# 曦云® 系列通用计算 GPU

## MXMACA® C++ 编程指南

CSPG-23006-020-F3_V03　　2024-03-15

沐曦专有和三级保密信息　　本文档受 NDA 管控

# 声明

# 更新记录

| 版本 | 日期 | 更新说明 |
| --- | --- | --- |
| V03 | 2024-03-15 | 新增曦云®系列 GPU 产品信息 |
| V02 | 2023-12-29 | 描述性修改及格式修正 |
| V01 | 2023-10-16 | 正式版本首次发布 |

# 目录

# 1 C++ 语言扩展

## 1.1 函数执行空间标识符

### 1.1.1 __global__

使用 __global__ 执行空间标识符声明的函数称为 kernel 函数，该函数：

- 在 device（设备侧）上执行。
- 可以被 host（主机侧）调用。
- 可以被 device（设备侧）调用。

一个 kernel 函数的返回类型必须是 void，且该函数不能是 class 的成员函数。详细信息参见执行配置。

对于 kernel 函数的调用默认是异步的。

### 1.1.2 __device__

使用 __device__ 执行空间标识符声明的函数称为 device（设备侧）函数，该函数：

- 在 device（设备侧）上执行。
- 只能被 device（设备侧）函数或 kernel 函数调用。

---

**注解:** __global__ 和 __device__ 执行空间标识符不能同时修饰同一个函数。

---

### 1.1.3 __host__

使用 __host__ 执行空间标识符声明的函数称为 host（主机侧）函数，该函数：

- 在 host（主机侧）执行。
- 只能被 host（主机侧）函数调用。

只使用 __host__ 执行空间标识符声明函数与不使用 __host__，__device__ 或者 __global__ 中的任意一个执行空间标识符的效果是等价的，两种方式声明的函数都表示该函数是 host（主机侧）函数。

---

**注解:** __global__ 与 __host__ 执行空间标识符不可以同时使用。但是 __device__ 与 __host__ 执行空间标识符可以同时使用，在这种情况下，函数将同时为 host 与 device 所编译。

---

宏 __MACA_ARCH__ 可以用来区分 host 与 device 之间的不同的代码路径：

```
__host__ __device__ void func() {
#ifdef __MACA_ARCH__
    // Device 代码路径
#else
    // Host 代码路径
#endif
    // Host/Device 可复用代码
}
```

### 1.1.4 未定义的行为

当：

- 定义了 __MACA_ARCH__，从 __global__，__device__ 或者 __host__ __device__ 函数调用 __host__ 函数。
- 未定义 __MACA_ARCH__，从 __host__ 函数调用 __device__ 函数。

出现上述两种情形之一的情况的时候称为“跨执行空间”的调用行为，这样的行为是未定义的。

### 1.1.5 __noinline__ 与 __forceinline__

__noinline__：该标识符可以提示编译器不要内联该函数。

__forceinline__：该函数标识符可以用来提示编译器强制内联该函数。

**注解：** __noinline__ 与 __forceinline__ 函数标识符不可以同时使用，并且都不能用于内联函数。

## 1.2 变量存储空间标识符

变量存储空间标识符标明一个变量在 device 上的内存位置。

一个 device 上定义的没有__device__，__constant__，和__shared__ 这些变量存储空间修饰符修饰的 automatic 变量一般会存储在寄存器中。但是在一些情况下，编译器会将其放入显存，这将会导致性能瓶颈。

### 1.2.1 __device__

__device__ 内存空间标识符声明一个存储在 device 上的变量。

在__constant__，__shared__，和__managed__ 中描述的内存空间标识符最多有一个可以与 __device__ 一起使用，来进一步表示变量所属的内存空间。如果没有任何一个这样的额外内存空间标识符出现，则该 device 变量：

- 存储于 global 内存空间。
- 拥有创建它的 MXMACA® 上下文的相同的生命周期。
- 每个设备上都有独自的对象。
- 可以被 grid 中所有的 thread 访问。

### 1.2.2 __constant__

__constant__ 内存空间标识符，可选择与 __device__ 一起使用，该 constant 变量：

- 存储于 constant 内存空间。
- 拥有与创建它的 MXMACA 上下文相同的生命周期。
- 每个设备上都有独立的对象。
- 可以被 grid 中所有的 thread 访问。

### 1.2.3 __shared__

__shared__ 内存空间标识符，可选择与 __device__ 一起使用，该 shared 变量：

- 存储于一个 block 的 shared 内存空间。
- 与该 block 共享生命周期。
- 每个 block 中都有一个独立的对象。
- 只能被所处的 block 线程访问。
- 没有固定的地址。

当在 shared 内存声明一个 external 数组，比如：

```
extern __shared__ float shared[];
```

数组的大小在 launch time 决定，参见执行配置。所有以这种方式声明的变量，在内存中有同样的起始地址，因此数组中的变量分布必须通过偏移量显式管理。例如：

```
short array0[128];
float array1[64];
int array2[256];
```

在动态分配的 shared 内存中，可以使用如下方式声明、初始化这些数组：

```
extern __shared__ float array[];
__device__ void func()      // __device__ 或者__global__ 函数
{
    short* array0 = (short *)array;
    float* array1 = (float*)&array0[128];
    int* array2 = (int*)&array1[64];
}
```

需要注意的是指针需要根据它指向的类型对齐，因此如下代码将无法正常工作（array1 没有 4 字节对齐）：

```
extern __shared__ float array[];
__device__ void func() // __device__ 或者 __global__ 函数
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[127];
}
```

内置向量类型的对齐要求参见表 1.1。

### 1.2.4 __managed__

__managed__ 内存空间标识符，可选择与 __device__ 一起使用，声明这样一个变量：

- 可以同时被 device 与 host 的代码引用，它的地址可以从 device 或者 host 函数中获取，也可以从 device 或者 host 函数中直接读取或者写入。
- 拥有与应用程序相同的生命周期。

## 1.3 内置类型

### 1.3.1 maca_bfloat16

maca_bfloat16 是一种通过截断 float 类型的后 16bit 而得到的半精度浮点格式，从编码角度而言，它就是一个尾数只有 7bit 长的 float，可以被视为用精度换内存的 float。

其内存结构如下所示：

- Sign bit（符号位）：1 bit
- Exponent width（指数位宽）：8 bits
- Significand precision（尾数精度）：7 bits

maca_bfloat16 的动态范围大约是 [-3.4e38, -1.18e-38] ∪ [1.18e-38, 3.4e38]，最多可具有三个有效的小数位数。在对精度要求不高的场景下，使用 maca_bfloat16 替代 float 可以显著减小数据拷贝的开销。

#### 1.3.1.1 maca_bfloat162 与 maca_bflaot16 关系

maca_bflaot162 为 maca_bflaot16 的 vector 2 形式，由两个 maca_bflaot16 组成，有高低 16 位的空间之分，共同占一个 32 位的空间。

maca_bflaot162 相关的函数都是高低 16 位的数据分别进行操作，而不是以 32 位的数据作为一个整体共同进行操作处理。

### 1.3.2 Half-precision floating-point format（半精度浮点格式）

与 bfloat16 不同，half 是一种新的 16 位浮点格式。几乎现在所有用法都遵循 IEEE 754-2008 标准，其中 16 位 half 格式称为 **binary16**，指数使用 5 位。这可以表示 ±65,504 范围内的值，大于 1 的最小值为 1 + 1/1024。

#### 1.3.2.1 IEEE 754 - 2008 半精度二进制浮点格式：binary16

IEEE 754 标准指定了一个 **binary16** 要有如下的格式：

- Sign bit（符号位）：1 bit
- Exponent width（指数位宽）：5 bits
- Significand precision（尾数精度）：11 bits （有 10 位被显式存储）

除非指数位全是 0，否则就会假定隐藏的起始位是 1。因此只有 10 位尾数在内存中被显示出来，而总精度是 11 位。据 IEEE 754 的说法，虽然尾数只有 10 位，但是尾数精度是 11 位的 ($\log_{10}(2^{11}) \approx 3.311$ 十进制数)。

#### 1.3.2.2 指数编码

半精度二进制浮点指数使用偏移二进制表示进行编码，零偏移量为 15；在 IEEE 754 标准中也称为指数偏差。

- Emin = $00001_2$ - $01111_2$ = -14
- Emax = $11110_2$ - $01111_2$ = 15
- 指数偏差 = $01111_2$ = 15

因此，为了获得真正的指数，必须从存储的指数中减去偏移量 15。

| 指数 | 尾数 $= 0$ | 尾数 $\neq 0$ | 方程 |
|---|---|---|---|
| $00000_2$ | 0，-0 | 次正规数 | $(-1)^{符号位} \times 2^{-14} \times 0.尾数位_2$ |
| $00001_2, \cdots, 11110_2$ | 正规数 | | $(-1)^{符号位} \times 2^{指数-15} \times 1.尾数位_2$ |
| $11111_2$ | ±infinity | NaN (quiet, signalling) | |

最小正次正规数为 $2^{-24} \approx 5.96 \times 10^{-8}$ 。最小正正规数为 $2^{-14} \approx 6.10 \times 10^{-5}$。最大可表示值为 $(2-2^{-10}) \times 2^{15} = 65504$。

#### 1.3.2.3 半精度示例

这些示例以位表示形式给出浮点值。这包括符号位、（偏置）指数和尾数。

| Binary | Hex | Value | Notes |
|---|---|---|---|
| 0 00000 0000000000 | 0000 | 0 | |
| 0 00000 0000000001 | 0001 | $2^{-14} \times (0 + 1/1024) \approx 0.000000059604645$ | 最小正次正规数 |
| 0 00000 1111111111 | 03ff | $2^{-14} \times (0 + 1023/1024) \approx 0.000060975552$ | 最大次正规数 |
| 0 00001 0000000000 | 0400 | $2^{-14} \times (1 + 0/1024) \approx 0.00006103515625$ | 最小正数 |
| 0 01101 0101010101 | 3555 | $2^{-2} \times (1 + 341/1024) \approx 0.33325195$ | 最接近 1/3 的值 |
| 0 01110 1111111111 | 3bff | $2^{-1} \times (1 + 1023/1024) \approx 0.99951172$ | 小于 1 的最大数字 |
| 0 01111 0000000000 | 3c00 | $2^0 \times (1 + 0/1024) = 1$ | 1 |
| 0 01111 0000000001 | 3c01 | $2^0 \times (1 + 1/1024) \approx 1.00097656$ | 大于 1 的最小数字 |
| 0 11110 1111111111 | 7bff | $2^{15} \times (1 + 1023/1024) = 65504$ | 最大数 |
| 0 11111 0000000000 | 7c00 | $\infty$ | 无穷大 |
| 1 00000 0000000000 | 8000 | -0 | |
| 1 10000 0000000000 | c000 | -2 | |
| 1 11111 0000000000 | fc00 | $-\infty$ | 负无穷大 |

默认情况下，因为有效位数为奇数，1/3 会像双精度一样向下舍入。超出舍入点的位是 0101…在最后一位不到一个单位的 1/2。

#### 1.3.2.4 half2 与 half 关系

half2 为 half 的 vector 2 形式，由两个 half 组成，有高低 16 位的空间之分，共同占一个 32 位的空间。

通常情况下，half2 相关的函数都是高低 16 位的数据分别进行操作，而不是以 32 位的数据作为一个整体共同进行操作处理。

## 1.4 内置向量类型

### 1.4.1 char，short，int，long，longlong，float，double

这些向量类型都是从基础的整数与浮点数类型中衍生而来，其本质为结构体。第一个，第二个，第三个和第四个分量可以通过字段 x，y，z，w 进行访问。它们都带有相同形式的构造函数 make_<type name>，比如：

```
int2 make_int2(int x, int y);
```

这样就可以使用数据 (x, y) 创建一个 int2 的向量类型。

详细的向量类型对齐要求参见表 1.1。

表 1.1 对齐要求

| 类型 | 对齐 |
|---|---|
| char1, uchar1 | 1 |
| char2, uchar2 | 2 |
| char3, uchar3 | 3 |
| char4, uchar4 | 4 |
| short1, ushort1 | 2 |
| short2, ushort2 | 4 |
| short3, ushort3 | 2 |
| short4, ushort4 | 8 |
| int1, uint1 | 4 |
| int2, uint2 | 8 |
| int3, uint3 | 4 |
| int4, uint4 | 16 |
| long1, ulong1 | 4 if sizeof(long) is equal to sizeof(int) 8, otherwise |
| long2, ulong2 | 8 if sizeof(long) is equal to sizeof(int), 16, otherwise |
| long3, ulong3 | 4 if sizeof(long) is equal to sizeof(int), 8, otherwise |
| long4, ulong4 | 16 |
| longlong1, ulonglong1 | 8 |
| longlong2, ulonglong2 | 16 |
| longlong3, ulonglong3 | 8 |
| longlong4, ulonglong4 | 16 |
| float1 | 4 |
| float2 | 8 |
| float3 | 4 |
| float4 | 16 |
| double1 | 8 |
| double2 | 16 |
| double3 | 8 |
| double4 | 16 |

### 1.4.2 dim3

这个类型是一个基于 uint3 的整数向量类型，它的用途是来指定维度的。在定义一个 dim3 类型的变量时，任何未指定的组成部分都会被默认初始化为 1。

## 1.5 内置变量

内置变量指定 grid 与 block 的维度以及 block 与 thread 的索引。它们的有效域只在 device 的函数中。

### 1.5.1 gridDim

这个变量是一个内置的 dim3 类型变量（参考dim3），其包含 grid 的维度信息。

### 1.5.2 blockIdx

这个变量是一个内置的 uint3 类型变量（参考char，short，int，long，longlong，float，double），其包含 grid 中的 block 索引。

### 1.5.3 blockDim

这个变量是一个内置的 dim3 类型变量（参考dim3），其包含 block 的维度信息。

### 1.5.4 threadIdx

这个变量是一个内置的 uint3 类型变量（参考char，short，int，long，longlong，float，double），其包含 block 中的 thread 索引。

### 1.5.5 warpSize

这个变量是一个内置的 int 类型变量，其包含 threads 的 warp 大小。

## 1.6 Memory Fence 函数

MXMACA 编程模型假定 device 是一个弱有序的内存模型，即不同的 MXMACA 线程把数据写入 shared 内存、global 内存、page-locked host 内存或者 peer device 内存的顺序并不一致。当没有同步操作时，两个 thread 同时读或者写一个相同内存地址是未知的行为。

在下面的例子中，线程 1 执行 writeXY()，而线程 2 执行 readXY()。

```
__device__ int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    Y = 20;
}
__device__ void readXY()
{
    int B = Y;
    int A = X;
}
```

这两个线程分别读、写相同的内存地址。由于任何数据竞争都是不确定的行为，并且是没有定义的语义，所以 A 与 B 的结果可以是任何值。

Memory Fence 函数可以强制保证内存访问的顺序一致性。Memory Fence 函数在强制排序的范围上有所不同，但它们独立于访问的内存空间（shared 内存、global 内存、page-locked host 内存以及 peer device 的内存）。

```
void __threadfence_block();
```

必要条件：

- 调用线程在调用 __threadfence_block() 之前的内存写操作行为，都会被 block 内所有已经调用了 __threadfence_block() 的线程观察到。
- 调用线程在调用 __threadfence_block() 之前的所有内存读操作，会在调用 __threadfence_block() 后的内存读操作之前执行，从而保证了有序性。

```
void __threadfence();
```

行为与 __threadfence_block() 类似。确保调用线程在调用 __threadfence() 之后，在其他线程调用 __threadfence() 之前，对内存不会进行写操作。注意，为了保证这个顺序正确，观察线程必须真正的观察内存，并且不应该对其进行缓存。

```
void __threadfence_system();
```

行为与 __threadfence_block() 类似。确保调用线程在调用 __threadfence_system() 之前，对内存的写操作，能被调用 __threadfence_system() 后的 device 侧线程、host 侧线程以及 peer device 线程，在调用线程写内存操作之前所观察到。

在之前的代码实例中，可以参照如下代码增加对应的 fence 函数：

```
__device__ int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    __threadfence();
    Y = 20;
}
__device__ void readXY()
{
    int B = Y;
    __threadfence();
    int A = X;
}
```

对于这段代码，可以观察得到的结果如下：

- A 的值为 1 并且 B 的值为 2。
- A 的值为 10 并且 B 的值为 2。
- A 的值为 10 并且 B 的值为 20。

第四个结果是不可能的，因为第一个写操作必须在第二个写操作之前可见。

- 如果 thread 1 和 2 属于同一个 block，使用 __threadfence_block() 就足够。
- 如果 thread 1 和 2 不属于同一个 block：
  - 如果它们是属于同一个 MXMACA device，必须使用 __threadfence()。
  - 如果它们属于不同的 MXMACA device，必须使用 __threadfence_system()。

一个常见的用例就是当线程使用其他线程产生的一些数据，如下面的代码所示，该 kernel 函数在一次调用中计算 N 个数字的数组的和。每个 block 首先对于数组的一个子集求和，并且将结果存储到全局内存中。当所有 block 执行完毕，最后一个 block 从全局内存读取这些数据和并且将它们相加以获得最终结

果。为了确定哪个 block 是最后完成的，每个 block 都原子地增加一个计数器，以表示它已经完成了部分求和计算以及存储（参考Atomic 函数 关于 atomic 函数的描述）。最后个 block 是收到计数器的值等于 gridDim.x-1。如果在存储部分和递增计数器之间没有设置栅栏，计数器可能在存储结果之前就递增了。因此可能到达 gridDim.x-1，让最后个 block 在内存更新之前开始读部分求和结果。

Memory fence 函数只影响现成的内存操作的顺序；它们不会确保内存操作对于其他线程是可见的（像 __syncthreads() 对于同一个 block 所做的事那样（参考同步函数））。在下面的代码示例中，result 的内存操作的可见性，是通过将其声明为 volatile 来确保的。

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float *array, unsigned int N,
                    volatile float *result)
{
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0)
    {
        // Thread 0 of each block stores the partial sum
        // to global memory. The compiler will use
        // a store operation that bypasses the L1 cache
        // since the "result" variable is declared as
        // volatile. This ensures that the threads of
        // the last block will read the correct partial
        // sums computed by all other blocks.
        result[blockIdx.x] = partialSum;
        // Thread 0 makes sure that the incrementation
        // of the "count" variable is only performed after
        // the partial sum has been written to global memory.
        __threadfence();
        // Thread 0 signals that it is done.
        unsigned int value = atomicInc(&count, gridDim.x);
        // Thread 0 determines if its block is the last
        // block to be done.
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    // Synchronize to make sure that each thread reads
    // the correct value of isLastBlockDone.
    __syncthreads();
    if (isLastBlockDone)
    {
        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0)
        {
            // Thread 0 of last block stores the total sum
            // to global memory and resets the count
            // varialble, so that the next kernel call
            // works properly.
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

## 1.7 同步函数

```
void __syncthreads();
```

一直等到 block 中的所有的 thread 到达这个点，并且这些 thread 进行的所有的在 __syncthreads() 之前的 global 与 shared 内存访问都被这个 block 中的所有的 thread 观测到。

__syncthreads() 是用来进行同 block 中的 thread 的协调沟通的。当 block 中的一些 thread 访问 shared 或者 global 内存中的相同的地址的时候，对于这些内存访问有一些潜在的“读在写之后”，“写在读之后”或者“写在写之后”的风险。这些数据风险可以通过在这些访问之间同步线程来规避。

__syncthreads() 也能出现在条件语句的代码中，但是仅当条件在线程块上计算的结果相同时，否则代码的执行可能会挂起或产生意外的副作用。

```
int __syncthreads_count(int predicate);
```

这个接口与 __syncthreads 相同，但是有额外的特性，它会为 block 的所有线程计算 predicate，并返回 predicate 计算为非零的线程数。

```
int __syncthreads_and(int predicate);
```

这个接口与 __syncthreads 相同，但是有额外的特性，它会为 block 的所有线程计算 predicate，并且 predicate 计算为非零的线程会返回非零。

```
int __syncthreads_or(int predicate);
```

这个接口与 __syncthreads 相同，但是有额外的特性，它会为 block 的所有线程计算 predicate，并且 predicate 计算为非零的线程会返回非零。

```
void __syncwarp(unsigned long mask=0xffffffffffffffffUL);
```

这个接口将会导致正在执行的 thread 一直等到所有的 warp 中 mask 标记的 lane 执行了 __syncwarp()（有相同的 mask）才会恢复执行。所有的 mask 中标记的未退出的 thread 必须以相同的 mask 执行一个对应的 __syncwarp()，否则结果是不确定的。

执行 __syncwarp() 保证参与 barrier 的 thread 之间的内存顺序。因此，一个 warp 中的 thread 希望能通过内存交流以保存内存，执行 __syncwarp，然后安全地读取 warp 中别的 thread 存储的值。

## 1.8 数学函数

半精度（half）类型介绍，参见Half-precision floating-point format（半精度浮点格式）。

### 1.8.1 半精度（half）算术函数

```
__device__ __half __habs (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的绝对值

**函数描述**

- 计算输入的半精度数字的绝对值，并返回结果。

```
__device__ __half __hadd (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，a 加 b 的和

**函数描述**

- 对 a 和 b 求和。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half __hadd_sat (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，a 加 b 的和。

**函数描述**

- 对 a 和 b 求和。把求到的结果向最近偶数舍入后，把结果限制在区间 [0.0, 1.0] 中，NaN 的结果刷新为 +0.0，返回结果。

```
__device__ __half __hdiv (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，a 除以 b 的商

**函数描述**

- 计算 a 除以 b 的商。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half __hfma (const __half a, const __half b, const __half c)
```

**参数**

a

- half，只读入参

b

- half，只读入参

c

- half，只读入参

**返回值**

- half，a 乘 b 加 c 的结果

**函数描述**

- 计算 a 乘 b 的积，再用这个积与 c 求和。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half __hfma_relu (const __half a, const __half b, const __
↪half c)
```

**参数**

a

- half，只读入参

b

- half，只读入参

c

- half，只读入参

**返回值**

- half，返回 a 乘 b 加 c 的结果

**函数描述**

- 以 relu 饱和度运算形式，计算 a 乘 b 的积，再用这个积与 c 求和。把求到的结果向最近偶数舍入后，返回结果。
- 如果结果为负数，则返回 0；如果结果为 NaN，则返回 canonical NaN。

```
__device__ __half __hfma_sat (const __half a, const __half b, const __half
↪c)
```

**参数**

a

- half，只读入参

b

- half，只读入参

c

- half，只读入参

**返回值**

- half，返回 a 乘 b 加 c 的结果

**函数描述**

- 计算 a 乘 b 的积，再用这个积与 c 求和。把求到的结果向最近偶数舍入后，把结果限制在区间 [0.0, 1.0] 上。
- 如果求和的结果为 NaN，则返回 +0.0。

```
__device__ __half __hmul (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，a 乘 b 的积

**函数描述**

- 计算 a 乘 b 的积。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half __hmul_sat (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，以饱和度的形式，返回 a 乘 b 的积

**函数描述**

- 计算 a 乘 b 的积。把求到的结果向最近偶数舍入后，把结果限制在区间 [0.0, 1.0] 上。
- 如果乘积的结果为 NaN，则返回 +0.0。

```
__device__ __half __hneg (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的取反值

**函数描述**

- 对 a 取反值，并返回结果。

```
__device__ __half __hsub (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，a 减 b 的差

**函数描述**

- 计算 a 减 b 的差。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half __hsub_sat (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，返回 a 减 b 的差

**函数描述**

- 计算 a 减 b 的差。把求到的结果向最近偶数舍入后，把结果限制在区间 [0.0, 1.0] 中，NaN 的结果刷新为 +0.0，并返回结果。

### 1.8.2 half2 算术函数

```
__device__ __half2 __h2div (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，a 除以 b 的商

**函数描述**

- 计算 a 除以 b 的商。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half2 __habs2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回 a 的绝对值

**函数描述**

- 对输入的 half2 结构中的两个 half 求绝对值，并返回结果。

```
__device__ __half2 __hadd2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，a 加 b 的和

**函数描述**

- 对 a 和 b 求和。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half2 __hadd2_sat (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 加 b 的和

**函数描述**

- 对 a 和 b 求和。把求到的结果向最近偶数舍入后，结果限制在区间 [0.0, 1.0] 上，返回结果。

```
__device__ __half2 __hcmadd (const __half2 a, const __half2 b, const __
↪half2 c)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

c

- half2，只读入参

**返回值**

- half2，a 乘 b 加 c 的结果

**函数描述**

- 函数会将入参理解为半精度的复数。
- 计算 a 乘 b 的积，再用这个积与 c 求和。

```
__device__ __half2 __hfma2 (const __half2 a, const __half2 b, const __
↪half2 c)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

c

- half2，只读入参

**返回值**

- half2，a 乘 b 加 c 的结果

**函数描述**

- 计算 a 乘 b 的积，再用这个积与 c 求和。把求到的结果向最近偶数舍入后，返回结果。

```
__device__ __half2 __hfma2_relu (const __half2 a, const __half2 b, const __
↪half2 c)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

c

- half2，只读入参

**返回值**

- half2，返回 a 乘 b 加 c 的结果

**函数描述**

- 以 relu 饱和度运算形式，计算 a 乘 b 的积，再用这个积与 c 求和。把求到的结果向最近偶数舍入后，返回结果。
- 如果结果为负数，则返回 0，如果结果为 NaN，则返回 canonical NaN。

```
__device__ __half2 __hfma2_sat (const __half2 a, const __half2 b, const __
↪half2 c)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

c

- half2，只读入参

**返回值**

- half2，返回 a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式计算 a 乘 b 的积，再用这个积与 c 求和。结果限制在 [0.0, 1.0] 范围中。
- 如果求和的结果为 NaN，则返回 +0.0。

```
__device__ __half2 __hmul2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式计算 a 乘 b 的积。

```
__device__ __half2 __hmul2_sat (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式计算 a 乘 b 的积。结果限制在 [0.0, 1.0] 范围中。
- 如果乘积的结果为 NaN，则返回 +0.0。

```
__device__ __half2 __hneg2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回 a 的取反值

**函数描述**

- 对输入的 half2 结构中的两个 half 取反值，并返回结果。

```
__device__ __half2 __hsub2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式计算 a 减 b 的差。

```
__device__ __half2 __hsub2_sat (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式计算 a 减 b 的差。把结果限制在区间 [0.0, 1.0] 上。

### 1.8.3 half 比较函数

```
__device__ bool __heq (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测 a 和 b 是否相等

**函数描述**

- 比较 a 和 b。如果相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hequ (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较 a 和 b 是否相等

**函数描述**

- 无序比较 a 和 b。如果相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hge (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测关系是否正确：a ⩾ b

**函数描述**

- 比较 a 和 b。如果 a ⩾ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hgeu (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a ⩾ b

**函数描述**

- 比较 a 和 b。如果 a ⩾ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hgt (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b。如果 a 大于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hgtu (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b。如果 a 大于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ int __hisinf (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- int，检测 a 是否为无穷

**函数描述**

- 如果 a 是负无穷，返回-1。
- 如果 a 是正无穷，返回 1。
- 如果 a 不是无穷，返回 0。

```
__device__ bool __hisnan (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- bool，检测 a 是否是 NaN

**函数描述**

- 如果 a 是 NaN，返回 true。
- 如果 a 不是 NaN，返回 false。

```
__device__ bool __hle (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测关系是否正确：a ⩽ b

**函数描述**

- 比较 a 和 b。如果 a ⩽ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hleu (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a ⩽ b

**函数描述**

- 无序比较 a 和 b。如果 a ⩽ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hlt (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测关系是否正确：a 小于 b

**函数描述**

- 比较 a 和 b。如果 a 小于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hltu (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 小于 b

**函数描述**

- 无序比较 a 和 b。如果 a 小于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ __half __hmax (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，返回 a 和 b 中，更大的值

**函数描述**

- 如果 a 大于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回另一个不是 NaN 的参数。
- 如果两个参数都是 NaN，返回 canonical NaN。

- 如果两个入参都是 0.0，+0.0 > -0.0。

```
__device__ __half __hmax_nan (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，返回 a 和 b 中，更大的值

**函数描述**

- 如果 a 大于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回 canonical NaN。
- 如果两个入参都是 0.0，+0.0 > -0.0。

```
__device__ __half __hmin (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，返回 a 和 b 中，更小的值

**函数描述**

- 比较 a 和 b。如果 a 小于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回另一个不是 NaN 的参数。
- 如果两个参数都是 NaN，返回 canonical NaN。
- 如果两个入参都是 0.0，+0.0 > -0.0。

```
__device__ __half __hmin_nan (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half，返回 a 和 b 中，更小的值

**函数描述**

- 比较 a 和 b。如果 a 小于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回 canonical NaN。

- 如果两个入参都是 0.0，+0.0 > -0.0。

```
__device__ bool __hne (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，检测关系是否正确：a 不等于 b

**函数描述**

- 比较 a 和 b。如果 a 不等于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hneu (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 不等于 b

**函数描述**

- 无序比较 a 和 b。如果 a 和 b 不相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

### 1.8.4 half2 比较函数

```
__device__ bool __hbeq2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a 等于 b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a 和 b 相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbequ2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 等于 b

**函数描述**

- 无序比较 a 和 b。如果两个 half2 结构中对应的的 half 值都相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hbge2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a $\geqslant$ b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a $\geqslant$ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbgeu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a $\geqslant$ b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a $\geqslant$ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hbgt2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a 大于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbgtu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a 大于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hble2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a $\leqslant$ b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a $\leqslant$ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbleu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a $\leqslant$ b

**函数描述**

- 无序比较 a 和 b 中对应的的 half 值。如果 a ⩽ b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hblt2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a 小于 b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a 小于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbltu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 小于 b

**函数描述**

- 无序比较 a 和 b 中对应的的 half 值。如果 a 小于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ bool __hbne2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，检测关系是否正确：a 不等于 b

**函数描述**

- 比较 a 和 b 中对应的的 half 值。如果 a 不等于 b，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 false。

```
__device__ bool __hbneu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- bool，无序比较，检测关系是否正确：a 不等于 b

**函数描述**

- 无序比较 a 和 b 中对应的的 half 值。如果 a 和 b 不相等，返回 true；否则，返回 false。
- 如果输入值为 NaN，返回 true。

```
__device__ __half2 __heq2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测 a 和 b 是否相等

**函数描述**

- 比较 a 和 b。如果相等，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hequ2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较 a 和 b 是否相等

**函数描述**

- 无序比较 a 和 b。如果相等，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

```
__device__ __half2 __hge2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测关系是否正确：a ⩾ b

**函数描述**

- 比较 a 和 b。如果 a ⩾ b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hgeu2 (const __half2 a, const __half2)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较，检测关系是否正确：a ⩾ b

**函数描述**

- 比较 a 和 b，如果 a ⩾ b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

```
__device__ __half2 __hgt2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b。如果 a 大于 b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hgtu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较，检测关系是否正确：a 大于 b

**函数描述**

- 比较 a 和 b。如果 a 大于 b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

```
__device__ __half2 __hisnan2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，a 中的各个 half 值是否为 NaN

**函数描述**

- 返回 half2。如果入参中的某个 half 值为 NaN，将返回值中对应的位置设为 1.0；否则，设置为 0.0。

```
__device__ __half2 __hle2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测关系是否正确：a ⩽ b

**函数描述**

- 比较 a 和 b。如果 a ⩽ b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hleu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较，检测关系是否正确：a ⩽ b

**函数描述**

- 无序比较 a 和 b。如果 a ⩽ b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

```
__device__ __half2 __hlt2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测关系是否正确：a 小于 b

**函数描述**

- 比较 a 和 b。如果 a 小于 b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hltu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较，检测关系是否正确：a 小于 b

**函数描述**

- 无序比较 a 和 b。如果 a 小于 b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

```
__device__ __half2 __hmax2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 和 b 中，更大的值

**函数描述**

- 如果 a 大于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回另一个不是 NaN 的参数。
- 如果两个参数都是 NaN，返回 canonical NaN。
- 如果两个被比较的值都是 0.0，+0.0 > -0.0。

```
__device__ __half2 __hmax2_nan (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 和 b 中，更大的值

**函数描述**

- 如果 a 大于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回 canonical NaN。
- 如果两个被比较的值都是 0.0，+0.0 > -0.0。

```
__device__ __half2 __hmin2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 和 b 中，更小的值

**函数描述**

- 比较 a 和 b。如果 a 小于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回另一个不是 NaN 的参数。
- 如果两个参数都是 NaN，返回 canonical NaN。
- 如果两个被比较的值都是 0.0，+0.0 > -0.0。

```
__device__ __half2 __hmin2_nan (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，返回 a 和 b 中，更小的值

**函数描述**

- 比较 a 和 b。如果小于 b，返回 a；否则，返回 b。
- 如果入参中有 NaN，返回 canonical NaN。
- 如果两个被比较的值都是 0.0，+0.0 > -0.0。

```
__device__ __half2 __hne2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，检测关系是否正确：a 不等于 b

**函数描述**

- 比较 a 和 b。如果 a 不等于 b，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <0.0, 0.0>。

```
__device__ __half2 __hneu2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，无序比较，检测关系是否正确：a 不等于 b

**函数描述**

- 无序比较 a 和 b。如果 a 和 b 不相等，返回 <1.0, 1.0>；否则，返回 <0.0, 0.0>。
- 如果输入值为 NaN，返回 <1.0, 1.0>。

### 1.8.5 半精度转换与数据移动函数

```
__host__ __device__ __half __double2half (const double a)
```

**参数**

a

- double，只读入参

**返回值**

- half，返回半精度类型的 a

**函数描述**

- 以向最近偶数舍入模式将 double 类型的 a 转换成半精度类型输出。

```
__host__ __device__ __half2 __float22half2_rn (const float2 a)
```

**参数**

a

- float2，只读入参

**返回值**

- half2，转换为 float2 类型的 a

**函数描述**

- 以向最近偶数舍入模式将 float2 类型的参数 a，转换成 half2 类型输出。
- 返回值中的 low 16 bits 为 a.x，high 16 bits 为 a.y。

```
__host__ __device__ __half __float2half (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 以向最近偶数舍入模式将 float 类型的参数 a，转换成半精度类型输出。

```
__host__ __device__ __half2 __float2half2_rn (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half2，由转成 half 类型的 a，组成的 half2

**函数描述**

- 以向最近偶数舍入模式将 float 类型的参数 a，转换成半精度类型，组成 half2 类型并输出。

```
__host__ __device__ __half __float2half_rd (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 以向下舍入模式将 float 类型的 a，转换成半精度类型输出。

```
__host__ __device__ __half __float2half_rn (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 以向最近偶数舍入模式将 float 类型的参数 a，转换成半精度类型输出。

```
__host__ __device__ __half __float2half_ru (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 以向上舍入模式将 float 类型的参数 a，转换成半精度类型输出。

```
__host__ __device__ __half __float2half_rz (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 以向零舍入模式将 float 类型的参数 a，转换成半精度类型输出。

```
__host__ __device__ __half2 __floats2half2_rn (const float a, const float
↪b)
```

