数据类型为fp16和bf16，输出类型支持fp16、bf16、fp32

支持的输入A矩阵shape为R8，R16，R32，支持的输入和输出format为linear和tiled

原型：

template <class inputT, class outputT, int tensor_format_in = 0, int tensor_format_out = 1> //T:input date type; 0:linear, 1:tiled

void mma_dte(void *A, void *B, void *C, int M, int N, int K);

mma_bf16_m8:

介绍：同mma_f32_bf16_m8，输入类型为bf16, 输出类型为bf16

mma_bf16_universal:

介绍：输入类型为bf16, 输出类型为bf16，同时支持A矩阵为R8、R16和R32

mma_f16_m8:

介绍：同mma_f32_bf16_m8，输入类型为f16, 输出类型为f16

mma_f16_universal:

介绍：输入类型为f16, 输出类型为f16，同时支持A矩阵为R8、R16和R32

mma_f32_bf16_universal:

介绍：输入类型为bf16, 输出类型为f32，同时支持A矩阵为R8、R16和R32

mma_f32_f16_universal:

介绍：输入类型为f16, 输出类型为f32，同时支持A矩阵为R8、R16和R32

mma_i32_i8_universal:

介绍：输入类型为int8, 输出类型为int32，同时支持A矩阵为R8、R16和R32

mma_bf16_mxi4_universal:

介绍：输入类型为mxint4, 输出类型为bf16，同时支持A矩阵为R8、R16和R32

mma_bf16_mxf4_universal:

介绍：输入类型为mxf4, 输出类型为bf16，同时支持A矩阵为R8、R16和R32

Attention

softmax

sofmtax_f32

介绍：

本函数是对linear format的数据进行softmax操作，数据来自于global memory

函数声明：

int softmax_f32(SiDeviceHandle dev, void *out, void *in, bool half_to_float, int outer_size, int dim_siz, int inner_size)；

说明：

dev是device handle，out是输出指针，in是输入指针，half_to_float暂时不用，outer_size代表多少条线性数据进行操作，注意若干条之间是紧密排列的线性关系，dim_size代表操作的每条数据有多少个元素，inner_size暂时不使用，默认设为1

数据对齐要求：

dim_size必须256对齐

softmax_bf16

介绍：

同上

函数声明：

int softmax_bf16(SiDeviceHandle dev, void *out, void *in, bool half_to_float, int outer_size, int dim_siz, int inner_size);

说明：

同上

数据对齐要求：

dim_size必须512对齐

softmax_tile

介绍：

本函数是对tile format的数据进行softmax操作

函数声明：

int softmax_tile_f32(SiDeviceHandle dev, void *out, void *in, bool half_to_float, int outer_size, int dim_siz, int inner_size);

说明：

dev是device handle，out是输出指针，in是输入指针，half_to_float暂时不用，outer_size代表多少条数据进行操作，注意是tile format，对于Q*Kt的结果，可以认为是结果score的行数，dim_size代表操作的每条数据有多少个元素，对于Q*Kt的结果，可以认为是结果score的列数，inner_size暂时不使用，默认设为1

数据对齐要求：

dim_size必须为8的倍数

outer_size必须是32的倍数

log_softmax

log_softmax_bf16

介绍：

本函数是对softmax_bf16的输出做log（以e为底数），数学上等价于：

$$\log\_\mathrm{softmax}(x_i) = (x_i - \max(x)) - \log\left(\sum_j e^{x_j - \max(x)}\right)$$

函数声明：

void log_softmax_bf16(sifmt::bfloat16 *p_in, sifmt::bfloat16 *p_out, float *etmp, bool half_to_float, int outer_size, int dim_size, int inner_size)

说明：

同softmax_bf16

数据对齐要求：

同softmax_bf16

RoPE

rope_bf16

介绍：计算输入数据的旋转位置编码，输入数据类型为bfloat16，sin值和cos值在线下提前算好。

函数声明：

void rope_bf16(SiDeviceHandle dev, void *d_in, void *sin, void *cos, void *d_out, int h_size, int seq_len);

dev为device handle. d_in为输入数据指针，d_out为输出数据指针， sin为sin数据指针， cos为cos数据指针，h_size为输入数据的hidden size, seq_len为sequence length。hidden size要求是1024的整数倍。sequence length没有颗粒度要求。

rms_norm

介绍：计算输入数据的RMSnorm,公式为

$$\frac{w * x}{\sqrt{mean(x^2) + eps}}$$

输入x为两个维度的tensor，分别是batch_size与normalized_size, normalized_size为低纬度，也是进行normlize计算的维度。

函数声明：

代码块

```
/// @brief Compute RMS (Root Mean Square) layer normalization
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param output Output normalized tensor
/// @param input Input tensor (shape: batch_size x normalized_size)
/// @param weight Weight tensor (shape: [original_normalized_size] if weight_opt=1)
/// @param eps Epsilon value for numerical stability
/// @param batch_size Batch size of input tensor
/// @param normalized_size Padded normalized dimension size (tile size aligned)
/// @param original_normalized_size Original (unpadded) normalized dimension size
/// @param weight_opt 1: apply weight; 0: no weight
/// @param eps_opt 1: use custom eps; 0: use default eps
template <typename T>
void rms_norm(const void* output, const void* input, const void* weight, const float eps, const int64_t batch_size, const int64_t normalized_size,  const int64_t original_normalized_size, int weight_opt, int eps_opt);

```

dev为device handle. output为输出数据指针，input为输出数据指针，weight为待乘的权重数据的指针，权重数据的size为normalized_size， eps为一个浮点数，通常为1e-6或1e-5。normalized_size需要对齐到tile size也就是1024B，如果原始输入数据不对齐，应该补0到对齐，并且将原始的数据size赋值给original_normalized_size. weight_opt 为1时表示计算中需要乘weight，否则不成乘weight。eps_opt为1是表示计算中需要加eps，否则不加eps。

fill

介绍：

输入为一个tensor，一个scalar，输出一个新的tensor，和输入tensor有相同的shape，且所有元素值为scalar。

函数声明：

template <typename T>

void fill(void *d_in, void *d_out, int dim0_size, int dim1_size, T value)

参数：

d_in为输入数据指针，d_out为输出数据指针，dim0_size为M维度大小，dim1_size为K维度大小(由于使用tile_reg，整体大小需满足1024B对齐)，value为需要填充的标量。

masked_fill

介绍：

输入为一个data tensor，一个mask tensor，一个scalar value，输出一个新的tensor，和输入tensor有相同的shape，且对应位置mask为1的元素值为输入的value。

函数声明：

template <typename T>

void masked_fill(void *d_data, void *d_mask, void *d_out, int data_dim0, int data_dim1, int mask_dim0, int mask_dim1, T value)

参数：

d_data为输入数据指针，d_mask为输入的掩码指针，d_out为输出数据指针，data_dim0为数据的外层维度大小，data_dim1为数据的内层维度大小，mask_dim0为掩码的外层维度大小，mask_dim1为掩码的内层维度大小，value为需要填充的标量。

掩码的shape和data的shape有三种情况：

1. mask_dim1 == data_dim1 && mask_dim0 == data_dim0: no broadcast

2. mask_dim1 == data_dim1 && mask_dim0 == 1: broadcast on dim0

3. mask_dim1 == 1 && mask_dim0 == data_dim0: broadcast on dim1

(掩码的shape和data的shape不一致，即2,3两种情况时，需满足data_dim1>=32)

sum

介绍：

输入为二维的tensor（shape为[M, N]），kernel为最后一个维度进行求和计算，并存储在输出tensor（shape为[M]）中。

函数声明：

代码块

```
/// @brief sum function
/// @tparam T Tensor datatype, only support sifmt::float32 for now
/// @param d_in Input tensor with shape [dim0_size, dim1_size]
/// @param d_out Output tensor with shape [dim0_size]
/// @param dim0_size Tensor shape 0
/// @param dim1_size Tensor shape 1
/// @param dim Dimension which sum calculate, only support -1 for now
template <typename T>
void sum(void *d_in, void *d_out, int dim0_size, int dim1_size, int dim = -1);
```

补充说明：

（1）初版只需要支持dim=-1，keepdim=true的参数配置

（2）keepdim相当于unsqueeze操作，框架可以处理，算子不需要做相关适配

（3）数制支持上，初版只需要支持float类型

（4）初版可以只考虑tensor为continuous的情况

（5）执行流程为：torch将输入tensor（shape为[A, B, C]）进行shape变换，转为[A * B, C]传入sikernel，sikernel对最后一维进行累加计算，输出shape为[A * B]的tensor，然后torch将其还原到[A, B, 1]的shape形式

（6）初版dim1_size参数，限制为32字节对齐（8个float），配合tile reg的load最小粒度要求

mul

介绍：输如两个二维tensor，计算他们的element wise的乘积，也支持其中一个为scalar另一个为tensor

函数声明：

代码块

```
/// @brief Multiply two tensors with specified dimensions (supports scalar broadcasting)
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in0 First input tensor (scalar if is_d_in0_scalar is true)
/// @param d_in1 Second input tensor (scalar if is_d_in1_scalar is true)
/// @param d_out Output tensor (shape: dim0_size x dim1_size)
/// @param dim0_size Size of dimension 0 (total size 1024B aligned)
/// @param dim1_size Size of dimension 1 (total size 1024B aligned)
/// @param is_d_in0_scalar True if d_in0 is a scalar
/// @param is_d_in1_scalar True if d_in1 is a scalar
template <typename T>//only sifmt::bfloat16, sifmt::float16, sifmt::float32 are supported
```

```
11  void mul(void *d_in0, void *d_in1, void *d_out, int dim0_size, int dim1_size,
    bool is_d_in0_scalar, bool is_d_in1_scalar);
12
13
14
```

补充说明：tensor的size需要时1024B对齐， 支持sifmt::bfloat16, sifmt::float16, sifmt::float32三种数据类型

div

介绍：

完成一维tensor的elementwise除法，被除数为tensor，除数可以为tensor或scalar，输出为tensor。

函数声明：

代码块

```
1  /// @brief Tensor division
2  /// @tparam T Datatype support [sifmt::float32, sifmt::bfloat16]
3  /// @param d_in0 Dividend tensor with shape [dim0_size]
4  /// @param d_in1 Divisor tensor with shape [dim0_size] or scalar if
   is_d_in1_scalar is true
5  /// @param d_out Output tensor with shape [dim0_size]
6  /// @param dim0_size Tensor shape param
7  /// @param is_d_in1_scalar Indicates d_in1 is scalar or tensor
8  template <typename T>
9  void div(void *d_in0, void *d_in1, void *d_out, int dim0_size, bool
   is_d_in1_scalar);
```

补充说明：

（1）数制支持上，初版需要支持float，bf16类型

（2）除数（输入2）需要支持tensor与scalar两种形式

（3）输出与被除数（输入1）的shape一致

（4）如果除数为tensor，那么除数与被除数shape应相同（如不同，则由框架处理进行broadcast），做elementwise除法

（5）对齐限制：硬件目前要求dim0_size * sizeof(T)为1024B对齐

（6）输入输出参数，均由框架进行预处理（reshape为一维并padding至符合对齐限制）

floor divide

介绍：

完成一维tensor的elementwise除法，并向下取整，被除数为tensor，除数可以为tensor或scalar，输出为tensor。

（1）数制支持上，初版需要支持int32类型，如果输入数据类型为int64，由框架进行int64->int32转换

（2）除数（输入2）需要支持tensor与scalar两种形式

（3）输出与被除数（输入1）的shape一致

（4）如果除数为tensor，那么除数与被除数shape应相同（如不同，则由框架处理进行broadcast），做elementwise除法

（5）对齐限制：全流程使用RVV Core，无对齐限制

（6）输入输出参数，均由框架进行预处理（reshape为一维）

内部接口调用形式：

代码块

```
/// @brief Tensor floor division
/// @tparam T Datatype support [int32_t]
/// @param[in] d_in0 Dividend tensor with shape [dim0_size]
/// @param[in] d_in1 Divisor tensor with shape [dim0_size] or scalar if is_d_in1_scalar is true
/// @param[out] d_out Output tensor with shape [dim0_size]
/// @param[in] dim0_size Tensor shape param
/// @param[in] is_d_in1_scalar Indicates d_in1 is scalar or tensor
template <typename T>
void floor_div(void *d_in0, void *d_in1, void *d_out, int dim0_size, bool is_d_in1_scalar);
```

bincount

介绍：

实现一维非负整数tensor的数据统计，统计0 ~ （min_len - 1）中每个自然数出现的次数。

函数声明：

代码块

```
/// @brief Count the frequency of each value in an array of non-negative ints
/// @tparam T Datatype support [int64_t]
/// @param d_in Input tensor with shape [dim_size]
/// @param d_out Output tensor with shape [min_len]
/// @param dim_size Input tensor size
/// @param min_len Output tensor size
template<typename T>
```

```
8   void bincount(void *d_in, void *d_out, T dim_size, T min_len);
```

补充说明：

（1）~~数制支持上，初版需要支持int64_t类型~~

（2）初版暂时不用支持weights参数

（3）输出tensor的长度上，初版先依赖min len，后续考虑是否支持max + 1的情况

（4）新情况变更，由于硬件限制，初版支持int32_t类型，由框架将数据进行64-32转换

eq

介绍：

比较输入的2个tensor（或tensor与scalar）对应位置的数值是否相等（也可同时实现比较是否不等的功能）

函数声明：

代码块

```
1   /// @brief Computes element-wise equality
2   /// @tparam T Datatype support [int32_t, sifmt:float32, sifmt::bfloat16]
3   /// @param[in] d_in0 Input tensor 0 with shape [dim0_size, dim1_size]
4   /// @param[in] d_in1 Input tensor 1 with shape [dim0_size, dim1_size] OR scalar
5   /// @param[out] d_out Output tensor with shape [dim0_size, dim1_size]
6   /// @param[in] dim0_size Tensor shape param 0
7   /// @param[in] dim1_size Tensor shape param 1
8   /// @param[in] is_d_in1_scalar Indicates tensor 1 is scalar OR tensor
9   /// @param[in] op_type Indicates OP is EQUAL(0) OR NOT EQUAL(1)
10  template <typename T>
11  void eq(void *d_in0, void *d_in1, void *d_out, int dim0_size, int dim1_size,
    bool is_d_in1_scalar = true, int op_type = 0);
```

补充说明：

（1）数制支持上，需要支持类型有：int32_t，bfloat16，float32

（2）初版只需要支持输入2为scalar的情况

（3）由于硬件限制，对于int64_t类型，由框架将数据进行64-32转换

nonzero

介绍：

返回tensor中非0元素的位置信息。

函数声明：

代码块

```
/// @brief Returns 2 tensor containing the indices of all non-zero elements of input
/// @tparam T Datatype support [int32_t]
/// @param[in] d_in Input tensor with shape [dim0_size, dim1_size]
/// @param[out] d_out0 Output tensor 0 with shape [*out_size]
/// @param[out] d_out1 Output tensor 1 with shape [*out_size]
/// @param[in] dim0_size Tensor shape param 0
/// @param[in] dim1_size Tensor shape param 1
/// @param[out] out_size Record nonzero element number, this param should be on DEVICE MEMORY
template <typename T>
void nonzero_2d(void *d_in, void *d_out0, void *d_out1, int dim0_size, int dim1_size, int *out_size);

/// @brief Returns 1 tensor containing the indices of all non-zero elements of input
/// @tparam T Datatype support [int32_t]
/// @param[in] d_in Input tensor with shape [dim0_size, dim1_size]
/// @param[out] d_out Output tensor with shape [*out_size, 2]
/// @param[in] dim0_size Tensor shape param 0
/// @param[in] dim1_size Tensor shape param 1
/// @param[out] out_size Record nonzero element number, this param should be on DEVICE MEMORY
template <typename T>
void nonzero_2d_no_tuple(void *d_in, void *d_out, int dim0_size, int dim1_size, int *out_size);
```

补充说明：

（1）初版支持as_tuple为ture/false的情况

as_tuple为true时，返回tensor为d个，分别存储不同维度的坐标值（二维输入即存储坐标的x值，y值）

as_tuple为false时，返回tensor为1个，shape为[MAX_NUM, DIM_NUM]

（2）初版支持int32_t类型，由框架将数据进行64-32转换

（3）初版仅实现二维输入接口，输出也为二维，按照最大值配置空间，假设input是[m, n]，输出每个tensor的大小为[m * n]，而每个输出tensor的真实大小，将存储于out_size返回参数中

as_tuple为true时，返回tensor为2个

as_tuple为false时，返回tensor为1个，shape为[m * n, 2]

index_select

介绍：

输入为1个二维数据tensor，1个一维索引tensor，索引的维度index_dim；输出为1个二维tensor，是输入tensor部分index的组合。

函数声明：

代码块

```
template <typename T>
void index_select(void *d_in0, void *d_in1, void *d_out, int output_size, int dim0_size, int dim1_size, int index_dim, int index_size);
```

参数：

1. d_in0为数据tensor，shape为(dim0_size,dim1_size)，size无对齐需求
2. d_in1为索引tensor，一维，长度为index_size
3. d_out为输出tensor
   a. index_dim=0时shape为(index_size,dim1_size)
   b. index_dim=1时shape为(dim0_size,index_size)
4. index_dim，索引的维度，为0或者1

gt_tensor

介绍：

输入一个二维data tensor，一个二维other tensor，输出为一个二维bool类型的tensor，表示data和other的元素的比较结果（element_data > element_other）

函数声明：

代码块

```
template <typename T>
void gt_tensor(void *d_in0, void *d_in1, void *d_out, int data_dim0, int data_dim1, int other_dim0, int other_dim1);
```