**参数**

a

- float，只读入参

b

- float，只读入参

**返回值**

- half2，将 a 和 b 转换成半精度类型后，组成 half2 类型并输出

**函数描述**

- 以向最近偶数舍入模式将 float 类型的参数 a 和 b，转换成半精度类型，组成 half2 类型并输出。

```
__host__ __device__ float2 __half22float2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- float2，转换为 float2 类型的 a

**函数描述**

- 将 half2 类型的参数 a，转换成 float2 类型后，输出。

```
__host__ __device__ float __half2float (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- float，转换为 float 类型的 a

**函数描述**

- 将 half 类型的参数 a，转换成 float 类型后，输出。

```
__device__ __half2 __half2half2 (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half2，用 half 类型的组成的 half2

**函数描述**

- 用 half 类型的 a，组成 half2 后，输出。

```
__device__ int __half2int_rd (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- int，转换为 int 类型的 h

**函数描述**

- 以向下舍入模式将 half 类型的参数 h，转换为 int 输出。

```
__device__ int __half2int_rn (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- int，转换成 int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 int 输出。

```
__device__ int __half2int_ru (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- int，转换成 int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 int 输出。

```
__host__ __device__ int __half2int_rz (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- int，转换成 int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 int 输出。

```
__device__ long long int __half2ll_rd (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- signed long long int，转换成 64 bit signed int 类型的 h

**函数描述**

- 以向下舍入模式将 half 类型的参数 h，转换为 64 bit signed int 输出。

```
__device__ long long int __half2ll_rn (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- signed long long int，转换成 64 bit signed int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 64 bit signed int 输出。

```
__device__ long long int __half2ll_ru (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- signed long long int，转换成 64 bit signed int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 64 bit signed int 输出。

```
__host__ __device__ long long int __half2ll_rz (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- signed long long int，转换成 64 bit signed int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 64 bit signed int 输出。

```
__device__ short int __half2short_rd (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- signed short int，转换成 signed short int 类型的 h

**函数描述**

- 以向下舍入模式将 half 类型的参数 a，转换为 signed short int 输出。

```
__device__ short int __half2short_rn (const __half h)
```

**参数**

h

- half 只读入参

**返回值**

- signed short int，转换成 signed short int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 signed short int 输出。

```
__device__ short int __half2short_ru (const __half h)
```

**参数**

h

- half 只读入参

**返回值**

- signed short int，转换成 signed short int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 signed short int 输出。

```
__host__ __device__ short int __half2short_rz (const __half h)
```

**参数**

h

- half 只读入参

**返回值**

- signed short int，转换成 signed short int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 signed short int 输出。

```
__device__ unsigned int __half2uint_rd (const __half h)
```

**参数**

h

- half 只读入参

**返回值**

- unsigned int，转换成 unsigned int 类型的 h

**函数描述**

- 在向下模式将 half 类型的参数 h，转换为 unsigned int 输出。

```
__device__ unsigned int __half2uint_rn (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned int，转换成 unsigned int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 unsigned int 输出。

```
__device__ unsigned int __half2uint_ru (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned int，转换成 unsigned int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 unsigned int 输出。

```
__host__ __device__ unsigned int __half2uint_rz (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned int，转换成 unsigned int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 unsigned int 输出。

```
__device__ unsigned long long int __half2ull_rd (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned long long int，转换成 unsigned 64 bit int 类型的 h

**函数描述**

- 以向下舍入模式将 half 类型的参数 h，转换为 unsigned 64 bit int 输出。

```
__device__ unsigned long long int __half2ull_rn (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned long long int，转换成 unsigned 64 bit int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 unsigned 64 bit int 输出。

```
__device__ unsigned long long int __half2ull_ru (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned long long int，转换成 unsigned 64 bit int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 unsigned 64 bit int 输出。

```
__host__ __device__ unsigned long long int __half2ull_rz (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned long long int，转换成 unsigned 64 bit int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 unsigned 64 bit int 输出。

```
__device__ unsigned short int __half2ushort_rd (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned short int，转换成 unsigned short int 类型的 h

**函数描述**

- 以向下舍入模式将 half 类型的参数 h，转换为 unsigned short int 输出。

```
__device__ unsigned short int __half2ushort_rn (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned short int，转换成 unsigned short int 类型的 h

**函数描述**

- 以向最近偶数舍入模式将 half 类型的参数 h，转换为 unsigned short int 输出。

```
__device__ unsigned short int __half2ushort_ru (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned short int，转换成 unsigned short int 类型的 h

**函数描述**

- 以向上舍入模式将 half 类型的参数 h，转换为 unsigned short int 输出。

```
__host__ __device__ unsigned short int __half2ushort_rz (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned short int，转换成 unsigned short int 类型的 h

**函数描述**

- 以向零舍入模式将 half 类型的参数 h，转换为 unsigned short int 输出。

```
__device__ short int __half_as_short (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- short int，转换成 signed short int 类型的 h

**函数描述**

- 将 half 类型的参数 h，转换为 signed short int 后，输出。

```
__device__ unsigned short int __half_as_ushort (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- unsigned short int，转换成 int 类型的 h

**函数描述**

- 将 half 类型的参数 h，转换为 unsigned short int 后输出。

```
__device__ __half2 __halves2half2 (const __half a, const __half b)
```

**参数**

a

- half，只读入参

b

- half，只读入参

**返回值**

- half2，a 和 b 组成的 half2

**函数描述**

- 将输入的 a 和 b 组合为 half2 后输出。
- 返回值中的 low 16 bits 为 a，high 16 bits 为 b。

```
__host__ __device__ float __high2float (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- float，转换为 float 类型的 a

**函数描述**

- 将类型为 16 bits half2 的 a，转换成 32 bits 的 float 类型后，输出。

```
__device__ __half __high2half (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half，转换为 half 类型的 a

**函数描述**

- 将类型为 half2 的 a，转成高 16bit 的 half 类型，输出。

```
__device__ __half2 __high2half2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，高 16bit 的 half2 类型

**函数描述**

- 提取 a 中的高 16bit 并返回一个新的 half2 值，其所含的 half 都等于输入的高 16bit。

```
__device__ __half2 __highs2half2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- half2，转换的浮点数中的高 16bit

**函数描述**

- 提取 a 和 b 中的高 16bit 的 half，组合并并返回一个新的 half2 值。
- 返回值中的 low 16 bits 为 a，high 16 bits 为 b。

```
__device__ __half __int2half_rd (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向下舍入模式将 int 类型的 i，转换成半精度的浮点类型输出。

```
__host__ __device__ __half __int2half_rn (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向最近偶数舍入模式将 int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __int2half_ru (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向上舍入模式将 int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ldca (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half2 __ldca (const __half2 *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ldcg (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half2 __ldcg (const __half2 *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ldcs (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half2 __ldcs (const __half2 *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ldcv (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half2 __ldcv (const __half2 *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ldg (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half2 __ldg (const __half2 *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ldlu (const __half *ptr)
```

**参数**

ptr

- memory location

**返回值**

- 入参指针指向的值

**函数描述**

- 输入指针，返回指针指向的值。

```
__device__ __half __ll2half_rd (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- half，将 i 转换成 half 结构

**函数描述**

- 以向下舍入模式将 signed 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__host__ __device__ __half __ll2half_rn (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- half，将 i 转换成 half 结构

**函数描述**

- 以向最近偶数舍入模式将 signed 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ll2half_ru (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- half，将 i 转换成 half 结构

**函数描述**

- 以向上舍入的模式将 signed 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ll2half_rz (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- half，将 i 转换成 half 结构

**函数描述**

- 以向零舍入模式将 signed 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__host__ __device__ float __low2float (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- float，a 中低 16bit 的部分，转换为 float

**函数描述**

- 将 half2 类型的参数 a，中低 16bit 的部分，转换为 32bit 的 float 类型，并输出。

```
__device__ __half __low2half (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half，输出 a 中低 16bit 的 half 值

**函数描述**

- 输出 half2 类型中，低 16bit 的 half 值。

```
__device__ __half2 __low2half2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，a 中低 16bit 的值，组合为 half2 类型后输出

**函数描述**

- 讲 half2 类型 a 中，低 16bit 的值，组合为 half2 类型后输出。

```
__device__ __half2 __lowhigh2highlow (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，交换 half 值的 a

**函数描述**

- 讲 half2 类型的 a 的 half 值交换，并输出。

```
__device__ __half2 __lows2half2 (const __half2 a, const __half2 b)
```

**参数**

a

- half2，只读入参

b

- half2，只读入参

**返回值**

- a 和 b 中低 16bit 的 half 的组合

**函数描述**

- 提取 half2 类型的 a 和 b 中，低 16bit 的 half 值，组合为 half2 类型后输出。

```
__device__ __half __shfl_down_sync (const unsigned mask, const __half var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half，只读入参

delta

- unsigned int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 2 字节的 word 作为一个 half。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过求 caller’s thread ID 和 delta 的和计算 source thread ID 的值。返回了 resulting thread ID 保存的 var 的值：这具有将 var 向下移动 delta threads 的效果。如果 width 小于 warpSize 则 warp 的每个子部分都表现为 starting logical thread ID 为 0 的单独实体。至于 __shfl_up_sync()，source thread 的 ID number 不会环绕 width 的值，因此 upper delta threads 将保持不变。

```
__device__ __half2 __shfl_down_sync (const unsigned mask, const __half2
↪var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half2，只读入参

delta

- unsigned int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 4 字节的 word 作为一个 half2。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过求 caller’s thread ID 和 delta 的和计算 source thread ID 的值。返回了 resulting thread ID 保存的 var 的值：这具有将 var 向下移动 delta threads 的效果。如果 width 小于 warpSize 则 warp 的每个子部分都表现为 starting logical thread ID 为 0 的单独实体。至于 __shfl_up_sync()，source thread 的 ID number 不会环绕 width 的值，因此 upper delta threads 将保持不变。

```
__device__ __half __shfl_sync (const unsigned mask, const __half var,
↪const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 2 字节的 word 作为一个 half。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 返回一个 ID 由 delta 给出的线程所持有的 var 的值。如果 width 小于 warpSize 则 warp 的每个子部分都表现为 starting logical thread ID 为 0 的单独实体。如果 delta 超过了 [0:width-1] 的范围，则返回的值应和 delta module width 所持有的值对应（即在同一子节内）。width 的值必须是一个 2 的幂。如果 width 不是 2 的幂或者一个大于 warpSize 的值，则 results 应为 undefined。

```
__device__ __half __shfl_up_sync (const unsigned mask, const __half var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half2，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 2 字节的 word 作为一个 half。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过将 caller’s lane ID 减去 delta 得到 source thread ID。返回 resulting lane ID 所持有的 var 的值：var 被 delta thread 向上移动。如果 width 小于 warpSize 则 warp 的每个子部分都表现为 starting logical thread ID 为 0 的单独实体。source thread ID 不会环绕 width 值，所以较低的 delta threads 不会被改变。width 的值必须是一个 2 的幂。如果 width 不是 2 的幂或者一个大于 warpSize 的值，则 results 应为 undefined。

```
__device__ __half2 __shfl_up_sync (const unsigned mask, const __half2 var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half2，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 4 字节的 word 作为一个 half2。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过将 caller’s lane ID 减去 delta 得到 source thread ID。返回 resulting lane ID 所持有的 var 的值：var 被 delta thread 向上移动。如果 width 小于 warpSize 则 warp 的每个子部分都表现为 starting logical thread ID 为 0 的单独实体。source thread ID 不会环绕 width 值，所以较低的 delta threads 不会被改变。width 的值必须是一个 2 的幂。如果 width 不是 2 的幂或者一个大于 warpSize 的值，则 results 应为 undefined。

```
__device__ __half __shfl_xor_sync (const unsigned mask, const __half var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 2 字节的 word 作为一个 half。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过对 caller’s thread ID 和 mask 进行按位异或计算来得到 source thread ID：返回由 resulting thread ID 所持有的 var 的值。如果 width 小于 warpSize 则每组 width consecutive thread 都能够访问 earlier groups 中 d 线程的元素，然而如果他们尝试访问 later groups 的线程，则会返回他们自己的 var。这种模式实现了一种蝶式寻址模式，例如用于 tree reduction 和 broadcast。

```
__device__ __half2 __shfl_xor_sync (const unsigned mask, const __half2 var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- half2，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- 从 source thread ID 返回一个引用了 var 的 4 字节的 word 作为一个 half2。如果 source thread ID 超过了 source 的范围或者 source thread 已经退出，则返 calling thread 自己的 var。

**函数描述**

- 通过对 caller’s thread ID 和 mask 进行按位异或计算来得到 source thread ID：返回由 resulting thread ID 所持有的 var 的值。如果 width 小于 warpSize 则每组 width consecutive thread 都能够访问 earlier groups 中 d 线程的元素，然而如果他们尝试访问 later groups 的线程，则会返回他们自己的 var。这种模式实现了一种蝶式寻址模式，例如用于 tree reduction 和 broadcast。

```
__device__ __half __short2half_rd (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- half，转换成半精度类型的 i

**函数描述**

- 以向下舍入模式将 int 类型的 i，转换成半精度浮点类型输出。

```
__host__ __device__ __half __short2half_rn (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- half，转换成半精度类型的 i

**函数描述**

- 以向最近偶数舍入模式将 int 类型的 i，转换成半精度浮点类型输出。

```
__device__ __half __short2half_ru (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- half，转换成半精度类型的 i

**函数描述**

- 以向上舍入模式将 int 类型的 i，转换成半精度浮点类型输出。

```
__device__ __half __short2half_rz (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- half，转换成半精度类型的 i

**函数描述**

- 以向零舍入模式将 int 类型的 i，转换成半精度浮点类型输出。

```
__device__ __half __short_as_half (const short int i)
```

**参数**

i

- shor int，只读入参

half

- reinterpreted 值

**函数描述**

- 将入参 i，reinterpret 成半精度浮点类型后，输出。

```
__device__ void __stcg (const __half *ptr, const __half value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stcg (const __half2 *ptr, const __half2 value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stcs (const __half *ptr, const __half value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stcs (const __half2 *ptr, const __half2 value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stwb (const __half *ptr, const __half value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stwb (const __half2 *ptr, const __half2 value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stwt (const __half *ptr, const __half value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ void __stwt (const __half2 *ptr, const __half2 value)
```

**参数**

ptr

- 内存位置

value

- 要被存储的值

```
__device__ __half __uint2half_rd (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 将 unsigned int 类型的 i，转换成半精度的浮点类型，round-down 后，输出。

```
__host__ __device__ __half __uint2half_rn (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向最近偶数舍入模式将 unsigned int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __uint2half_ru (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向上舍入模式将 unsigned int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __uint2half_rz (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向零舍入模式将 unsigned int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ull2half_rd (const unsigned long long int i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向下舍入模式将 unsigned 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__host__ __device__ __half __ull2half_rn (const unsigned long long int i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向最近偶数舍入模式将 unsigned 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ull2half_ru (const unsigned long long int i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向上舍入模式将 unsigned 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ull2half_rz (const unsigned long long int i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向零舍入模式将 unsigned 64bit int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ushort2half_rd (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向下舍入模式将 unsigned short int 类型的 i，转换成半精度的浮点类型输出。

```
__host__ __device__ __half __ushort2half_rn (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向最近偶数舍入模式将 unsigned short int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ushort2half_ru (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向上舍入模式将 unsigned short int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ushort2half_rz (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- half，转换为 half 类型的 i

**函数描述**

- 以向零舍入模式将 unsigned short int 类型的 i，转换成半精度的浮点类型输出。

```
__device__ __half __ushort_as_half (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- half，reinterpreted 值

**函数描述**

- 将入参 i，reinterpret 成半精度浮点类型后，输出。

### 1.8.6 half 数学函数

```
__device__ __half hceil (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- half，返回不小于 h 的最小的 integer 值

**函数描述**

- 计算不小于 h 的最小 interger 值。

```
__device__ __half hcos (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的 cosine 值

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half cosine 值。

```
__device__ __half hexp (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 自然指数函数。

```
__device__ __half hexp10 (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的十进制自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 十进制自然指数函数。

```
__device__ __half hexp2 (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的二进制自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 二进制自然指数函数。

```
__device__ __half hfloor (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- half，返回小于或等于 h 的最大 integer 值。

**函数描述**

- 计算小于或等于 h 的最大 integer 值。

```
__device__ __half hlog (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的自然对数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 自然对数。

```
__device__ __half hlog10 (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的十进制对数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 十进制对数。

```
__device__ __half hlog2 (const __half a)
```

**参数**

h

- half，只读入参

**返回值**

- half，返回最近 h 的 integer

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 的二进制对数。

```
__device__ __half hrcp (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的倒数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 的倒数。

```
__device__ __half hrint (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- half，返回最接近 h 的整数值。

**函数描述**

- 用半精度的格式把 h 舍入到最接近的整数值，对于正好一半的情况下舍入到最接近的偶数值。

```
__device__ __half hrsqrt (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，返回 a 的平方根倒数

**函数描述**

- 以向最近偶数舍入模式计算输入 a 的 half 的平方根倒数。

```
__device__ __half hsin (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，a 的正弦值

**函数描述**

- 以向最近偶数舍入模式计算输入的 a 的正弦值。

```
__device__ __half hsqrt (const __half a)
```

**参数**

a

- half，只读入参

**返回值**

- half，a 的平方根。

**函数描述**

- 以向最近偶数舍入模式计算输入的 a 的平方根。

```
__device__ __half htrunc (const __half h)
```

**参数**

h

- half，只读入参

**返回值**

- half，返回截断 integer

**函数描述**

- 将 h 舍入为最接近的 integer，其幅度不超过 h。

### 1.8.7 half2 数学函数

```
__device__ __half2 h2ceil (const __half2 h)
```

**参数**

h

- half2，只读入参

**返回值**

- half2，返回不小于 h 的最小 integer 向量

**函数描述**

- 对于向量 h 的每个分量，计算不小于 h 的最小 interger 值。

```
__device__ __half2 h2cos (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素 cosine 值

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的 cosine 值。

```
__device__ __half2 h2exp (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的自然指数函数。

```
__device__ __half2 h2exp10 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素十进制自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的十进制自然指数函数。

```
__device__ __half2 h2exp2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素二进制自然指数函数

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的二进制自然指数函数。

```
__device__ __half2 h2floor (const __half2 h)
```

**参数**

h

- half2，只读入参

**返回值**

- half2，返回小于或等于 h 的最大 integer 向量。

**函数描述**

- 对于向量 h 的每个分量，计算小于或等于 h 的最大 integer 值。

```
__device__ __half2 h2log (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素自然对数

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 自然对数。

```
__device__ __half2 h2log10 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素十进制自然对数

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 十进制自然对数。

```
__device__ __half2 h2log2 (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回 a 的二进制对数。

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 二进制对数。

```
__device__ __half2 h2rcp (const __half2 a)
```

**参数**

h

- half2，只读入参

**返回值**

- half2，返回四舍五入 integer 的向量

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 倒数。

```
__device__ __half2 h2rint (const __half2 h)
```

**参数**

h

- half2，只读入参

**返回值**

- half2，返回四舍五入 integer 的向量

**函数描述**

- 以向最近偶数舍入模式将向量 a 中每个 half 值转换为半精度浮点型后，输出转换后的 half2。

```
__device__ __half2 h2rsqrt (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素倒数平方根

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的倒数平方根。

```
__device__ __half2 h2sin (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素 sine 值。

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的 sine 值。

```
__device__ __half2 h2sqrt (const __half2 a)
```

**参数**

a

- half2，只读入参

**返回值**

- half2，返回向量 a 的元素平方根

**函数描述**

- 以向最近偶数舍入模式计算输入向量 a 的 half2 的平方根。

```
__device__ __half2 h2trunc (const __half2 h)
```

**参数**

h

- half2，只读入参

**返回值**

- half2，返回截断

**函数描述**

- 将向量 h 的每个分量四舍五入为幅度不超过 h 的最接近的整数。

bfloat16 类型介绍，参见maca_bfloat16。

### 1.8.8 Bfloat16 算术函数

```
__device__ maca_bfloat16 __habs (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 的绝对值

**函数描述**

- 计算 maca_bfloat16 的绝对值。

```
__device__ maca_bfloat16 __hadd (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 与 b 的和

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的加法。

```
__device__ maca_bfloat16 __hadd_sat (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 与 b 的和

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的加法，并且结果被限制在 [0.0, 1.0] 之间。
- 若结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat16 __hdiv (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 除以 b 的商

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的除法。

```
__device__ maca_bfloat16 __hfma (const maca_bfloat16 a, const maca_
↪bfloat16 b, const maca_bfloat16 c)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

C

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的融合乘加运算。

```
__device__ maca_bfloat16 __hfma_relu (const maca_bfloat16 a, const maca_
↪bfloat16 b,
const maca_bfloat16 c)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

c

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的融合乘加运算，并将运算结果进行 relu 变换。
- 若结果小于 0.0 则视为 0.0。NaN 被视为标准 NaN。

```
__device__ maca_bfloat16 __hfma_sat (const maca_bfloat16 a, const maca_
↪bfloat16 b,
const maca_bfloat16 c)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

c

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的融合乘加运算，并且结果被限制在 [0.0, 1.0] 之间。
- 若结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat16 __hmul (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的乘法。

```
__device__ maca_bfloat16 __hmul_sat (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的乘法，并且结果被限制在 [0.0, 1.0] 之间。
- 若结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat16 __hneg (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 的相反数

**函数描述**

- 计算 maca_bfloat16 的相反数。

```
__device__ maca_bfloat16 __hsub (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的减法。

```
__device__ maca_bfloat16 __hsub_sat (const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat16 的减法，并且结果被限制在 [0.0, 1.0] 之间。
- 若结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

### 1.8.9 Bfloat162 算术函数

```
__device__ maca_bfloat162 __habs2 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，对向量 a 的两个分量取绝对值后的向量

**函数描述**

- 计算 maca_bfloat162 中的两个 maca_bfloat16 分量的绝对值。

```
__device__ maca_bfloat162 __hadd2 (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 与 b 的和

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 向量加法。

```
__device__ maca_bfloat162 __hadd2_sat (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 与 b 的和

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量加法，并且每个分量的结果被限制在 [0.0, 1.0] 之间。
- 若分量的结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat162 __h2div (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 除以 b 的商

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 向量除法。

```
__device__ maca_bfloat162 __hcmadd (const maca_bfloat162 a, const maca_
↪bfloat162 b,
const maca_bfloat162 c)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

c

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，复数 a 乘 b 加 c 的结果

**函数描述**

- 将输入向量视为复数，并执行融合乘加运算。

```
__device__ maca_bfloat162 __hfma2 (const maca_bfloat162 a, const maca_
↪bfloat162 b,
const maca_bfloat162 c)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

C

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量融合乘加运算。

```
__device__ maca_bfloat162 __hfma2_relu (const maca_bfloat162 a,
const maca_bfloat162 b, const maca_bfloat162 c)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

c

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量融合乘加运算，并将运算结果进行 relu 变换。
- 若分量的结果小于 0.0 则视为 0.0。NaN 被视为标准 NaN。

```
__device__ maca_bfloat162 __hfma2_sat (const maca_bfloat162 a, const maca_
↪bfloat162 b,
const maca_bfloat162 c)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

c

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 乘 b 加 c 的结果

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量融合乘加运算，并且结果被限制在 [0.0, 1.0] 之间。
- 若分量的结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat162 __hmul2 (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量乘法。

```
__device__ maca_bfloat162 __hmul2_sat (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 乘 b 的积

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量乘法，并且结果被限制在 [0.0, 1.0] 之间。
- 若分量的结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

```
__device__ maca_bfloat162 __hneg2 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，对向量 a 的两个分量取相反数后的向量

**函数描述**

- 计算 maca_bfloat162 中的两个 maca_bfloat16 分量的相反数。

```
__device__ maca_bfloat162 __hsub2 (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量减法。

```
__device__ maca_bfloat162 __hsub2_sat (const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，向量 a 减 b 的差

**函数描述**

- 以向最近偶数舍入模式执行 maca_bfloat162 的向量减法，并且结果被限制在 [0.0, 1.0] 之间。
- 若分量的结果大于 1.0 则视为 1.0，小于 0.0 则视为 0.0。NaN 被视为 +0.0。

### 1.8.10 Bfloat16 比较函数

```
__device__ bool __heq(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 等于 b”的判定结果

**函数描述**

- 判定两个 maca_bfloat16 是否相等。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hequ(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 等于 b”的判定结果

**函数描述**

- 判定两个 maca_bfloat16 是否相等。
- 如果任一输入为 NaN，则判定结果为 true。

```
__device__ bool __hge(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 大于或等于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否大于或等于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hgeu(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 大于或等于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否大于或等于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 true。

```
__device__ bool __hgt(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 大于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否大于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hgtu(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 大于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否大于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 true。

```
__device__ int __hisinf(const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- int，“a 为无穷大”的判定结果

**函数描述**

- 判定 maca_bfloat16 的值是否为无穷大。
- 正无穷的判定结果为 1，负无穷为-1，不是无穷大则为 0。

```
__device__ bool __hisnan(const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- bool，“a 为 NaN”的判定结果

**函数描述**

- 判定 maca_bfloat16 的值是否为 NaN。

```
__device__ bool __hle(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 小于或等于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否小于或等于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hleu(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 小于或等于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否小于或等于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 true。

```
__device__ bool __hlt(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 小于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否小于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hltu(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 小于 b”的判定结果

**函数描述**

- 判定第一个参数 maca_bfloat16 是否小于第二个参数 maca_bfloat16。
- 如果任一输入为 NaN，则判定结果为 true。

```
__device__ maca_bfloat16 __hmax(const maca_bfloat16 a, const maca_bfloat16
↪b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，值等于 a 和 b 中更大的那一个

**函数描述**

- 获取两个 maca_bfloat16 中值更大的那一个。
- NaN 视为小于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat16 __hmax_nan(const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16，值等于 a 和 b 中更大的那一个

**函数描述**

- 获取两个 maca_bfloat16 中值更大的那一个。
- NaN 视为大于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat16 __hmin(const maca_bfloat16 a, const maca_bfloat16
↪b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，值等于 a 和 b 中更小的那一个

**函数描述**

- 获取两个 maca_bfloat16 中值更小的那一个。
- NaN 视为大于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat16 __hmin_nan(const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，值等于 a 和 b 中更小的那一个

**函数描述**

- 获取两个 maca_bfloat16 中值更小的那一个。
- NaN 视为小于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ bool __hne(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 不等于 b”的判定结果

**函数描述**

- 判定两个 maca_bfloat16 是否不相等。
- 如果任一输入为 NaN，则判定结果为 false。

```
__device__ bool __hneu(const maca_bfloat16 a, const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- bool，“a 不等于 b”的判定结果

**函数描述**

- 判定两个 maca_bfloat16 是否不相等。
- 如果任一输入为 NaN，则判定结果为 true。

### 1.8.11 Bfloat162 比较函数

```
__device__ bool __hbeq2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbequ2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ bool __hbge2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都大于或等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都大于或等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbgeu2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都大于或等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都大于或等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ bool __hbgt2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都大于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都大于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbgtu2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都大于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都大于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ bool __hble2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都小于或等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都小于或等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbleu2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都小于或等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都小于或等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ bool __hblt2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都小于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都小于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbltu2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都小于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都小于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ bool __hbne2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都不等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都不等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 false。

```
__device__ bool __hbneu2(const maca_bfloat162 a, const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- bool，“a 的分量都不等于 b 的分量”的判定结果

**函数描述**

- 判定第一个 maca_bfloat162 向量的两个分量是否都不等于第二个 maca_bfloat162 的分量。
- 如果参数中存在 NaN，则判定结果为 true。

```
__device__ maca_bfloat162 __heq2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对两个向量的分量进行相等判定。
- 对应分量的值相等时判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hequ2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行相等判定。
- 相等时判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

```
__device__ maca_bfloat162 __hge2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行大于或等于判定。
- 第一个 maca_bfloat162 向量的分量大于或等于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hgeu2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行大于或等于判定。
- 第一个 maca_bfloat162 向量的分量大于或等于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

```
__device__ maca_bfloat162 __hgt2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行大于判定。
- 第一个 maca_bfloat162 向量的分量大于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hgtu2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行大于判定。
- 第一个 maca_bfloat162 向量的分量大于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

```
__device__ maca_bfloat162 __hisnan2(const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量是否为 NaN 的结果

**函数描述**

- 分别判定输入向量的两个分量的值是否为 NaN，如果是 NaN，结果中相应的分量会被赋予 1.0 的值；否则，赋予 0.0 的值。

```
__device__ maca_bfloat162 __hle2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行小于或等于判定。
- 第一个 maca_bfloat162 向量的分量小于或等于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hleu2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行小于或等于判定。
- 第一个 maca_bfloat162 向量的分量小于或等于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

```
__device__ maca_bfloat162 __hlt2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行小于判定。
- 第一个 maca_bfloat162 向量的分量小于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hltu2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对向量的两个分量进行小于判定。
- 第一个 maca_bfloat162 向量的分量小于第二个的向量的对应分量时，判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

```
__device__ maca_bfloat162 __hmax2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，分别对两个向量的分量提取更大的值的结果

**函数描述**

- 分别对两个输入向量的对应的分量进行大小比较，将值更大的分量作为结果以构建新的向量。
- NaN 视为小于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat16 __hmax2_nan(const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，分别对两个向量的分量提取更大的值的结果

**函数描述**

- 分别对两个输入向量的对应的分量进行大小比较，将值更大的分量作为结果以构建新的向量。
- NaN 视为大于任何数。

- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat162 __hmin2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，分别对两个向量的分量提取更小的值的结果

**函数描述**

- 分别对两个输入向量的对应的分量进行大小比较，将值更小的分量作为结果以构建新的向量。
- NaN 视为大于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat16 __hmin2_nan(const maca_bfloat16 a, const maca_
↪bfloat16 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，分别对两个向量的分量提取更小的值的结果

**函数描述**

- 分别对两个输入向量的对应的分量进行大小比较，将值更小的分量作为结果以构建新的向量。
- NaN 视为小于任何数。
- 对于 0 的判定，+0.0 > -0.0。

```
__device__ maca_bfloat162 __hne2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对两个向量的分量进行不相等判定。

- 对应分量的值不相等时判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 0.0。

```
__device__ maca_bfloat162 __hneu2(const maca_bfloat162 a, const maca_
↪bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用于指示两个分量的比较结果

**函数描述**

- 分别对两个向量的分量进行不相等判定。
- 对应分量的值不相等时判定结果的对应分量会被赋予 1.0 的值；否则，赋予 0.0。
- 若分量的值为 NaN 会使得判定结果被赋予 1.0。

### 1.8.12 Bfloat16 精度转换与数据移动函数

```
__host__ __device__ float2 __bfloat1622float2 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- float2，由 a 转换的向量

**函数描述**

- 将 maca_bfloat162 向量转换为 float2 向量。

```
__device__ maca_bfloat162 __bfloat162bfloat162 (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat162，用 a 同时作为两个分量而构建的向量

**函数描述**

- 使用一个 maca_bfloat16 参数来构建 maca_bfloat162 向量，向量中的两个分量的值均与参数相等。

```
__host__ __device__ float  __bfloat162float (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- float，值由 a 转换得到

**函数描述**

- 将 maca_bfloat16 转换 32 位浮点数。

```
__device__ int  __bfloat162int_rd (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为带符号的 32 位整数。

```
__device__ int  __bfloat162int_rn (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为带符号的 32 位整数。

```
__device__ int  __bfloat162int_ru (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- int，值由 h 转换得到

**函数描述**

- 用向上舍入的模式，将 maca_bfloat16 类型的浮点数 h 转换成 signed integer 类型。

```
__host__ __device__ int  __bfloat162int_rz (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为带符号的 32 位整数。

```
__device__ long long int  __bfloat1621l_rd (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- long long int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为带符号的 64 位整数。

```
__device__ long long int  __bfloat1621l_rn (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- long long int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为带符号的 64 位整数。

```
__device__ long long int  __bfloat1621l_ru (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- long long int，值由 h 转换得到

**函数描述**

- 用向上舍入的模式，将 maca_bfloat16 类型的浮点数 h 转换成 64 位 signed integer 类型。

```
__host__ __device__ long long int  __bfloat1621l_rz (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- long long int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为带符号的 64 位整数。

```
__device__ short int  __bfloat162short_rd (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- short int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为带符号的 16 位整数。

```
__device__ short int  __bfloat162short_rn (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- short int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为带符号的 16 位整数。

```
__device__ short int  __bfloat162short_ru (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- short int，值由 h 转换得到

**函数描述**

- 用向上舍入的模式，将 maca_bfloat16 类型的浮点数 h 转换成 signed short integer 类型。

```
__host__ __device__ short int  __bfloat162short_rz (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- short int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为带符号的 16 位整数。

```
__device__ unsigned int  __bfloat162uint_rd (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为无符号的 32 位整数。

```
__device__ unsigned int  __bfloat162uint _rn (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为无符号的 32 位整数。

```
__device__ unsigned int  __bfloat162uint_ru (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned int，值由 h 转换得到

**函数描述**

- 用向上舍入的模式，将 maca_bfloat16 类型的浮点数 h 转换成 unsigned ineger 类型。

```
__host__ __device__ unsigned int  __bfloat162uint_rz (const maca_bfloat16
↪h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为无符号的 32 位整数。

```
__device__ unsigned long long int  __bfloat162ull_rd (const maca_bfloat16
↪h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned long long int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为无符号的 64 位整数。

```
__device__ unsigned long long int  __bfloat162ull _rn (const maca_bfloat16
↪h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned long long int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为无符号的 64 位整数。

```
__device__ unsigned long long int  __bfloat162ull_ru (const maca_bfloat16
↪h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned long long int，值由 h 转换得到

**函数描述**

- 以向上舍入模式将 maca_bfloat16 类型转换为无符号的 64 位整数。

```
__host__ __device__ unsigned long long int  __bfloat162ull_rz (const maca_
↪bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned long long int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为无符号的 64 位整数。

```
__device__ unsigned short int  __bfloat162ushort_rd (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned short int，值由 h 转换得到

**函数描述**

- 以向下舍入模式将 maca_bfloat16 类型转换为无符号的 16 位整数。

```
__device__ unsigned short int  __bfloat162ushort _rn (const maca_bfloat16
↪h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned short int，值由 h 转换得到

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 类型转换为无符号的 16 位整数。

```
__device__ unsigned short int  __bfloat162ushort_ru (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned short int，值由 h 转换得到

**函数描述**

- 以向上舍入模式将 maca_bfloat16 类型转换为无符号的 16 位整数。

```
__host__ __device__ unsigned short int __bfloat162ushort_rz (const maca_
↪bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- unsigned short int，值由 h 转换得到

**函数描述**

- 以向零舍入模式将 maca_bfloat16 类型转换为无符号的 16 位整数。