参数：

1. d_in0为data tensor，shape为(data_dim0,data_dim1)
2. d_in1为other tensor，shape为(other_dim0,other_dim1)；支持broadcast成data tensor相同的shape。

3. d_out为输出tensor，元素为bool类型，shape和d_in0一样

ge_scalar

介绍：

输入一个二维data tensor，一个scalar；输出为一个二维bool类型的tensor，表示data的元素和scalar的比较结果（element_data >= scalar）

函数声明：

代码块

```
template <typename T>
void ge_scalar(void *d_in0, T d_in1, void *d_out, int dim0_size, int
dim1_size);
```

参数：

1. d_in0为输入的tensor，shape为(dim0_size,dim1_size)
2. d_in1为输入的scalar
3. d_out为输出的tensor，元素为bool类型，shape和d_in0一样

lt_scalar

介绍：

输入一个二维data tensor，一个scalar；输出为一个二维bool类型的tensor，表示data的元素和scalar的比较结果（element_data < scalar）

函数声明：

代码块

```
template <typename T>
void lt_scalar(void *d_in0, T d_in1, void *d_out, int dim0_size, int
dim1_size);
```

参数：

1. d_in0为输入的tensor，shape为(dim0_size,dim1_size)
2. d_in1为输入的scalar
3. d_out为输出的tensor，元素为bool类型，shape和d_in0一样

Topk

介绍：输入一个二维的data tensor，在内层维度上找到最大或最小的k个元素，输出这些元素的值和索引，支持排序后输出。

函数声明：

代码块

```
/// @brief get the largest or smallest K numbers and the indices form tensor self
/// @param T Datatype support [sifmt::float32, sifmt::bfloat16, sifmt::float16]
/// @param values Output data pointer
/// @param indices Output data index pointer
/// @param self Input data pointer
/// @param batch_size Batch size of input data
/// @param size The size from which to select the k values
/// @param k K number
/// @param largest True for largest k values, False for smallest k values
/// @param sorted True for sorting the result,  False for no sorting
template <typename T>
void topk(void* values, void* indices, const void* self, const int64_t batch_size, const int64_t size, int k, bool largest, bool sorted);

```

补充说明：目前使用RVV core以堆排序的方式实现，每个rvv core负责一个vector。性能需要评估。

rsqrt

介绍：rsqrt用于计算输入张量每个元素的平方根的倒数。返回一个新张量，其中每个元素是 `input` 中相应元素的平方根的倒数。

函数声明：

代码块

```
//calculate the reciprocal of the square-root of the input tensor with size dim0*dim1.
//total size should be 1024B algined
template <typename T>//only sifmt::bfloat16, sifmt::float16, sifmt::float32 are supported
void rsqrt(void *d_in, void *d_out, int dim0_size, int dim1_size );
```

参数：

1. d_in为输入的tensor，shape为(dim0_size,dim1_size)
2. d_out为输出的tensor，shape同输入，数据类型同输入

mean_dim

介绍：输入一个二维的tensor，按照给定的维度(0/1)计算均值，返回的tensor在指定维度上的数据量为1.

函数声明：

代码块

```
/// @brief  Returns the mean value of each row of the tensor in the given dimension.
/// @tparam T Datatype support [float32, bfloat16], the dataype of input and output should be the same, can be different in the future.
/// @param d_out Output tensor with shape
/// @param d_in  Input tensor with shape [dim0_size, dim1_size]
/// @param dim0_size Dimension 0 size of input tensor
/// @param dim1_size Dimension 1 size of input tensor, should be 1024B aligned for tile core calculation, pad value 0
/// @param original_dim1_size Original dimension 1 size of input tensor before pad
/// @param dim The dimension or dimensions to reduce
template <typename T>
void mean_dim(void* d_out, void* d_in, int dim0_size, int dim1_size, int origianl_dim1_size, int dim);
```

参数：

1. d_in为输入的tensor，shape为(dim0_size,dim1_size)
2. d_out为输出的tensor，若dim=1，shape为（dim0_size,1）；若dim=0，shape为（1，dim1_size）
3. original_dim1_size为有效的dim1_size, 若使用tile core，dim1_size为original_dim1_size pad 0之后按照1024B对齐的数值。
4. dim指定需要进行mean的维度。

silu_and_mul

介绍：*Applies SiLU activation to first half of input and multiplies with second half*

* @note Memory Layout:
* - Input tensor has shape [..., 2*d] where last dimension contains pairs of values
* - Output tensor has shape [..., d]
* - Supports non-contiguous tensors through explicit stride parameters
*
* @note Behavior:
* - Computes out[i] = SiLU(input[i]) * input[i+d] for i in [0,d)

* - SiLU(x) = x * sigmoid(x)
* - Optimized implementation for different hardware and data types

函数声明：

代码块

```
/// @brief Applies SiLU activation to first half of input and multiplies with second half
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input tensor [dim0_size, 2*dim1_size]
/// @param d_out Output tensor [dim0_size, dim1_size]
/// @param dim0_size Size of dimension 0 for the input/output tensor
/// @param dim1_size Size of dimension 1 for the output tensor (total size must be 1024B aligned)
template <typename T>
void silu_and_mul(void *d_in, void *d_out, int dim0_size, int dim1_size);
```

说明：

1. 数据类型支持bfloat16，float32和float16，通过template设置
2. 使用tile_core计算，dim1_size需要按照1024B对齐。目前llama和deepseek，在dim1_size上已经满足1024B对齐的要求，暂时不考虑不对齐pad的情况
3. dim1_size是output的dimension 1 size。input的dimension 1 size是2*dim1_size

max

介绍：

计算2D tensor在某一个维度上的最大值。输入为一个2D tensor，一个指定的比较维度；输出为一个最大值tensor，和一个最大值对应的下标tensor。如果有多个相同的最大值，选择第一个作为输出。

函数声明：

代码块

```
/// @brief Compute the maximum values for a 2D tensor. Only support reduce in one given dimension.
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input 2D tensor (shape: dim0_size x dim1_size)
/// @param d_out0 Output tensor containing the maximum values.
///               In default, keepdim is true, so the output shape is 1*dim0_size or 1*dim1_size (depending on dim).
/// @param d_out1 Output tensor containing the indices of maximum values, same shape with d_out0.
```

```
/// @param dim0_size Size of dimension 0 for the input tensor
/// @param dim1_size Size of dimension 1 for the input tensor
/// @param dim Dimension along which to compute max
template <typename T>
void max(void *d_in, void *d_out0, void *d_out1, int dim0_size, int dim1_size,
int dim);
```

amax

介绍：

计算2D tensor在某一个（或多个）维度上的最大值。输入为一个2D tensor，一个指定的比较维度；输出为一个最大值tensor。如果有多个相同的最大值，选择第一个作为输出。

函数声明：

代码块

```
/// @brief Compute the maximum values for a 2D tensor. Supposed to support
reduce in multi-dimensions, currently only one.
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16,
sifmt::float32]
/// @param d_in Input 2D tensor (shape: dim0_size x dim1_size)
/// @param d_out Output tensor containing the maximum values.
///              In default, keepdim is true, so the output shape is
1*dim0_size or 1*dim1_size (depending on dim)
/// @param dim0_size Size of dimension 0 for the input tensor
/// @param dim1_size Size of dimension 1 for the input tensor
/// @param dim Dimension along which to compute max
template <typename T>
void amax(void *d_in, void *d_out, int dim0_size, int dim1_size, int dim);
```

说明：

多维度上的reduce待支持。

argmax

介绍：

计算2D tensor在某一个维度上的最大值的下标。输入为一个2D tensor，一个指定的比较维度；输出为一个最大值对应的下标tensor。如果有多个相同的最大值，选择第一个下标作为输出。

函数声明：

代码块

```cpp
/// @brief Compute the indices of the maximum values for a 2D tensor. For each row (along dimension 0), finds the index of the maximum value in the corresponding column (dimension 1).
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input 2D tensor (shape: dim0_size x dim1_size)
/// @param d_out Output tensor containing the indices of the maximum values.
///              In default, keepdim is true, so the output shape is 1*dim0_size or 1*dim1_size (depending on dim)
/// @param dim0_size Size of dimension 0 for the input tensor
/// @param dim1_size Size of dimension 1 for the input tensor
/// @param dim Dimension along which to compute argmax
template <typename T>
void argmax(void *d_in, void *d_out, int dim0_size, int dim1_size, int dim);
```

sigmoid

介绍：

计算2D Tensor的sigmoid函数值。
$f(x) = sigmoid(x) = 1/(1 + e^{-x})$

函数声明：

代码块

```cpp
/// @brief Computes the sigmoid activation function
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input tensor
/// @param d_out Output tensor
/// @param dim0_size Size of dimension 0 for the input/output tensor
/// @param dim1_size Size of dimension 1 for the input/output tensor (total size must be 1024B aligned)
template <typename T>
void sigmoid(void *d_in, void *d_out, int dim0_size, int dim1_size);
```

triu_tril

介绍：输入为一个二维tensor，输出上三角矩阵（triu）或下三角矩阵（tril）

- `diagonal`：对角线偏移量：
  - 大于 0：从主对角线向上偏移
  - 等于 0：主对角线
  - 小于 0：从主对角线向下偏移

- `upper`：操作模式：
  - true: 上三角（保留对角线及以上的元素）
  - false: 下三角（保留对角线及以下的元素）
- `is_out`：输出模式：
  - true: 将结果写入 d_out，保持 d_in 不变
  - false: 原地修改 d_in，忽略 d_out

函数声明：

代码块

```
/// @brief Generate upper/lower triangular matrix mask (supports in-place operation)
/// @tparam T Supported data types [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input matrix pointer (shape: [dim0_size, dim1_size])
/// @param d_out Output matrix pointer (same shape as input)
/// @param dim0_size Number of matrix rows
/// @param dim1_size Number of matrix columns
/// @param diagonal Diagonal offset:
///                 > 0: Offset upward from main diagonal
///                 = 0: Main diagonal
///                 < 0: Offset downward from main diagonal
/// @param upper Operation mode:
///              true: Upper triangular (preserve elements on/above diagonal)
///              false: Lower triangular (preserve elements on/below diagonal)
/// @param is_out Output mode:
///              true: Write result to d_out, keep d_in unchanged
///              false: Modify d_in in-place, ignore d_out
template <typename T>
void triu_tril(void *d_in, void *d_out, int dim0_size, int dim1_size, int diagonal, bool upper, bool is_out);
```

fused_add_rms_norm

介绍：实现残差连接与 RMS 归一化的融合操作，支持原地更新残差张量

- d_in: 输入 / 输出张量，形状为 [dim0_size x dim1_size]，输出结果会覆盖此内存
- d_residual: 残差张量，形状与 d_in 相同
- d_weight: 缩放权重张量，形状为 [dim1_size]
- dim0_size: 输入张量的第 0 维大小
- dim1_size: 输入张量的第 1 维大小

- epsilon: float类型的小常数
- inplace_residual: 控制是否原地更新残差张量
  - true: 将 input + residual 的结果存入 d_residual
  - false: 不修改 d_residual

函数声明：

代码块

```
/// @brief Fused add residual and RMS normalization operation (supports in-place residual update)
/// @tparam T Supported data types [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input/output tensor (shape: dim0_size x dim1_size)
/// @param d_residual Residual tensor (shape: dim0_size x dim1_size)
/// @param d_weight Weight tensor for scaling (shape: [dim1_size])
/// @param dim0_size Size of dimension 0 for the input tensor
/// @param dim1_size Size of dimension 1 for the input tensor
/// @param epsilon Small value added to variance for numerical stability
/// @param inplace_residual When true:
///                         - Stores (input + residual) in d_residual
template <typename T>
void fused_add_rms_norm(void *d_in, void *d_residual, void *d_weight, int64_t dim0_size, int64_t dim1_size, float epsilon, bool inplace_residual);
```

Sin

代码块

```
/// @brief Compute sin(x) for each element in input tensor
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input tensor (shape: dim0_size x dim1_size)
/// @param d_out Output tensor (sin values, shape: dim0_size x dim1_size)
/// @param dim0_size Size of dimension 0 (total size 1024B aligned)
/// @param dim1_size Size of dimension 1 (total size 1024B aligned)
template <typename T>//only sifmt::bfloat16, sifmt::float16, sifmt::float32 are supported
void sin(void *d_in, void *d_out, int dim0_size, int dim1_size );

```

Cos

```
/// @brief Compute cos(x) for each element in input tensor
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16,
sifmt::float32]
/// @param d_in Input tensor (shape: dim0_size x dim1_size)
/// @param d_out Output tensor (cos values, shape: dim0_size x dim1_size)
/// @param dim0_size Size of dimension 0 (total size 1024B aligned)
/// @param dim1_size Size of dimension 1 (total size 1024B aligned)
template <typename T>//only sifmt::bfloat16, sifmt::float16, sifmt::float32
are supported
void cos(void *d_in, void *d_out, int dim0_size, int dim1_size );

```

Reciprocal

代码块

```
/// @brief Compute reciprocal (1/x) for each element in input tensor
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16,
sifmt::float32]
/// @param d_in Input tensor (shape: dim0_size x dim1_size)
/// @param d_out Output tensor (reciprocal square root values, shape:
dim0_size x dim1_size)
/// @param dim0_size Size of dimension 0 (total size 1024B aligned)
/// @param dim1_size Size of dimension 1 (total size 1024B aligned)
template <typename T>//only sifmt::bfloat16, sifmt::float16, sifmt::float32
are supported
void reciprocal(void *d_in, void *d_out, int dim0_size, int dim1_size );
```

Sort

代码块

```
/// @brief Sort the input array and return sorted values and indices
/// @param values Output buffer for sorted values
/// @param indices Output buffer for indices
/// @param self Input array to sort
/// @param batch_size Number of arrays to sort
/// @param size Length of each array
/// @param descending If true, sort in descending order; if false, sort in
ascending order
/// @param stable If true, maintain relative order of equal elements
template<typename T>
```

```
10  void sort(void* values, void* indices, const void* self, const int64_t
    batch_size, const int64_t size, bool descending, bool stable);
11
```

index

介绍：对二维输入张量进行索引操作或原地赋值操作，支持 index 和 index_put 两种模式。

- is_put: 操作模式
  - true: index_put（根据索引将 d_value 写入 d_self 指定位置，可累加或覆盖）
  - false: index（根据索引从 d_self 提取元素，写入 d_out）
- accumulate: index_put 模式下的累加方式
  - true: 累加（d_self 指定位置加上 d_value）
  - false: 覆盖（d_self 指定位置被 d_value 覆盖）
- d_index0, d_index1: 索引张量，指定要操作的元素位置，二者长度相同
- d_value: 写入值，可以是标量或与索引长度一致的张量
- d_out: 输出张量，index 模式下存放提取结果，index_put 模式下存放修改后的结果
- self_dim0, self_dim1: 输入张量 d_self 的形状
- index_size: 索引张量长度
- value_dim0, value_dim1: d_value 的形状，value_dim0*value_dim1=1 或 index_size
- output_dim0, output_dim1: 输出张量的形状

代码块

```
1   /// @brief Perform tensor indexing or in-place value assignment operations.
    Here're some notes:
2   ///        1. d_index0 and d_index1 have the same size.
3   ///        2. d_value is a scalar, or has the same size as d_index0.
4   /// @tparam T Supported data types [sifmt::bfloat16, sifmt::float16,
    sifmt::float32]
5   /// @param d_self Input tensor to index (shape: [self_dim0, self_dim1])
6   /// @param d_index0 dim0 index tensor
7   /// @param d_index1 dim1 index tensor
8   /// @param d_value Value tensor to put in
9   /// @param d_out Output tensor (shape: [output_dim0, output_dim1])
10  /// @param is_put Operation mode:
11  ///              true: index_put (modify d_self with d_value)
12  ///              false: index (extract values from d_self)
13  /// @param accumulate For index_put only:
14  ///              true: add values to existing elements
```

```
///                false: overwrite existing elements
/// @param self_dim0 Size of input tensor dimension 0
/// @param self_dim1 Size of input tensor dimension 1
/// @param index_size Size of d_index0 and d_index1
/// @param value_dim0 Size of value tensor dimension 0, value_dim0*value_dim1 = 1 or index0_size
/// @param value_dim1 Size of value tensor dimension 1
/// @param output_dim0 Size of output tensor dimension 0
/// @param output_dim1 Size of output tensor dimension 1
template <typename T>
void index(void *d_self, int *d_index0, int *d_index1, void *d_value, void *d_out, bool is_put, bool accumulate, int self_dim0,
           int self_dim1, int index_size, int value_dim0, int value_dim1, int output_dim0, int output_dim1);

```

compare

介绍：对两个张量进行逐元素比较操作，支持大于等于（GE）、大于（GT）、小于等于（LE）、小于（LT）四种比较方式，结果为布尔类型张量。

- OP: 比较操作类型
  - GE: 大于等于（>=）
  - GT: 大于（>）
  - LE: 小于等于（<=）
  - LT: 小于（<）
- d_data: 输入张量，形状为 [data_dim0, data_dim1]
- d_other: 比较张量，可以与输入张量形状一致或可广播
- d_out: 输出布尔张量，形状为 [data_dim0, data_dim1]
- data_dim0, data_dim1: 输入张量的形状
- other_dim0, other_dim1: 比较张量的形状（支持广播）