```
__device__ short int __bfloat16_as_short (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- short int，值由 h 重新解释得到

**函数描述**

- 按位将 maca_bfloat16 重新解释为带符号的 16 位整数。

```
__device__ unsigned short int __bfloat16_as_ushort (const maca_bfloat16 h)
```

**参数**

h * maca_bfloat16，只读入参

**返回值**

- unsigned short int，值由 h 重新解释得到

**函数描述**

- 按位将 maca_bfloat16 重新解释为无符号的 16 位整数。

```
__host__ __device__ maca_bfloat16  __double2bfloat16 (const double a)
```

**参数**

a

- double，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向最近偶数舍入模式将 64 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat162  __float22bfloat162_rn (const float2 a)
```

**参数**

a

- float2，只读入参

**返回值**

- maca_bfloat162，由 a 转换的向量

**函数描述**

- 以向最近偶数舍入模式将 32 位浮点数向量转换为 maca_bfloat162 向量。

```
__host__ __device__ maca_bfloat16  __float2bfloat16 (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向最近偶数舍入模式将 32 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat162  __float2bfloat162_rn (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat162，由 a 转换的向量

**函数描述**

- 以向最近偶数舍入模式将 32 位浮点数类型转换为 maca_bfloat16，并用这个 maca_bfloat16 构建一个 maca_bfloat162 向量。

```
__host__ __device__ maca_bfloat16  __float2bfloat16_rd (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向下舍入模式将 32 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __float2bfloat16_rn (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向最近偶数舍入模式将 32 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __float2bfloat16_ru (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向上舍入模式将 32 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __float2bfloat16_rz (const float a)
```

**参数**

a

- float，只读入参

**返回值**

- maca_bfloat16，值由 a 转换得到

**函数描述**

- 以向零舍入模式将 32 位浮点数类型转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat162  __floats2bfloat162_rn (const float a,
↪const float b)
```

**参数**

a

- float，只读入参

b

- float，只读入参

**返回值**

- maca_bfloat162，由 a 和 b 构建得到的向量

**函数描述**

- 以向最近偶数舍入模式将两个 32 位浮点数类型转换为两个 maca_bfloat16，并用这两个 maca_bfloat16 构建一个 maca_bfloat162 向量。

```
__device__ maca_bfloat162  __halves2bfloat162 (const maca_bfloat16 a,
↪const maca_bfloat16 b)
```

**参数**

a

- maca_bfloat16，只读入参

b

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat162，由 a 和 b 构建得到的向量

**函数描述**

- 用两个 maca_bfloat16 参数构建一个 maca_bfloat162 向量，第一个参数被赋予到向量的低 16 位，第二个参数被赋予到向量的高 16 位。

```
__device__ maca_bfloat16  __high2bfloat16 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat16，a 向量的高 16 位分量

**函数描述**

- 提取 maca_bfloat162 向量的高 16 位分量。

```
__device__ maca_bfloat162  __high2bfloat162 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用 a 向量的高 16 位分量构建出的向量

**函数描述**

- 提取 maca_bfloat162 向量的高 16 位分量，以此构建出两个分量的值都等于提取值的 maca_bfloat162 向量。

```
__host__ __device__ float  __high2float (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- float，值由 a 向量的高 16 位分量转换得到

**函数描述**

- 提取 maca_bfloat162 向量的高 16 位分量，并将该分量转换为 32 为浮点数。

```
__device__ maca_bfloat162  __highs2bfloat162 (const maca_bfloat162 a,
↪const maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用 a 和 b 向量的高 16 位分量构建出的向量

**函数描述**

- 分别提取两个 maca_bfloat162 向量的高 16 位分量，构建出一个新的 maca_bfloat162 向量。

```
__device__ maca_bfloat16 __int2bfloat16_rd (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将带符号的 32 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __int2bfloat16_rn (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将带符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16 __int2bfloat16_ru (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将带符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __int2bfloat16_rz (const int i)
```

**参数**

i

- int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将带符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ldca (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldca (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat162*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ldcg (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldcg (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat162*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ldcs (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldcs (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat162*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ldcv (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldcv (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat126*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ldg (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldg (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat162*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ldlu (const maca_bfloat16 *ptr)
```

**参数**

ptr

- maca_bfloat16*，指针

**返回值**

- maca_bfloat16，ptr 指针指向的值

```
__device__ maca_bfloat162  __ldlu (const maca_bfloat162 *ptr)
```

**参数**

ptr

- maca_bfloat162*，指针

**返回值**

- maca_bfloat162，ptr 指针指向的值

```
__device__ maca_bfloat16  __ll2bfloat16_rd (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将带符号的 64 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __ll2bfloat16_rn (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将带符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ll2bfloat16_ru (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将带符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ll2bfloat16_rz (const long long int i)
```

**参数**

i

- long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将带符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat162  __low2bfloat162 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat16，用 a 向量的低 16 位分量构建出的向量

**函数描述**

- 提取 maca_bfloat162 向量的低 16 位分量，以此构建出两个分量的值都等于提取值的 maca_bfloat162 向量。

```
__host__ __device__ float  __low2float (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- float，值由 a 向量的高 16 位分量转换得到

**函数描述**

- 提取 maca_bfloat162 向量的低 16 位分量，并将该分量转换为 32 为浮点数。

```
__device__ maca_bfloat162  __lowhigh2highlow (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，将 a 向量的两个分量的值互换后得到

**函数描述**

- 将 maca_bfloat162 向量的高低分量的值互换。

```
__device__ maca_bfloat162  __lows2bfloat162 (const maca_bfloat162 a, const
↪maca_bfloat162 b)
```

**参数**

a

- maca_bfloat162，只读入参

b

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162，用 a 和 b 向量的低 16 位分量构建出的向量

**函数描述**

- 分别提取两个 maca_bfloat162 向量的低 16 位分量，构建出一个新的 maca_bfloat162 向量。

```
__device__ maca_bfloat16 __shfl_down_sync (const unsigned mask, const maca_
↪bfloat16 var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat16，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat16，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回向后序号偏移量等于 delta 的线程中的变量 var 的值。
- 若偏移后的线程超出了 width 指定的组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat162 __shfl_down_sync (const unsigned mask, const
↪maca_bfloat162 var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat162，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat162，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回向后序号偏移量等于 delta 的线程中的变量 var 的值。

- 若偏移后的线程超出了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat16 __shfl_sync (const unsigned mask, const maca_
↪bfloat16 var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat16，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat16，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回序号等于 delta 的线程中的变量 var 的值。
- 若序号超过了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat162 __shfl_sync (const unsigned mask, const maca_
↪bfloat162 var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat162，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat162，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回序号等于 delta 的线程中的变量 var 的值。

- 若序号超过了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat16 __shfl_up_sync (const unsigned mask, const maca_
↪bfloat16 var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat16，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat16，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回向前序号偏移量等于 delta 的线程中的变量 var 的值。
- 若偏移后的线程超出了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat162 __shfl_up_sync (const unsigned mask, const maca_
↪bfloat162 var,
const unsigned int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat162，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat162，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会返回向前序号偏移量等于 delta 的线程中的变量 var 的值。

- 若偏移后的线程超出了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat16 __shfl_xor_sync (const unsigned mask, const maca_
↪bfloat16 var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat16，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat16，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。
- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会将其自身的序号与 delta 进行一次异或操作，将操作得到的结果作为线程序号，从序号代表的线程中取得 var 值。
- 若序号超出了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat162 __shfl_xor_sync (const unsigned mask, const maca_
↪bfloat162 var,
const int delta, const int width)
```

**参数**

mask

- unsigned int，只读入参

var

- maca_bfloat16，只读入参

delta

- int，只读入参

width

- int，只读入参

**返回值**

- maca_bfloat162，线程间交换数据得到的值

**函数描述**

- 指定一组线程互相交换数据。

- 以 width 将同一个 wave 中的线程分组，同一组的线程从零开始计算序号，被 mask 指定的线程会将其自身的序号与 delta 进行一次异或操作，将操作得到的结果作为线程序号，从序号代表的线程中取得 var 值。
- 若序号超出了组大小，或该线程没有被 mask 指定，则直接返回入参 var。
- mask 中的每一个位都对应着一个线程，将位的值设置为 1 即代表指定了该线程进行数据交换。

```
__device__ maca_bfloat16 __short2bfloat16_rd (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将带符号的 16 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16 __short2bfloat16_rn (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将带符号的 16 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16 __short2bfloat16_ru (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将带符号的 16 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16 __short2bfloat16_rz (const short int i)
```

**参数**

i

- short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将带符号的 16 位整数转换为 maca_bfloat16。

```
__device__ void __stcg (const maca_bfloat16 *ptr, const maca_bfloat16
↪value)
```

**参数**

ptr

- maca_bfloat16*，指针

value

- maca_bfloat16，要存储到 ptr 指向的内存中的值

```
__device__ void __stcg (const maca_bfloat162 *ptr, const maca_bfloat162
↪value)
```

**参数**

ptr

- maca_bfloat162*，指针

value

- maca_bfloat162，要存储到 ptr 指向的内存中的值

```
__device__ void __stcs (const maca_bfloat16 *ptr, const maca_bfloat16
↪value)
```

**参数**

ptr

- maca_bfloat16*，指针

value

- maca_bfloat16，要存储到 ptr 指向的内存中的值

```
__device__ void __stcs (const maca_bfloat162 *ptr, const maca_bfloat162
↪value)
```

**参数**

ptr

- maca_bfloat162*，指针

value

- maca_bfloat162，要存储到 ptr 指向的内存中的值

```
__device__ void __stwb (const maca_bfloat16 *ptr, const maca_bfloat16
↪value)
```

**参数**

ptr

- maca_bfloat16*，指针

value

- maca_bfloat16，要存储到 ptr 指向的内存中的值

```
__device__ void __stwb (const maca_bfloat162 *ptr, const maca_bfloat162
↪value)
```

**参数**

ptr

- maca_bfloat162*，指针

value

- maca_bfloat162，要存储到 ptr 指向的内存中的值

```
__device__ void __stwt (const maca_bfloat16 *ptr, const maca_bfloat16
↪value)
```

**参数**

ptr

- maca_bfloat16*，指针

value

- maca_bfloat16，要存储到 ptr 指向的内存中的值

```
__device__ void __stwt (const maca_bfloat162 *ptr, const maca_bfloat162
↪value)
```

**参数**

ptr

- maca_bfloat162*，指针

value

- maca_bfloat162，要存储到 ptr 指向的内存中的值

```
__device__ maca_bfloat16  __uint2bfloat16_rd (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将无符号的 32 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __uint2bfloat16_rn (const unsigned int
↪i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将无符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __uint2bfloat16_ru (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将无符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __uint2bfloat16_rz (const unsigned int i)
```

**参数**

i

- unsigned int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将无符号的 32 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ull2bfloat16_rd (const unsigned long long int
↪i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将无符号的 64 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __ull2bfloat16_rn (const unsigned long
↪long int i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将无符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ull2bfloat16_ru (const unsigned long long int
↪i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将无符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ull2bfloat16_rz (const unsigned long long int
↪i)
```

**参数**

i

- unsigned long long int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将无符号的 64 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ushort2bfloat16_rd (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向下舍入模式将无符号的 16 位整数转换为 maca_bfloat16。

```
__host__ __device__ maca_bfloat16  __ushort2bfloat16_rn (const unsigned
↪short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向最近偶数舍入模式将无符号的 16 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ushort2bfloat16_ru (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向上舍入模式将无符号的 16 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ushort2bfloat16_rz (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- maca_bfloat16，值由 i 转换得到

**函数描述**

- 以向零舍入模式将无符号的 16 位整数转换为 maca_bfloat16。

```
__device__ maca_bfloat16  __ushort_as_bfloat16 (const unsigned short int i)
```

**参数**

i

- unsigned short int，只读入参

**返回值**

- maca_bfloat16，值由 i 重新解释得到

**函数描述**

- 按位将无符号的 16 位整数重新解释为 maca_bfloat16。

### 1.8.13 Bfloat16 数学函数

```
__device__ maca_bfloat16  hceil (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 计算大于输入的最小整数。

```
__device__ maca_bfloat16  hcos (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的余弦值。

```
__device__ maca_bfloat16  hexp (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的自然指数函数。

```
__device__ maca_bfloat16  hexp10 (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的 10 进制指数函数。

```
__device__ maca_bfloat16  hexp2 (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的 2 进制指数函数。

```
__device__ maca_bfloat16  hfloor (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16,

**函数描述**

- 计算小于输入的最大整数。

```
__device__ maca_bfloat16 hlog (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的自然对数

```
__device__ maca_bfloat16 hlog10 (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的 10 进制对数

```
__device__ maca_bfloat16 hlog2 (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的 2 进制对数

```
__device__ maca_bfloat16 hrcp (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的倒数。

```
__device__ maca_bfloat16 hrint (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式将 maca_bfloat16 转换为最接近的整数。

```
__device__ maca_bfloat16 hrsqrt (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的倒数平方根。

```
__device__ maca_bfloat16 hsin (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的正弦值。

```
__device__ maca_bfloat16 hsqrt (const maca_bfloat16 a)
```

**参数**

a

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 以向最近偶数舍入模式计算 maca_bfloat16 的平方根。

```
__device__ maca_bfloat16 htrunc (const maca_bfloat16 h)
```

**参数**

h

- maca_bfloat16，只读入参

**返回值**

- maca_bfloat16

**函数描述**

- 将入参截断，只保留整数部分。

### 1.8.14 Bfloat162 数学函数

```
__device__ maca_bfloat162  h2ceil (const maca_bfloat162 h)
```

**参数**

h

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 对于输入向量的两个分量，分别计算大于分量的最小整数。

```
__device__ maca_bfloat162  h2cos (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的 maca_bfloat16 的余弦值。

```
__device__ maca_bfloat162  h2exp (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的自然指数函数。

```
__device__ maca_bfloat162  h2exp10 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的 10 进制自然指数函数。

```
__device__ maca_bfloat162  h2exp2 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的 2 进制自然指数函数。

```
__device__ maca_bfloat162  hfloor (const maca_bfloat162 h)
```

**参数**

h

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 对于输入向量的两个分量，分别计算小于分量的最大整数。

```
__device__ maca_bfloat16 h2log (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的自然对数。

```
__device__ maca_bfloat16 h2log10 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的 10 进制自然对数。

```
__device__ maca_bfloat16 h2log2 (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的 2 进制自然对数。

```
__device__ maca_bfloat162 hrcp (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的倒数。

```
__device__ maca_bfloat162 hrint (const maca_bfloat162 h)
```

**参数**

h

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别将输入向量的两个分量转换为最接近的整数。

```
__device__ maca_bfloat16 h2rsqrt (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的倒数平方根。

```
__device__ maca_bfloat162 h2sin (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的正弦值。

```
__device__ maca_bfloat16 h2sqrt (const maca_bfloat162 a)
```

**参数**

a

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 以向最近偶数舍入模式，分别计算输入向量的两个分量的平方根。

```
__device__ maca_bfloat162 htrunc (const maca_bfloat162 h)
```

**参数**

h

- maca_bfloat162，只读入参

**返回值**

- maca_bfloat162

**函数描述**

- 分别将输入向量的两个分量截断，只保留整数部分。

### 1.8.15 单精度数学函数

```
__device__ float acosf (float x)
```

计算输入参数的反余弦函数值。

**返回值**

- 当输入参数 x 取值范围为 `[-1, 1]` 时，结果取值范围为 `[0, pi]`，以弧度为单位。
- acosf(1) 返回 +0。
- 当 x 不在 `[-1, 1]` 之间时，acosf(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反余弦函数值。

```
__device__ float acoshf (float x)
```

计算输入参数的反双曲余弦函数值。

**返回值**

- 返回值在区间 `[0, +∞]` 之间。
- acoshf(1) 返回 0。
- 当 x 在 `[-∞, 1]` 之间时，acoshf(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反双曲余弦函数值。

```
__device__ float asinf (float x)
```

计算输入参数的反正弦函数值。

**返回值**

- 当输入参数 x 取值范围为 `[-1, 1]` 时，结果取值范围为 `[-pi / 2, +pi / 2]`，以弧度为单位。
- asinf(1) 返回 +0。
- 当 x 不在 `[-1, 1]` 之间时，asinf(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反正弦函数值。

```
__device__ float asinhf (float x)
```

计算输入参数的反双曲正弦函数值。

**返回值**

- asinhf(0) 返回 1。

**函数描述**

- 计算输入参数 x 的反双曲正弦函数值。

```
__device__ float atan2f (float y, float x)
```

计算第一个和第二个输入参数之比的反正切。

**返回值**

- 结果以弧度为单位，在区间 [-Π, +Π]
- atan2f(0, 1) 返回 +0

**函数描述**

- 计算 y / x 的反正切函数值。结果的象限决定于 x、y 的符号。

```
__device__ float atanf (float x)
```

计算输入参数的反正切函数值。

**返回值**

- 结果以弧度为单位，在区间 [-pi / 2, +pi / 2]
- atanf(0) 返回 +0

**函数描述**

- 计算输入参数 x 的反正切函数值。

```
__device__ float atanhf (float x)
```

计算输入参数的双曲反正切函数值。

**返回值**

- atanhf(±0) 返回 ±0。
- atanhf(±1) 返回 ±∞
- 当 x 不在区间 [-1, 1] 时，atanhf(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的双曲反正切函数值。

```
__device__ float cbrtf (float x)
```

计算输入参数的立方根。

**返回值**

- 返回 $x^{1/3}$。
- cbrtf(±0) 返回 ±0。
- cbrtf(±∞) 返回 ±∞

**函数描述**

- 计算输入参数 x 的立方根，$x^{1/3}$。

```
__device__ float ceilf (float x)
```

计算输入参数的向上取整值。

**返回值**

- ceilf(±0) 返回 ±0。
- ceilf(±∞) 返回 ±∞

**函数描述**

- 计算不小于输入参数 x 的最小整数。

```
__device__ float copysignf (float x, float y)
```

以第二个参数的符号，创建具有第一个参数大小的值。

**返回值**

- 返回一个值，它有 x 的大小与 y 的符号。

**函数描述**

- 计算具有 x 的大小与 y 的符号的值。

```
__device__ float cosf (float x)
```

计算输入参数的余弦函数值

**返回值**

- cosf(±0) 返回 1。
- cosf(±∞) 返回 NaN。

**函数描述**

- 计算输入参数 x（以弧度为单位）的余弦值。

```
__device__ float coshf (float x)
```

计算输入参数的双曲余弦函数值。

**返回值**

- coshf(±0) 返回 1。
- coshf(±∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的双曲余弦函数值。

```
__device__ float cospif (float x)
```

计算输入参数 $x \times \Pi$ 的余弦函数值。

**返回值**

- cospif(±0) 返回 1。
- cospif(±∞) 返回 NaN。

**函数描述**

- 计算 $x \times \Pi$ 的余弦函数值，x 是输入参数。

```
__device__ float cyl_bessel_i0f (float x)
```

计算输入参数的 0 阶正则修正圆柱贝塞尔函数的值。

**返回值**

- 返回输入参数的 0 阶正则修正圆柱贝塞尔函数的值。

**函数描述**

- 计算输入参数的 0 阶正则修正圆柱贝塞尔函数的值，$I_0(x)$。

```
__device__ float cyl_bessel_i1f (float x)
```

计算输入参数的 1 阶正则修正圆柱贝塞尔函数的值。

**返回值**

- 返回输入参数的 1 阶正则修正圆柱贝塞尔函数的值。

**函数描述**

- 计算输入参数的 1 阶正则修正圆柱贝塞尔函数的值，$I_1(x)$。

```
__device__ float erfcf (float x)
```

计算输入参数的互补误差函数。

**返回值**

- erfcf(-∞) 返回 2。
- erfcf(+∞) 返回 +0。

**函数描述**

- 计算输入参数 x 的互补误差函数，1 - erf(x)。

```
__device__ float erfcinvf (float y)
```

计算输入参数的逆互补误差函数。

**返回值**

- erfcinvf(0) 返回 +∞。
- erfcinvf(2) 返回 -∞。

**函数描述**

- 对于 y 在区间 [0, 2] 内，计算输入参数 y 的逆互补误差函数。逆互补误差函数寻找符合下面条件的 x 的值：y = erfc(x)，$0 \leqslant y \leqslant 2$，并且$-\infty \leqslant x \leqslant \infty$。

```
__device__ float erfcxf (float x)
```

计算输入参数的缩放互补误差函数。

**返回值**

- erfcxf(-∞) 返回 +∞。
- erfcxf(+∞) 返回 +0。
- 如果正确计算的值的范围超出了单精度浮点数的表示范围，erfcxf(x) 返回 +∞。

**函数描述**

- 计算输入参数 x 的缩放互补误差函数 $e^{x^2}erf(x)$。

```
__device__ float erff (float x)
```

计算输入参数的错误函数。

**返回值**

- erff(±0) 返回 ±0。
- erff(±∞) 返回 ±1。

**函数描述**

- 计算输入参数 x 的错误函数。

```
__device__ float erfinvf (float y)
```

计算输入参数的逆误差函数。

**返回值**

- erfinvf(1) 返回 +∞。
- erfinvf(-1) 返回 -∞。

**函数描述**

- 对于 y 在区间 [-1, 1] 内，计算输入参数 y 的逆误差函数。逆误差函数寻找符合下面条件的 x 的值：y = erf(x)，0 ⩽ y ⩽ 2，并且-∞ ⩽ x ⩽ ∞。

```
__device__ float exp10f (float x)
```

计算输入参数的基数 10 指数。

**返回值**

- 返回 $10^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$10^x$。

```
__device__ float exp2f (float x)
```

计算输入参数的基数 2 指数。

**返回值**

- 返回 $2^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$2^x$。

```
__device__ float expf (float x)
```

计算输入参数的基数 e 指数。

**返回值**

- 返回 $e^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$e^x$。

```
__device__ float expm1f (float x)
```

计算输入参数的基数 e 指数减去 1。

**返回值**

- 返回 $e^x$ - 1。

**函数描述**

- 计算输入参数 x 的基数 10 指数减去 1，$e^x$ - 1。

```
__device__ float fabsf (float x)
```

计算输入参数的基数 e 指数减去 1。

**返回值**

- fabsf(±∞) 返回 +∞。
- fabsf(±0) 返回 0。

**函数描述**

- 计算输入参数 x 的绝对值。

```
__device__ float fdimf (float x, float y)
```

计算 x 与 y 之间的正数差。

**返回值**

- 返回 x 与 y 之间的正数差。
- 如果 x > y，fdimf(x, y) 返回 x - y。
- 如果 x ⩽ y，fdimf(x, y) 返回 +0。

**函数描述**

- 计算 x 与 y 之间的正数差。当 x > y 时，结果为 x - y；否则，是 +0。

```
__device__ float fdividef (float x, float y)
```

两个浮点数做除法。

**返回值**

- x / y。

**函数描述**

- 计算 x / y。

```
__device__ float floorf (float x)
```

计算不大于 x 的最大整数。

**返回值**

- 返回值以浮点数返回。
- floorf(±∞) 返回 ±∞
- floorf(±0) 返回 ±0。

**函数描述**

- 计算不大于 x 的最大整数。

```
__device__ float fmaf (float x, float y, float z)
```

以一个操作计算单精度 `x * y + z`。

**返回值**

返回舍入的单精度 `x * y + z` 的结果

- `fmaf(±∞, ±0, z)` 返回 NaN。
- `fmaf(±0, ±∞, z)` 返回 NaN。
- 如果 x * y 是确切的 +∞，`fmaf(x, y, -∞)` 返回 NaN。
- 如果 x * y 是确切的-∞，`fmaf(x, y, +∞)` 返回 NaN。

**函数描述**

- 以一个三元操作计算单精度 `x * y + z`。在计算之后，值需要舍入一次。

```
__device__ float fmaxf (float x, float y)
```

计算入参的最大值。

**返回值**

- 返回入参的最大值。
- 如果两个参数都是 NaN，返回 NaN。
- 如果一个参数是 NaN，返回另一个不是 NaN 的参数。

**函数描述**

- 计算入参 x，y 的最大值。将 NaN 参数视作无效数据。如果一个参数是 NaN，另一个参数是合法数字值，数字值要被返回。

```
__device__ float fminf (float x, float y)
```

计算入参的最小值。

**返回值**

- 返回入参的最小值。
- 如果两个参数都是 NaN，返回 NaN。
- 如果一个参数是 NaN，返回另一个不是 NaN 的参数。

**函数描述**

- 计算入参 x，y 的最小值。将 NaN 参数视作无效数据。如果一个参数是 NaN，另一个参数是合法数字值，数字值要被返回。

```
__device__ float fmodf (float x, float y)
```

计算 x/y 的浮点余数。

**返回值**

- 返回 x/y 的浮点余数。
- 如果 y 不是 0，fmodf(±0, y) 返回 ±0。
- 如果 x 是有限值，fmodf(x, ±∞) 返回 x。
- 如果 x 是 ±∞ 或者 y 是 0，fmodf(x, y) 返回 NaN。
- 如果某一个参数是 NaN，返回 NaN。

**函数描述**

- 计算 x/y 的浮点余数。这个函数计算的浮点余数具体的是 x - n * y，这里的 n 是 x / y，其分数部分被截断。计算出的结果会有 x 相同的符号，并且它的值大小比 y 的值大小小。

```
__device__ float frexpf (float x, int *nptr)
```

提取浮点值的尾数和指数。

**返回值**

- 返回分数分量 m。
- frexp(0, nptr) 对于分数分量返回 0，对于整数分量返回 0。
- frexp(±0, nptr) 返回 ±0 并且 nptr 指向的内存赋值 0。
- frexp(±∞, nptr) 返回 ±∞，并且在 nptr 指向的内存赋值一个不确定的值。
- frexp(NaN, y) 返回 NaN，并且在 nptr 指向的内存赋值一个不确定的值。

**函数描述**

- 将浮点值 x 分解为归一化分数元素的分量 m 和指数的另一项 n。m 的绝对值将大于或等于 0.5 且小于 1.0 或等于 0；$x = m * 2^n$。整数指数 n 将存储在 nptr 指向的位置。

```
__device__ float hypotf (float x, float y)
```

计算 x 与 y 的平方和的平方根。

**返回值**

- 斜边的长度 sqrt($x^2 + y^2$)。如果正确计算的值溢出，返回 + 无穷。如果正确计算结果向下溢出，返回 0。

**函数描述**

- 计算直角三角形斜边的长度，该直角三角形的两侧具有长度 x 和 y，且没有过度溢出或下溢。

```
__device__ float ilogbf (float x)
```

计算参数的无偏整数指数。

**返回值**

- 如果成功，返回参数的无偏指数。
- ilogbf(0) 返回 INT_MIN。
- ilogbf(NaN) 返回 INT_MIN。
- 如果 x 是 $\infty$ 或者确切值比 INT_MAX 大，ilogbf(x) 返回 INT_MAX。
- 如果确切值比 INT_MIN 小，ilogbf(x) 返回 INT_MIN。
- 注意，上面的操作不涉及 FP_ILOGB0、FP_ILOGBNAN。

**函数描述**

- 计算参数 x 的无偏整数指数。

```
__device__ bool isfinite (float a)
```

确认参数是否是有穷的。

**返回值**

- 当且仅当 a 是一个有穷值的时候返回 true。

**函数描述**

- 确认参数 a 是否是有穷的。

```
__device__ bool isinf (float a)
```

确认参数是否是无穷的。

**返回值**

- 当且仅当 a 是一个无穷值的时候返回 true。

**函数描述**

- 确认参数 a 是否是无穷的。

```
__device__ bool isnan (float a)
```

确认参数是否是 NaN。

**返回值**

- 当且仅当 a 是一个 NaN 的时候返回 true。

**函数描述**

- 确认参数 a 是否是 NaN。

```
__device__ float j0f (float x)
```

计算输入参数的第一类 0 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 0 阶贝塞尔函数的值。
- j0f(±∞) 返回 +0。
- j0f(NaN) 返回 NaN。

**函数描述**

- 计算输入参数 x 的第一类 0 阶贝塞尔函数的值，$J_0(x)$。

```
__device__ float j1f (float x)
```

计算输入参数的第一类 1 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 1 阶贝塞尔函数的值。
- j1f(±0) 返回 ±0。
- j1f(±∞) 返回 ±0。
- j1f(NaN) 返回 NaN。

**函数描述**

- 计算输入参数 x 的第一类 1 阶贝塞尔函数的值，$J_1(x)$。

```
__device__ float jnf (int n, float x)
```

计算输入参数的第一类 n 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 n 阶贝塞尔函数的值
- jnf(NaN) 返回 NaN。
- jnf(n, x) 在 n < 0 时返回 ±0。
- jnf(n, +∞) 返回 +0。

**函数描述**

- 计算输入参数 x 的第一类 n 阶贝塞尔函数的值，$J_1(x)$。

```
__device__ float ldexpf (float x, int exp)
```

计算 $x * 2^{exp}$ 的值。

**返回值**

- 如果正确计算的值超出单精度范围，ldexpf(x) 返回 ±∞。

**函数描述**

- 计算入参 x、exp 下面的数学公式：$x * 2^{exp}$ 的值。

```
__device__ float lgammaf (float x)
```

计算输入参数 gamma 函数绝对值的自然对数。

**返回值**

- lgammaf(1) 返回 +0。
- lgammaf(2) 返回 +0。
- 如果正确计算的值超出单精度表示范围，lgammaf(x) 返回 ±∞。
- 如果 x ⩽ 0 且是个整数，lgammaf(x) 返回 +∞。
- lgammaf(-∞) 返回 -∞。

- lgammaf(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的 gamma 函数绝对值的自然对数。

```
__device__ long long int llrintf (float x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的整数值。

**函数描述**

- 将 x 四舍五入到最接近的整数值，一半的情况下四舍五入到最接近的偶数整数值。如果结果超出返回类型的范围，则结果未定义。

```
__device__ long long int llroundf (float x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的整数值。

**函数描述**

- 将 x 四舍五入到最接近的整数值，一半的情况下从零四舍五入。如果结果超出返回类型的范围，则结果未定义。

```
__device__ float log10f (float x)
```

计算输入参数以 10 为底的对数。

**返回值**

- log10f(±0) 返回 -∞。
- log10f(1) 返回 +0。
- 当 x < 0 时，log10f(x) 返回 NaN。
- log10f(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 以 10 为底的对数。

```
__device__ float log1pf (float x)
```

计算 $\log_e(1+x)$ 的值。

**返回值**

- log1pf(±0) 返回 ±0。
- log1pf(-1) 返回 -∞。
- 当 x < -1 时，log1pf(x) 返回 NaN。
- log1pf(+∞) 返回 +∞。

**函数描述**

- 计算 $\log_e(1+x)$ 的值。

```
__device__ float log2f (float x)
```

计算输入参数以 2 为底的对数。

**返回值**

- log2f(±0) 返回 -∞。
- log2f(1) 返回 +0。
- 当 x < 0 时，log2f(x) 返回 NaN。
- log2f(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 以 2 为底的对数。

```
__device__ float logbf (float x)
```

计算输入参数 x 的指数的浮点表示形式。

**返回值**

- logbf(±0) 返回 -∞。
- logbf(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的指数的浮点表示形式。

```
__device__ float logf (float x)
```

计算输入参数的自然对数。

**返回值**

- logf(±0) 返回 -∞。
- logf(1) 返回 +0。
- 当 x < 0 时，logf(x) 返回 NaN。
- logf(+∞) 返回 +∞。

**函数描述**

计算输入参数的自然对数。

```
__device__ long int lrintf (float x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

将 x 四舍五入到最接近的整数值，一半的情况下四舍五入到最接近的偶数整数值。如果结果超出返回类型的范围，则结果未定义。

```
__device__ long int lrintf (float x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

将 x 四舍五入到最接近的整数值，一半的情况下从零四舍五入。如果结果超出返回类型的范围，则结果未定义。

```
__device__ float max (const float a, const float b)
```

计算输入的浮点数的最大值。

**函数描述**

计算输入参数 a 和 b 的最大值，行为与函数 fmaxf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ float min (const float a, const float b)
```

计算输入的浮点数的最小值。

**函数描述**

计算输入参数 a 和 b 的最小值，行为与函数 fminf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ float modff (float x, float *iptr)
```

将输入参数分解为分数和整数部分。

**返回值**

- modff(±x, iptr) 返回一个与 x 具有相同的符号的结果。
- modff(±∞, iptr) 返回 ±0 并且在 iptr 指向的内存中存入 ±∞。
- modff(NaN, iptr) 返货 NaN 并且在 iptr 指向的内存中存入 NaN。

**函数描述**

将参数 x 分解为分数和整数部分。分数部分存储在参数 iptr 中。分数和整数部分的符号与参数 x 相同。

```
__device__ float nanf (const char *tagp)
```

返回“Not a Number”值。

**返回值**

- nanf(tagp) 返回 NaN。

**函数描述**

返回安静 NaN 的表示。参数 tagp 选择一种可能的表示形式。

```
__device__ float nearbyintf (float x)
```

将输入参数四舍五入到最接近的整数。

**返回值**

- nearbyintf(±0) 返回 ±0。
- nearbyintf(±∞) 返回 ±∞。

**函数描述**

将参数 x 舍入为单精度浮点格式的整数值。

```
__device__ float nextafterf (float x, float y)
```

返回 y 方向上参数 x 之后的下一个可表示的单精度浮点值。

**返回值**

- nextafterf(x, y) = y 如果 x == y。
- nextafterf(x, y) = NaN 如果 x 或 y 是 NaN。

**函数描述**

计算沿 y 方向 x 之后的下一个可表示的单精度浮点值。例如，如果 y 大于 x，则 nextafterf() 返回大于 x 的最小可表示数字。

```
__device__ float norm3df (float a, float b, float c)
```

计算参数的三个坐标的平方和的平方根。

**返回值**

- 返回三维向量长度 sqrt($a^2 + b^2 + c^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

**函数描述**

计算欧几里德空间中三维向量的长度，无过度溢出或下溢。

```
__device__ float norm4df (float a, float b, float c, float d)
```

计算参数的四个坐标的平方和的平方根。

**返回值**

- 返回四维向量长度 sqrt($a^2 + b^2 + c^2 + d^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

**函数描述**

计算欧几里德空间中四维向量的长度，无过度溢出或下溢。

```
__device__ float normcdff (float y)
```

计算标准正态累积分布函数。

**返回值**

- normcdff(+∞) 返回 1。
- normcdff(-∞) 返回 +0。

**函数描述**

计算输入 y 的标准正态分布函数的值。

```
__device__ float normcdfinvf (float y)
```

计算标准正态累积分布函数。

**返回值**

- normcdfinvf(0) 返回 -∞。
- normcdfinvf(1) 返回 +∞。
- 如果 x 不在区间 [0, 1]，normcdfinvf(x) 返回 NaN。

**函数描述**

计算输入参数 y 的标准正态累积分布函数的逆。这个函数的输入值的取值范围在区间 (0,1)。