代码块

```
enum class CompareOp{
    GE,  // Greater than or equal
    GT,  // Greater than
    LE,  // Less than or equal
    LT,  // Less than
};

/// @brief Element-wise comparison operation (supports GE/GT/LE/LT)
```

```
/// @tparam T Supported data types [sifmt::bfloat16, sifmt::float16,
sifmt::float32]
/// @tparam OP Comparison operator selected from CompareOp enum
/// @param d_data Input tensor (shape: data_dim0 x data_dim1)
/// @param d_other Comparison tensor (broadcastable to input shape)
/// @param d_out Boolean output tensor (shape: data_dim0 x data_dim1)
/// @param data_dim0 Size of input tensor's dimension 0
/// @param data_dim1 Size of input tensor's dimension 1
/// @param other_dim0 Size of comparison tensor's dimension 0
/// @param other_dim1 Size of comparison tensor's dimension 1
template<typename T, CompareOp OP>
void compare(void* d_data, void* d_other, void* d_out, int data_dim0, int
data_dim1, int other_dim0, int other_dim1);

```

bitwise

介绍：对输入张量执行逐元素按位操作，支持 AND、OR、NOT 三种操作。输入和输出张量 shape 必须一致（[dim0_size, dim1_size]），支持 int 和 bool 类型。

- OP: 操作类型
  - AND：逐元素按位与（out[i] = input[i] & other[i]）
  - OR：逐元素按位或（out[i] = input[i] | other[i]）
  - NOT：逐元素按位取反（int 类型 out[i] = ~input[i]，bool 类型 out[i] = !input[i]，此时 other 被忽略）
- input: 输入张量指针，形状 [dim0_size, dim1_size]
- other: 第二输入张量指针，形状 [dim0_size, dim1_size]，仅 AND/OR 操作需要
- out: 输出张量指针，形状 [dim0_size, dim1_size]
- dim0_size, dim1_size: 输入/输出张量的两个维度大小

代码块

```
enum class BitwiseOp {
  AND,
  OR,
  NOT,
};
/// @brief Perform element-wise bitwise operation (AND, OR, NOT) on input
tensors.
/// @tparam T Supported data types [int, bool]
/// @tparam BitwiseOp Bitwise operation type: AND, OR, NOT
/// @param input Input tensor pointer (shape: [dim0_size, dim1_size])
/// @param other Second input tensor pointer (shape: [dim0_size, dim1_size])
```

```
/// @param out Output tensor pointer (shape: [dim0_size, dim1_size])
/// @param dim0_size Size of dimension 0
/// @param dim1_size Size of dimension 1
/// For AND/OR: out[i] = input[i] <op> other[i]
/// For NOT:    out[i] = ~input[i] (other is ignored)
template <typename T, BitwiseOp OP>
void bitwise(void *input, void *other, void *out, int64_t dim0_size, int64_t dim1_size);

```

rotary_embedding

介绍：对 query 和 key 张量应用旋转位置编码（Rotary Positional Embedding, RoPE），用于为注意力机制引入位置信息。要求 hidden_size 为 1024 的倍数，head_size 当前仅支持 128。cos_sin_cache 需提前计算好。支持 GPT-NeoX 风格（is_neox=true）。

- positions：位置索引张量，shape [seq_len]
- query：待加位置编码的 query 张量，shape [seq_len, hidden_size]
- key：待加位置编码的 key 张量，shape [seq_len, hidden_size]
- cos_sin_cache：预计算的 cos/sin 缓存，shape [seq_len, head_size]
- positions_sizes/strides：positions 张量的 shape/stride，通常为 {seq_len} 和 {1}
- query_sizes/strides：query 张量的 shape/stride，通常为 {seq_len, hidden_size} 和 {hidden_size, 1}
- key_sizes/strides：key 张量的 shape/stride，通常为 {seq_len, hidden_size} 和 {hidden_size, 1}
- cache_sizes/strides：cos_sin_cache 的 shape/stride，通常为 {seq_len, head_size} 和 {head_size, 1}
- positions_ndims/query_ndims/key_ndims/cache_ndims：各张量的维度数
- head_size：每个 attention head 的维度（必须为 2 的幂，当前仅支持 128）
- is_neox：是否采用 GPT-NeoX 风格的 rotary embedding（当前仅支持 true）

代码块

```
// hidden_size must be a multiple of 1024
// head_size must be 128 currently
/// @brief Apply rotary positional embeddings to query and key tensors, hidden_size must be multiple of 1024
/// @tparam T Supported data types [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param positions Position indices tensor, shape [seq_len]
/// @param query Query tensor to apply rotary embedding, shape [seq_len, hidden_size]
```

```cpp
/// @param key Key tensor to apply rotary embedding, shape [seq_len, hidden_size]
/// @param cos_sin_cache Precomputed cosine/sine values, shape [seq_len, head_size]
/// @param positions_sizes Sizes of positions tensor, value: {seq_len}
/// @param positions_strides Strides of positions tensor, value: {1}
/// @param query_sizes Sizes of query tensor, value: {seq_len, hidden_size}
/// @param query_strides Strides of query tensor, value: {hidden_size, 1}
/// @param key_sizes Sizes of key tensor, value: {seq_len, hidden_size}
/// @param key_strides Strides of key tensor, value: {hidden_size, 1}
/// @param cache_sizes Sizes of cos_sin_cache tensor, value: {seq_len, head_size}
/// @param cache_strides Strides of cos_sin_cache tensor, value: {head_size, 1}
/// @param positions_ndims Number of dimensions in positions tensor (value: 1)
/// @param query_ndims Number of dimensions in query tensor (value: 2)
/// @param key_ndims Number of dimensions in key tensor (value: 2)
/// @param cache_ndims Number of dimensions in cos_sin_cache tensor (value: 2)
/// @param head_size Size of each attention head (must be power of two, e.g., 128)
/// @param is_neox Use GPT-NeoX style rotary embeddings (currently only true supported)
template <typename T>
void rotary_embedding(const int64_t *positions,
    void *query,
    void *key,
    const void *cos_sin_cache,
    const int64_t *positions_sizes,
    const int64_t *positions_strides,
    const int64_t *query_sizes,
    const int64_t *query_strides,
    const int64_t *key_sizes,
    const int64_t *key_strides,
    const int64_t *cache_sizes,
    const int64_t *cache_strides,
    int64_t positions_ndims,
    int64_t query_ndims,
    int64_t key_ndims,
    int64_t cache_ndims,
    int64_t head_size,
    bool is_neox);
```

all_any

介绍：对 2D 张量的指定维度执行 **all** 或 **any** 归约操作，用于判断张量中是否**所有元素**（`isAll == true`）或**至少一个元素**（`isAll == false`）满足特定条件。支持的数据类型为 `bool` 和 `uint8_t`。

- `in`：输入张量指针，shape 为 `[in_dim0, in_dim1]`
- `out`：输出张量指针，shape 为 `[out_dim0, out_dim1]`
- `in_dim0`：输入张量维度 0 的大小
- `in_dim1`：输入张量维度 1 的大小
- `out_dim0`：输出张量维度 0 的大小
- `out_dim1`：输出张量维度 1 的大小
- `red_dim`：归约操作的维度：
  - `0` 表示对维度 0 进行归约
  - `1` 表示对维度 1 进行归约
  - `-1` 表示对两个维度都进行归约（输出张量应为 `[1, 1]`）

注意：若 `red_dim == -1`，输出张量的两个维度大小都应为 `1`。

代码块

```
/// @brief Compute all or any reduction across a specified dimension of a 2D tensor.
/// @tparam T Supported data types [bool, uint8_t]
/// @tparam isAll If true, compute all reduction; if false, compute any reduction
/// @param in Input tensor pointer (shape: [in_dim0, in_dim1])
/// @param out Output tensor pointer (shape: [out_dim0, out_dim1])
/// @param in_dim0 Size of input tensor dimension 0
/// @param in_dim1 Size of input tensor dimension 1
/// @param out_dim0 Size of output tensor dimension 0
/// @param out_dim1 Size of output tensor dimension 1
/// @param red_dim Dimension to reduce along (0: dim0, 1: dim1, -1: both dimensions)
/// @note out_dim0 and out_dim1 should be 1 if red_dim is -1.
template <typename T, int isAll>
void all_any(void *in, void *out, int64_t in_dim0, int64_t in_dim1, int64_t out_dim0,int64_t out_dim1, int red_dim);
```