```
__device__ float normf (int dim, const float *p)
```

计算任意数量坐标的平方和的平方根。

**返回值**

- 返回 dim 维向量长度 sqrt($p_0^2 + p_1^2 + \cdots + p_{dim-1}^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

**函数描述**

计算向量 p 的长度，向量 p 的维数作为参数传递，不会出现过度溢出或下溢。

```
__device__ float powf (float x, float y)
```

计算第一个参数的值与第二个参数的幂。

**返回值**

- powf(±0, y) 对于 y 小于 0 的整数，返回 ±∞。
- powf(±0, y) 对于 y 一个大于 0 的整数奇数返回 ±0。
- powf(±0, y) 对于 y > 0 且不是一个奇数整数返回 +0。
- powf(-1, ±∞) 返回 1。
- powf(+1, y) 对于任何 y 甚至 NaN，返回 1。
- powf(x, ±0) 对于任何 x 甚至 NaN，返回 1。
- powf(x, y) 对于有穷 x < 0 且有穷非整数 y 返回 NaN。
- powf(x, -∞) 对于 |x| < 1 返回 +∞。
- powf(x, -∞) 对于 |x| > 1 返回 +0。
- powf(x, +∞) 对于 |x| < 1 返回 +0。
- powf(x, +∞) 对于 |x| > 1 返回 +∞。
- powf(-∞, y) 对于 y 是一个小于 0 的奇数返回-0。
- powf(-∞, y) 对于 y < 0 且不是奇数返回 +0。
- powf(-∞, y) 对于 y 是一个大于 0 的奇数返回 -∞。
- powf(-∞, y) 对于 y > 0 且不是奇数返回 +∞。
- powf(+∞, y) 对于 y < 0 返回 +0。
- powf(+∞, y) 对于 y > 0 返回 +∞。

**函数描述**

计算 x 的值为 y 的幂。

```
__device__ float rcbrtf (float x)
```

计算倒数立方根函数。

**返回值**

- rcbrt(±0) 返回 ±∞
- rcbrt(±∞) 返回 ±0

**函数描述**

计算 x 的倒数立方根函数。

```
__device__ float remainderf (float x, float y)
```

计算单精度浮点余数。

**返回值**

- remainderf(x, 0) 返回 NaN。
- remainderf(±∞, y) 返回 NaN。
- remainderf(x, ±∞) 对于有穷 x 返回 x。

**函数描述**

计算非零 y 的 x 除以 y 的单精度浮点余数 r，因此 r = x - ny。n 的值是最接近 x / y 的整数。

```
__device__ float remquof (float x, float y, int *quo)
```

计算单精度浮点余数和部分商。

**返回值**

- 返回余数。
- remquof(x, 0, quo) 返回 NaN。
- remquof(±∞, y, quo) 返回 NaN。
- remquof(x, ±∞, quo) 返回 x。

**函数描述**

以与 remainderf() 函数相同的方式计算双精度浮点余数。参数 quo 在 x 除以 y 后返回部分商。值 quo 与 x/y 有相同的符号，可能不是精确商，但与低阶 3 位中的精确商一致。

```
__device__ float rhypotf (float x, float y)
```

计算两个参数平方和的平方根。

**返回值**

- 返回斜边长度倒数值 1 / sqrt($x^2$ + $y^2$)。如果平方根溢出，返回 0。如果平方根向下溢出，返回 +∞。

**函数描述**

计算直角三角形斜边长度倒数，该直角三角形的两侧具有长度 x 和 y，没有过度溢出或下溢。

```
__device__ float rintf (float x)
```

将输入四舍五入为最接近的浮点类型的整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

以浮点格式将 x 四舍五入为最接近的整数值，将一半大小写四舍五入为最接近的偶数整数值。

```
__device__ float rnorm3df (float a, float b, float c)
```

计算参数三个坐标的平方和的平方根的倒数。

**返回值**

- 返回三维向量长度的倒数 1 / sqrt($a^2$ + $b^2$ + $c^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 +∞。

**函数描述**

计算欧几里德空间中三维向量长度倒数，无过度溢出或下溢。

```
__device__ float rnorm4df (float a, float b, float c, float d)
```

计算参数四个坐标的平方和的平方根的倒数。

**返回值**

- 返回四维向量长度的倒数 1 / sqrt($a^2 + b^2 + c^2 + d^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 $+\infty$。

**函数描述**

计算欧几里德空间中四维向量长度倒数，无过度溢出或下溢。

```
__device__ float rnormf (int dim, const float *p)
```

计算任意数量坐标的平方和的平方根的倒数。

**返回值**

- 返回 dim 维向量长度的倒数 1 / sqrt($p_0^2 + p_1^2 + \cdots + p_{dim-1}^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 $+\infty$。

**函数描述**

计算欧几里德空间中 dim 维向量 p 长度倒数，无过度溢出或下溢。

```
__device__ float roundf (float x)
```

四舍五入到最接近的浮点整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

以浮点格式将 x 四舍五入到最接近的整数值，中间的大小写从零四舍五入。

```
__device__ float rsqrtf (float x)
```

计算输入值平方根的倒数。

**返回值**

- 返回 1 / sqrt(x)。
- rsqrtf($+\infty$) 返回 +0
- rsqrtf($\pm 0$) 返回 $\pm\infty$
- rsqrtf(x) 如果 x < 0 返回 NaN。

**函数描述**

计算输入参数 x 的平方根的倒数。

```
__device__ float scalblnf (float x, long int n)
```

按 2 的整数幂缩放浮点输入。

**返回值**

- 返回 x * $2^n$。
- scalblnf($\pm 0$, n) 返回 $\pm 0$。
- scalblnf(x, 0) 返回 x。
- scalblnf($\pm\infty$, n) 返回 $\pm\infty$。

**函数描述**

按 $2^n$ 缩放浮点输入 x。

```
__device__ bool signbit (float a)
```

返回输入的符号位。

**返回值**

- 当且仅当 a 是负数返回 true。

**函数描述**

确认浮点数是不是负数。

```
__device__ void sincosf (float x, float *sptr, float *cptr)
```

计算第一个输入参数的正弦和余弦。

**返回值**

- 无。

**函数描述**

计算第一个输入参数 x（以弧度为单位）的正弦和余弦。正弦和余弦的结果分别写入第二个参数 sptr 和第三个参数 cptr。

```
__device__ void sincospif (float x, float *sptr, float *cptr)
```

计算第一个输入参数 * Π 的正弦和余弦。

**返回值**

- 无。

**函数描述**

计算第一个输入参数 x（以弧度为单位）* Π 的正弦和余弦。正弦和余弦的结果分别写入第二个参数 sptr 和第三个参数 cptr。

```
__device__ float sinf (float x)
```

计算输入参数的正弦。

**返回值**

- sinf(±0) 返回 ±0。
- sinf(±∞) 返回 NaN。

**函数描述**

计算输入参数 x 的正弦。

```
__device__ float sinhf (float x)
```

计算输入参数的双曲正弦。

**返回值**

- sinhf(±0) 返回 ±0。
- sinhf(±∞) 返回 ±∞。

**函数描述**

计算输入参数 x 的双曲正弦。

```
__device__ float sinpif (float x)
```

计算输入参数 * Π 的正弦。

**返回值**

- sinf(±0) 返回 ±0。

- sinf(±∞) 返回 NaN。

**函数描述**

计算输入参数 x * Π 的正弦。

```
__device__ float sqrtf (float x)
```

计算输入参数的平方根。

**返回值**

- sqrtf(±0) 返回 ±0。
- sqrtf(+∞) 返回 +∞。
- 如果 x < 0，sqrtf(x) 返回 x。

**函数描述**

计算输入参数 x 的平方根。

```
__device__ float tanf (float x)
```

计算输入参数的正切。

**返回值**

- tanf(±0) 返回 ±0。
- tanf(±∞) 返回 NaN。

**函数描述**

计算输入参数 x 的正切。

```
__device__ float tanhf (float x)
```

计算输入参数的双曲正切。

**返回值**

- tanhf(±0) 返回 ±0。

**函数描述**

计算输入参数 x 的双曲正切。

```
__device__ float tgammaf (float x)
```

计算输入参数的 gamma 函数。

**返回值**

- tgammaf(±0) 返回 ±∞。
- tgammaf(2) 返回 +1。
- 如果计算值超出单精度表示范围，tgammaf(x) 返回 ±∞。
- 如果 x < 0 且是个整数，tgammaf(x) 返回 NaN。
- tgammaf(-∞) 返回 NaN。
- tgammaf(+∞) 返回 +∞。

**函数描述**

计算输入参数 x 的 gamma 函数。

```
__device__ float truncf (float x)
```

将输入参数截断为整数部分。

**返回值**

- 返回截断的整数部分。

**函数描述**

将 x 四舍五入到最接近的整数值，该整数值的大小不超过 x。

```
__device__ float y0f (float x)
```

为输入参数计算第二类 0 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 0 阶贝塞尔函数的值。
- y0f(0) 返回 -∞。
- y0f(x) 对于 x < 0 返回 NaN。
- y0f(+∞) 返回 +0。
- y0f(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 0 阶贝塞尔函数的值，$Y_0(x)$。

```
__device__ float y1f (float x)
```

为输入参数计算第二类 1 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 1 阶贝塞尔函数的值。
- y1f(0) 返回 -∞。
- y1f(x) 对于 x < 0 返回 NaN。
- y1f(+∞) 返回 +0。
- y1f(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 1 阶贝塞尔函数的值，$Y_1(x)$。

```
__device__ float ynf (int n, float x)
```

为输入参数计算第二类 n 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 n 阶贝塞尔函数的值。
- ynf(0) 返回 -∞。
- ynf(x) 对于 x < 0 返回 NaN。
- ynf(+∞) 返回 +0。
- ynf(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 n 阶贝塞尔函数的值，$Y_n(x)$。

### 1.8.16 双精度数学函数

```
__device__ double acos (double x)
```

计算输入参数的反余弦函数值。

**返回值**

- 结果以弧度位单位，在范围 [0, Π] 之间，x 取值范围在 [-1, 1] 之间。
- acos(1) 返回 +0。
- 当 x 不在 [-1, 1] 之间时，acos(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反余弦函数值。

```
__device__ double acosh (double x)
```

计算输入参数的反双曲余弦函数值。

**返回值**

- 返回值在区间 [0, +∞] 之间。
- acosh(1) 返回 0。
- 当 x 在 [-∞, 1] 之间时，acosh(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反双曲余弦函数值。

```
__device__ double asin (double x)
```

计算输入参数的反正弦函数值。

**返回值**

- 结果以弧度位单位，在范围 [-Π / 2, + Π / 2] 之间，x 取值范围在 [-1, 1] 之间。
- asin(1) 返回 +0。
- 当 x 不在 [-1, 1] 之间时，asin(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的反正弦函数值。

```
__device__ double asinh (double x)
```

计算输入参数的反双曲正弦函数值。

**返回值**

- asinh(0) 返回 1。

**函数描述**

- 计算输入参数 x 的反双曲正弦函数值。

```
__device__ double atan (double x)
```

计算输入参数的反正切函数值。

**返回值**

- 结果以弧度位单位，在区间 [-Π / 2, +Π / 2]。
- atan(0) 返回 +0。

**函数描述**

- 计算输入参数 x 的反正切函数值。

```
__device__ double atan2 (double y, double x)
```

计算第一个和第二个输入参数之比的反正切。

**返回值**

- 结果以弧度位单位，在区间 [-Π, +Π]。
- atan2(0, 1) 返回 +0。

**函数描述**

- 计算 y / x 的反正切函数值。结果的象限决定于 x、y 的符号。

```
__device__ double atanh (double x)
```

计算输入参数的双曲反正切函数值。

**返回值**

- atanh(±0) 返回 ±0。
- atanh(±1) 返回 ±∞
- 当 x 不在区间 [-1, 1] 时，atanh(x) 返回 NaN。

**函数描述**

- 计算输入参数 x 的双曲反正切函数值。

计算输入参数的立方根。

**返回值**

- 返回 $x^{1/3}$。
- cbrt(±0) 返回 ±0。
- cbrt(±∞) 返回 ±∞

**函数描述**

- 计算输入参数 x 的立方根，$x^{1/3}$。

```
__device__ double ceil (double x)
```

计算输入参数的向上取整值。

**返回值**

- ceil(±0) 返回 ±0。
- ceil(±∞) 返回 ±∞

**函数描述**

- 计算不小于输入参数 x 的最小整数。

```
__device__ double copysign (double x, double y)
```

创造具有给定大小的值，复制第二个值的符号。

**返回值**

- 返回具有 x 的大小与 y 的符号的值。

**函数描述**

- 计算具有 x 的大小与 y 的符号的值。

```
__device__ double cos (double x)
```

计算输入参数的余弦函数值。

**返回值**

- cos(0) 返回 1。
- cos(±∞) 返回 NaN。

**函数描述**

- 计算输入参数 x（弧度位单位）的余弦值。

```
__device__ double cosh (double x)
```

计算输入参数的双曲余弦函数值。

**返回值**

- cosh(0) 返回 1。
- cosh(±∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的双曲余弦函数值。

```
__device__ double cospi (double x)
```

计算输入参数 × Π 的余弦函数值。

**返回值**

- cospi(0) 返回 1。
- cospi(±∞) 返回 NaN。

**函数描述**

- 计算 x × Π 的余弦函数值，x 是输入参数。

```
__device__ double cyl_bessel_i0 (double x)
```

计算输入参数的 0 阶正则修改圆柱贝塞尔函数的值。

**返回值**

- 返回输入参数的 0 阶正则修改圆柱贝塞尔函数的值。

**函数描述**

- 计算输入参数的 0 阶正则修改圆柱贝塞尔函数的值，$I_0(x)$。

```
__device__ double cyl_bessel_i1 (double x)
```

计算输入参数的 1 阶正则修改圆柱贝塞尔函数的值。

**返回值**

- 返回输入参数的 1 阶正则修改圆柱贝塞尔函数的值。

**函数描述**

- 计算输入参数的 1 阶正则修改圆柱贝塞尔函数的值，$I_1(x)$。
- 计算输入参数 x 的缩放互补误差函数。

```
__device__ double erf (double x)
```

计算输入参数的错误函数。

**返回值**

- erf(±0) 返回 ±0。
- erf(±∞) 返回 ±1。

**函数描述**

- 计算输入参数 x 的错误函数。

```
__device__ double erfc (double x)
```

计算输入参数的互补误差函数。

**返回值**

- erfc(-∞) 返回 2。
- erfc(+∞) 返回 +0。

**函数描述**

- 计算输入参数 x 的互补误差函数，1 - erf(x)。

```
__device__ double erfcinv (double y)
```

计算输入参数的逆互补误差函数。

**返回值**

- erfcinv(0) 返回 +∞。
- erfcinv(2) 返回 -∞。

**函数描述**

- 计算输入参数 y 的逆互补误差函数，对于 y 在区间 [0, 2] 上。逆互补误差函数寻找符合下面条件的 x 的值：y = erfc(x)，0 ⩽ y ⩽ 2，并且-∞ ⩽ x ⩽ ∞。

```
__device__ double erfcx (double x)
```

计算输入参数的缩放互补误差函数。

**返回值**

- erfcx(-∞) 返回 +∞。
- erfcx(+∞) 返回 +0。
- 如果正确计算的值的范围超出了单精度浮点数的表示范围，erfcx(x) 返回 +∞。

**函数描述**

```
__device__ double erfinv (double y)
```

计算输入参数的逆误差函数。

**返回值**

- erfinv(1) 返回 +∞。
- erfinv(-1) 返回 -∞。

**函数描述**

- 计算输入参数 y 的逆误差函数，y 在区间 [-1, 1] 上。逆误差函数寻找符合下面条件的 x 的值：y = erf(x)，0 ⩽ y ⩽ 2，并且-∞ ⩽ x ⩽ ∞。

```
__device__ double exp (double x)
```

计算输入参数的基数 e 指数。

**返回值**

- 返回 $e^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$e^x$。

```
__device__ double exp10 (double x)
```

计算输入参数的基数 10 指数。

**返回值**

- 返回 $10^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$10^x$。

```
__device__ double exp2 (double x)
```

计算输入参数的基数 2 指数。

**返回值**

- 返回 $2^x$。

**函数描述**

- 计算输入参数 x 的基数 10 指数，$2^x$。

```
__device__ double expm1 (double x)
```

计算输入参数的基数 e 指数减去 1。

**返回值**

- 返回 $e^x$ - 1。

**函数描述**

- 计算输入参数 x 的基数 10 指数减去 1，$e^x$ - 1。

```
__device__ double fabs (double x)
```

计算输入参数的基数 e 指数减去 1。

**返回值**

- fabs($\pm\infty$) 返回 $+\infty$。
- fabs($\pm0$) 返回 0。

**函数描述**

- 计算输入参数 x 的绝对值。

```
__device__ double fdim (double x, double y)
```

计算 x 与 y 之间的正数差。

**返回值**

- 返回 x 与 y 之间的正数差。
- 如果 x > y，fdim(x, y) 返回 x - y。
- 如果 x $\leqslant$ y，fdim(x, y) 返回 +0。

**函数描述**

- 计算 x 与 y 之间的正数差。当 x > y 时结果为 x - y；否则，为 +0。

```
__device__ double floor (double x)
```

计算不大于 x 的最大整数。

**返回值**

- 返回值以浮点数返回。
- floor(±∞) 返回 ±∞
- floor(±0) 返回 ±0。

**函数描述**

- 计算不大于 x 的最大整数。

```
__device__ double fma (double x, double y, double z)
```

以一个操作计算单精度 `x * y + z`。

**返回值**

返回舍入的单精度 `x * y + z` 的结果：

- `fma(±∞, ±0, z)` 返回 NaN。
- `fma(±0, ±∞, z)` 返回 NaN。
- 如果 x * y 是确切的 +∞，`fma(x, y, -∞)` 返回 NaN。
- 如果 x * y 是确切的-∞，`fma(x, y, +∞)` 返回 NaN。

**函数描述**

- 以一个三元操作计算单精度 `x * y + z`。在计算之后，值需要舍入一次。

```
__device__ double fmax (double, double)
```

计算入参的最大值。

**返回值**

- 返回入参的最大值。
- 如果两个参数都是 NaN，返回 NaN。
- 如果一个参数是 NaN，返回另一个不是 NaN 的参数。

**函数描述**

- 计算入参 x，y 的最大值。将 NaN 参数视作无效数据。如果一个参数是 NaN，另一个参数是合法数字值，数字值要被返回。

```
__device__ double fmin (double x, double y)
```

计算入参的最小值。

**返回值**

- 返回入参的最小值。
- 如果两个参数都是 NaN，返回 NaN。
- 如果一个参数是 NaN，返回另一个不是 NaN 的参数。

**函数描述**

- 计算入参 x，y 的最小值。将 NaN 参数视作无效数据。如果一个参数是 NaN，另一个参数是合法数字值，数字值要被返回。

```
__device__ double fmod (double x, double y)
```

计算 x/y 的浮点余数。

**返回值**

- 返回 x/y 的浮点余数。
- 如果 y 不是 0，fmod(±0, y) 返回 ±0。
- 如果 x 是有限值，fmod(x, ±∞) 返回 x。
- 如果 x 是 ±∞ 或者 y 是 0，fmod(x, y) 返回 NaN。
- 如果某一个参数是 NaN，返回 NaN。

**函数描述**

- 计算 x/y 的浮点余数。这个函数计算的浮点余数具体的是 x - n * y，这里的 n 是 x / y，其分数部分被截断。计算出的结果会有 x 相同的符号，并且它的值大小比 y 的值大小小。

```
__device__ double frexp (double x, int *nptr)
```

提取浮点值的尾数和指数。

**返回值**

- 返回分数分量 m。
- frexp(0, nptr) 对于分数分量返回 0，对于整数分量返回 0。
- frexp(±0, nptr) 返回 ±0 并且 nptr 指向的内存赋值 0。
- frexp(±∞, nptr) 返回 ±∞，并且在 nptr 指向的内存赋值一个不确定的值。
- frexp(NaN, y) 返回 NaN，并且在 nptr 指向的内存赋值一个不确定的值。

**函数描述**

- 将浮点值 x 分解为归一化分数元素的分量 m 和指数的另一项 n。m 的绝对值将大于或等于 0.5 且小于 1.0 或等于 0；$x = m * 2^n$。整数指数 n 将存储在 nptr 指向的位置。

```
__device__ double hypot (double x, double y)
```

计算 x 与 y 的平方和的平方根。

**返回值**

- 斜边的长度 $sqrt(x^2 + y^2)$。如果正确计算的值溢出，返回 + 无穷。如果正确计算结果向下溢出，返回 0。

**函数描述**

- 计算直角三角形斜边的长度，该直角三角形的两侧具有长度 x 和 y，且没有过度溢出或下溢。

```
__device__ int ilogb (double x)
```

计算参数的无偏整数指数。

**返回值**

- 如果成功，返回参数的无偏指数。
- ilogb(0) 返回 INT_MIN。
- ilogb(NaN) 返回 INT_MIN。
- 如果 x 是 ∞ 或者确切值比 INT_MAX 大，ilogb(x) 返回 INT_MAX。
- 如果确切值比 INT_MIN 小，ilogb(x) 返回 INT_MIN
- 注意，上面的操作不涉及 FP_ILOGB0、FP_ILOGBNAN。

**函数描述**

- 计算参数 x 的无偏整数指数。

```
__device__ bool isfinite (double a)
```

确认参数是否是有穷的。

**返回值**

- 当且仅当 a 是一个有穷值的时候返回 true。

**函数描述**

- 确认参数 a 是否是有穷的。

```
__device__ bool isinf (double a)
```

确认参数是否是无穷的。

**返回值**

- 当且仅当 a 是一个无穷值的时候返回 true。

**函数描述**

- 确认参数 a 是否是无穷的。

```
__device__ bool isnan (double a)
```

确认参数是否是 NaN。

**返回值**

- 当且仅当 a 是一个 NaN 的时候返回 true。

**函数描述**

- 确认参数 a 是否是 NaN。

```
__device__ double j0 (double x)
```

计算输入参数的第一类 0 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 0 阶贝塞尔函数的值。
- j0($\pm\infty$) 返回 +0。
- j0(NaN) 返回 NaN。

**函数描述**

- 计算输入参数 x 的第一类 0 阶贝塞尔函数的值，$J_0(x)$。

```
__device__ double j1 (double x)
```

计算输入参数的第一类 1 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 1 阶贝塞尔函数的值。
- j1($\pm 0$) 返回 $\pm 0$。
- j1($\pm\infty$) 返回 $\pm 0$。
- j1(NaN) 返回 NaN。

**函数描述**

- 计算输入参数 x 的第一类 1 阶贝塞尔函数的值，$J_1(x)$。

```
__device__ double jn (int n, double x)
```

计算输入参数的第一类 n 阶贝塞尔函数的值。

**返回值**

- 输入参数的第一类 n 阶贝塞尔函数的值。
- jn(NaN) 返回 NaN。
- jn(n, x) 在 n < 0 时返回 ±0。
- jn(n, +∞) 返回 +0。

**函数描述**

- 计算输入参数 x 的第一类 n 阶贝塞尔函数的值，$J_n(x)$。

```
__device__ double ldexp (double x, int exp)
```

计算 x * $2^{exp}$ 的值。

**返回值**

- 如果正确计算的值超出单精度范围，ldexp(x) 返回 ±∞。

**函数描述**

- 计算入参 x、exp 下面的数学公式：x * $2^{exp}$ 的值。

```
__device__ double lgamma (double x)
```

计算输入参数 gamma 函数绝对值的自然对数。

**返回值**

- lgamma(1) 返回 +0。
- lgamma(2) 返回 +0。
- 如果正确计算的值超出单精度表示范围，lgamma(x) 返回 ±∞。
- 如果 x ⩽ 0 且是个整数，lgamma(x) 返回 +∞。
- lgamma(-∞) 返回 -∞。
- lgamma(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的 gamma 函数绝对值的自然对数。

```
__device__ long long int llrint (double x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的整数值。

**函数描述**

- 将 x 四舍五入到最接近的整数值，一半的情况下四舍五入到最接近的偶数整数值。如果结果超出返回类型的范围，则结果未定义。

```
__device__ long long int llround (double x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的整数值。

**函数描述**

- 将 x 四舍五入到最接近的整数值，一半的情况下从零四舍五入。如果结果超出返回类型的范围，则结果未定义。

```
__device__ double log (double x)
```

计算输入参数的自然对数。

**返回值**

- log(±0) 返回 -∞。
- log(1) 返回 +0。
- 当 x < 0，log(x) 时返回 NaN。
- log(+∞) 返回 +∞。

**函数描述**

计算输入参数的自然对数。

```
__device__ double log10 (double x)
```

计算输入参数以 10 为底的对数。

**返回值**

- log10(±0) 返回 -∞。
- log10(1) 返回 +0。
- 当 x < 0 时，log10(x) 返回 NaN。
- log10(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 以 10 为底的对数。

```
__device__ double log1p (double x)
```

计算 $\log_e(1+x)$ 的值。

**返回值**

- log1p(±0) 返回 ±0。
- log1p(-1) 返回 -∞。
- 当 x < -1 时，log1p(x) 返回 NaN。
- log1p(+∞) 返回 +∞。

**函数描述**

- 计算 $\log_e(1+x)$ 的值。

```
__device__ double log2 (double x)
```

计算输入参数以 2 为底的对数。

**返回值**

- log2(±0) 返回 -∞。
- log2(1) 返回 +0。
- 当 x < 0 时，log2(x) 返回 NaN。

- log2(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 以 2 为底的对数。

```
__device__ double logb (double x)
```

计算输入参数 x 的指数的浮点表示形式。

**返回值**

- logb(±0) 返回 -∞。
- logb(+∞) 返回 +∞。

**函数描述**

- 计算输入参数 x 的指数的浮点表示形式。

```
__device__ long int lrint (double x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

将 x 四舍五入到最接近的整数值，一半的情况下四舍五入到最接近的偶数整数值。如果结果超出返回类型的范围，则结果未定义。

```
__device__ long int lround (double x)
```

将输入四舍五入到最接近的整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

将 x 四舍五入到最接近的整数值，一半的情况下从零四舍五入。如果结果超出返回类型的范围，则结果未定义。

```
__device__ double max (const double a, const float b)
```

计算输入的浮点数的最大值。

**函数描述**

把 float 参数转为 double，计算输入参数 a 和 b 的最大值，行为与函数 fmaxf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double max (const float a, const double b)
```

计算输入的浮点数的最大值。

**函数描述**

把 float 参数转为 double，计算输入参数 a 和 b 的最大值，行为与函数 fmaxf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double max (const double a, const double b)
```

计算输入的浮点数的最大值。

**函数描述**

计算输入参数 a 和 b 的最大值，行为与函数 fmaxf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double min (const double a, const float b)
```

计算输入的浮点数的最小值。

**函数描述**

把 float 参数转为 double，计算输入参数 a 和 b 的最小值，行为与函数 fminf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double min (const float a, const double b)
```

计算输入的浮点数的最小值。

**函数描述**

把 float 参数转为 double，计算输入参数 a 和 b 的最小值，行为与函数 fminf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double min (const double a, const double b)
```

计算输入的浮点数的最小值。

**函数描述**

计算输入参数 a 和 b 的最小值，行为与函数 fminf() 一致。

**注解：** 这个函数与 std:: 里面的函数不同。

```
__device__ double modf (double x, double *iptr)
```

将输入参数分解为分数和整数部分。

**返回值**

- modf(±x, iptr) 返回一个与 x 具有相同的符号的结果。
- modf(±∞, iptr) 返回 ±0 并且在 iptr 指向的内存中存入 ±∞。
- modf(NaN, iptr) 返货 NaN 并且在 iptr 指向的内存中存入 NaN。

**函数描述**

将参数 x 分解为分数和整数部分。分数部分存储在参数 iptr 中。分数和整数部分的符号与参数 x 相同。

```
__device__ double nan (const char *tagp)
```

返回“Not a Number”值。

**返回值**

- nan(tagp) 返回 NaN。

**函数描述**

返回安静 NaN 的表示。参数 tagp 选择一种可能的表示形式。

```
__device__ double nearbyint (double x)
```

将输入参数四舍五入到最接近的整数。

**返回值**

- nearbyint(±0) 返回 ±0。
- nearbyint(±∞) 返回 ±∞。

**函数描述**

将参数 x 舍入为单精度浮点格式的整数值。

```
__device__ double nextafter (double x, double y)
```

返回 y 方向上参数 x 之后的下一个可表示的单精度浮点值。

**返回值**

- 如果 x == y，nextafter(x, y) = y。
- 如果 x 或 y 是 NaN，nextafter(x, y) = NaN。

**函数描述**

计算沿 y 方向 x 之后的下一个可表示的单精度浮点值。例如，如果 y 大于 x，则 nextafterf() 返回大于 x 的最小可表示数字。

```
__device__ double norm (int dim, const double *p)
```

计算任意数量坐标的平方和的平方根。

**返回值**

- 返回 dim 维向量长度 sqrt($p_0^2 + p_1^2 + \cdots + p_{dim-1}^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

```
__device__ double norm3d (double a, double b, double c)
```

计算参数的三个坐标的平方和的平方根。

**返回值**

- 返回三维向量长度 sqrt($a^2 + b^2 + c^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

**函数描述**

计算欧几里德空间中三维向量的长度，无过度溢出或下溢。

```
__device__ double norm4d (double a, double b, double c, double d)
```

计算参数的四个坐标的平方和的平方根。

**返回值**

- 返回四维向量长度 sqrt($a^2 + b^2 + c^2 + d^2$)。如果计算结果溢出，则返回 +∞，如果计算结果向下溢出，则返回 0。

**函数描述**

计算欧几里德空间中四维向量的长度，无过度溢出或下溢。

```
__device__ double normcdf (double y)
```

计算标准正态累积分布函数。

**返回值**

- normcdf(+∞) 返回 1。
- normcdf(-∞) 返回 +0。

**函数描述**

计算输入 y 的标准正态分布函数的值。

```
__device__ double normcdfinv (double y)
```

计算标准正态累积分布函数。

**返回值**

- normcdfinv(0) 返回 -∞。
- normcdfinv(1) 返回 +∞。
- 如果 x 不在区间 [0, 1]，normcdfinv(x) 返回 NaN。

**函数描述**

计算输入参数 y 的标准正态累积分布函数的逆。这个函数的输入值的取值范围在区间 (0,1)。

计算向量 p 的长度，向量 p 的维数作为参数传递，不会出现过度溢出或下溢。

```
__device__ double pow (double x, double y)
```

计算第一个参数的值与第二个参数的幂。

**返回值**

- pow(±0, y) 对于 y 小于 0 的整数，返回 ±∞。
- pow(±0, y) 对于 y 一个大于 0 的整数奇数返回 ±0。
- pow(±0, y) 对于 y > 0 且不是一个奇数整数返回 +0。
- pow(-1, ±∞) 返回 1。
- pow(+1, y) 对于任何 y 甚至 NaN，返回 1。
- pow(x, ±0) 对于任何 x 甚至 NaN，返回 1。
- pow(x, y) 对于有穷 x < 0 且有穷非整数 y 返回 NaN。
- pow(x, -∞) 对于 |x| < 1 返回 +∞。
- pow(x, -∞) 对于 |x| > 1 返回 +0。
- pow(x, +∞) 对于 |x| < 1 返回 +0。
- pow(x, +∞) 对于 |x| > 1 返回 +∞。
- pow(-∞, y) 对于 y 是一个小于 0 的奇数返回-0。
- pow(-∞, y) 对于 y < 0 且不是奇数返回 +0。
- pow(-∞, y) 对于 y 是一个大于 0 的奇数返回 -∞。
- pow(-∞, y) 对于 y > 0 且不是奇数返回 +∞。
- pow(+∞, y) 对于 y < 0 返回 +0。
- pow(+∞, y) 对于 y > 0 返回 +∞。

**函数描述**

计算 x 的值为 y 的幂。

```
__device__ double rcbrt (double x)
```

计算倒数立方根函数。

**返回值**

- rcbrt(±0) 返回 ±∞。
- rcbrt(±∞) 返回 ±0。

**函数描述**

计算 x 的倒数立方根函数。

```
__device__ double remainder (double x, double y)
```

计算单精度浮点余数。

**返回值**

- remainderf(x, 0) 返回 NaN。
- remainderf(±∞, y) 返回 NaN。
- remainderf(x, ±∞) 对于有穷 x 返回 x。

**函数描述**

计算非零 y 的 x 除以 y 的单精度浮点余数 r，因此 r = x - ny。n 的值是最接近 x / y 的整数。

```
__device__ double remquo (double x, double y, int *quo)
```

计算单精度浮点余数和部分商。

**返回值**

- 返回余数。
- remquo(x, 0, quo) 返回 NaN。
- remquo(±∞, y, quo) 返回 NaN。
- remquo(x, ±∞, quo) 返回 x。

**函数描述**

以与 remainder() 函数相同的方式计算双精度浮点余数。参数 quo 在 x 除以 y 后返回部分商。值 quo 与 x/y 有相同的符号，可能不是精确商，但与低阶 3 位中的精确商一致。

```
__device__ double rhypot (double x, double y)
```

计算两个参数平方和的平方根。

**返回值**

- 返回斜边长度倒数值 1 / sqrt($x^2$ + $y^2$)。如果平方根溢出，返回 0。如果平方根向下溢出，返回 +∞。

**函数描述**

计算直角三角形斜边长度倒数，该直角三角形的两侧具有长度 x 和 y，没有过度溢出或下溢。

```
__device__ double rint (double x)
```

将输入四舍五入为最接近的浮点整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

以浮点格式将 x 四舍五入为最接近的整数值，将一半大小写四舍五入为最接近的偶数整数值。

```
__device__ double rnorm (int dim, const double *p)
```

计算任意数量坐标的平方和的平方根的倒数。

**返回值**

- 返回 dim 维向量长度的倒数 1 / sqrt($p_0^2 + p_1^2 + \cdots + p_{dim-1}^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 $+\infty$。

**函数描述**

计算欧几里德空间中 dim 维向量 p 长度倒数，无过度溢出或下溢。

```
__device__ double rnorm3d (double a, double b, double c)
```

计算参数三个坐标的平方和的平方根的倒数。

**返回值**

- 返回三维向量长度的倒数 1 / sqrt($a^2 + b^2 + c^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 $+\infty$。

**函数描述**

计算欧几里德空间中三维向量长度倒数，无过度溢出或下溢。

```
__device__ double rnorm4d (double a, double b, double c, double d)
```

计算参数四个坐标的平方和的平方根的倒数。

**返回值**

- 返回四维向量长度的倒数 1 / sqrt($a^2 + b^2 + c^2 + d^2$)。如果平方根溢出返回 0，如果平方根向下溢出，返回 $+\infty$。

**函数描述**

计算欧几里德空间中四维向量长度倒数，无过度溢出或下溢。

```
__device__ double round (double x)
```

四舍五入到最接近的浮点整数值。

**返回值**

- 返回舍入后的值。

**函数描述**

以浮点格式将 x 四舍五入到最接近的整数值，中间的大小写从零四舍五入。

```
__device__ double rsqrt (double x)
```

计算输入值平方根的倒数。

**返回值**

- 返回 1 / sqrt(x)。
- rsqrt($+\infty$) 返回 +0
- rsqrt($\pm 0$) 返回 $\pm\infty$
- 如果 x < 0，rsqrt(x) 返回 NaN。

**函数描述**

计算输入参数 x 的平方根的倒数。

```
__device__ double scalbln (double x, long int n)
```

按 2 的整数幂缩放浮点输入。

**返回值**

- 返回 x * $2^n$。
- scalblnf(±0, n) 返回 ±0。
- scalblnf(x, 0) 返回 x。
- scalblnf(±∞, n) 返回 ±∞。

**函数描述**

按 $2^n$ 缩放浮点输入 x。

```
__device__ double scalbln (double x, int n)
```

按 2 的整数幂缩放浮点输入。

**返回值**

- 返回 x * $2^n$。
- scalbln(±0, n) 返回 ±0。
- scalbln(x, 0) 返回 x。
- scalbln(±∞, n) 返回 ±∞。

**函数描述**

按 $2^n$ 缩放浮点输入 x。

```
__device__ bool signbit (double a)
```

返回输入的符号位。

**返回值**

- 当且仅当 a 是负数返回 true。

**函数描述**

确认浮点数是不是负数。

```
__device__ float sin (double x)
```

计算输入参数的正弦。

**返回值**

- sin(±0) 返回 ±0。
- sin(±∞) 返回 NaN。

**函数描述**

计算输入参数 x 的正弦。

```
__device__ void sincos (double x, double *sptr, double *cptr)
```

计算第一个输入参数的正弦和余弦。

**返回值**

- 无。

**函数描述**

计算第一个输入参数 x（以弧度为单位）的正弦和余弦。正弦和余弦的结果分别写入第二个参数 sptr 和第三个参数 cptr。

```
__device__ void sincospi (double x, double *sptr, double *cptr)
```

计算第一个输入参数 * Π 的正弦和余弦。

**返回值**

- 无。

**函数描述**

计算第一个输入参数 x（以弧度为单位）* Π 的正弦和余弦。正弦和余弦的结果分别写入第二个参数 sptr 和第三个参数 cptr。

```
__device__ double sinh (double x)
```

计算输入参数的双曲正弦。

**返回值**

- sinh(±0) 返回 ±0。
- sinh(±∞) 返回 ±∞。

**函数描述**

计算输入参数 x 的双曲正弦。

```
__device__ double sinpi (double x)
```

计算输入参数 * Π 的正弦。

**返回值**

- sin(±0) 返回 ±0。
- sin(±∞) 返回 NaN。

**函数描述**

计算输入参数 x * Π 的正弦。

```
__device__ double sqrt (double x)
```

计算输入参数的平方根。

**返回值**

- sqrt(±0) 返回 ±0。
- sqrt(+∞) 返回 +∞。
- 如果 x < 0，sqrt(x) 返回 x。

**函数描述**

计算输入参数 x 的平方根。

```
__device__ double tan (double x)
```

计算输入参数的正切。

**返回值**

- tan(±0) 返回 ±0。
- tan(±∞) 返回 NaN。

**函数描述**

计算输入参数 x 的正切。

```
__device__ double tanh (double x)
```

计算输入参数的双曲正切。

**返回值**

- tanhf(±0) 返回 ±0。

**函数描述**

计算输入参数 x 的双曲正切。

```
__device__ double tgamma (double x)
```

计算输入参数的 gamma 函数。

**返回值**

- tgamma(±0) 返回 ±∞。
- tgamma(2) 返回 +1。
- 如果计算值超出单精度表示范围，tgamma(x) 返回 ±∞。
- 如果 x < 0 且是个整数，tgamma(x) 返回 NaN。
- tgamma(-∞) 返回 NaN。
- tgamma(+∞) 返回 +∞。

**函数描述**

计算输入参数 x 的 gamma 函数。

```
__device__ double trunc (double x)
```

将输入参数截断为整数部分。

**返回值**

- 返回截断的整数部分。

**函数描述**

将 x 四舍五入到最接近的整数值，该整数值的大小不超过 x。

```
__device__ double y0 (double x)
```

为输入参数计算第二类 0 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 0 阶贝塞尔函数的值。
- y0(0) 返回 -∞。
- y0(x) 对于 x < 0 返回 NaN。
- y0(+∞) 返回 +0。
- y0(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 0 阶贝塞尔函数的值，$Y_0(x)$。

```
__device__ double y1 (double x)
```

为输入参数计算第二类 1 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 1 阶贝塞尔函数的值。
- y1(0) 返回 -∞。
- y1(x) 对于 x < 0 返回 NaN。
- y1(+∞) 返回 +0。
- y1(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 1 阶贝塞尔函数的值，$Y_1(x)$。

```
__device__ double yn (int n, double x)
```

为输入参数计算第二类 n 阶贝塞尔函数的值。

**返回值**

- 返回计算的第二类 n 阶贝塞尔函数的值。
- yn(0) 返回 -∞。
- yn(x) 对于 x < 0 返回 NaN。
- yn(+∞) 返回 +0。
- yn(NaN) 返回 NaN。

**函数描述**

计算输入参数 x 的第二类 n 阶贝塞尔函数的值，$Y_n(x)$。

### 1.8.17 整数型度数学函数

```
__device__ int abs (int a)
```

**参数**

a

- int

**返回值**

- int

**函数描述**

- 计算 a 的绝对值。

```
__device__ long int labs (long int a)
```

**参数**

a

- long int

**返回值**

- long int

**函数描述**

- 计算 a 的绝对值。

```
__device__ long long int llabs (long long int a)
```

**参数**

a

- long long int

**返回值**

- long long long int

**函数描述**

- 计算 a 的绝对值。

```
__device__ long long int llmax (const long long int a, const long long int
b)
```

**参数**

a

- long long int

b

- long long int

**返回值**

- long long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ long long int llmin (const long long int a, const long long int
b)
```

**参数**

a

- long long int

b

- long long int

**返回值**

- long long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ unsigned long long int max (const unsigned long long int a,
↪const
long long int b)
```

**参数**

a

- long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned long long int max (const long long int a, const
 ↪unsigned
long long int b)
```

**参数**

a

- long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned long long int max (const unsigned long long int a,
 ↪const
unsigned long long int b)
```

**参数**

a

- unsigned long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ long long int max (const long long int a, const long long int b)
```

**参数**

a

- long long int

b

- long long int

**返回值**

- long long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ unsigned long int max (const unsigned long int a, const long int
b)
```

**参数**

a

- unsigned long int

b

- long int

**返回值**

- unsigned long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned long int max (const long int a, const unsigned long int
b)
```

**参数**

a

- long int

b

- unsigned long int

**返回值**

- unsigned long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned long int max (const unsigned long int a, const unsigned
long int b)
```

**参数**

a

- unsigned long int

b

- unsigned long int

**返回值**

- unsigned long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ long int max (const long int a, const long int b)
```

**参数**

a

- long int

b

- long int

**返回值**

- long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ unsigned int max (const unsigned int a, const int b)
```

**参数**

a

- unsigned int

b

- int

**返回值**

- unsigned int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned int max (const int a, const unsigned int b)
```

**参数**

a

- int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最大值。

```
__device__ unsigned int max (const unsigned int a, const unsigned int b)
```

**参数**

a

- unsigned int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ int max (const int a, const int b)
```

**参数**

a

- int

b

- int

**返回值**

- int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ unsigned long long int min (const unsigned long long int a,
 ↪const
long long int b)
```

**参数**

a

- unsigned long long int

b

- long long int

**返回值**

- unsigned long long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned long long int min (const long long int a, const
 ↪unsigned
long long int b)
```

**参数**

a

- long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned long long int min (const unsigned long long int a,
 ↪const
unsigned long long int b)
```

**参数**

a

- unsigned long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ long long int min (const long long int a, const long long int b)
```

**参数**

a

- long long int

b

- long long int

**返回值**

- long long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ unsigned long int min (const unsigned long int a, const long int
b)
```

**参数**

a

- unsigend long int

b

- long int

**返回值**

- unsigned long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned long int min (const long int a, const unsigned long int
b)
```

**参数**

a

- long int

b

- unsigned long int

**返回值**

- unsigned long int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned long int min (const unsigned long int a, const unsigned
long int b)
```

**参数**

a

- unsigned long int

b

- unsigned long int

**返回值**

- unsigned long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ long int min (const long int a, const long int b)
```

**参数**

a

- long int

b

- long int

**返回值**

- long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ unsigned int min (const unsigned int a, const int b)
```

**参数**

a

- unsigned int

b

- int

**返回值**

- unsigned int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned int min (const int a, const unsigned int b)
```

**参数**

a

- int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 执行整数类型提升后，返回 a 和 b 中的最小值。

```
__device__ unsigned int min (const unsigned int a, const unsigned int b)
```

**参数**

a

- unsigned int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ int min (const int a, const int b)
```

**参数**

a

- int

b

- int

**返回值**

- int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ unsigned long long int ullmax (const unsigned long long int a,
const unsigned long long int b)
```

**参数**

a

- unsigned long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ unsigned long long int ullmin (const unsigned long long int a,
const unsigned long long int b)
```

**参数**

a

- unsigned long long int

b

- unsigned long long int

**返回值**

- unsigned long long int

**函数描述**

- 返回 a 和 b 中的最小值。

```
__device__ unsigned int umax (const unsigned int a, const unsigned int b)
```

**参数**

a

- unsigned int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 返回 a 和 b 中的最大值。

```
__device__ unsigned int umin (const unsigned int a, const unsigned int b)
```

**参数**

a

- unsinged int

b

- unsigned int

**返回值**

- unsigned int

**函数描述**

- 返回 a 和 b 中的最小值。

### 1.8.18 单精度 intrinsic

```
__device__ float __cosf (float x)
```

计算输入参数的快速近似余弦值（`cos(x)`）。

**返回值**

- 返回 `cos(x)` 。

**函数描述**

- 计算输入参数 x 的快速近似余弦值，以弧度为单位。

```
__device__ float __exp10f (float x)
```

计算输入参数的快速近似基数 10 指数。

**返回值**

- 返回 $10^x$。

**函数描述**

- 计算输入参数 x 的快速近似基数 10 指数，$10^x$。

```
__device__ float __expf (float x)
```

计算输入参数的快速近似基数 e 指数。

**返回值**

- 返回 $e^x$。

**函数描述**

- 计算输入参数 x 的快速近似基数 e 的指数，$e^x$。

```
__device__ float __fadd_rd (float x, float y)
```

以向下舍入模式把两个浮点数加起来。

**返回值**

- 返回 `x + y`。

**函数描述**

- 以向下舍入模式把两个浮点数 x，y 加起来。

```
__device__ float __fadd_rn (float x, float y)
```

以向最近偶数舍入模式把两个浮点数加起来。

**返回值**

- 返回 `x + y`。

**函数描述**

- 以最近偶数舍入模式把两个浮点数 x，y 加起来。

```
__device__ float __fadd_ru (float x, float y)
```

以向上舍入模式把两个浮点数加起来。

**返回值**

- 返回 `x + y`。

**函数描述**

- 以向上舍入模式把两个浮点数 x，y 加起来。

```
__device__ float __fadd_rz (float x, float y)
```

以向零舍入模式把两个浮点数加起来。

**返回值**

- 返回 `x + y`。

**函数描述**

- 以向零舍入模式把两个浮点数 x，y 加起来。

```
__device__ float __fdiv_rd (float x, float y)
```

以向下舍入模式计算两个浮点数的商。

**返回值**

- 返回 x / y。

**函数描述**

- 以向下舍入模式计算两个浮点数的商。

```
__device__ float __fdiv_rn (float x, float y)
```

以向最近偶数舍入模式计算两个浮点数的商。

**返回值**

- 返回 x / y。

**函数描述**

- 以向最近偶数舍入模式计算两个浮点数的商。

```
__device__ float __fdiv_ru (float x, float y)
```

以向上舍入模式计算两个浮点数的商。

**返回值**

- 返回 x / y。

**函数描述**

- 以向上舍入模式计算两个浮点数的商。

```
__device__ float __fdiv_rz (float x, float y)
```

以向零舍入模式计算两个浮点数的商。

**返回值**

- 返回 x / y。

**函数描述**

- 以向零舍入模式计算两个浮点数的商。

```
__device__ float __fdividef (float x, float y)
```

计算输入参数的快速近似除法。

**返回值**

- 返回 x / y。
- `__fdividef(∞, y)` 对于 $2^{216} < |y| < 2^{218}$ 返回 NaN。
- `__fdividef(x, y)` 对于 $2^{216} < |y| < 2^{218}$ 和有穷 x 返回 0。

**函数描述**

- 计算输入参数的快速近似除法 x / y。

```
__device__ float __fmaf_ieee_rd (float x, float y, float z)
```

以向下舍入模式计算融合乘法加法运算，忽略-ftz=true 的编译器标志。

**函数描述**

- 行为与 `__fmaf_rd(x, y, z)` 一致。区别在于处理非规范化的输入和输出：-ftz 编译器标志无效。

```
__device__ float __fmaf_ieee_rn (float x, float y, float z)
```

以向最近偶数舍入模式计算融合乘法加法运算，忽略-ftz=true 的编译器标志。

**函数描述**

- 行为与 `__fmaf_rn(x, y, z)` 一致。区别在于处理非规范化的输入和输出：-ftz 编译器标志无效。

```
__device__ float __fmaf_ieee_ru (float x, float y, float z)
```

以向最近偶数舍入模式计算融合乘法加法运算，忽略-ftz=true 的编译器标志。

**函数描述**

- 行为与 `__fmaf_ru(x, y, z)` 一致。区别在于处理非规范化的输入和输出：-ftz 编译器标志无效。

```
__device__ float __fmaf_ieee_rz (float x, float y, float z)
```

以向最近偶数舍入模式计算融合乘法加法运算，忽略-ftz=true 的编译器标志。

**函数描述**

- 行为与 `__fmaf_rz(x, y, z)` 一致。区别在于处理非规范化的输入和输出：-ftz 编译器标志无效。

```
__device__ float __fmaf_rd (float x, float y, float z)
```

以向下舍入模式计算单精度 `x * y + z`。

**返回值**

返回舍入的单精度 `x * y + z` 的结果：

- `fmaf(±∞, ±0, z)` 返回 NaN。
- `fmaf(±0, ±∞, z)` 返回 NaN。
- 如果 x * y 是确切的 +∞，`fmaf(x, y, -∞)` 返回 NaN。
- 如果 x * y 是确切的-∞，`fmaf(x, y, +∞)` 返回 NaN。

**函数描述**

- 以向下舍入模式计算单精度 `x * y + z`。

```
__device__ float __fmaf_rn (float x, float y, float z)
```

以向最近偶数舍入模式计算单精度 `x * y + z`。

**返回值**

返回舍入的单精度 `x * y + z` 的结果：

- `fmaf(±∞, ±0, z)` 返回 NaN。
- `fmaf(±0, ±∞, z)` 返回 NaN。
- 如果 x * y 是确切的 +∞，`fmaf(x, y, -∞)` 返回 NaN。
- 如果 x * y 是确切的-∞，`fmaf(x, y, +∞)` 返回 NaN。

**函数描述**

- 以向最近偶数舍入模式计算单精度 `x * y + z`。

```
__device__ float __fmaf_ru (float x, float y, float z)
```

以向上舍入模式计算单精度 `x * y + z`。

**返回值**

返回舍入的单精度 `x * y + z` 的结果：

- fmaf(±∞, ±0, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的 +∞，fmaf(x, y, -∞) 返回 NaN。
- 如果 x * y 是确切的-∞，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向上舍入模式计算单精度 x * y + z。

```
__device__ float __fmaf_rz (float x, float y, float z)
```

以向零舍入模式计算单精度 x * y + z。

**返回值**

返回舍入的单精度 x * y + z 的结果：

- fmaf(±∞, ±0, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的 +∞，fmaf(x, y, -∞) 返回 NaN。
- 如果 x * y 是确切的-∞，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向零舍入模式计算单精度 x * y + z。

```
__device__ float __fmul_rd (float x, float y, float z)
```

以向下舍入模式计算单精度 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向下舍入模式计算单精度 x * y。

```
__device__ float __fmul_rn (float x, float y, float z)
```

以向最近偶数舍入模式计算单精度 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向最近偶数舍入模式计算单精度 x * y。

```
__device__ float __fmul_ru (float x, float y, float z)
```

以向上舍入模式计算单精度 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向上舍入模式计算单精度 x * y。

```
__device__ float __fmul_rz (float x, float y, float z)
```

以向零舍入模式计算单精度 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向零舍入模式计算单精度 `x * y`。

```
__device__ float __frcp_rd (float x)
```

以向下舍入模式计算 `1 / x`。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向下舍入模式计算 `1 / x`。

```
__device__ float __frcp_rn (float x)
```

以向最近偶数舍入模式计算 `1 / x`。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向最近偶数舍入模式计算 `1 / x`。

```
__device__ float __frcp_ru (float x)
```

以向上舍入模式计算 `1 / x`。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向上舍入模式计算 `1 / x`。

```
__device__ float __frcp_rz (float x)
```

以向零舍入模式计算 `1 / x`。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向零舍入模式计算 `1 / x`。

```
__device__ float __frsqrt_rn (float x)
```

以向最近偶数舍入模式计算 `1 / sqrt(x)`。

**返回值**

- 返回 `1 / sqrt(x)`。

**函数描述**

- 以向最近偶数舍入模式计算 `1 / sqrt(x)`。

```
__device__ float __fsqrt_rd (float x)
```

以向下舍入模式计算 `sqrt(x)`。

**返回值**

- 返回 sqrt(x)。

**函数描述**

- 以向下舍入模式计算 sqrt(x) 。

```
__device__ float __fsqrt_rn (float x)
```

以向最近偶数舍入模式计算 sqrt(x) 。

**返回值**

- 返回 sqrt(x)。

**函数描述**

- 以向最近偶数舍入模式计算 sqrt(x) 。

```
__device__ float __fsqrt_ru (float x)
```

以向上舍入模式计算 sqrt(x) 。

**返回值**

- 返回 sqrt(x) 。

**函数描述**

- 以向上舍入模式计算 sqrt(x) 。

```
__device__ float __fsqrt_rz (float x)
```

以向零舍入模式计算 sqrt(x) 。

**返回值**

- 返回 sqrt(x)。

**函数描述**

- 以向零舍入模式计算 sqrt(x) 。

```
__device__ float __fsub_rd (float x, float y)
```

以向下舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向下舍入模式计算 x - y。

```
__device__ float __fsub_rn (float x, float y)
```

以向最近偶数舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向最近偶数舍入模式计算 x - y。

```
__device__ float __fsub_ru (float x, float y)
```

以向上舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向上舍入模式计算 x - y。

```
__device__ float __fsub_rz (float x, float y)
```

以向零舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向零舍入模式计算 x - y。

```
__device__ float __log10f (float x)
```

计算 $\log_{10}(x)$。

**返回值**

- 返回 $\log_{10}(x)$。

**函数描述**

- 计算 $\log_{10}(x)$。

```
__device__ float __log2f (float x)
```

计算 $\log_2(x)$。

**返回值**

- 返回 $\log_2(x)$。

**函数描述**

- 计算 $\log_2(x)$。

```
__device__ float __logf (float x)
```

计算 $\log_e(x)$。

**返回值**

- 返回 $\log_e(x)$。

**函数描述**

- 计算 $\log_e(x)$。

```
__device__ float __powf (float x, float y)
```

计算 $x^y$。

**返回值**

- 返回 $x^y$。

**函数描述**

- 计算 $x^y$。

```
__device__ float __saturatef (float x)
```

将输入参数映射为 [+0.0, 1.0]。

**返回值**

- 如果 x < 0，__saturatef(x) 返回 0。
- 如果 x > 1，__saturatef(x) 返回 1。
- 如果 0 ⩽ x ⩽ 1，__saturatef(x) 返回 x。
- __saturatef(NaN) 返回 0。

**函数描述**

- 把输入参数映射到区间 [+0.0, 1.0]。

```
__device__ void __sincosf (float x, float *sptr, float *cptr)
```

计算第一个输入参数的正弦和余弦的快速近似值。

**返回值**

- 无。

**函数描述**

- 计算第一个输入参数 x 的正弦和余弦的快速近似值（以弧度为单位）。正弦和余弦的结果分别写入第二个参数 sptr 和第三个参数 cptr。

```
__device__ float __sinf (float x)
```

计算输入参数的正弦的快速近似值。

**返回值**

- x 的正弦值。

**函数描述**

- 以弧度为单位，计算 x 的正弦值。

```
__device__ float __tanf (float x)
```

计算输入参数的正切的快速近似值。

**返回值**

- x 的正切值。

**函数描述**

- 以弧度为单位，计算 x 的正切值。

### 1.8.19 双精度 intrinsic

这一节描述仅在 device 代码中支持的整数型 intrinsic 函数。不需要在程序中包含任何头文件就可以使用这些函数。

```
__device__ double __dadd_rd (double x, double y)
```

以向下舍入模式把两个浮点值加起来。

**返回值**

- 返回 x + y。

**函数描述**

- 以向下舍入模式把两个浮点值 x，y 加起来。

```
__device__ double __dadd_rn (double x, double y)
```

以向最近偶数舍入模式把两个浮点值加起来。

**返回值**

- 返回 x + y。

**函数描述**

- 以向最近偶数舍入模式把两个浮点值 x，y 加起来。

```
__device__ double __dadd_ru (double x, double y)
```

以向上舍入模式把两个浮点值加起来。

**返回值**

- 返回 x + y。

**函数描述**

- 以向上舍入模式把两个浮点值 x，y 加起来。

```
__device__ double __dadd_rz (double x, double y)
```

以向零舍入模式把两个浮点值加起来。

**返回值**

- 返回 x + y。

**函数描述**

- 以向零舍入模式把两个浮点值 x，y 加起来。

```
__device__ double __ddiv_rd (double x, double y)
```

以向下舍入模式计算 x / y。

**返回值**

- 返回 x / y。

**函数描述**

- 以向下舍入模式计算 x / y。

```
__device__ double __ddiv_rn (double x, double y)
```

以向最近偶数舍入模式计算 x / y。

**返回值**

- 返回 x / y。

**函数描述**

- 以向最近偶数舍入模式计算 x / y。

```
__device__ double __ddiv_ru (double x, double y)
```

以向上舍入模式计算 x / y。

**返回值**

- 返回 x / y。

**函数描述**

- 以向上舍入模式计算 x / y。

```
__device__ double __ddiv_rz (double x, double y)
```

以向零舍入模式计算 x / y。

**返回值**

- 返回 x / y。

**函数描述**

- 以向零舍入模式计算 x / y。

```
__device__ double __dmul_rd (double x, double y)
```

以向下舍入模式计算 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向下舍入模式计算 x * y。

```
__device__ double __dmul_rn (double x, double y)
```

以向最近偶数舍入模式计算 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向最近偶数舍入模式计算 x * y。

```
__device__ double __dmul_ru (double x, double y)
```

以向上舍入模式计算 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向上舍入模式计算 x * y。

```
__device__ double __dmul_rz (double x, double y)
```

以向零舍入模式计算 x * y。

**返回值**

- 返回 x * y。

**函数描述**

- 以向零舍入模式计算 x * y。

```
__device__ double __drcp_rd (double x)
```

以向下舍入模式计算 1 / x。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向下舍入模式计算 1 / x。

```
__device__ double __drcp_rn (double x)
```

以向最近偶数舍入模式计算 1 / x。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向最近偶数舍入模式计算 1 / x。

```
__device__ double __drcp_ru (double x)
```

以向上舍入模式计算 1 / x。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向上舍入模式计算 1 / x。

```
__device__ double __drcp_rz (double x)
```

以向零舍入模式计算 1 / x。

**返回值**

- 返回 1 / x。

**函数描述**

- 以向零舍入模式计算 1 / x。

```
__device__ double __dsqrt_rd (double x)
```

以向下舍入模式计算 `sqrt(x)`。

**返回值**

- 返回 `sqrt(x)`。

**函数描述**

- 以向下舍入模式计算 `sqrt(x)`。

```
__device__ double __dsqrt_rn (double x)
```

以向最近偶数舍入模式计算 `sqrt(x)`。

**返回值**

- 返回 `sqrt(x)`。

**函数描述**

- 以向最近偶数舍入模式计算 `sqrt(x)`。

```
__device__ double __dsqrt_ru (double x)
```

以向上舍入模式计算 `sqrt(x)`。

**返回值**

- 返回 `sqrt(x)`。

**函数描述**

- 以向上舍入模式计算 `sqrt(x)`。

```
__device__ double __dsqrt_rz (double x)
```

以向零舍入模式计算 sqrt(x) 。

**返回值**

- 返回 sqrt(x) 。

**函数描述**

- 以向零舍入模式计算 sqrt(x) 。

```
__device__ double __dsub_rd (double x, double y)
```

以向下舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向下舍入模式计算 x - y。

```
__device__ double __dsub_rn (double x, double y)
```

以向最近偶数舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向最近偶数舍入模式计算 x - y。

```
__device__ double __dsub_ru (double x, double y)
```

以向上舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向上舍入模式计算 x - y。

```
__device__ double __dsub_rz (double x, double y)
```

以向零舍入模式计算 x - y。

**返回值**

- 返回 x - y。

**函数描述**

- 以向零舍入模式计算 x - y。

```
__device__ double __fma_rd (double x, double y, double z)
```

以向下舍入模式作为单个操作计算 x * y + z 。

**返回值**

x * y + z 作为单个操作返回的舍入值。

- fmaf(±∞, ±0, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的正无穷，fmaf(x, y, -∞) 返回 NaN。

- 如果 x * y 是确切的负无穷，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向下舍入模式作为单个操作计算 x * y + z。

```
__device__ double __fma_rn (double x, double y, double z)
```

以向最近偶数舍入模式作为单个操作计算 x * y + z。

**返回值**

x * y + z 作为单个操作返回的舍入值。

- fmaf(±∞, ±, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的正无穷，fmaf(x, y, -∞) 返回 NaN。
- 如果 x * y 是确切的负无穷，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向最近偶数舍入模式作为单个操作计算 x * y + z。

```
__device__ double __fma_ru (double x, double y, double z)
```

以向上舍入模式作为单个操作计算 x * y + z。

**返回值**

x * y + z 作为单个操作返回的舍入值。

- fmaf(±∞, ±, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的正无穷，fmaf(x, y, -∞) 返回 NaN。
- 如果 x * y 是确切的负无穷，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向上舍入模式作为单个操作计算 x * y + z。

```
__device__ double __fma_rz (double x, double y, double z)
```

以向零舍入模式作为单个操作计算 x * y + z。

**返回值**

x * y + z 作为单个操作返回的舍入值。

- fmaf(±∞, ±, z) 返回 NaN。
- fmaf(±0, ±∞, z) 返回 NaN。
- 如果 x * y 是确切的正无穷，fmaf(x, y, -∞) 返回 NaN。
- 如果 x * y 是确切的负无穷，fmaf(x, y, +∞) 返回 NaN。

**函数描述**

- 以向零舍入模式作为单个操作计算 x * y + z。

### 1.8.20 整数型 intrinsic

这一节描述仅在 device 代码中支持的整数型 intrinsic 函数。不需要在程序中包含任何头文件就可以使用这些函数。

```
__device__ unsigned int __brev (unsigned int x)
```

把一个无符号整数中的 32 个 bit 反转。

**返回值**

- 返回 bit 反转之后的 x 的值，比如返回值的第 N 个 bit 对应 x 的第 31 - N 个 bit。

**函数描述**

- 把无符号整数 x 中的 32 个 bit 反转。

```
__device__ unsigned long long int __brevll (unsigned long long int x)
```

把一个 64 位无符号整数中的 64 个 bit 顺序反转。

**返回值**

- 返回 bit 反转之后的 x 的值，比如返回值的第 N 个 bit 对应 x 的第 63 - N 个 bit。

**函数描述**

- 把 64 位无符号整数 x 中的 64 个 bit 反转。

```
__device__ unsigned int __byte_perm (unsigned int x, unsigned int y,
↪unsigned int s)
```

从两个 32 位无符号整数中返回选择的字节。

**返回值**

- 返回值这样计算：`result[n] := input[selector[n]]` 这里的 `result[n]` 是 r 的第 n 个字节。

**函数描述**

- `byte_perm(x,y,s)` 返回一个 32 位的整数，这个整数由 8 个输入的字节（整数 x，y）中提供 4 个字节组成，由选择器 s 选择这 4 个字节。

输入的字节这样索引：`input[0] = x<7:0> input[1] = x<15:8> input[2] = x<23:16> input[3] = x<31:24> input[4] = y<7:0> input[5] = y<15:8> input[6] = y<23:16> input[7] = y<31:24>` 。选择器按照下面的方式组成（选择器的高 16-bit 是不会用到的）：`selector[0] = s<2:0> selector[1] = s<6:4> selector[2] = s<10:8> selector[3] = s<14:12>`

```
__device__ int __clz (int x)
```

返回 32 位整数中连续的高位零位数。

**返回值**

- 返回一个介于 0 和 32（包括 0 和 32）之间的值，表示零位的数目。

**函数描述**

- 从 x 的最高有效位（位 31）开始计算最前面连续零位的数量。

```
__device__ int __clzll (long long int x)
```

返回 64 位整数中连续的高位零位数。

**返回值**

- 返回一个介于 0 和 64（包括 0 和 64）之间的值，表示零位的数目。

**函数描述**

- 从 x 的最高有效位（位 63）开始计算最前面连续零位的数量。

```
__device__ int __ffs (int x)
```

查找 32 位整数中设置为 1 的最低有效位的位置。

**返回值**

- 返回一个介于 0 和 32（含 0 和 32）之间的值，表示第一个设置为 1 的位的位置。
- `__ffs(0)` 返回 0。

**函数描述**

- 查找 x 中设置为 1 的第一个最低有效位的位置。

```
__device__ int __ffsll (long long int x)
```

查找 64 位整数中设置为 1 的最低有效位的位置。

**返回值**

- 返回一个介于 0 和 64（含 0 和 64）之间的值，表示第一个设置为 1 的位的位置。
- `__ffsll(0)` 返回 0。

**函数描述**

- 查找 x 中设置为 1 的第一个（最低有效）位的位置。

```
__device__ unsigned int __funnelshift_l (unsigned int lo, unsigned int hi,
unsigned int shift)
```

连接 `hi : lo`，左移 `shift & 31` 位，返回最高的 32 位。

**返回值**

- 返回移位后的 64 位值的最高的 32 位的值。

**函数描述**

- 将参数 lo 和 hi 连接形成的 64 位值向左偏移参数 shift 指定的量。参数 lo 保存 64 位源值的位 31:0，参数 hi 保存 64 位源值的位 63:32。源 64 位值左移 `shift & 31` 位。返回结果的最高有效 32 位。

```
__device__ unsigned int __funnelshift_lc (unsigned int lo, unsigned int hi,
unsigned int shift)
```

连接 `hi : lo`，左移 `min(shift, 32)` 位，返回最高的 32 位。

**返回值**

- 返回移位后的 64 位值的最高的 32 位的值。

**函数描述**

- 将参数 lo 和 hi 串联形成的 64 位值向左偏移参数 shift 指定的量。参数 lo 保存 64 位源值的位 31:0，参数 hi 保存 64 位源值的位 63:32。源 64 位值左移 `min(shift, 32)` 位。返回结果的最高有效 32 位。

```
__device__ unsigned int __funnelshift_r (unsigned int lo, unsigned int hi,
unsigned int shift)
```

连接 `hi : lo`，右移 `shift & 31` 位，返回最低的 32 位。

**返回值**

- 返回移位后的 64 位值的最高的 32 位的值。

**函数描述**

- 将参数 lo 和 hi 连接形成的 64 位值向左偏移参数 shift 指定的量。参数 lo 保存 64 位源值的位 31:0，参数 hi 保存 64 位源值的位 63:32。源 64 位值右移 `shift & 31` 位。返回结果的最低有效 32 位。

```
__device__ unsigned int __funnelshift_rc (unsigned int lo, unsigned int hi,
unsigned int shift)
```

连接 hi : lo，右移 min(shift, 32) 位，返回最低的 32 位。

**返回值**

- 返回移位后的 64 位值的最高的 32 位的值。

**函数描述**

- 将参数 lo 和 hi 连接形成的 64 位值向左偏移参数 shift 指定的量。参数 lo 保存 64 位源值的位 31:0，参数 hi 保存 64 位源值的位 63:32。源 64 位值右移 min(shift, 32) 位。返回结果的最低有效 32 位。

```
__device__ int __hadd (int, int)
```

计算有符号输入参数的平均值，避免在加法中间结果中出现溢出。

**返回值**

- 返回一个表示这两个输入的有符号平均值的有符号整数。

**函数描述**

- 用 (x + y) >> 1 计算有符号整数输入的参数 x 和 y 的平均值，避免在加法中间结果中出现溢出。

```
__device__ int __mul24 (int x, int y)
```

计算两个整数最低有效 24 位乘积的最低有效 32 位。

**返回值**

- 返回乘积 x*y 的最低有效 32 位。

**函数描述**

- 计算 x 和 y 的最低有效 24 位的乘积的最低有效 32 位。忽略 x 和 y 的高阶 8 位。

```
__device__ long long int __mul64hi (long long int x, long long int y)
```

计算两个 64 位整数乘积的最高有效 64 位。

**返回值**

- 返回乘积 x*y 的最高有效 64 位。

**函数描述**

- 计算 128 位乘积 x*y 的最高有效 64 位，其中 x 和 y 是 64 位整数。

```
__device__ int __mulhi (int x, int y)
```

计算两个 32 位整数乘积的最高有效 32 位。

**返回值**

- 返回乘积 x*y 的最高有效 32 位。

**函数描述**

- 计算 64 位乘积 x*y 的最高有效 32 位，其中 x 和 y 是 32 位整数。

```
__device__ int __popc (unsigned int x)
```

计算 32 位整数中设置为 1 的位数。

**返回值**

- 返回一个介于 0 和 32（含 0 和 32）之间的值，表示设置位的数目。

**函数描述**

- 计算 x 中设置为 1 的位数。

```
__device__ int __popcll (unsigned long long int x)
```

计算 64 位整数中设置为 1 的位数。

**返回值**

- 返回一个介于 0 和 64（含 0 和 64）之间的值，表示设置位的数目。

**函数描述**

- 计算 x 中设置为 1 的位数。

```
__device__ int __rhadd (int, int)
```

计算有符号输入参数的四舍五入平均值，避免中间和溢出。

**返回值**

- 返回一个有符号整数值，表示两个输入的有符号四舍五入平均值。

**函数描述**

- 用 `( x + y + 1 ) >> 1` 计算有符号输入参数 x 和 y 的平均值，避免中间和溢出。

```
__device__ unsigned int __sad (int x, int y, unsigned int z)
```