Embeding

flash_attn_decode

介绍：

decode阶段attention的计算逻辑，具体可参考 4.Paged Attention

函数声明：

代码块

```
/// @brief Paged decode attention for vLLM
/// B - batch size
/// QH - Query head number
/// KVH - Key and Value head number
/// D - head dim
/// @tparam T Datatype support [sifmt::bfloat16]
/// @param[in] q Query input tensor with shape [B, 1, QH, D]
/// @param[in] q_sizes Query input tensor shape param
/// @param[in] q_ndim Query input tensor shape dimension
/// @param[in] kcache Key cache input tensor with shape [num_blocks, page_block_size, KVH, D]
/// @param[in] kcache_sizes Key cache input tensor shape param
/// @param[in] kcache_ndim Key cache input tensor shape dimension
/// @param[in] vcache Value cache input tensor with shape [num_blocks, page_block_size, KVH, D]
/// @param[in] vcache_sizes Value cache input tensor shape param
/// @param[in] vcache_ndim Value cache input tensor shape dimension
/// @param[in] k_ Intermediate key tensor
/// @param[in] k_sizes Intermediate key tensor shape param
/// @param[in] k_ndim Intermediate key tensor shape dimension
/// @param[in] v_ Intermediate value tensor
/// @param[in] v_sizes Intermediate value tensor shape param
/// @param[in] v_ndim Intermediate value tensor shape dimension
/// @param[in] seqlens_k_ Key sequence lengths tensor
/// @param[in] seqlens_k_sizes Key sequence lengths tensor shape param
/// @param[in] seqlens_k_ndim Key sequence lengths tensor shape dimension
/// @param[in] rotary_cos_ Precomputed cosine values for rotary position encoding
/// @param[in] rotary_cos_sizes Precomputed cosine values tensor shape param
/// @param[in] rotary_cos_ndim Precomputed cosine values tensor shape dimension
/// @param[in] rotary_sin_ Precomputed sine values for rotary position encoding
/// @param[in] rotary_sin_sizes Precomputed sine values tensor shape param
/// @param[in] rotary_sin_ndim Precomputed sine values tensor shape dimension
/// @param[in] cache_batch_idx_ Cache batch index tensor
/// @param[in] cache_batch_idx_sizes Cache batch index tensor shape param
/// @param[in] cache_batch_idx_ndim Cache batch index tensor shape dimension
/// @param[in] leftpad_k_ Left padding for key tensor
/// @param[in] leftpad_k_sizes Left padding for key tensor shape param
/// @param[in] leftpad_k_ndim Left padding for key tensor shape dimension
/// @param[in] block_table_ Page block table tensor
/// @param[in] block_table_sizes Page block table tensor shape param
/// @param[in] block_table_ndim Page block table tensor shape dimension
/// @param[in] alibi_slopes_ A bias of alibi_slope tensor with shape [B, QH]
```

```cpp
/// @param[in] alibi_slopes_sizes alibi_slope tensor shape param
/// @param[in] alibi_slopes_ndim alibi_slope tensor shape dimension
/// @param[out] out_ Output tensor with shape [B, 1, QH, D]
/// @param[in] out_sizes Output tensor shape param
/// @param[in] out_ndim Output tensor shape dimension
/// @param[out] softmax_lse_ Softmax log-sum-exp tensor
/// @param[in] softmax_lse_sizes Softmax log-sum-exp tensor shape param
/// @param[in] softmax_lse_ndim Softmax log-sum-exp tensor shape dimension
/// @param[in] softmax_scale Softmax scale, default to 1 / sqrt(D)
/// @param[in] is_causal Whether to apply causal attention mask
/// @param[in] window_size_left Left window size for sliding window local attention
/// @param[in] window_size_right Right window size for sliding window local attention
/// @param[in] softcap Anything > 0 activates softcapping attention
/// @param[in] is_rotary_interleaved If true, rotary combines indices 0 & 1, else indices 0 & rotary_dim / 2
/// @param[in] num_splits Number of splits
template<typename T>
void flash_attn_decode(
  void* q, // Tensor
  const int64_t* q_sizes,
  int q_ndim,

  void* kcache, // Tensor
  const int64_t* kcache_sizes,
  int kcache_ndim,

  void* vcache, // Tensor
  const int64_t* vcache_sizes,
  int vcache_ndim,

  void* k_, // Tensor
  const int64_t* k_sizes,
  int k_ndim,

  void* v_, // Tensor
  const int64_t* v_sizes,
  int v_ndim,

  int* seqlens_k_, // Tensor
  const int64_t* seqlens_k_sizes,
  int seqlens_k_ndim,

  void* rotary_cos_, // Tensor
  const int64_t* rotary_cos_sizes,
  int rotary_cos_ndim,
```

```
85
86      void* rotary_sin_, // Tensor
87      const int64_t* rotary_sin_sizes,
88      int rotary_sin_ndim,
89
90      int* cache_batch_idx_, // Tensor
91      const int64_t* cache_batch_idx_sizes,
92      int cache_batch_idx_ndim,
93
94      int* leftpad_k_, // Tensor
95      const int64_t* leftpad_k_sizes,
96      int leftpad_k_ndim,
97
98      int* block_table_, // Tensor
99      const int64_t* block_table_sizes,
100     int block_table_ndim,
101
102     float* alibi_slopes_, // Tensor
103     const int64_t* alibi_slopes_sizes,
104     int alibi_slopes_ndim,
105
106     void* out_, // Tensor
107     const int64_t* out_sizes,
108     int out_ndim,
109
110     void* softmax_lse_, // Tensor
111     const int64_t* softmax_lse_sizes,
112     int softmax_lse_ndim,
113
114     const float softmax_scale,
115     bool is_causal,
116     int window_size_left,
117     int window_size_right,
118     const float softcap,
119     bool is_rotary_interleaved,
120     int num_splits
121 );
```

reshape_and_cache_flash

介绍：

reshape_and_cache_flash将key和value写入到KV Cache中，具体可参考：融合算子-copy_kv_cache

函数声明：

代码块

```
/**
 * @brief Reshapes and caches key-value tensors for Flash Attention
 * supported datatype(dtype):
 * [TMAP_DTYPE_FP16, TMAP_DTYPE_BF16, TMAP_DTYPE_FP32]
 *
 * @param[in] key              Key tensor [num_tokens, num_heads, head_size]
 * @param[in] value            Value tensor [num_tokens, num_heads, head_size]
 * @param[in, out] key_cache   Key cache tensor [num_blocks, block_size, num_heads, head_size]
 * @param[in, out] value_cache Value cache tensor [num_blocks, block_size, num_heads, head_size]
 * @param[in] slot_mapping     Mapping from token indices to cache slots [num_tokens]
 * @param[in] key_sizes        Sizes of key tensor
 * @param[in] key_strides      Strides of key tensor
 * @param[in] value_sizes      Sizes of value tensor
 * @param[in] value_strides    Strides of value tensor
 * @param[in] key_cache_sizes  Sizes of key cache tensor
 * @param[in] key_cache_strides Strides of key cache tensor
 * @param[in] value_cache_sizes Sizes of value cache tensor
 * @param[in] value_cache_strides Strides of value cache tensor
 * @param[in] slot_mapping_sizes Sizes of slot mapping tensor
 * @param[in] slot_mapping_strides Strides of slot mapping tensor
 * @param[in] key_ndims         Number of dimensions in key tensor
 * @param[in] value_ndims       Number of dimensions in value tensor
 * @param[in] key_cache_ndims   Number of dimensions in key cache tensor
 * @param[in] value_cache_ndims Number of dimensions in value cache tensor
 * @param[in] slot_mapping_ndims Number of dimensions in slot mapping tensor
 * @param[in] num_tokens       Number of tokens to process
 * @param[in] num_heads        Number of attention heads
 * @param[in] head_size        Size of each attention head
 * @param[in] block_size       Size of each cache block
 * @param[in] key_dtype        Data type of key tensor
 * @param[in] value_dtype      Data type of value tensor
 * @param[in] key_cache_dtype  Data type of key cache tensor
 * @param[in] value_cache_dtype Data type of value cache tensor
 * @param[in] kv_cache_type    Type of KV cache quantization ("fp8", "fp8_e4m3", "auto", etc.)
 * @param[in] k_scale          Scale factor for key quantization [1] or [num_heads]
 * @param[in] v_scale          Scale factor for value quantization [1] or [num_heads]
 * @param[in] k_scale_size     Size of k_scale tensor (must be 1 or num_heads)
```

```
 * @param[in] v_scale_size    Size of v_scale tensor (must be 1 or num_heads)
 *
 * @note Memory Layout:
 *       - Input tensors (key, value) have shape [num_tokens, num_heads,
head_size]
 *       - Cache tensors have shape [num_blocks, block_size, num_heads,
head_size]
 *       - Supports non-contiguous tensors through explicit stride parameters
 *
 * @note Scale Factors:
 *       - k_scale can be either a single value [1] or per-head values
[num_heads]
 *       - v_scale can be either a single value [1] or per-head values
[num_heads]
 *       - When size is 1, the same scale is applied to all heads
 *       - When size is num_heads, each head uses its own scale factor
 *
 * @note Behavior:
 *       - Reshapes key and value tensors to match the cache format
 *       - Maps tokens to their corresponding cache slots using slot_mapping
 *       - Supports optional quantization to FP8 format with scaling
 *       - Ignores tokens with negative slot indices (padding tokens)
 */
void reshape_and_cache_flash(
    const void* key,                    // Key tensor data pointer
    const void* value,                  // Value tensor data pointer
    void* key_cache,                    // Key cache tensor data pointer
    void* value_cache,                  // Value cache tensor data pointer
    const int32_t* slot_mapping,        // Slot mapping data pointer
    const int64_t* key_sizes,           // Key tensor sizes
    const int64_t* key_strides,         // Key tensor strides
    const int64_t* value_sizes,         // Value tensor sizes
    const int64_t* value_strides,       // Value tensor strides
    const int64_t* key_cache_sizes,     // Key cache tensor sizes
    const int64_t* key_cache_strides,   // Key cache tensor strides
    const int64_t* value_cache_sizes,   // Value cache tensor sizes
    const int64_t* value_cache_strides, // Value cache tensor strides
    const int64_t* slot_mapping_sizes,  // Slot mapping tensor sizes
    const int64_t* slot_mapping_strides,// Slot mapping tensor strides
    int key_ndims,                       // Number of dimensions in key tensor
    int value_ndims,                     // Number of dimensions in value
tensor
    int key_cache_ndims,                 // Number of dimensions in key cache
tensor
    int value_cache_ndims,               // Number of dimensions in value
cache tensor
```

```
    int slot_mapping_ndims,          // Number of dimensions in slot mapping tensor
    int num_tokens,                  // Number of tokens to process
    int num_heads,                   // Number of attention heads
    int head_size,                   // Size of each attention head
    int block_size,                  // Size of each cache block
    int key_dtype,                   // Data type of key tensor
    int value_dtype,                 // Data type of value tensor
    int key_cache_dtype,             // Data type of key cache tensor
    int value_cache_dtype,           // Data type of value cache tensor
    const char* kv_cache_type,       // Type of KV cache quantization
    const float* k_scale,            // Scale factor for key quantization
    const float* v_scale,            // Scale factor for value quantization
    int k_scale_size,                // Size of k_scale tensor (1 or num_heads)
    int v_scale_size                 // Size of v_scale tensor (1 or num_heads)
);
```