计算 `|x - y| + z`，绝对差之和。

**返回值**

- 返回 `|x - y| + z`。

**函数描述**

- 计算 `|x - y| + z`，一个 32 位的第三个参数与第一个参数 x，第二个参数 y 的绝对值之差的和的整数。输入的 x 与 y 是 32 位有符号整数，z 是一个 32 位的无符号整数。

```
__device__ unsigned int __uhadd (unsigned int, unsigned int)
```

计算无符号输入参数的平均值，避免中间和溢出。

**返回值**

- 返回一个无符号整数值，表示两个输入的无符号平均值。

**函数描述**

- 将用 `(x+y)>>1` 计算无符号输入参数 x 和 y 的平均值，避免中间和溢出。

```
__device__ unsigned int __umul24 (unsigned int x, unsigned int y)
```

计算两个无符号整数的最低有效 24 位乘积的最低有效 32 位。

**返回值**

- 返回乘积 x*y 的最低有效 32 位。

**函数描述**

- 计算 x 和 y 的最低有效 24 位的乘积的最低有效 32 位。忽略 x 和 y 的高阶 8 位。

```
__device__ unsigned long long int __umul64hi (unsigned long long int x,
↪unsigned long long int y)
```

计算两个 64 位无符号整数乘积的最高有效 64 位。

**返回值**

- 返回乘积 x*y 的最高有效 64 位。

**函数描述**

- 计算 128 位乘积 x*y 的最高有效 64 位，其中 x 和 y 是 64 位无符号整数。

```
__device__ unsigned int __umulhi (unsigned int x, unsigned int y)
```

计算两个 32 位无符号整数乘积的最高有效 32 位。

**返回值**

- 返回乘积 x*y 的最高有效 32 位。

**函数描述**

- 计算 64 位乘积 x*y 的最高有效 32 位，其中 x 和 y 是 32 位无符号整数。

```
__device__ unsigned int __urhadd (unsigned int, unsigned int)
```

计算无符号输入参数的四舍五入平均值，避免加法中间结果溢出。

**返回值**

- 返回一个无符号整数值，表示两个输入的无符号舍入平均值。

**函数描述**

- 用 `(x + y) >> 1` 计算有符号整数输入的参数 x 和 y 的平均值，避免加法中间结果溢出。

```
__device__ unsigned int __usad (unsigned int x, unsigned int y, unsigned
↪int z)
```

计算 `|x - y| + z`，绝对差之和。

**返回值**

- 返回 `|x - y| + z`。

**函数描述**

- 计算 `|x - y| + z`，一个 32 位的第三个参数与第一个参数 x，第二个参数 y 的绝对值之差的和的整数。输入的 x，y 与 z 是 32 位的无符号整数。

### 1.8.21 类型转换 intrinsic

这一节描述仅在设备上支持的类型转换 intrinsic 函数。不需要在程序中包含任何头文件就可以使用这些函数。

```
__device__ float __double2float_rd (double x)
```

把一个 double 用向下舍入模式转换成一个 float。

**返回值**

- 返回转换的结果。

**函数描述**

- 把双精度浮点数的值 x 用向下舍入模式（向负无限舍入）转换成一个单精度浮点数的值。

```
__device__ float __double2float_rn (double x)
```

把一个 double 用向最近偶数舍入模式转换成一个 float。

**返回值**

- 返回转换的结果。

**函数描述**

- 把双精度浮点数的值 x 用向最近偶数舍入模式转换成一个单精度浮点数的值。

```
__device__ float __double2float_ru (double x)
```

把一个 double 用向上舍入模式转换成一个 float。

**返回值**

- 返回转换的结果。

**函数描述**

- 把双精度浮点数的值 x 用向上舍入模式转换成一个单精度浮点数的值。

```
__device__ float __double2float_rz (double x)
```

把一个 double 用向零舍入模式转换成一个 float。

**返回值**

- 返回转换的结果。

**函数描述**

- 把双精度浮点数的值 x 用向零舍入模式转换成一个单精度浮点数的值。

```
__device__ int __double2hiint (double x)
```

将 double 中的高 32 位重新解释为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 中的高 32 位重新解释为有符号整数。

```
__device__ int __double2int_rd (double x)
```

把一个 double 用向下舍入模式转换成一个有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向下舍入模式转换成一个有符号整数值。

```
__device__ int __double2int_rn (double x)
```

把一个 double 用向最近偶数舍入模式转换成一个有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向最近偶数舍入模式转换成一个有符号整数值。

```
__device__ int __double2int_ru (double x)
```

把一个 double 用向上舍入模式转换成一个有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向上舍入模式转换成一个有符号整数值。

```
__device__ int __double2int_rz (double x)
```

把一个 double 用向零舍入模式转换成一个有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向零舍入模式转换成一个有符号整数值。

```
__device__ long long int __double2ll_rd (double x)
```

把一个 double 用向下舍入模式转换成一个 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向下舍入模式转换成一个 64 位有符号整数。

```
__device__ long long int __double2ll_rn (double x)
```

把一个 double 用向最近偶数舍入模式转换成一个 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向最近偶数舍入模式转换成一个 64 位有符号整数。

```
__device__ long long int __double2ll_ru (double x)
```

把一个 double 用向上舍入模式转换成一个 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向上舍入模式转换成一个 64 位有符号整数。

```
__device__ long long int __double2ll_rz (double x)
```

把一个 double 用向零舍入模式转换成一个 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向零舍入模式转换成一个 64 位有符号整数。

```
__device__ int __double2loint (double x)
```

将 double 中的低 32 位重新解释为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 中的低 32 位重新解释为有符号整数。

```
__device__ unsigned int __double2uint_rd (double x)
```

把一个 double 用向下舍入模式转换成一个 unsigned int。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向下舍入模式转换成一个无符号整数值。

```
__device__ unsigned int __double2uint_rn (double x)
```

把一个 double 用向最近偶数舍入模式转换成一个无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向最近偶数舍入模式转换成一个无符号整数值。

```
__device__ unsigned int __double2uint_ru (double x)
```

把一个 double 用向上舍入模式转换成一个无符号整数。

**返回值**

- 返回转换的结果

**函数描述**

- 将 double 数据 x 用向上舍入模式转换成一个无符号整数值。

```
__device__ unsigned int __double2uint_rz (double x)
```

把一个 double 用向零舍入模式转换成一个无符号整数值。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向零舍入模式转换成一个无符号整数值。

```
__device__ unsigned long long int __double2ull_rd (double x)
```

把一个 double 用向下舍入模式转换成一个 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向下舍入模式转换成一个 64 位无符号整数。

```
__device__ unsigned long long int __double2ull_rn (double x)
```

把一个 double 用向最近偶数舍入模式转换成一个 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 用向最近偶数舍入模式转换成一个 64 位无符号整数。

```
__device__ unsigned long long int __double2ull_ru (double x)
```

把一个 double 用向上舍入模式转换成一个 64 位无符号整数。

**返回值**

- 返回转换的结果

**函数描述**

- 将 double 数据 x 用向上舍入模式转换成一个 64 位无符号整数值。

```
__device__ unsigned long long int __double2ull_rz (double x)
```

把一个 double 用向零舍入模式转换成一个 64 位无符号整数。

**返回值**

- 返回转换的结果

**函数描述**

- 将 double 数据 x 用向零模式转换成一个 64 位无符号整数值。

```
__device__ long long int __double_as_longlong(double x)
```

将 double 中的位重新解释为 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 中的位重新解释为 64 位有符号整数。

```
__device__ int __float2int_rd(double x)
```

以向下舍入模式将浮点转换为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 以向下舍入模式将浮点转换为有符号整数。

```
__device__ int __float2int_rn(double x)
```

以向最近偶数舍入模式将浮点转换为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 以向最近偶数舍入模式将浮点转换为有符号整数。

```
__device__ int __float2int_ru(double x)
```

以向上舍入模式将浮点转换为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 以向上舍入模式将浮点转换为有符号整数。

```
__device__ int __float2int_rz(double x)
```

以向零舍入模式将浮点转换为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 double 数据 x 以向零舍入模式将浮点转换为有符号整数。

```
__device__ long long int __float2ll_rd (float x)
```

以向下舍入模式将浮点转换为 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向下舍入模式将浮点转换为 64 位有符号整数。

```
__device__ long long int __float2ll_rn (float x)
```

以向最近偶数舍入模式将浮点转换为 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向最近偶数舍入模式将浮点转换为 64 位有符号整数。

```
__device__ long long int __float2ll_ru (float x)
```

以向上舍入模式将浮点转换为 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向上舍入模式将浮点转换为 64 位有符号整数。

```
__device__ long long int __float2ll_rz (float x)
```

以向零舍入模式将浮点转换为 64 位有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向零舍入模式将浮点转换为 64 位有符号整数。

```
__device__ unsigned int __float2uint_rd (float x)
```

以向下舍入模式将浮点转换为无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向下舍入模式将浮点转换为无符号整数。

```
__device__ unsigned int __float2uint_rn (float x)
```

以向最近偶数舍入模式将浮点转换为无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向最近偶数舍入模式将浮点转换为无符号整数。

```
__device__ unsigned int __float2uint_ru (float x)
```

以向上舍入模式将浮点转换为无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向上舍入模式将浮点转换为无符号整数。

```
__device__ unsigned int __float2uint_rz (float x)
```

以向零舍入模式将浮点转换为无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向零舍入模式将浮点转换为无符号整数。

```
__device__ unsigned long long int __float2ull_rd (float x)
```

以向下舍入模式将浮点转换为 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向下舍入模式将浮点转换为 64 位无符号整数。

```
__device__ unsigned long long int __float2ull_rn (float x)
```

以向最近偶数舍入模式将浮点转换为 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向最近偶数舍入模式将浮点转换为 64 位无符号整数。

```
__device__ unsigned long long int __float2ull_ru (float x)
```

以向上舍入模式将浮点转换为 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向上舍入模式将浮点转换为 64 位无符号整数。

```
__device__ unsigned long long int __float2ull_rz (float x)
```

以向零舍入模式将浮点转换为 64 位无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 以向零舍入模式将浮点转换为 64 位无符号整数。

```
__device__ int __float_as_int (float x)
```

将浮点中的位重新解释为有符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 中的位重新解释为有符号整数。

```
__device__ unsigned int __float_as_uint (float x)
```

将浮点中的位重新解释为无符号整数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 float 数据 x 中的位重新解释为无符号整数。

```
__device__ double __hiloint2double (int hi, int lo)
```

将高和低 32 位整数值重新解释为双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 将 hi 的整数值重新解释为双精度浮点值的高 32 位，将 lo 的整数值重新解释为相同双精度浮点值的低 32 位。

```
__device__ double __int2double_rn (int x)
```

把一个有符号整数转换成双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 转换成一个双精度浮点数的值。

```
__device__ float __int2float_rd (int x)
```

把一个有符号整数用向下舍入模式转换成单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 用向下舍入模式转换成一个单精度浮点数的值。

```
__device__ float __int2float_rn (int x)
```

把一个有符号整数用向最近偶数舍入模式转换成单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 用向最近偶数舍入模式转换成一个单精度浮点数的值。

```
__device__ float __int2float_ru (int x)
```

把一个有符号整数用向上舍入模式转换成单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 用向上舍入模式转换成一个单精度浮点数的值。

```
__device__ float __int2float_rz (int x)
```

把一个有符号整数用向零舍入模式转换成单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 用向零舍入模式转换成一个单精度浮点数的值。

```
__device__ float __int_as_float (int x)
```

将整数中的位重新解释为浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把有符号整数 x 中的位重新解释为浮点数。

```
__device__ double __ll2double_rd (long long int x)
```

把一个 64 位有符号整数用向下舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向下舍入模式转换成一个双精度浮点数。

```
__device__ double __ll2double_rn (long long int x)
```

把一个 64 位有符号整数用向最近偶数舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向最近偶数舍入模式转换成一个双精度浮点数。

```
__device__ double __ll2double_ru (long long int x)
```

把一个 64 位有符号整数用向上舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向上舍入模式转换成一个双精度浮点数。

```
__device__ double __ll2double_rz (long long int x)
```

把一个 64 位有符号整数用向零舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向零舍入模式转换成一个双精度浮点数。

```
__device__ float __ll2float_rd (long long int x)
```

把一个 64 位有符号整数用向下舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向下舍入模式转换成一个单精度浮点数。

```
__device__ float __ll2float_rn (long long int x)
```

把一个 64 位有符号整数用向最近偶数舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向最近偶数舍入模式转换成一个单精度浮点数。

```
__device__ float __ll2float_ru (long long int x)
```

把一个 64 位有符号整数用向上舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向上舍入模式转换成一个单精度浮点数。

```
__device__ float __ll2float_rz (long long int x)
```

把一个 64 位有符号整数用向零舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 用向零舍入模式转换成一个单精度浮点数。

```
__device__ double __longlong_as_double (long long int x)
```

将 64 位有符号整数中的位重新解释为双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把 64 位有符号整数 x 中的位重新解释为双精度浮点数。

```
__device__ double __uint2double_rn (unsigned int x)
```

把一个无符号整数转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 转换成一个双精度浮点数。

```
__device__ float __uint2float_rd (unsigned int x)
```

把一个无符号整数用向下舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向下舍入模式转换成一个单精度浮点数。

```
__device__ float __uint2float_rn (unsigned int x)
```

把一个无符号整数用向最近偶数舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向最近偶数舍入模式转换成一个单精度浮点数。

```
__device__ float __uint2float_ru (unsigned int x)
```

把一个无符号整数用向上舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向上舍入模式转换成一个单精度浮点数。

```
__device__ float __uint2float_rz (unsigned int x)
```

把一个无符号整数用向零舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向零舍入模式转换成一个单精度浮点数。

```
__device__ float __uint_as_float (unsigned int x)
```

将无符号整数中的位重新解释为浮点。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 中的位重新解释为浮点。

```
__device__ double __ull2double_rd (unsigned long long int x)
```

把一个 64 位无符号整数用向下舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向下舍入模式转换成一个双精度浮点数。

```
__device__ double __ull2double_rn (unsigned long long int x)
```

把一个 64 位无符号整数用向最近偶数舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向最近偶数舍入模式转换成一个双精度浮点数。

```
__device__ double __ull2double_ru (unsigned long long int x)
```

把一个 64 位无符号整数用向上舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向上舍入模式转换成一个双精度浮点数。

```
__device__ double __ull2double_rz (unsigned long long int x)
```

把一个 64 位无符号整数用向零舍入模式转换成一个双精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向零舍入模式转换成一个双精度浮点数。

```
__device__ float __ull2float_rd (unsigned long long int x)
```

把一个 64 位无符号整数用向下舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向下舍入模式转换成一个单精度浮点数。

```
__device__ float __ull2float_rn (unsigned long long int x)
```

把一个 64 位无符号整数用向最近偶数舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向最近偶数舍入模式转换成一个单精度浮点数。

```
__device__ float __ull2float_ru (unsigned long long int x)
```

把一个 64 位无符号整数用向上舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向上舍入模式转换成一个单精度浮点数。

```
__device__ float __ull2float_rz (unsigned long long int x)
```

把一个 64 位无符号整数用向零舍入模式转换成一个单精度浮点数。

**返回值**

- 返回转换的结果。

**函数描述**

- 把无符号整数 x 用向零舍入模式转换成一个单精度浮点数。

## 1.8.22 SIMD intrinsic

```
__device__ unsigned int __vabs2 (unsigned int a)
```

计算每半字的绝对值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成，然后每个部分计算绝对值。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabs4 (unsigned int a)
```

计算每字节的绝对值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，然后每个部分（字节）计算绝对值。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsdiffs2 (unsigned int a, unsigned int b)
```

计算每半字绝对值差。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算绝对值的差。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsdiffs4 (unsigned int a, unsigned int b)
```

计算每字节的绝对值差。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算绝对值的差。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsdiffu2 (unsigned int a, unsigned int b)
```

计算每半字无符号整数差的绝对值：|a - b|。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算差的绝对值。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsdiffu4 (unsigned int a, unsigned int b)
```

计算每字节的绝对值差。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算差的绝对值。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsss2 (unsigned int a)
```

计算每半字有符号饱和值的绝对值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算有符号饱和值的绝对值。局部结果重新组装成最后的结果，然后以 `unsigned int` 返回计算结果。

```
__device__ unsigned int __vabsss4 (unsigned int a)
```

计算每字节有符号饱和值的绝对值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算有符号饱和值的绝对值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vadd2 (unsigned int a, unsigned int b)
```

计算每半字无符号的和：a + b。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算无符号加法的结果。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vabsss4 (unsigned int a, unsigned int b)
```

计算每字节无符号的和：a + b。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算无符号加法的结果。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vaddss2 (unsigned int a, unsigned int b)
```

计算每半字有符号和的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算有符号和的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vaddss4 (unsigned int a, unsigned int b)
```

计算每字节有符号和的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算有符号和的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vaddus2 (unsigned int a, unsigned int b)
```

计算每半字无符号和的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计算无符号和的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vaddus4 (unsigned int a, unsigned int b)
```

计算每字节无符号和的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计算无符号和的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vavgs2 (unsigned int a, unsigned int b)
```

计算每半字有符号四舍五入平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计有符号平均值，结果四舍五入。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vavgs4 (unsigned int a, unsigned int b)
```

计算每字节有符号四舍五入平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计有符号平均值，结果四舍五入。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vavgu2 (unsigned int a, unsigned int b)
```

计算每半字无符号四舍五入平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，函数计无符号平均值，结果四舍五入。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vavgu4 (unsigned int a, unsigned int b)
```

计算每字节无符号四舍五入平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，函数计无符号平均值，结果四舍五入。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vcmpeq2 (unsigned int a, unsigned int b)
```

计算每半字比较。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果它们相等，局部结果为 0xffff；否则为 0x0000。比如 __vcmpeq2(0x1234aba5, 0x1234aba6) 返回 0xffff0000。

```
__device__ unsigned int __vcmpeq4 (unsigned int a, unsigned int b)
```

计算每字节比较。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果它们相等，局部结果为 0xff；否则为 0x00。比如 __vcmpeq4(0x1234aba5, 0x1234aba6) 返回 0xffffff00。

```
__device__ unsigned int __vcmpges2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较：a >= b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpges2(0x1234aba5, 0x1234aba6) 返回 0xffff0000。

```
__device__ unsigned int __vcmpges4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较：a >= b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xff；否则为 0x00。比如 __vcmpges4(0x1234aba5, 0x1234aba6) 返回 0xffffff00。

```
__device__ unsigned int __vcmpgeu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较：a >= b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpgeu2(0x1234aba5, 0x1234aba6) 返回 0xffff0000。

```
__device__ unsigned int __vcmpgeu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较：a >= b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xff；否则为 0x00。比如 __vcmpgeu4(0x1234aba5, 0x1234aba6) 返回 0xffffff00。

```
__device__ unsigned int __vcmpgts2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较：a > b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpgts2(0x1234aba5, 0x1234aba6) 返回 0x00000000。

```
__device__ unsigned int __vcmpgts4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较：a > b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xff；否则为 0x00。比如 __vcmpgts4(0x1234aba5, 0x1234aba6) 返回 0x00000000。

```
__device__ unsigned int __vcmpgtu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较：a > b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpgtu2(0x1234aba5, 0x1234aba6) 返回 0x00000000。

```
__device__ unsigned int __vcmpgtu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较：a > b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a >= part_b，局部结果为 0xff；否则为 0x00。比如 __vcmpgtu4(0x1234aba5, 0x1234aba6) 返回 0x00000000。

```
__device__ unsigned int __vcmples2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较：a <= b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a <= part_b，局部结果为 0xffff；否则为 0x0000。比如__vcmples2(0x1234aba5, 0x1234aba6) 返回 0xffffffff。

```
__device__ unsigned int __vcmples4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较：a <= b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a <= part_b，局部结果为 0xff；否则为 0x00。比如__vcmples4(0x1234aba5, 0x1234aba6) 返回 0xffffffff。

```
__device__ unsigned int __vcmpleu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较：a <= b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a <= part_b，局部结果为 0xffff；否则为 0x0000。比如__vcmpleu2(0x1234aba5, 0x1234aba6) 返回 0xffffffff。

```
__device__ unsigned int __vcmpleu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较：a <= b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a <= part_b，局部结果为 0xff；否则为 0x00。比如__vcmpleu4(0x1234aba5, 0x1234aba6) 返回 0xffffffff。

```
__device__ unsigned int __vcmplts2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较：a < b ? 0xffff : 0

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a < part_b，局部结果为 0xffff；否则为 0x0000。比如__vcmplts2(0x1234aba5, 0x1234aba6) 返回 0x0000ffff。

```
__device__ unsigned int __vcmplts4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较：a < b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a < part_b，局部结果为 0xff；否则为 0x00 。比如 __vcmplts4(0x1234aba5, 0x1234aba6) 返回 0x000000ff。

```
__device__ unsigned int __vcmpltu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较：a < b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a < part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpltu2(0x1234aba5, 0x1234aba6) 返回 0x0000ffff。

```
__device__ unsigned int __vcmpltu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较：a < b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a < part_b，局部结果为 0xff；否则为 0x00 。比如 __vcmpltu4(0x1234aba5, 0x1234aba6) 返回 0x000000ff。

```
__device__ unsigned int __vcmpne2 (unsigned int a, unsigned int b)
```

计算每半字比较：a != b ? 0xffff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，如果 part_a != part_b，局部结果为 0xffff；否则为 0x0000。比如 __vcmpeq2(0x1234aba5, 0x1234aba6) 返回 0x0000ffff。

```
__device__ unsigned int __vcmpne4 (unsigned int a, unsigned int b)
```

计算每字节比较: a != b ? 0xff : 0。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，如果 part_a != part_b，局部结果为 0xff；否则为 0x00 。比如 __vcmpeq4(0x1234aba5, 0x1234aba6) 返回 0x000000ff。

```
__device__ unsigned int __vhaddu2 (unsigned int a, unsigned int b)
```

计算每半字无符号平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算无符号平均值。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vhaddu4 (unsigned int a, unsigned int b)
```

计算每字节无符号平均值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算无符号平均值。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vmaxs2 (unsigned int a, unsigned int b)
```

计算每半字有符号最大值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算有符号最大值。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vmaxs4 (unsigned int a, unsigned int b)
```

计算每字节有符号最大值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算有符号最大值。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vmaxu2 (unsigned int a, unsigned int b)
```

计算每半字无符号最大值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算无符号最大值。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vmaxu4 (unsigned int a, unsigned int b)
```

计算每字节无符号最大值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算无符号最大值。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vmins2 (unsigned int a, unsigned int b)
```

计算每半字有符号最小值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算有符号最小值。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vmins4 (unsigned int a, unsigned int b)
```

计算每字节有符号最小值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算有符号最小值。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vminu2 (unsigned int a, unsigned int b)
```

计算每半字无符号最小值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算无符号最小值。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vminu4 (unsigned int a, unsigned int b)
```

计算每字节无符号最大值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算无符号最小值。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vneg2 (unsigned int a)
```

计算每半字的相反数。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成，然后每个部分计算相反数。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vneg4 (unsigned int a)
```

计算每字节的相反数。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，然后每个部分（字节）计算相反数。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vnegss2 (unsigned int a)
```

计算每半字有符号相反数的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成，然后每个部分计算有符号相反数的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vnegss4 (unsigned int a)
```

计算每字节有符号相反数的饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，然后每个部分（字节）计算有符号相反数的饱和值。局部结果重新组装成最后的结果，然后以 unsigned int 返回计算结果。

```
__device__ unsigned int __vsads2 (unsigned int a, unsigned int b)
```

计算每半字有符号差值的绝对值的和。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算有符号差值的绝对值，然后求和。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vsads4 (unsigned int a, unsigned int b)
```

计算每字节有符号差值的绝对值的和。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算有符号差值的绝对值，然后求和。局部结果组合成最终结果，并以 unsigned int 返回。

```
__device__ unsigned int __vsadu2 (unsigned int a, unsigned int b)
```

计算每半字无符号差值的绝对值的和。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，计算有符号差值的绝对值，然后求和。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vsadu4 (unsigned int a, unsigned int b)
```

计算每字节无符号差值的绝对值的和。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，计算有符号差值的绝对值，然后求和。局部结果组合成最终结果，并以 `unsigned int` 返回。

```
__device__ unsigned int __vseteq2 (unsigned int a, unsigned int b)
```

计算每半字比较。

**返回值**

- 如果 a = b, 返回 1 否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a == part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vseteq4 (unsigned int a, unsigned int b)
```

计算每字节比较。

**返回值**

- 如果 a = b, 返回 1 否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a == part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetges2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较。

**返回值**

- 如果 a >= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a >= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetges4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较。

**返回值**

- 如果 a >= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a >= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgeu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较。

**返回值**

- 如果 a >= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a >= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgeu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较。

**返回值**

- 如果 a >= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a >= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgts2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较。

**返回值**

- 如果 a > b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a > part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgts4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较。

**返回值**

- 如果 a > b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a > part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgtu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较。

**返回值**

- 如果 a > b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a > part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetgtu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较。

**返回值**

- 如果 a > b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a > part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetles2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较。

**返回值**

- 如果 a <= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a <= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetles4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较。

**返回值**

- 如果 a <= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a <= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetleu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较。

**返回值**

- 如果 a <= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a <= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetleu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较。

**返回值**

- 如果 a <= b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a <= part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetlts2 (unsigned int a, unsigned int b)
```

计算每半字有符号比较。

**返回值**

- 如果 a < b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a < part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetlts4 (unsigned int a, unsigned int b)
```

计算每字节有符号比较。

**返回值**

- 如果 a < b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a < part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetltu2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较。

**返回值**

- 如果 a < b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a < part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetltu4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较。

**返回值**

- 如果 a < b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a < part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetne2 (unsigned int a, unsigned int b)
```

计算每半字无符号比较。

**返回值**

- 如果 a != b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，判断 part_a != part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsetne4 (unsigned int a, unsigned int b)
```

计算每字节无符号比较。

**返回值**

- 如果 a != b, 返回 1；否则返回 0。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，判断 part_a != part_b，如果每个部分都满足，返回 1。

```
__device__ unsigned int __vsub2 (unsigned int a, unsigned int b)
```

计算每半字减法。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 2 个部分，每个部分有 2 个字节组成。对每个对应的部分，做减法操作，局部结果最后组成最终结果并以 `unsigned int` 返回。

```
__device__ unsigned int __vsub4 (unsigned int a, unsigned int b)
```

计算每字节减法。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分离成 4 个部分，每个部分 1 个字节组成。对每个对应的部分，做减法操作，局部结果最后组成最终结果并以 `unsigned int` 返回。

```
__device__ unsigned int __vsubss2 (unsigned int a, unsigned int b)
```

计算每半字的无符号或有符号减法，取无符号整数范围内饱和值。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分成 2 个部分，每部分由 2 个字节组成。对每个对应的部分，做有符号饱和减法，每部分的结果组成最终结果以 `unsigned int` 返回。

```
__device__ unsigned int __vsubss4 (unsigned int a, unsigned int b)
```

计算有符号的每字节饱和减法。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分成 4 个部分，每部分由 1 个字节组成。对每个对应的部分，做有符号饱和减法，每部分的结果组成最终结果以 `unsigned int` 返回。

```
__device__ unsigned int __vsubus2 (unsigned int a, unsigned int b)
```

计算无符号的每半字饱和减法。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分成 2 个部分，每部分由 2 个字节组成。对每个对应的部分，做无符号饱和减法，每部分的结果组成最终结果以 `unsigned int` 返回。

```
__device__ unsigned int __vsubus4 (unsigned int a, unsigned int b)
```

计算无符号的每字节饱和减法。

**返回值**

- 返回计算结果。

**函数描述**

- 把 4 字节参数分成 4 个部分，每部分由 1 个字节组成。对每个对应的部分，做无符号饱和减法，每部分的结果组成最终结果以 `unsigned int` 返回。

## 1.9 Texture 函数

### 1.9.1 Texture Object API

#### 1.9.1.1 tex1Dfetch()

```
template<class T>
T tex1Dfetch(mcTextureObject_t texObj, int x);
```

**参数**

texObj

- mcTextureObject_t，一维 texture object 对象

x

- int，整型 texture 坐标，必须是非归一化整数型值。

**返回值**

- T，T 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型。

**函数描述**

从使用线性空间创建的一维 texture object 中获取整数型 texture 坐标 x 所在位置的元素值。坐标 x 必须是非归一化坐标，且只支持 border 和 clamp 两种寻址模式，不支持 texture filter 操作，对于 8bit 或者 16bit 整数型，可以将输出数据提升成单精度浮点型输出。

#### 1.9.1.2 tex1D()

```
template<class T>
T tex1D(mcTextureObject_t texObj, float x);
```

**参数**

texObj

- mcTextureObject_t，一维 texture object 对象

x

- float, 浮点 texture 坐标，可以是归一化坐标 (-1.0~1.0) 或者 (0~1.0), 也可以是非归一化浮点坐标。

**返回值**

- T，T 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型。

**函数描述**

从使用 MXMACA 数组创建的一维 texture object 中，获取浮点 texture 坐标 x 所在位置的元素值。

## 1.9.2 Texture Reference API

### 1.9.2.1 tex1Dfetch()

```
template <class DataType>
Type tex1Dfetch(texture<DataType, mcTextureType1D, mcReadModeElementType>
↪texRef,
int x);
```

**参数**

texRef

- texture<DataType，mcTextureType1D，mcReadModeElementType>，一维 texture reference，DataType 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型。

x

- int，整型 texture 坐标，必须是非归一化整数型值。

**返回值**

- Type，Type 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型，与模板类型 DateType 一致。

```
template <class DataType>
Type tex1Dfetch(texture<DataType, mcTextureType1D,
↪mcReadModeNormalizedFloat> texRef,
int x);
```

**参数**

texRef

- texture<DataType，mcTextureType1D，mcReadModeElementType>，一维 texture reference，DataType 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型。

x

- int，整型 texture 坐标，可以是归一化坐标 (-1.0~1.0) 或者 (0~1.0)，也可以是非归一化浮点坐标。

**返回值**

- Type，Type 的可取范围为 float 及其 int1，int2，int4 向量类型。

**函数描述**

从使用线性空间创建的一维 texture reference 中，获取整数型 texture 坐标 x 所在位置的元素值。坐标 x 必须是非归一化坐标，且只支持 border 和 clamp 两种寻址模式，不支持 texture filter 操作，对于 8bit 或者 16bit 整数型，可以将输出数据提升成单精度浮点型输出。

### 1.9.2.2 tex1D()

```
template<class DataType > Type
tex1D(texture<DataType, mcTextureType1D, mcReadModeElementType> texRef,
↪float x)
```

**参数**

texRef

- texture<DataType, mcTextureType1D, mcReadModeElementType>，一维 texture reference，DataType 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型。

x

- float，浮点 texture 坐标

**返回值**

- Type，Type 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型，与模板类型 DateType 一致。

```
template<class DataType > Type
tex1D(texture<DataType, mcTextureType1D, mcReadModeNormalizedFloat> texRef,
↪ float x)
```

**参数**

texRef

- texture<DataType, mcTextureType1D, mcReadModeElementType>，一维 texture reference，DataType 的可取范围为 char, unsigned char, short, unsigned short, int, unsigned int, float 及其 int1，int2，int4 向量类型

x

- float，浮点 texture 坐标

**返回值**

- Type，Type 的可取范围为 float 及其 int1，int2，int4 向量类型。

**函数描述**

从使用 MXMACA 数组创建的一维 texture reference 中，获取浮点 texture 坐标 x 所在位置的元素值。

## 1.10 只读数据缓存区加载函数

```
T __ldg(const T* address);
```

函数返回数据类型 T 在地址 address 的数据，这里的数据类型 T 可以是 char、signed char、short、int、long、long long unsigned char、unsigned short、unsigned int、unsigned long、unsigned long long、char2、char4、short2、short4、int2、int4、longlong2 uchar2、uchar4、ushort2、ushort4、uint2、uint4、ulonglong2 float、float2、float4、double 或者 double2。当包含头文件 maca_fp16.h 时，数据类型 T 也可以是 __half 或者 __half2。同样的，当包含头文件 maca_bfloat16.h 时，数据类型 T 也可以是 maca_bfloat16 或者 maca_bfloat162。该操作的数据被缓存在只读数据缓存区域。

## 1.11 使用缓存提示的加载函数

```
T __ldcg(const T* address);
T __ldca(const T* address);
T __ldcs(const T* address);
T __ldlu(const T* address);
T __ldcv(const T* address);
```

这些函数返回数据类型 T 在地址 address 的数据，这里的数据类型 T 可以是 char、signed char、short、int、long、long long unsigned char、unsigned short、unsigned int、unsigned long、unsigned long long、char2、char4、short2、short4、int2、int4、longlong2 uchar2、uchar4、ushort2、ushort4、uint2、uint4、ulonglong2 float、float2、float4、double 或者 double2。当包含头文件 maca_fp16.h

时，数据类型 T 也可以是 __half 或者 __half2。同样的，当包含头文件 maca_bfloat16.h 时，数据类型 T 也可以是 maca_bfloat16 或者 maca_bfloat162。这些操作使用各自对应的缓存方式。

## 1.12 使用缓存提示的存储函数

```
void __stwb(T* address, T value);
void __stcg(T* address, T value);
void __stcs(T* address, T value);
void __stwt(T* address, T value);
```

这些函数在地址 address 上存储数据类型为 T 的数据 value，这里的数据类型 T 可以是 char、signed char、short、int、long、long long、unsigned char、unsigned short、unsigned int、unsigned long、unsigned long long、char2、char4、short2、short4、int2、int4、longlong2、uchar2、uchar4、ushort2、ushort4、uint2、uint4、ulonglong2、float、float2、float4、double 或者 double2。当包含头文件 maca_fp16.h 时，数据类型 T 也可以是 __half 或者 __half2。同样的，当包含头文件 maca_bfloat16.h 时，数据类型 T 也可以是 maca_bfloat16 或者 maca_bfloat162。这些操作使用各自对应的缓存方式。

## 1.13 时间函数

```
clock_t clock();
long long int clock64();
```

当在执行 device 代码的时候，返回每个多处理器计数器的值，该值在每个时钟周期递增。在 kernel 开始以及结束的时候对这个计数器取样，取两次样本的差值，并且记录每个线程为每个线程提供设备为完全执行线程所需的时钟周期数的度量的结果，但是不是 device 确切的消耗在执行这些指令上的时钟周期数。前一个数字比后一个数字大，因为线程时间是切片的。

## 1.14 Atomic 函数

Atomic 函数是对 global 或 shared 的内存空间执行“读-改-写”的原子操作。例如，`atomicAdd()` 从 global 或者 shared 的内存空间的某个地址读取一个字，对该字加 1 后将结果写回到相同的地址上。该操作是原子的，它保证该操作可以在不受其他线程干扰的情况下执行（在操作完成之前，没有其他线程可以访问该地址）。Atomic 函数与 memory fence 不同，其不具有同步或者其他约束内存操作顺序的行为（参考Memory Fence 函数 以了解更多）。Atomic 函数仅可被用于设备侧函数中。

Atomic 函数的原子范围：

- system 范围原子：当前程序中所有 MXMACA 线程具备原子性，包括系统中的其他 CPU 和 GPU。这些函数使用“_system”做为后缀，例如 atomicAdd_system。
- device 范围原子：与当前线程处于同一个 device 的所有 MXMACA 线程具备原子性。这些函数没有扩展后缀，例如 atomicAdd。
- block 范围原子：与当前线程处于同一个 block 的所有 MXMACA 线程具备原子性。这些函数使用“_block”做为后缀，例如 atomicAdd_block。

下例中，使用“atomicAdd_system”函数更新了地址 addr 的值：

```
__global__ void mykernel(int *addr) {
    atomicAdd_system(addr, 10);
}
```

**注解:** 大多数的原子操作均可通过“atomicCAS”函数实现。

## 1.14.1 算术函数

### 1.14.1.1 atomicAdd()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address, unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 *address, __half2 val);
__half atomicAdd(__half *address, __half val);
maca_bfloat162 atomicAdd(maca_bfloat162 *address, maca_bfloat162 val);
maca_bfloat16 atomicAdd(maca_bfloat16 *address, maca_bfloat16 val);
```

atomicAdd 函数读取 global 或者 shared 内存（address）的原始数据（old），计算“old + val”后将结果写回 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

### 1.14.1.2 atomicSub()

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address, unsigned int val);
```

atomicSub 函数读取 global 或者 shared 内存（address）的原始数据（old），计算“old - val”后将结果写回 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

### 1.14.1.3 atomicExch()

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address, unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

atomicExch 函数读取 global 或者 shared 内存（address）的原始数据（old），将 val 值写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

### 1.14.1.4 atomicMin()

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address, unsigned int val);
unsigned long long int atomicMin(unsigned long long int* address,
                                 unsigned long long int val);
```

atomicMin 函数读取 global 或者 shared 内存（address）的原始数据（old），比较 old 和 val 后（记较小的值为 min）将 min 值写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.1.5 atomicMax()

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address, unsigned int val);
unsigned long long int atomicMax(unsigned long long int* address,
                                 unsigned long long int val);
```

atomicMax 函数读取 global 或者 shared 内存（address）的原始数据（old），比较 old 和 val 后（记较大的值为 max）将 max 值写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.1.6 atomicInc()

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
```

atomicInc 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“(old >= val) ? 0 : (old + 1)”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.1.7 atomicDec()

```
unsigned int atomicDec(unsigned int* address, unsigned int val);
```

atomicDec 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“(old == 0) || (old >= val) ? 0 : (old - 1)”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.1.8 atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address, unsigned int compare,
↪unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare, unsigned
↪long long int val);
unsigned short int atomicCAS(unsigned short int *address, unsigned short
↪int compare,
                             unsigned short int val);
```

atomicCAS 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“old == compare ? val : old”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

### 1.14.2 位运算函数

#### 1.14.2.1 atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address, unsigned int val);
unsigned long long int atomicAnd(unsigned long long int* address,
                                 unsigned long long int val);
```

atomicAnd 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“old & val”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.2.2 atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address, unsigned int val);
unsigned long long int atomicOr(unsigned long long int* address,
                                unsigned long long int val);
```

atomicOr 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“old | val”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

#### 1.14.2.3 atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address, unsigned int val);
unsigned long long int atomicXor(unsigned long long int* address,
                                 unsigned long long int val);
```

atomicXor 函数读取 global 或者 shared 内存（address）的原始数据（old），计算算式“old ^ val”后将结果写入 address 地址中，并返回 old 数据。该函数保证对应的（system，device，block 中的一种）原子性。

## 1.15 地址空间谓词函数

在这一节中描述的函数当参数是一个空指针的时候行为未定义。

### 1.15.1 __isGlobal()

```
__device__ unsigned int __isGlobal(const void *ptr);
```

如果 ptr 包含 global 内存空间中对象的通用地址，则返回 1；否则返回 0。

### 1.15.2 __isShared()

```
__device__ unsigned int __isShared(const void *ptr);
```

如果 ptr 包含 shared 内存空间中对象的通用地址，则返回 1；否则返回 0。

### 1.15.3 __isLocal()

```
__device__ unsigned int __isLocal(const void *ptr);
```

如果 ptr 包含 local 内存空间中对象的通用地址，则返回 1；否则返回 0。

## 1.16 地址空间转换函数

### 1.16.1 __cvta_generic_to_global()

```
__device__ size_t __cvta_generic_to_global(const void *ptr);
```

返回对通用地址指针 ptr 转换到 global 内存地址空间的地址的值。

### 1.16.2 __cvta_generic_to_shared()

```
__device__ size_t __cvta_generic_to_shared(const void *ptr);
```

返回对通用地址指针 ptr 转换到 shared 内存地址空间的地址的值。

### 1.16.3 __cvta_generic_to_constant()

```
__device__ size_t __cvta_generic_to_constant(const void *ptr);
```

返回对通用地址指针 ptr 转换到 constant 内存地址空间的地址的值。

### 1.16.4 __cvta_generic_to_local()

```
__device__ size_t __cvta_generic_to_local(const void *ptr);
```

返回对通用地址指针 ptr 转换到 local 内存地址空间的地址的值。

### 1.16.5 __cvta_global_to_generic()

```
__device__ size_t __cvta_global_to_generic(size_t rawbits);
```

返回对值 rawbits 转换所得的 global 内存的通用地址指针。

### 1.16.6 __cvta_shared_to_generic()

```
__device__ size_t __cvta_shared_to_generic(size_t rawbits);
```

返回对值 rawbits 转换所得的 shared 内存的通用地址指针。

### 1.16.7 __cvta_constant_to_generic()

```
__device__ size_t __cvta_constant_to_generic(size_t rawbits);
```

返回对值 rawbits 转换所得的 constant 内存的通用地址指针。

### 1.16.8 __cvta_local_to_generic()

```
__device__ size_t __cvta_local_to_generic(size_t rawbits);
```

返回对值 rawbits 转换所得的 local 内存通用地址指针。

## 1.17 Alloca 函数

### 1.17.1 alloca()

```
__host__ __device__ void * alloca(size_t size);
```

alloca 函数的作用是在调用者的堆栈帧中分配 size 字节的内存，返回一个指向已分配内存的指针。当从 device 代码调用该函数时，内存的开头对齐为 16 字节。当返回 alloca() 的调用者时，分配的内存会自动释放。

例子：

```
__device__ void foo(unsigned int num) {
        int *ptr = (int *)alloca(num * sizeof(int));
        // use of ptr
        ...
}
```

## 1.18 编译器优化提示函数

### 1.18.1 __builtin_assume_aligned()

```
void * __builtin_assume_aligned (const void *exp, size_t align)
```

让编译器假设参数指针至少是按照 align 字节对齐的，并且返回参数指针。

例子：

```
void *res = __builtin_assume_aligned(ptr, 32);  // 编译器可以假设 res 是按照最少 32 字节对齐
```

三个参数的版本：

```
void * __builtin_assume_aligned (const void *exp, size_t align,
<integral type> offset)
```

让编译器假设 (char *)exp - offset 至少是按照 align 字节对齐的，并且返回参数指针。

例子：

```
void *res = __builtin_assume_aligned(ptr, 32, 8); // 编译器可以假设"(char
↪*)res - 8"是按照最少 32 字节对齐的
```

### 1.18.2 __builtin_assume()

```
void __builtin_assume(bool exp)
```

让编译器假设这个布尔参数值是 true。如果参数的值在运行时不是 true，那么此行为不明确。

例子：

```
__device__ int get(int *ptr, int idx) {
    __builtin_assume(idx <= 2);
    return ptr[idx];
}
```

### 1.18.3 __assume()

```
void __assume(bool exp)
```

让编译器假设这个布尔参数值是 true，如果在运行时这个参数的值不是 true，那么该行为不明确。

例子：

```
__device__ int get(int *ptr, int idx) {
    __assume(idx <= 2);
    return ptr[idx];
}
```

### 1.18.4 __builtin_expect()

```
long __builtin_expect (long exp, long c)
```

指示编译器期望为 `exp == c`，并且返回 exp 的值。通常用于向编译器指示分支预测信息。

例子：

```
// 指示编译器"var == 0" 有很大可能性是对的，因此这个 if 代码块的 body 在运行时很可能会执行
if (__builtin_expect (var, 0))
    doit ();
```

### 1.18.5 __builtin_unreachable()

```
void __builtin_unreachable(void)
```

指示编译器控制流从未到达调用此函数的位置。如果控制流在运行时确实达到此点，则程序行为不明确。

例子：

```
// 指示编译器 default 分支永远不会到达。
switch (in) {
case 1: return 4;
case 2: return 10;
default: __builtin_unreachable();
}
```

### 1.18.6 限制

__assume() 函数只在 host 编译器使用 c1.exe 时才支持，其他的几个函数都是全平台支持，受以下限制：

- 如果 host 编译器支持该函数，则可以从翻译单元中的任何位置调用该函数。
- 否则，必须从 __device__ / __global__ 函数体中调用该函数，或者仅在定义了 __MACA_ARCH__ 宏时调用该函数。

## 1.19 Warp Vote 函数

```
int __all_sync(unsigned mask, int predicate);

int __any_sync(unsigned mask, int predicate);

unsigned __ballot_sync(unsigned mask, int predicate);

unsigned long __activemask();
```

warp vote 函数对一个 warp 里指定 (unmasked) 的 threads 执行一个 reduction-and-broadcast 操作。这些函数从该 warp 中的每一个 thread 都获取整数 predicate 作为输入，并将该值与 0 比较。比较的结果在 warp 的活动线程中结合（reduced），最后根据以下指定方法将返回值广播至参与的线程：

**__all_sync(unsigned mask, predicate)**

对于 mask 指定的所有的未退出的线程的 predicate 值评估，当且仅当所有的线程的 predicate 都是非零的才返回非零。

**__any_sync(unsigned mask, predicate)**

对于 mask 指定的所有的未退出的线程的 predicate 值评估，当且仅当所有的线程的 predicate 有一个非零的才返回非零。

**__ballot_sync(unsigned mask, predicate)**

对于 mask 指定的所有的未退出的线程的 predicate 值评估，返回一个第 N 个 bit 为 1 当且仅当 warp 中的第 N 个线程是 active 线程且它的 predicate 非零。

**__activemask()**

返回一个 64 位的当前 warp 中所有的 active 线程的 mask 的整数。当且仅当在调用 __activemask() 时 warp 中第 N 个线程是 actived，则将第 N 个 bit 位置 1；否则置 0。已经退出程序的线程总是会被标记为 inactive。注意 warp 线程同步于 __activemask() 调用处，但不保证后续指令的同步性，除非后续指令为 warp 同步函数。

约束：参数“mask”中设置为 1 的线程必须为 active 线程。

## 1.20 Warp Match 函数

__match_any_sync 和 __match_all_sync 实现了在 warp 内线程之间的变量广播比较操作。

### 1.20.1 提要

```
unsigned long __match_any_sync(unsigned long mask, T value);
unsigned long __match_all_sync(unsigned long mask, T value, int *pred);
```

数据类型 T 可以是 int、unsigned int、long、unsigned long、long long、unsigned long long、float 或者 double。

### 1.20.2 描述

`__match_sync()` 接口允许对 value 值在 mask 标记的线程同步后在 warp 中跨线程广播比较。

**__match_any_sync**

返回与 mask 线程中有相同 value 的线程掩码。

**__match_all_sync**

如果与全部 mask 线程具有相同的 value 值返回该 mask；否则返回 0。如果所有 mask 线程具有相同的 value 值将指示变量 pred 设置为 true；否则设置为 false。

这些带有 `_sync` 的 match intrinsic 有一个 mask 参数表明参与这个操作的线程。每个 bit 都对应着一个线程的 laneID，每一个参与的线程对应的 bit 必须设置为 1 来确保它们在硬件执行内部函数之前正确聚合。所有 mask 标识的未退出的线程必须使用相同的 mask 执行相同的 intrinsic，否则结果会是不确定的。

约束：mask 中设置为 1 的线程必须为 active 线程。

## 1.21 Warp Reduce 函数

`__reduce_sync(unsigned mask, T value)` 在 mask 标记的线程同步后对 value 数据执行一个规约操作。对于 {add，min，max} 操作 T 可以是无符号和有符号的，对于 {and，or，xor} 操作 T 仅支持无符号。

### 1.21.1 提要

```
// add/min/max
unsigned int __reduce_add_sync(unsigned long mask, unsigned int value);
unsigned int __reduce_min_sync(unsigned long mask, unsigned int value);
unsigned int __reduce_max_sync(unsigned long mask, unsigned int value);
int __reduce_add_sync(unsigned long mask, int value);
int __reduce_min_sync(unsigned long mask, int value);
int __reduce_max_sync(unsigned long mask, int value);

// and/or/xor
unsigned int __reduce_and_sync(unsigned long mask, unsigned int value);
unsigned int __reduce_or_sync(unsigned long mask, unsigned int value);
unsigned int __reduce_xor_sync(unsigned long mask, unsigned int value);
```

### 1.21.2 描述

**__reduce_add_sync, __reduce_min_sync, __reduce_max_sync**

返回在 mask 标记线程中对 value 执行算数求和、最小或者最大规约操作的结果。

**__reduce_and_sync, __reduce_or_sync, __reduce_xor_sync**

返回在 mask 标记线程中对 value 执行逻辑 AND、OR 或者 XOR 规约操作的结果。

mask 指示了该调用中参与运算的线程。每个 bit 都对应着一个线程的 laneID，每一个参与的线程对应的 bit 必须设置为 1 来确保它们在硬件执行内部函数之前正确聚合。所有 mask 标识的未退出的线程必须使用相同的 mask 执行相同的 intrinsic，否则结果会是不确定的。

约束：mask 中设置为 1 的线程必须为 active 线程。

## 1.22 Warp Shuffle 函数

`__shfl_sync`、`__shfl_up_sync`、`__shfl_down_sync` 和 `__shfl_xor_sync` 实现在 warp 中的线程之间的变量相互交换。

### 1.22.1 提要

T 可以是 int、unsigned int、long、unsigned long、long long、unsigned long long、float、double、half 或者 half2。

### 1.22.2 描述

__shfl_sync() 接口允许不通过 shared 内存实现 warp 中的线程之间的变量交换。warp 中所有 active 线程（mask 指示的线程）同时进行交换，每个线程根据类型移动 4 个或 8 个字节。

warp 中的 thread 被称为 lane，可以有一个在 0~warpSize-1 之间的索引。支持四种源 lane 寻址模式：

**__shfl_sync()**

直接从索引的 lane 拷贝。

**__shfl_up_sync()**

从调用线程的 lane 索引相对减少差值之后的 laneID 对应的 lane 处拷贝。

**__shfl_sync()**

从调用线程的 lane 索引相对增加差值之后的 laneID 对应的 lane 处拷贝。

**__shfl_sync()**

基于自己 laneID 的异或操作获得的 laneID 对应的 lane 处拷贝。

线程只能从另一个 active 的参与 `__shfl_sync()` 命令的线程处读取数据，如果目标线程是 inactive 的，那么检索到的值是未定义的。

所有的 `__shfl_sync()` intrinsic 都有一个可选择添加的参数 `width`，它将会改变这些 intrinsic 的行为。`width` 的值必须是 2 的正整数次幂，如果不是或者比 warpSize 大，结果将是未定义的。

`__shfl_sync()` 返回 `srcLane` 指定 ID 的线程的 `var` 变量的值。如果 width 比 warpSize 小，那么每一个 warp 的子 group 将作为一个独立的实体，拥有一个逻辑的从 0 开始的 laneID 系统。如果 `srcLane` 超过了 [0:width-1]，返回的结果对应的将由 srcLane mod width 指定的线程的 var 的值决定。

`__shfl_up_sync()` 通过 caller 线程的 laneID 减少 `delta` 来计算 srcLaneID。计算出的 srcLaneID 对应的线程的 var 的值将被返回：实际上，var 再 warp 中向上移动了 `delta` 个 lane。如果 width 比 warpSize 小，那么每一个 warp 的子 group 将作为一个独立的实体，拥有一个逻辑的从 0 开始的 laneID 系统。srcLaneID 不会环绕 width 改变，因此低 laneID 的 lane 的值将不会改变。

`__shfl_down_sync()` 通过 caller 线程的 laneID 增加 `delta` 来计算 srcLaneID。计算出的 srcLaneID 对应的线程的 var 的值将被返回：实际上，var 再 warp 中向上移动了 `delta` 个 lane。如果 width 比 warpSize 小，那么每一个 warp 的子 group 将作为一个独立的实体，拥有一个逻辑的从 0 开始的 laneID 系统。srcLaneID 不会环绕 width 改变，因此低 laneID 的 lane 的值将不会改变。

`__shfl_xor_sync()` 通过 caller 线程的 laneID 与 `laneMask` 做与运算来计算 srcLaneID。计算出的 srcLaneID 对应的线程的 var 的值将被返回。如果 width 比 warpSize 小，每组宽度连续的线程能够访问

之前线程组中的元素，但是如果它们想访问之后组的线程，它们自己的 var 的值会被返回。这种模式实现了一种蝴蝶寻址模式，如用于树缩减和广播的模式。

这些 intrinsic 有一个 mask 参数表明参与这个操作的线程。每个 bit 都对应着一个线程的 laneID，每一个参与的线程对应的 bit 必须设置为 1 来确保它们在硬件执行内部函数之前正确聚合。所有 mask 标识的未退出的线程必须使用相同的 mask 执行相同的 intrinsic，否则结果会是不确定的。

约束：mask 中设置为 1 的线程必须为 active 线程。

### 1.22.3 例子

```
#include <stdio.h>
__global__ void bcast(int arg)
{
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)                          // 注意：除了第零个 lane，
        value = arg;                          // 其他线程不会使用这个变量
    value = __shfl_sync(0xffffffffffffffffUL, value, 0); // 同步 warp 中所有线程，并且从 lane 0 处取得“value”
    if (value != arg)
        printf("Thread %d failed.\n", threadIdx.x);
}
int main()
{
    bcast<<<1, 64>>>(1234);
    mcDeviceSynchronize();
    return 0;
}
```

## 1.23 Nanosleep 函数

__nanosleep 是设备侧的以纳秒为单位的休眠函数。

### 1.23.1 提要

```
void __nanosleep(unsigned ns);
```

### 1.23.2 描述

**__nanosleep**

__nanosleep(ns) 支持 thread 以近似纳秒单位的休眠。该单位受运行主频影响，故仅为近似纳秒级为单位，不应用于实时性要求极高的时间度量场景。

### 1.23.3 例子

```
__device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}

__device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

## 1.24 Warp matrix 函数

C++ warp matrix 操作用来求解形如 D=A*B+C 的矩阵乘加问题。这些操作需要 warp 中所有线程共同完成，它们所处的条件分支必须相同，否则执行结果是未定义的。

### 1.24.1 描述

以下所有函数和类型都定义在 wmma 命名空间中。

```
template <typename Use, int m, int n, int k, typename T, typename Layout =
 ↪void>
class fragment;

void load_matrix_sync(fragment<...> &f, const T* p, unsigned ldm);
void load_matrix_sync(fragment<...> &f, const T* p, unsigned ldm, layout_t
 ↪layout);
void store_matrix_sync(T* p, const fragment<...> &f, unsigned ldm, layout_
 ↪t layout);
void fill_fragment(fragment<...> &f, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...>
 ↪ &b,
const fragment<...> &c, bool satf = false);
```

**fragment**

warp 中所有线程共同存储矩阵的数据结构。矩阵分片到内存中的映射方式是不确定的。

定义矩阵分片时必须明确模板中的参数。第一个模板参数指示矩阵分片如何参与矩阵运算。Use 可以接受的类型有：

- matrix_a 将 fragment 用作第一个被乘数 A。
- matrix_b 将 fragment 用作第二个被乘数 B。
- accumulator 将 fragment 用作累加数或者目标结果（C 或 D）。

m、n 和 k 指示了矩阵的 shape。每个矩阵的 shape 取决于它的角色。对于 matrix_a 为 m x k，对于 matrix_b 为 k x n，对于 accumulator 为 m x n。

数据类型 T 对于被乘数可以是 double、float、__half、maca_bfloat16、signed char 或者 unsigned char，对于累加数可以是 double float int 和 __half。必须为 matrix_a 和 matrix_b 矩阵分片指定 Layout 模板参数。row_major 或 col_major 分别表示矩阵的行或列的元素在内存中是

连续的。accumulator 矩阵的 Layout 参数应保留默认值 void 。当加载或者存储累加数的时候再指定行或列布局。

**load_matrix_sync**

warp 中的所有线程都执行到 load_matrix_sync 时从内存中加载矩阵。p 指向矩阵在内存中的第一个元素。ldm 描述了连续行（行主序）或列（列主序）之间的元素个数。如果矩阵分片是 accumulator，layout 参数必须是 mem_row_major 或者 mem_col_major 。对于 matrix_a 和 matrix_b 的矩阵分片，内存布局从分片的 Layout 参数中推断。p、ldm、layout 以及 f 的所有模板参数在同一 warp 的所有线程中必须相同。

**store_matrix_sync**

warp 中的所有线程都执行到 store_matrix_sync 时将矩阵存储到内存中。p 指向矩阵在内存中的第一个元素。ldm 描述了连续行（行主序）或列（列主序）之间的元素个数。如果矩阵分片是 accumulator，layout 参数必须是 mem_row_major 或者 mem_col_major 。p、ldm、layout 以及 f 的所有模板参数在同一 warp 的所有线程中必须相同。

**fill_fragment**

使用变量 v 的值填充矩阵分片。因为分片到内存中的映射方式是不确定的，需要 warp 中的所有线程调用该函数并提供相同的参数值。

**mma_sync**

warp 中的所有线程都执行到 mma_sync 时执行矩阵乘加运算 D=A*B+C 。支持 C=A*B+C 的原地操作。warp 中的所有线程的每个矩阵分片的模板参数和 satf 值必须相同。此外，模板参数 m、n 和 k 必须在分片 A、B、C 和 D 之间匹配。该函数必须被 warp 中的所有线程调用，否则行为是未定义的。

如果 satf （saturate to finite value）是 true，则为目标结果附加以下数值属性：

- 如果一个元素结果为正无穷大，则调整为 +MAX_NORM。
- 如果一个元素结果为负无穷大，则调整为 -MAX_NORM。
- 如果一个元素结果为 NaN，则调整为 0。

由于矩阵分片到内存中的映射方式是不确定的，因此必须在调用 store_matrix_sync 后才能从内存中访问单个元素。有一种特殊场景是 warp 中的所有线程对所有分片元素执行相同操作，此时可以使用以下类成员直接访问元素。

```
enum fragment<Use, m, n, k, T, Layout>::num_elements;
T fragment<Use, m, n, k, T, Layout>::x[num_elements];
```

例如，以下代码将 accumulator 矩阵缩放一半。

```
wmma::fragment<wmma::accumulator, 16, 16, 16, float> frag;
float alpha = 0.5f;
/*...*/
for (int t = 0; t < frag.num_elements; t++)
  frag.x[t] *= alpha;
```

### 1.24.2 其他浮点类型

**maca_bfloat16**

该数据格式是另一种 fp16 格式，其范围与 f32 相同，但精度有所降低（7bits）。可以引入 maca_bfloat16.h 头文件然后使用 maca_bfloat16 数据格式。使用 maca_bfloat16 的矩阵分片需要与 float 类型的累加数组合使用。支持的形状和操作与 __half 保持一致。

**tf32**

该类型是一种特殊的浮点格式，范围与 f32 相同，但精度有所降低（>=10bits）。该格式的布局在设备端已定义实现，可以在 WMMA 操作中直接使用这种浮点格式。数据格式之间的转换是在设备端完成的，用户不需要手动进行。

该类型唯一支持的矩阵形状是 16x16x8（m-n-k）。

### 1.24.3 双精度类型

使用该类型对应的函数时必须使用 `double` 类型的 `fragment`。

### 1.24.4 元素类型和矩阵形状

下表是矩阵乘加运算支持的 `matrix_a`、`matrix_b` 和 `accumulator` 的组合：

| Matrix A | Matrix B | Accumulator | Matrix Size (m-n-k) |
|---|---|---|---|
| __half | __half | float | 16x16x16 |
| __half | __half | float | 32x8x16 |
| __half | __half | float | 8x32x16 |
| __half | __half | __half | 16x16x16 |
| __half | __half | __half | 32x8x16 |
| __half | __half | __half | 8x32x16 |
| unsigned char | unsigned char | int | 16x16x16 |
| unsigned char | unsigned char | int | 32x8x16 |
| unsigned char | unsigned char | int | 8x32x16 |
| signed char | signed char | int | 16x16x16 |
| signed char | signed char | int | 32x8x16 |
| signed char | signed char | int | 8x32x16 |

其他浮点类型支持：

| Matrix A | Matrix B | Accumulator | Matrix Size (m-n-k) |
|---|---|---|---|
| maca_bfloat16 | maca_bfloat16 | float | 16x16x16 |
| maca_bfloat16 | maca_bfloat16 | float | 32x8x16 |
| maca_bfloat16 | maca_bfloat16 | float | 8x32x16 |
| precision::tf32 | precision::tf32 | float | 16x16x8 |

双精度类型支持：

| Matrix A | Matrix B | Accumulator | Matrix Size (m-n-k) |
|---|---|---|---|
| double | double | double | 8x8x4 |

### 1.24.5 示例

以下代码实现了在单个 warp 里进行 16x16x16 矩阵乘法。

```
#include <mma.h>

__global__ void wmma_ker(half *a, half *b, float *c) {
  // 声明矩阵分片
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

  // 输出初始化为 0
  wmma::fill_fragment(c_frag, 0.0f);
```

(下页继续)

(续上页)

```
  // 加载输入
  wmma::load_matrix_sync(a_frag, a, 16);
  wmma::load_matrix_sync(b_frag, b, 16);

  // 执行矩阵乘法
  wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

  // 保存输出
  wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

## 1.25 Asynchronous Data Copies

MXMACA 引入 memcpy_asyn API 允许设备隐式管理异步数据拷贝。memcpy_async 特性允许 MXMACA 内核重叠数据移动和计算。

### 1.25.1 memcpy_async API

memcpy_async API 由 maca_async.h 头文件提供。

`cooperative_groups::memcpy_async` 使用 `cooperative_groups::wait` 同步。

拷贝和计算模型通过 shared 内存预存数据。

通常应用这样一种拷贝和计算模型：

- 从 global 内存取数据
- 将数据存储到 shared 内存中
- 在 shared 内存数据上进行运算，最终将结果写回到 global 内存中。

下面的两个小节将描述这个模型在有/无 memcpy_async 的情况下的表达方式：

### 1.25.2 没有 memcpy_async

在没有 memcpy_async 的情况下，拷贝阶段表示如下：`shared[local_idx] = global[global_idx]`。这个 global 到 shared 的内存拷贝将会扩展为从 global 内存读取数据到寄存器，然后从寄存器写数据到 shared 内存。

当这个模型出现在迭代计算中，每一个 thread block 需要在从 global 到 shared 内存的拷贝之后进行同步，来确保在计算流程开始之前所有的到 shared 内存的写入都已经完成。thread block 在计算阶段结束之后也需要再次同步，防止在所有的 thread 完成计算之前在 shared 内存的覆盖写入。这个模型在下面的代码段描述：

```
__global__ void without_memcpy_async(uint32_t*       global_out,
                                     uint32_t const* global_in,
                                     uint32_t        size,
                                     uint32_t        num_batch_per_block)
{
    auto grid  = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
```

(下页继续)

(续上页)

```
    // 输入的 size 应该与 num_batch_per_block * grid.size() 相等
    assert(size == num_batch_per_block * grid.size());

    // block.size() * sizeof(int) 字节
    extern __shared__ uint32_t shared[];

    uint32_t local_idx = block.thread_rank();

    for (uint32_t batch = 0; batch < num_batch_per_block; ++batch) {
        // 计算 global 内存中当前 block 的 batch index
        uint32_t block_batch_idx =
            block.group_index().x * block.size() + grid.size() * batch;

        uint32_t global_idx = block_batch_idx + local_idx;

        shared[local_idx] = global_in[global_idx];

        // 等待所有线程完成
        block.sync();

        // 计算并将结果写回 global 内存
        global_out[global_idx] =
            shared[local_idx] + shared[block.size() - local_idx - 1];

        // 等待使用 shared 内存计算完成
        block.sync();
    }
}
```

### 1.25.3 有 memcpy_async

当具备 memcpy_async 的时候，从 global 内存到 shared 内存的赋值：`shared[local_idx] = global_in[global_idx]`；可以被替换成为 cooperative_groups 里面的异步拷贝操作：

```
cooperative_groups::memcpy_async(group, shared, global_in + batch_idx,
                                 sizeof(int) * block.size());
```

memcpy_async API 从 global 内存起始地址 global_in + batch_idx 拷贝 sizeof(int) * block.size() 字节的数据到 shared 内存。这个操作就像由另外一个线程执行一样，在完成复制之后，它与当前线程通过 wait 同步。在数据复制操作完成之前，修改 global 数据或者读取/写入 shared 数据会引入数据竞争。

```
__global__ void with_memcpy_async(uint32_t*       global_out,
                                  uint32_t const* global_in,
                                  uint32_t        size,
                                  uint32_t        num_batch_per_block)
{
    auto grid  = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();

    // 输入的 size 应该与 num_batch_per_block * grid.size() 相等
    assert(size == num_batch_per_block * grid.size());

    // block.size() * sizeof(int) 字节
    extern __shared__ uint32_t shared[];
```

(下页继续)

(续上页)

```
    uint32_t local_idx = block.thread_rank();

    for (uint32_t batch = 0; batch < num_batch_per_block; ++batch) {
        // 计算 global 内存中当前 block 的 batch index
        uint32_t block_batch_idx =
            block.group_index().x * block.size() + grid.size() * batch;

        uint32_t global_idx = block_batch_idx + local_idx;

        // 整个 thread-group 协作地拷贝整个 batch 到 shared 内存:
        cooperative_groups::memcpy_async(block,
                                         shared,
                                         global_in + block_batch_idx,
                                         sizeof(uint32_t) * block.size());

        // Join 所有线程，等待所有的 copy 完成
        cooperative_groups::wait(block);


        // 计算并将结果写回 global 内存
        global_out[global_idx] =
            shared[local_idx] + shared[block.size() - local_idx - 1];

        // 等待使用 shared 内存计算完成
        block.sync();
    }
}
```

## 1.26 Assert 函数

断言函数，在设备侧函数中调用该函数以阻止致命的错误。

```
void assert(int expression);
```

当 expression 等于 0 时，阻止当前 kernel 执行。并且每个设备侧线程（expression 为 0 时）会按如下格式打印 stderr：

```
<filename>:<line number>:<function>:
block: [blockIdx.x, blockIdx.y, blockIdx.z],
thread: [threadIdx.x, threadIdx.y, threadIdx.z],
Assertion `<expression>` failed.
```

## 1.27 Trap 函数

可在任意设备线程中调用 `__trap()` 以触发 trap 动作。

```
void __trap();
```

终止 kernel 的执行并在 host 侧引发中断。

## 1.28 Breakpoint 函数

在任意设备线程中调用 `__brkpt()` 来停止当前 kernel 的执行。

```
void __brkpt();
```

## 1.29 格式化输出

设备侧的 printf 函数可以对设备侧的信息进行格式化输出。函数原型如下：

```
int printf(const char *format[, arg, ...]);
```

与标准 C 库的 printf 函数类似，设备侧 printf 函数根据输入的 format 参数，将输入的参数格式化的输出到主机端的流。其返回值与标准 C 库中的 printf 一致。

设备侧的 printf 函数会被设备侧的每个活动线程执行，其执行结果以 warp 为单位保序。

### 1.29.1 Format Specifiers

与标准 C 库的 printf 一致，其拥有这样的格式: `%[flags][width][.precision][size]type`

目前支持:

- Flags: #、“空格”、0、+、-
- Width: *、“0-9”
- Precision: “0-9”
- Size: h、l、ll
- Type: “%diouxXfFeEgGaAcspn”

### 1.29.2 限制

Printf 输出的最终格式由主机端决定。这意味着格式字符串必须由主机系统的编译器和 C 库控制。我们将努力确保 MXMACA 的 printf 函数支持的格式说明符的形成来自最常见的主机编译器的通用子集，但确切的行为将取决于主机操作系统。

如格式说明符所述，printf() 将接受有效标志和类型的所有组合，但其无法确定在最终输出格式化的主机系统上组合有效。如果程序发出包含无效组合的格式字符串，输出可能未定义。

### 1.29.3 例子

如下代码：

```
#include <stdio.h>

__global__ void helloMXMACA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
```

(下页继续)

(续上页)

```
    helloMXMACA<<<1, 5>>>(1.2345f);
    return 0;
}
```

将会输出：

```
Hello thread 0, f=1.2345
Hello thread 1, f=1.2345
Hello thread 2, f=1.2345
Hello thread 3, f=1.2345
Hello thread 4, f=1.2345
```

注意每个线程均会执行 printf 命令（这里不考虑有条件分支的场景），因此输出行与 grid 中启动的线程一样多。当 printf 函数作用与条件分支（假设条件变量与 threadIdx.x 相关）中，则各线程可能执行不同的打印。

如下代码：

```
#include <stdio.h>

__global__ void helloMXMACA(float f)
{
    if (threadIdx.x == 0)
        printf("Hello thread %d, f=%f\n", threadIdx.x, f) ;
}

int main()
{
    helloMXMACA<<<1, 5>>>(1.2345f);
    return 0;
}
```

将会输出:

```
Hello thread 0, f=1.2345
```

if() 语句限制了仅有线程 0 将调用 printf，因此只能看到一行输出。

## 1.30 动态 Global 内存操作

```
__device__  void* malloc(size_t size);
```

```
__device__  void free( void * ptr);
```

从 global 内存中动态的申请和释放 buffer。

```
__device__  void* memcpy(void* dest, const  void* src, size_t size);
```

从 `src` 指向的内存地址复制 `size` 字节到 `dest` 指向的内存地址。

```
__device__ void* memset(void* ptr, int value, size_t size);
```