concat and cache mla

介绍：

vllm中，DeepSeek V3结构的模型，所需要的KV cache管理算子，具体可参考：融合算子-cache_mla

函数声明：

代码块

```
/// @brief Concatenates kv_c and k_pe, and caches the result in kv_cache.
/// @tparam T Data type of kv_c and k_pe (e.g., sifmt::float16, sifmt::float32, sifmt::bfloat16)
/// supported datatype(dtype):
/// [sifmt::float16, sifmt::float32, sifmt::bfloat16]
/// @param kv_c [num_tokens, kv_lora_rank]
/// @param k_pe [num_tokens, pe_dim]
/// @param kv_cache [num_blocks, block_size, (kv_lora_rank + pe_dim)]
/// @param slot_mapping [num_tokens]
/// @param block_stride stride 0 of kv_cache
/// @param entry_stride Stride 1 of kv_cache
/// @param kv_c_stride stride 0 of kv_c
/// @param k_pe_stride stride 0 of k_pe
/// @param kv_lora_rank dim 1 of kv_c
/// @param pe_dim  dim 1 of k_pe
```

```
/// @param block_size dim 1 of kv_cache
/// @param num_tokens number of tokens
/// @param scale scale factor for quantization
template <typename T>
void concat_and_cache_mla(
    const void* kv_c,             // [num_tokens, kv_lora_rank]
    const void* k_pe,             // [num_tokens, pe_dim]
    void* kv_cache,               // [num_blocks, block_size, (kv_lora_rank + pe_dim)]
    const int64_t* slot_mapping,  // [num_tokens]
    const int block_stride,                    //
    const int entry_stride,                    //
    const int kv_c_stride,                     //
    const int k_pe_stride,                     //
    const int kv_lora_rank,                    //
    const int pe_dim,                          //
    const int block_size,                      //
    const int num_tokens,                      //
    const float* scale                         //
);
```

gather_cache

介绍：

将KV cache汇总至指定位置，具体可参考：融合算子-gather_cache

函数声明：

代码块

```
/// @brief Gathers cache blocks from src_cache based on block_table and cu_seq_lens.
/// @tparam T Data type of src_cache and dst
/// supported datatype(dtype):
/// [uint32_t, uint16_t, uint8_t]
/// @param src_cache source cache tensor with shape [NUM_BLOCKS, BLOCK_SIZE, ENTRIES]
/// @param dst destination tensor with shape [TOT_TOKENS, ENTRIES]
/// @param block_table paged block table with shape[BATCH, BLOCK_INDICES]
/// @param cu_seq_lens cumsum sequence length tensor with shape [BATCH + 1]
/// @param batch_size batch size of request
/// @param block_size size of each cache block
/// @param entry_size size of each entry in the cache block
/// @param block_table_stride stride 0 of block_table
```

```
/// @param cache_block_stride stride 0 of src_cache
/// @param cache_entry_stride stride 1 of src_cache
/// @param dst_entry_stride stride 0 of dst
/// @param seq_starts nullptr or [BATCH]
template <typename T>
void gather_cache(
    const void* src_cache,        // [NUM_BLOCKS, BLOCK_SIZE, ENTRIES]
    void* dst,                    // [TOT_TOKENS, ENTRIES]
    const int32_t* block_table,   // [BATCH, BLOCK_INDICES]
    const int32_t* cu_seq_lens,   // [BATCH + 1]
    const int32_t batch_size,
    const int32_t block_size,
    const int32_t entry_size,
    const int64_t block_table_stride,
    const int64_t cache_block_stride,
    const int64_t cache_entry_stride,
    const int64_t dst_entry_stride,
    const int32_t* seq_starts)    // nullptr or [BATCH]
);
```

arange

- 介绍：生成一个从起始值到结束值按照指定步长递增的一维张量。
inType和outType不同时，支持：
  - inType=int, outType=bf16
  - inType=f32, outType=bf16

代码块

```
/// @tparam inType 支持的输入数据类型 [sifmt::bfloat16, sifmt::float32, int]
/// @tparam outType 支持的输出数据类型 [sifmt::bfloat16, sifmt::float32, int]
/// @param start 序列的起始值
/// @param end 序列的结束值（不包含）
/// @param step 相邻两个值之间的步长
/// @param out 输出张量，包含生成的序列
/// @param out_size 输出张量的大小
template <typename inType, typename outType>
void arange(const inType& start, const inType& end, const inType& step, void
*out, int64_t out_size);
```

pow

代码块

```
/// @brief 计算 base 张量中每个元素的幂
/// @tparam T 数据类型支持 [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param base 底数张量
/// @param exp 指数张量
/// @param out 输出张量
template <typename T>
void pow(sikernel::Tensor<T>& base, sikernel::Tensor<T>& exp,
sikernel::Tensor<T>& out);

```

isin

代码块

```
/// @brief 检查 query 张量中的每个元素是否存在于 table 张量中
/// @tparam T 数据类型支持 [int]
/// @param query 查询张量
/// @param table 表张量
/// @param out 输出张量
/// @param invert 是否反转结果（true 表示“不在”，false 表示“在”）
template <typename T>
void is_in(sikernel::Tensor<T> &query, sikernel::Tensor<T> &table,
sikernel::Tensor<bool> &out, bool invert);
```

cumsum

代码块

```
/// @brief 计算指定维度上的元素累加和
/// @tparam T 数据类型支持 [sifmt::bfloat16, sifmt::float16, sifmt::float32,
int]
/// @param self 输入张量
/// @param out 输出张量
/// @param dim 要进行累加和的维度
/// @note 输出张量的形状与输入张量相同。
template <typename T>
void cumsum(sikernel::Tensor<T>& self, sikernel::Tensor<T>& out, int64_t dim);
```

where

代码块

```
/// @brief Accoding to the condition tensor, get the element from self tensor or other tensor
/// @tparam T Datatype support [sizeof(T) = 1, 2 or 4]
/// @param out: out tensor
/// @param condition: condition tensor
/// @param self: self tensor
/// @param out_ndims: out_shape rank
/// @param out_shapes: the array of out_shapes
/// @param out_strides: the array of out_strides
/// @param self_strides: the array of self_strides
/// @param other_strides: the array of other_strides
template<typename T>
void where(const void* out, const void* condition, const void* self, const void* other, int out_ndims, const int64_t* out_shapes, const int64_t* out_strides, const int64_t* self_strides, const int64_t* other_strides);
```

replication_pad

代码块

```
/// @brief Accoding to pad info to pad the feature map
/// @tparam T Datatype support [sizeof(T) = 1, 2 or 4]
/// @param self: input tensor
/// @param output: output tensor
/// @param padding: padding params(left right top bottom front back)
 /// @param padding_arr_size: pad dim size
template<typename T>
void replication_pad_out(const sikernel::Tensor<T>& self, sikernel::Tensor<T>& output, const void* padding, int padding_arr_size);
```

reflection_pad

代码块

```
/// @brief Accoding to pad info to pad the feature map
/// @tparam T Datatype support [sizeof(T) = 1, 2 or 4]
/// @param self: input tensor
/// @param output: output tensor
/// @param padding: padding params(left right top bottom front back)
/// @param padding_arr_size: pad dim size
template<typename T>
void reflection_pad_out(const sikernel::Tensor<T>& self, sikernel::Tensor<T>& output, const void* padding, int padding_arr_size);
```

uniform

功能：生成指定范围内的均匀分布随机数。函数签名：

代码块

```
template <typename T>
void uniform(T from, T to, T *out, int64_t size, uint64_t seed, uint64_t
offset);
```

参数：

- from：下界（含）
- to：上界（不含）
- out：输出随机值
- size：元素个数
- seed：随机种子
- offset：生成器偏移

支持类型：sifmt::float32, sifmt::bfloat16, int

exponential

功能：逐元素计算e^x

代码块

```
template <typename T>
void exponential(void *self, void *out, float lambda, int64_t size);
```

参数：

- self：输入
- out：输出
- lambda：暂未使用
- size：元素个数

支持类型：sifmt::bfloat16, sifmt::float16, sifmt::float32

注意需要128B对齐

---

clamp

功能：将值限制在 [min, max] 范围内。函数签名：

```
template <typename T>
void clamp(void *self, void *out, void *min, void *max, int64_t self_size,
int64_t min_size, int64_t max_size);
```

参数：

- self：输入
- out：输出
- min：下界
- max：上界
- self_size：输入元素个数
- min_size：min 元素个数
- max_size：max 元素个数

支持类型：sifmt::bfloat16, sifmt::float16, sifmt::float32

conj_physical

功能：计算复数的物理共轭（取反虚部）。函数签名：

```
void conj_physical(sifmt::float32 *self, sifmt::float32 *out, int64_t size);
```

参数：

- self：输入（交错存储：实部0,虚部0,实部1,虚部1,...）
- out：输出（交错存储：实部0,-虚部0,实部1,-虚部1,...）
- size：复数个数

polar

功能：将极坐标 (r,θ)(*r*,*θ*) 转换为笛卡尔坐标 (x,y)

```
void polar(sifmt::float32 *theta, sifmt::float32 *r, sifmt::float32 *out,
int64_t size);
```

参数：

- theta：角度（弧度）
- r：半径
- out：输出坐标（格式：x0,y0,x1,y1,...）
- size：点数

输出大小：size * 2

min

功能：按指定维度计算最小值并返回索引。函数签名：

代码块

```
template <typename T>
void min(sikernel::Tensor<T>& self, sikernel::Tensor<T>& out_value,
sikernel::Tensor<int64_t>& out_index, int64_t dim);
```

参数：

- self：输入
- out_value：最小值
- out_index：最小值索引
- dim：计算维度

支持类型：sifmt::bfloat16, int

代码块

```
void count_expert_num_tokens(
    int *topk_ids,
    int *expert_num_tokens,
    long num_experts,
    long topk_numel,
    int *expert_map,
    bool has_expert_map);
```

ffn

silu

介绍：

输入为一个二维tensor，计算每个元素的silu值，计算公式为：
$f(x) = x * sigmoid(x) = x/(1 + e^{-x})$

函数声明：

代码块

```
/// @brief Computes the SiLU (Sigmoid Linear Unit) activation function, where the output for each element is x * sigmoid(x)
/// @tparam T Datatype support [sifmt::bfloat16, sifmt::float16, sifmt::float32]
/// @param d_in Input tensor
/// @param d_out Output tensor
/// @param dim0_size Size of dimension 0 for the input/output tensor
/// @param dim1_size Size of dimension 1 for the input/output tensor (total size must be 1024B aligned)
template <typename T>
void silu(void *d_in, void *d_out, int dim0_size, int dim1_size);
```

补充：

使用tile reg计算，dim0_size*dim1_size需1024B对齐。

misc

gather

介绍：按照索引从原tensor中提取对应的数据。具体来说，它根据指定的维度（dim）和索引（index）来从输入tensor中选取数据，并输出一个新的tensor。

out[i][j][k] = input[index[i][j][k]][j][k] # if dim == 0
out[i][j][k] = input[i][index[i][j][k]][k] # if dim == 1
out[i][j][k] = input[i][j][index[i][j][k]] # if dim == 2

函数声明：

代码块

```
// Returns a new tensor whose shape is same with index, the values is gathered along an axis specified by dim from input tensor.
// d_in is the input tensor with shape (in_dim0, in_dim1)
// d_index is the index tensor with shape (index_dim0, index_dim1)
// d_out is the output tensor
template <typename T>
```

```
6   void gather(void *d_in, void *d_index, void *d_out, int dim, int in_dim0, int
    in_dim1, int index_dim0, int index_dim1);
```

scatter

介绍：将scalar或者tensor按照索引写入原tensor。具体来说，它根据指定的维度（dim）和索引（index）来指定写入的位置。

self**[index[i][j][k]][j][k] = src[i][j][k]** *# if dim == 0*
self**[i][index[i][j][k]][k] = src[i][j][k]** *# if dim == 1*
self**[i][j][index[i][j][k]] = src[i][j][k]** *# if dim == 2*

函数声明：

代码块

```
1   /// @brief Writes a scalar value into input tensor at the indices specified in
    the index tensor for dimension = dim
2   /// @tparam T Datatype support [bool]
3   /// @param d_in Input tensor (shape: in_dim0 x in_dim1)
4   /// @param d_index Index tensor (shape: index_dim0 x index_dim1)
5   /// @param Svalue Scalar bool value to write into the input tensor
6   /// @param dim Dimension along which to gather (0: dim0, 1: dim1)
7   /// @param in_dim0 Size of input tensor dimension 0
8   /// @param in_dim1 Size of input tensor dimension 1
9   /// @param index_dim0 Size of index tensor dimension 0
10  /// @param index_dim1 Size of index tensor dimension 1
11  template <typename T>
12  void scatter(void *d_in, void *d_index, bool Svalue, int dim, int in_dim0, int
    in_dim1, int index_dim0, int index_dim1);
```

hp_to_mx

代码块

```
1   /// @brief Converts a high-precision tensor (such as float16, float32,
    bfloat16) to a low-precision tensor (such as mxint8, mxfloat6e3m2 or
    mxfloat6e2m3).
2   /// @tparam MX_T Target low-precision type (e.g., sifmt::mxint8,
    sifmt::mxfloat6e3m2, sifmt::mxfloat6e2m3)
3   /// @tparam HP_T Source high-precision type (e.g., sifmt::float16,
    sifmt::float32, sifmt::bfloat16)
4   /// @param mx_ptr Output tensor pointer (low-precision)
5   /// @param hp_ptr Input tensor pointer (high-precision)
6   /// @param batch_size Batch size
7   /// @param dim0 Size of dimension 0
```

```
/// @param dim1 Size of dimension 1
/// @note Supports multiple type conversions. See MX_T and HP_T template parameters for supported types.
/// @note Data alignment requirements:
///       - For mxint8 output, total element count (batch_size * dim0 * dim1) must be aligned to 1024.
///       - For mxf6e3m2/mxf6e2m3 output, total element count must be aligned to 1024*4.
template <typename MX_T, typename HP_T>
void hp_to_mx(void* hp_ptr, void* mx_ptr, int batch_size, int dim0, int
```

Supported list

categary	name	dataty
	tile_mma_32x32_dte_bf16_bf16_bf16_tiled_tensor	bf16->
	tile_mma_32x32_bf16_f8e4m3_f8e4m3_tiled_tensor	f8e4m
Blas.L1.dot	tile_dot_f16	f16->f1
	copy_from	any
	concat	any
attention.softmax	softmax_f32	f32
	softmax_bf16	bf16
	softmax_tile	f32
attention.act_quant	act_quant	bf16->
attention.rms_norm	rms_norm	bf16/f1
attention.masked_fill	masked_fill	any
attention.sum	sum	f32->f3
attention.floor_div	floor_div_f32	f32->f3
	floor_div_bf16	bf16->
attention.index_select	index_select	f32/f16
attention.topk	topk	bf16/f1
attention.gt_tensor	gt_tensor	f32/f16
attention.ge_scalar	ge_scalar	f32/f16
attention.bincount	bincount	int64_
attention.mean_dim	mean_dim_f32	f32->f3
	mean_dim_bf16	bf16->
misc.gather	gather	f32->f3
attention.eq	eq_s32	i32->b
	eq_f32	f32->b
	eq_bf16	bf16->
attention.nonzero	nonzero_2d_s32	i32->i3
	nonzero_2d_b8	bool->
	nonzero_2d_no_tuple_s32	i32->i3
	nonzero_2d_no_tuple_b8	bool->
attention.max	max	f32/f16
attention.amax	amax	f32/f16
attention.argmax	argmax	f32/f16
misc.scatter	scatter	bool

misc.scatter	scatter	bool
attention.index	index	f32/f16
	index_put	f32/f16
attention.compare	CompareOp: gt、ge、lt、le	f32/f16
attention.bitwise	BitwiseOp: and、or、not	f32/f16
attention.decode_attn	flash_attn_decode	bf16
attention.all_any	all_any	bool/u
attention.arange	arange	f32/f16
attention.pow	pow	f32/f16
attention.isin	isin	i32
attention.cumsum	cumsum	f32/f16
attention.uniform	uniform	f32/bf1
attention.exponential	exponential	f32/f16
attention.clamp	clamp	f32/f16
attention.conj_physical	conj_physical	f32
attention.polar	polar	f32
attention.min	min	bf16/i