将 `src` 指向的地址开始的 `size` 字节的内存块里面的值设置为 value（解析成 `unsigned char`）

`malloc()` 函数从 device heap 中分配至少 `size` 字节内存并返回一个指向这块分配的内存的指针。如果没有足够的内存则返回 NULL。返回的指针确保是 16 字节边界对齐。

### 1.30.1 堆内存分配

在任何 `malloc()` 或 `free()` 加载到上下文之前，device 的堆内存有一个固定的大小被指定。如果使用 `malloc()` 而没有指定固定的堆大小，一个默认的 8 MB 堆内存会被分配。

下面的 api 函数提供获取、设置堆大小的能力：

- mcDeviceGetLimit(size_t* size, mcLimitMallocHeapSize)
- mcDeviceSetLimit(mcLimitMallocHeapSize, size_t size)

堆大小保证最少 `size` 字节，`mcDeviceGetLimit` 返回当前要求的堆大小。

### 1.30.2 与 host 内存交互的 api

### 1.30.3 例子

#### 1.30.3.1 Per Thread 分配

如下代码：

```
#include <stdlib.h>
#include <stdio.h>
__global__ void mallocTest()
{
    size_t size = 123;
    char* ptr = (char*)malloc(size);
    memset(ptr, 0, size);
    printf("Thread %d got pointer: %p\n", threadIdx.x, ptr);
    free(ptr);
}
int main()
{
    // 设置一个 128 MB 的堆大小. 注意这个动作必须在任何 kernel launch 之前完成
    mcDeviceSetLimit(mcLimitMallocHeapSize, 128*1024*1024);
    mallocTest<<<1, 5>>>();
    mcDeviceSynchronize();
    return 0;
}
```

将会输出：

```
Thread 0 got pointer: 00057020

Thread 1 got pointer: 0005708c

Thread 2 got pointer: 000570f8

Thread 3 got pointer: 00057164

Thread 4 got pointer: 000571d0
```

#### 1.30.3.2 Per Thread block 分配

```
#include <stdlib.h>
__global__ void mallocTest()
{
    __shared__ int* data;
    // block 中的第一个线程执行分配动作并通过 shared 内存将指针与其 block 中所有其他的线程共享，
    // 因此可以轻松地合并访问。每个线程分配 64 字节。
    if (threadIdx.x == 0)
    {
        size_t size = blockDim.x * 64;
        data = (int*)malloc(size);
    }
    __syncthreads();
    // 检查是否失败
    if (data == NULL)
        return;
    // 把线程 ID 写入指针，保证合并
    int* ptr = data;
    for (int i = 0; i < 64; ++i)
        ptr[i * blockDim.x + threadIdx.x] = threadIdx.x;
    // 保证在释放之前所有的线程都完毕
    __syncthreads();
    // 只有一个线程执行释放的动作
    if (threadIdx.x == 0)
        free(data);
}
int main()
{
    mcDeviceSetLimit(mcLimitMallocHeapSize, 128 * 1024 * 1024);
    mallocTest<<<10, 128>>>();
    mcDeviceSynchronize();
    return 0;
}
```

## 1.31 执行配置

任何对于 __global__ 函数的调用必须明确“执行配置”。执行配置定义了将会在 device 上执行的函数的 grid、block 的维度，同时也定义了相关的流。

执行配置通过在函数名与带括号的参数列表之间插入形如 <<<Dg, Db, Ns, S>>> 的表达式来确定，其中：

- Dg 的数据类型为 dim3，该参数确定了 grid 的维度与大小，Dg.x * Dg.y * Dg.z 的值为启动的 block 的数量。
- Db 的数据类型为 dim3，该参数确定了 block 的维度与大小，Db.x * Db.y * Db.z 的值为每个 block 中 thread 的数量。
- Ns 的数据类型为 size_t，该参数指定了除静态分配的 shared 内存之外，本次调用中每个 block 可动态分配的 shared 内存的大小（单位：字节）。这些动态分配的内存可以被任何声明为 external 的数组（__shared__）使用；该参数为可选参数，默认为 0。
- S 的数据类型为 mcStream_t，该参数指定了相关的流对象。该参数为可选参数，默认为 0。

例如，一个 kernel 函数声明为：

```
__global__ void Func(float* parameter);
```

可以使用如下方式调用该 kernel 函数：

```
Func<<< Dg, Db, Ns >>>(parameter);
```

执行配置的参数会在函数参数计算之前被计算。

如果 Dg 或者 Db 超出 device 允许的最大值，则函数调用会失败。当 Ns 超出 device 可用的最大 shared 内存时，将会减少静态分配所需的内存。

## 1.32 Launch Bounds（启动边界）

每个 kernel 使用的寄存器越少，多处理器上的 thread 与 block 就可能存在的越多，这样可以提升性能。

因此，编译器使用启发式方法来最小化寄存器使用量，同时将寄存器溢出和指令计数保持在最小。

应用程序可以通过在 `__global__` 函数中使用 `__launch_bounds__()` 标识符的形式，向编译器提供附加信息来有选择地帮助这些启发式。

```
__global__ void
__launch_bounds__(maxThreadsPerBlock)
MyKernel(...)
{
    ...
}
```

- maxThreadsPerBlock 确定每次程序启动 MyKernel 的时候，每个 block 的最大线程数量。

如果指定了启动边界，编译器首先从它们派生出内核应该使用的寄存器数量的上限，以确保 max-ThreadsPerBlock 线程可以驻留在多处理器上。编译器以以下方式优化寄存器的使用：

如果初始寄存器的使用高于，编译器会进一步降低它，直到它小于或等于，通常以牺牲更多的本地内存或更多的指令数量为代价。

## 1.33 #pragma unroll

编译器在默认情况下会展开具有较小次数的循环。也可以使用 `#pragma unroll` 指令指示编译器展开任意循环。该指令的位置必须在一个循环前面，同时也只作用于这个循环。指令后可以跟随一个可选的整型常数表达式（ICE）。当没有指定 ICE 并且循环次数是常数的时候，编译器将会将循环完全展开。当 ICE 的值为 1 时，编译器将不会展开循环。当 ICE 的值是一个非正整数或者大于 `int` 类型最大值的时候，编译器将忽略该指令。

例子：

```
struct S1_t
{
    static const int value = 4;
};
template <int X, typename T2>
__device__ void foo(int *p1, int *p2)
{
// 没有参数，循环将被完全展开
#pragma unroll
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 2;
// unroll value = 8
#pragma unroll(X + 1)
```

(下页继续)

(续上页)

```
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 4;
// unroll value = 1, 不展开循环
#pragma unroll 1
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 8;
// unroll value = 4
#pragma unroll(T2::value)
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 16;
}
__global__ void bar(int *p1, int *p2)
{
    foo<7, S1_t>(p1, p2);
}
```

## 1.34 关于内联汇编

MXMACA 暂时不支持内嵌汇编，如有需要可联系沐曦客户支持部门协助。

# 2 Cooperative Groups

## 2.1 介绍

Cooperative Groups 是一个 MXMACA 编程模型的扩展，目的是组织线程之间的交互。Cooperative Groups 允许开发者表达交互线程的粒度，帮助他们进行更有效的并行分解。

## 2.2 编程模型概念

Cooperative Groups 编程模型描述 MXMACA thread blocks 内部以及相互之间的同步模式。它为应用提供定义他们自己的线程分组的方法，以及同步它们的接口。还提供新的强制执行特定限制的 launch APIs 来保证同步的功能。这些原语在 MXMACA 中启用新的协作并行模型，包括生产者-消费者并行性，机会并行性，以及整个 Grid 的整体同步。

Cooperative Groups 编程模型由以下元素构成：

- 表示协作线程分组的数据类型；
- 获取 MXMACA launch API 定义的隐式分组的操作（比如 thread blocks）；
- 将现有分组划分为新分组的操作集合；
- 数据移动和操作的算法集合（比如 memcpy_async，reduce，scan）；
- 同步组内所有线程的操作；
- 检查组属性的操作；
- 暴露低级别、特定于组且通常由硬件加速的操作集合。

Cooperative Groups 中的主要概念是对象命名其中一部分的线程集。这种将组表示为一级程序对象的方式改进了软件组合，因为集合函数可以接收表示参与线程组的显式对象。该对象还使程序员的意图明确，从而消除了从而消除了代码脆弱的不合理架构假设、对编译器优化的不良限制以及与 GPU 的更好的兼容性。

要编写高效的代码，最好使用专门的 group（通用会失去很多编译时优化），并通过引用将这些组对象传递给打算以某种写作方式使用这些线程的函数。

要使用 Cooperative Groups，请包含头文件：

```
#include <maca_cooperative_groups.h>
```

并使用 Cooperative Groups 的命名空间：

```
using namespace cooperative_groups;
// 或者使用一个别名防止集体算法污染命名空间
namespace cg = cooperative_groups;
```

代码可以使用 mxcc 正常编译，但是如果你想使用 memcpy_async，reduce 或者 scan 的功能，并且你的 host 侧编译器的默认方言不是 c++11 或更高，你需要在命令行添加–std=c++11。

### 2.2.1 组示例

为了说明组的概念，此示例尝试执行块范围的规约求和，如果没有 cooperative groups，写代码的时候在实现上有隐藏的约束：

```
__device__ int sum(int *x, int n) {
    __syncthreads();
    return total;
}

__global__ void parallel_kernel(float *x) {
    // ...
    // 整个线程块都必须调用 sum
    sum(x, n);
}
```

所有线程块的线程都必须到达 __syncthreads() barrier，但是这个约束对于想使用 sum 的开发者来说是隐藏的。使用 Cooperative Groups，一个更好的实现方式：

```
__device__ int sum(const thread_block& g, int *x, int n) {
    // ...
    g.sync();
    return total;
}

__global__void parallel_kernel(...) {
    // ...
    // 整个线程块都必须调用 sum
    thread_block tb = this_thread_block();
    sum(tb, x, n);
}
```

## 2.3 组类型

### 2.3.1 隐式组

隐式组代表内核的启动配置。无论内核是如何编写的，它始终具有一定数量的 thread、block 和 block 维度、单个 grid 和 grid 维度。另外，如果使用多设备协同启动 API，可以有多个 grid（每个设备一个 grid）。这些组为分解成更细粒度的组提供了一个起点，这些组通常是硬件加速的，并且更专门用于开发人员正在解决的问题。

尽管您可以在代码的任何位置创建隐式组，但这样做很危险。为隐式组创建句柄是一个集体操作–组中的所有线程都必须参与。如果组不是在所有线程都到达的分支中创建的，则可能导致死锁或者数据损坏。因此，建议您预先为隐式组创建一个句柄（尽可能早，在任何分支发生之前）并在整个内核中使用这个句柄。

#### 2.3.1.1 Thread Block Group

大家对于这样的 thread block 定义的 thread group 已经非常了解了。Cooperative Group 扩展引入一个新的数据类型，thread_block，来显示地表示 kernel 中的这一概念。

```
class thread_block;
// 如下方式构造：
thread_block g = this_thread_block();
```

(下页继续)

(续上页)

```
// Public 成员函数:
static void sync(); // 同步组内的线程
static unsigned int thread_rank(); // 在 [0, num_threads) 区间内为调用线程排序
static dim3 group_index(); // 在启动的 grid 中当前 block 的 3-D index
static dim3 thread_index(); // 在启动的 block 中当前 thread 的 3-D index
static dim3 dim_threads(); // 以 units 为单位的启动的 block 的尺寸
static unsigned int num_threads(); // 当前 group 中总的线程数
// 传统成员函数 (别名):
static unsigned int size(); // 当前 group 中总的线程数 (num_threads 的别名)
static dim3 group_dim(); // 启动的 block 的尺寸 (dim_threads 的别名)
```

例子:

```
/// 从 global 内存加载数据到 shared 内存
__global__ void kernel(int *globalInput) {
    __shared__ int x;
    thread_block g = this_thread_block();
    // 选一个 leader
    if (g.thread_rank() == 0) {
        // 一次从 global 中加载所有 thread 需要的数据到 shared
        x = (*globalInput);
    }
    // 在加载数据到 shared 内存完毕之后，如果 block 中所有的线程需要使用这个数据，你需
    g.sync(); // 等价于 __syncthreads();
}
// 注意： group 中所有的线程都必须参与集体操作，否则行为是未定义的。
// thread_block 数据类型是从更加 generic 的类型 thread_group 派生出来的，thread_
↪group
// 可以用来表示更广泛的 group
```

#### 2.3.1.2 Grid Group

这个 group 对象表示在一个单独的 grid 中启动的所有 threads。除了 sync() 之外的接口任何时间都可以调用，但是为了整个 grid 同步，你需要使用 cooperative launch API（mcLaunchCooperativeKernel)。

```
class grid_group;
// 如下构造方式:
grid_group g = this_grid();
// Public 成员函数:
bool is_valid() const; // 返回当前 grid_group 是否可以同步
void sync() const; // 同步当前 group 中所有的线程
static unsigned long long thread_rank(); // 调用线程在 [0, num_threads) 上的排
序
static unsigned long long block_rank(); // 调用 block 在 [0, num_blocks) 上
的排序
static unsigned long long num_threads(); // group 中 threads 的总数目
static unsigned long long num_blocks(); // group 中 block 的总数目
static dim3 dim_blocks(); // 以 units 为单位的启动的 block 的大小
static dim3 block_index(); // 在启动的 grid 中当前 block 的 3-D index
// 传统成员函数 (别名)
static unsigned long long size(); // 当前 group 中总的线程数 (num_threads 的别
名)
static dim3 group_dim(); // 启动的 grid 的尺寸 (dim_blocks 的别名)
```

#### 2.3.1.3 Multi Grid Group

这个 group 对象表示一个 multi-device cooperative launch 里面所有 devices 的所有 threads。与 grid_group 不同，所有的 API 需要你使用合适的启动 API（mcLaunchCooperativeKernelMultiDevice）。

```
class multi_grid_group;
// 如下构造方式
// 内核必须通过 cooperative multi-device API 启动
multi_grid_group g = this_multi_grid();
// Public 成员函数：
bool is_valid() const; // 返回当前 multi_grid_group 是否可以同步
void sync() const; // 同步当前 group 中所有的线程
unsigned long long num_threads(); // group 中 threads 的总数目
unsigned long long thread_rank(); // 调用线程在 [0, num_threads) 上的排序
unsigned int grid_rank(); // 调用线程在 [0, num_grids) 上的排序
// 传统成员函数（别名）
static unsigned long long size(); // 当前 group 中总的线程数 (num_threads 的别
名)
// 描述：所有 devices 不推荐使用 multi_grid_group。
```

### 2.3.2 显式组

#### 2.3.2.1 Thread Block Tile

一个模板化版本的 tiled group, 模板参数用来指定 tile 的 size，这样在编译时知道 size，才有可能实现更优化的执行。

```
template <unsigned int Size, typename ParentT = void>
class thread_block_tile;
// 如下构造方式
template <unsigned int Size, typename ParentT>
_CG_QUALIFIER thread_block_tile<Size, ParentT> tiled_partition(const
↪ParentT &g)
```

Size 必须是 2 的幂，并且不大于 64。

Parent T 是分割出这个子 group 的父类型。它是自动推断的，但是存储这个信息的 void 值将会在 group 句柄中存储，而不是这个类型中。

```
// Public 成员函数
void sync() const; // 同步 group 内的线程
unsigned long long num_threads() const; // group 中线程总数
unsigned long long thread_rank() const; // 调用线程在 [0, num_threads) 上的排
序
unsigned long long meta_group_size() const; // 返回 parent group 分割创建的组
的数量
unsigned long long meta_group_rank() const; // 从父 group 划分的 tiles 集合中
的组的线性秩
// （由 meta_group_size 限定）
T shfl_up(T var, int delta) const; // 参考 Warp Shuffle Functions
T shfl_down(T var, int delta) const; // 参考 Warp Shuffle Functions
T shfl_xor(T var, int delta) const; // 参考 Warp Shuffle Functions
T any(int predicate) const; // 参考 Warp Vote Functions
T all(int predicate) const; //参考 Warp Vote Functions
T ballot(int predicate) const; // 参考 Warp Vote Functions
T match_any(T val) const; // 参考 Warp Match Functions
```

(下页继续)

(续上页)

```
T match_all(T val, int &pred) const; // 参考 Warp Match Functions
// 传统成员函数（别名）
unsigned long long size() const; // 当前 group 中总的线程数 (num_threads 的别名)
```

在 C++11 及更高版本标准机型编译时，shfl, shfl_up, shfl_down 与 shfl_xor 这些函数接收任意类型的对象。意味着只要满足以下条件，就可以 shuffle 非整数型的类型：

- 具有可复制性：比如，is_trivially_copyable<T>::value == true
- sizeof(T) <= 32

例子：

```
// 下面的代码将会创建两组 tiled groups, size 分别是 32 和 4：
// 后者将源代码编码在类型中，而前者将其存储在句柄中
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
thread_block_tile<4, thread_block> tile4 = tiled_partition<4>(block);
```

**注解：** 这里要使用模板化的数据结构 `thread_block_tile`，分组的 size 通过模板参数传递给 `tiled_partition` 而非函数参数。

### Warp-Synchronous 代码模型

开发者可能会写出之前隐式地假定 warp size 的 warp-synchronize 代码，并且根据这个数字编写代码。现在需要显式地指定这个数字。

```
__global__ void cooperative_kernel(...) {
    // 获取默认的“当前 thread block” 分组
    thread_block my_block = this_thread_block();

    // 细分为 32 个 thread 的分组，tiled 子分组
    // tiled 子分组将父分组均匀地划分为相邻的 thread 集合
    // ——在这种情况下，每个分组有一个 warp size 数量的 thread
    auto my_tile = tiled_partition<64>(my_block);

    // 下面的操作将会只被每个 block 的前 64 个 thread 组成的
    // tiled 分组执行
    if (my_tile.meta_group_rank() == 0) {
        // ...
        my_tile.sync();
    }
}
```

### Single thread 分组

表示当前 thread 的分组可以通过 `this_thread` 这个函数获取：

```
thread_block_tile<1> this_thread();
```

下面的 `memcpy_async` API 使用一个 `thread_group`，来从源到目标拷贝一个元素：

```
#include <maca_cooperative_groups.h>
#include <maca_async.h>

cooperative_groups::memcpy_async(cooperative_groups::this_thread(),
dest, src, sizeof(int));
```

#### 2.3.2.2 Coalesced Groups

在 MXMACA 的 SIMT 架构下，在硬件层面，多处理器以一组 64 线程（称作 warps）的方式执行。如果在应用代码中存在数据依赖的条件分支，导致一个 warp 内的 thread 岔道，那么 warp 串行地执行每个分支，同时 disable 掉不在当前执行分支上的线程。在分支上的，保持 active 的线程被称作 coalesced。Cooperative Groups 有能力去发现，并且创建一个包含所有的 coalesced 线程的分组。

通过 `coalesced_threads()` 构造这个分组的句柄是机会主义的。它返回那个调用点当时 active thread 组成的集合，并且不保证哪一个 thread 是被返回的（只要他们是 active 的），或者说它们在整个执行过程中将会保持 coalesced（它们在执行过程中作为一个集体，但是之后也可能再次分岔）。

```
class coalesced_group;
// 通过如下方式构造：
coalesced_group active = coalesced_threads();
// Public 成员函数：
void sync() const; // 同步 group 内的线程
unsigned long long num_threads() const; // group 中线程总数
unsigned long long thread_rank() const; // 调用线程在 [0, num_threads) 上的排
序
unsigned long long meta_group_size() const; // 返回 parent group 分割创建的组
的数量
// 如果这个 group 是通过查询活动线程集创建的，
// 比如 coalesced_threads(), meta_group_size() 值为 1
unsigned long long meta_group_rank() const; // 从父 group 划分的 tiles 集合中
的
//  （由 meta_group_size 限定）
T shfl_up(T var, int delta) const; // 参考 Warp Shuffle Functions
T shfl_down(T var, int delta) const; // 参考 Warp Shuffle Functions
T shfl_xor(T var, int delta) const; // 参考 Warp Shuffle Functions
T any(int predicate) const; // 参考 Warp Vote Functions
T all(int predicate) const; //参考 Warp Vote Functions
T ballot(int predicate) const; // 参考 Warp Vote Functions
T match_any(T val) const; // 参考 Warp Match Functions
T match_all(T val, int &pred) const; // 参考 Warp Match Functions
// 传统成员函数（别名）
unsigned long long size() const; // 当前 group 中总的线程数 (num_threads 的别名
```

在 C++11 及更高版本标准机型编译时，shfl, shfl_up, shfl_down 与 shfl_xor 这些函数接收任意类型的对象。意味着只要满足以下条件，就可以 shuffle 非整数型的类型：

- 具有可复制性：比如，is_trivially_copyable<T>::value == true
- sizeof(T) <= 32

例子：

```
__global__ void kernel(int *gIn) {
    if (threadIdx.x == *gIn) {
        coalesced_group active = coalesced_threads();
        active.sync();
    }
}
```

### Discovery Pattern

开发者通常需要与当前 active 的线程集合共事。没有对存在的线程进行假设，而是开发人员使用碰巧存在的线程。可以通过下面的“聚合 warp 中线程之间的原子增量”的例子看出来：

```
{
    unsigned int writemask = __activemask();
    unsigned int total = __popc(writemask);
    unsigned int prefix = __popc(writemask & __lanemask_lt());
    // 发现 laneID 最小的 active thread
    int elected_lane = __ffs(writemask) - 1;
    int base_offset = 0;
    if (prefix == 0) {
        base_offset = atomicAdd(p, total);
    }
    base_offset = __shfl_sync(writemask, base_offset, elected_lane);
    int thread_offset = prefix + base_offset;
    return thread_offset;
}
```

使用 Cooperative Groups 重新写这个功能如下：

```
{
    using cg = namespace cooperative_groups;
    cg::coalesced_group g = cg::coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.num_threads());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

## 2.4 组分割

### 2.4.1 tiled_partition

```
template <unsigned int Size, typename ParentT>
thread_block_tile<Size, ParentT> tiled_partition(const ParentT& g);

thread_group tiled_partition(const thread_group& parent, unsigned int
↪tilesz);
```

`tiled_partition` 方法是一组分割父组到一个一维的，行优先的，子分组的平铺。将一共创建 `(size(parent) / tilesz)` 个子分组，因此父组的 size 必须是可以被 Size 整除的数字。允许的父组的类型是 `thread_block` 或者 `thread_block_tile`。

实现可能导致调用的 thread 一直等到所有的父组的成员调用了这个操作，之后才恢复执行。功能限制于本地的硬件条件，1/2/4/8/16/32/64，并且 `cooperative_groups::size(parent)` 必须比参数 Size 大。

**例子：**

```
/// 下面的代码将会创建一个 32-thread tile
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
```

我们也可以把这些组分割成更小的组，每个组的 size 是 4：

```
auto tile4 = tiled_partition<4>(tile32);
// 或使用一个通用的组
// thread_group tile4 = tiled_partition(tile32, 4);
```

假设我们要包含下面的代码行：

```
if (tile4.thread_rank() == 0) printf("Hello from tile4 rank 0\n");
```

那么这个表达将会被 block 中每第四个线程打印。

### 2.4.2 labeled_partition

```
coalesced_group labeled_partition(const coalesced_group& g, int label);
template <unsigned int Size>
coalesced_group labeled_partition(const thread_block_tile<Size>& g, int
↪label);
```

`labeled_partition` 方法是一个分割父组到一个一维的子分组（组内线程都是 coalesced）的操作集合。实现将会评估条件 label 然后把有同样 label 值的线程放到相同的组。

实现可能导致调用的 thread 一直等到所有的父组的成员调用了这个操作，之后才恢复执行。

**注解：** 这个功能仍然在测试中以后可能会有轻微改动。

### 2.4.3 binary_partition

```
coalesced_group binary_partition(const coalesced_group& g, bool pred);
template<unsigned int Size>
coalesced_group binary_partition(const thread_block_tile<Size>& g, bool
↪pred);
```

`binary_partition()` 方法是一个分割父组到一个一维的子分组（组内线程都是 coalesced）的操作集合。实现将会评估条件 pred 然后把有同样 pred 值的线程放到相同的组。它是 `labeled_partition()` 的一种特殊形式，label 只能为 0 或 1。

实现可能导致调用的 thread 一直等到所有的父组的成员调用了这个操作，之后才恢复执行。

**注解：** 这个功能仍然在测试中以后可能会有轻微改动。

## 2.5 组操作集合

### 2.5.1 同步

#### 2.5.1.1 sync

```
cooperative_groups::sync(T& group);
```

`sync` 函数同步指定的分组内的线程。T 可以是任何存在的分组类型，它们都支持同步操作。如果这个分组是一个 grid_group 或者一个 multi_grid_group，内核必须使用合适的 cooperative 启动接口来启动。

## 2.5.2 数据移动

### 2.5.2.1 memcpy_async

memcpy_async 是一个组级别的内存拷贝操作集合，它对非阻塞的全局到共享内存的内存交换应用硬件加速支持。对于分组内的特定的线程集合，memcpy_async 会通过单独的一个流水步骤移动指定数量的字节或者输入类型的元素。此外，为了获取最佳的性能表现，需要全局以及共享内存按照 16 字节对齐。只有当拷贝源是全局内存并且拷贝目标是共享内存，并且两块内存编码都必须是 16，8 或者 4 字节对齐的时候这个操作才是异步的，否则它就是一个通常意义上的内存拷贝操作。异步内存拷贝的数据应该只能在 wait 之后读取，表示把数据移动到共享内存的操作阶段已经结束了。

必须等待所有未处理的请求可能会失去一些灵活性（但是会获得简单性）。为了有效地重叠数据传输和执行，能够在等待和操作请求 N 的同时启动 N+1 memcpy_async 请求很重要。

**用法 1**

```
template <typename TyGroup, typename TyElem, typename TyShape>
void memcpy_async(
    const TyGroup &group,
    TyElem *__restrict__ _dst,
    const TyElem *__restrict__ _src,
    const TyShape &shape
);
```

执行 shape 字节数的数据拷贝。

**用法 2**

```
template <typename TyGroup, typename TyElem,
typename TyDstLayout, typename TySrcLayout>
void memcpy_async(
    const TyGroup &group,
    TyElem *__restrict__ dst,
    const TyDstLayout &dstLayout,
    const TyElel *__restrict__ src,
    const TySrcLayout &srcLayout
);
```

执行 min(dstLayout, srcLayout) 个元素的拷贝。

**代码生成要求：** C++11

头文件 maca_async.h 需要被包含。

**例子：**

```
/// 这个例子在 block 上把 elementsPerThreadBlock 个有效数据从全局内存拷贝到
/// 限制大小的共享内存（elementsInShared）。
#include <maca_cooperative_groups.h>
#include <maca_async.h>

namespace cg = cooperative_groups;

__global__ void kernel(int *global_data) {
    cg::thread_block tb = cg::this_thread_block();
    const size_t elementsPerThreadBlock = 16 * 1024;
    const size_t elementsInShared = 128;
    __shared__ int local_smem[elementsInShared];

    size_t copy_count;
```

(下页继续)

(续上页)

```
    size_t index = 0;
    while (index < elementsPerThreadBlock) {
        cg::memcpy_async(tb, local_smem, elementsInShared, global_data +
↪index,
                         elementsPerThreadBlock - index);
        copy_count = min(elementsInShared, elementsPerThreadBlock - index);
        cg::wait(tb);
        // 操作 local_smem
        index += copy_count;
    }
}
```

## 2.6 Grid 同步

在介绍 Cooperative Groups 之前，MXMACA 编程模型只允许在内核完成边界处的 thread block 之间进行同步。内核边界带来了一种隐含的状态无效，以及潜在的性能影响。

举个例子，在具体的应用场景，应用有很多的小内核，每一个内核表示一个流水进程中的一个状态。对于当前的 MXMACA 编程模型这些内核的存在是必要的，以保证运行在一个确定的流水线状态的 thread block 在下一个流水线状态之前输出了下一个流水线状态可能需要的数据。在这样的场景下，提供全局 thread block 同步的能力将会允许应用重构以拥有当给定阶段完成时能在设备上同步的持久 thread block。

想要一个内核的整个 grid 同步，你只需要简单的使用 `grid.sync()` 函数：

```
grid_group grid = this_grid();
grid.sync();
```

而且在启动内核的时候，使用 MXMACA 运行时启动 API `mcLaunchCooperativeKernel` 替代 <<<…>>> 执行配置语法是必要的。

**例子：**

要保证 GPU 上的 thread block 的共同驻留，启动的 block 的数量需要仔细思考。比如像下面的尽可能多的启动 SM 可以启动的 thread block：

```
int device = 0;
mcDeviceProp deviceProp;
mcGetDeviceProperties(&deviceProp, dev);
mcLaunchCooperativeKernel((void*)my_kernel, deviceProp.multiProcessorCount,
numThreads, args);
```

或者，你可以通过使用占用计算器计算每个 SM 可以同时容纳多少 block 来最大限度地提高暴露的并行性，如下所示：

```
/// 下面的代码将会启动一个在默认 stream 上的最大限度占用 GPU 的内核
int numBlocksPerSm = 0;
// my_kernel 启动的线程数量
int numThreads = 128;
mcDeviceProp deviceProp;
mcGetDeviceProperties(&deviceProp, dev);
mcOccupancyMaxActiveBlocksPerMultiProcessor(&numBlocksPerSm,
my_kernel, numThreads, 0);
// 启动
void* kernelArgs[] = { /* 添加内核参数 */ };
dim3 dimBlock(numThreads, 1, 1);
```

(下页继续)

(续上页)

```
dim3 dimGrid(deviceProp.multiProcessorCount * numBlocksPerSm, 1, 1);
mcLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock, kernelArgs);
```

推荐通过查询设备的 mcDevAttrCooperativeLaunch 属性来先确认设备支持 cooperative launch：

```
int dev = 0;
int supportsCoopLaunch = 0;
mcDeviceGetAttribute(&supportsCoopLaunch, mcDevAttrCooperativeLaunch, dev);
```

如果设备 0 的属性显示支持 cooperative launch，supportsCoopLaunch 将会被设为 1。

## 2.7 多设备同步

为了支持使用 Cooperative Groups 进行多设备之间的同步，必须使用 MXMACA 启动接口 mcLaunchCooperativeKernelMultiDevice。这个与现存的 MXMACA 接口明显不同的接口允许一个主机线程在多个设备上启动一个内核。除了 mcLaunchCooperativeKernel 的限制与保证之外，这个接口有其他的语法：

- 这个接口保证启动是原子的，比如：如果这个接口调用成功，提供的线程块的数量将会在所有指定的设备上启动。
- 通过这个接口启动的函数必须是相同的。驱动不会在这一点上有显示的检查，因为是不可行的。这一点需要应用来保证。
- 在提供的 mcLaunchParams 中没有映射到相同设备的两个条目。
- 这个启动的所有目标设备必须有相同的计算能力——主要版本以及次要版本。
- 所有设备上每个 grid 的 block size, grid size 以及共享内存的数量都必须相同。这意味着每个设备上可以启动的 block 数量的最大值是受有最少 SM 数量的设备的限制的。
- 任何用户定义的 __device__，__constant__ 或者 __managed__ 的出现在模块中的设备侧全局变量拥有正在启动的 CU 函数，并在每个设备上独立实例化。用户有责任给这些全局变量合适的初始化。

可以通过启用对等访问（通过对所有参与的设备设置 mcCtxEnablePeerAccess 或者 mcDeviceEnablePeerAccess）。

启动参数应该使用一个结构体的数组定义（一个设备一个结构体对象），并且使用 mcLaunchCooperativeKernelMultiDevice 启动。

**例子：**

```
mcDeviceProp deviceProp;
mcGetDeviceCount(&numGpus);

// 每个设备的启动参数
mcLaunchParams *launchParams =
                (mcLaunchParams*)malloc(sizeof(mcLaunchParams) * numGpus);
mcStream_t *streams = (mcStream_t*)malloc(sizeof(mcStream_t) * numGpus);

// 启动过程中拷贝内核启动参数
// 也可以使用每个设备独立的内核参数拷贝，但是函数/内核的签名以及名字必须相同
void* kernelArgs = { /* 添加内核参数 */ }

for (int i = 0; i < numGpus; i++) {
    mcSetDevice(i);
    // 每个设备的 stream，但是每个设备也可以使用默认的 NULL stream
    mcStreamCreate(&streams[i]);
```

(下页继续)

(续上页)

```
    // 循环其他设备以及设置 mcDeviceEnablePeerAccess 以获取一个更快的 barrier 的实现
}

// 所有设备必须相同的计算能力并且有相同的启动配置
// 这里查询设备 0 是必要的
mcGetDeviceProperties(&deviceProp[i], 0);
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount, 1, 1);
for (int i = 0; i < numGpus; i++) {
    launchParamsList[i].func = (void*)my_kernel;
    launchParamsList[i].gridDim = dimGrid;
    launchParamsList[i].blockDim = dimBlock;
    launchParamsList[i].sharedMem = 0;
    launchParamsList[i].stream = streams[i];
    launchParamsList[i].args = kernelArgs;
}
mcLaunchCooperativeKernelMultiDevice(launchParams, numGpus);
```

此外，就像 grid 级别的同步，设备侧代码看起来也很像：

```
multi_grid_group multi_grid = this_multi_grid();
multi_grid.sync();
```

但是代码需要用独立编译的方式，传递-rdc=true 到 mxcc。

推荐通过查询设备的 `mcDevAttrCooperativeMultiDeviceLaunch` 属性来先确认设备支持多设备 cooperative launch：

```
int dev = 0;
int supportsMdCoopLaunch = 0;
mcDeviceGetAttribute(&supportsMdCoopLaunch,
mcDevAttrCooperativeMultiDeviceLaunch, dev);
```

如果设备 0 的属性显示支持多设备 cooperative launch，`supportsMdCoopLaunch` 将会被设为 1。

# 3 C++ 语言支持

关于 mxcc 支持的 ISO C++ 标准，参见表 3.1。

表 3.1 mxcc 支持的 ISO C++ 标准

| 语言标准 | Flag | 支持程度 |
| --- | --- | --- |
| C++98/C++03 | -std=c++98 | 除了 export 完全支持 |
| C++11 | -std=c++11 | 完全支持 |
| C++14 | -std=c++14 | 完全支持 |
| C++17 | -std=c++17 | 部分支持 |
| C++20 | -std=c++20 | 部分支持 |
| C++2b（暂定 C++23） | -std=c++2b | 部分支持 |

## 3.1 C++98 支持状态

mxcc 支持了除了 export（C++11 中移除）之外的所有的 C++98 标准的特性。

## 3.2 C++11 支持状态

mxcc 支持 C++11 所有的特性，可以使用-std=c++11 的选项应用 C++11。

## 3.3 C++14 支持状态

mxcc 支持 C++14 所有的特性，可以使用-std=c++14 的选项应用 C++14。

## 3.4 C++17 以及之后标准支持状态

mxcc 不完全支持 C++17 及之后 C++ 标准，不建议使用。