曦云® 系列通用计算 GPU

mcCUB API 参考

CSRD-23013-020-F3_V04　　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
V04	2024-03-15	新增曦云®系列 GPU 产品信息
V03	2024-01-25	描述性修改及格式修正
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 介绍

1.1 什么是 CUB

CUB 是一个只包含头文件的 C++ 库，可为 MXMACA（MetaX Advanced Compute Architecture）编程模型的每一层提供最先进的、可重用的软件组件，包括并行原语和许多实用程序。并行原语包括线程束（warp）级别的“集合”原语、块级别的“集合”原语和设备级别的原语。这些实用程序包括复杂的迭代器、线程和线程块 I/O、设备、内核和存储管理。

1.2 安装

安装 MXMACA 工具包将把 CUB 和 Thrust 的头文件复制到系统的标准 MXMACA include 目录中。

```
# header location
${MACA_PATH}/include/cub
${MACA_PATH}/include/thrust
```

不需要单独构建 CUB。要在代码中使用 CUB 原语，只需：

- 获取最新的 CUB 发行版
- 在 MXMACA C++ 源代码中，通过 #include 指令包含 `cub/cub.cuh` 这个“伞”（master）头文件。（或者，通过 #include 指令包含特定的头文件，用来定义需要的 CUB 原语。）
- 使用 MXMACA 的 mxcc 编译器编译您的程序，指定一个 `-I<path-to-CUB> include-path` 标志，以引用 CUB 和 Thrust 头文件库的位置。

1.3 在应用程序中使用 CUB

以下的代码段说明了一个 MXMACA 内核，其中每块 BLOCK_THREADS 个线程将共同加载、排序和存储自己（BLOCK_THREADS * ITEMS_PER_THREAD）个整数键：

```
// myCubApp1.cu
#include <mc_runtime.h>
#include <cub/cub.cuh>
//
// 块排序 MXMACA 核函数
//
template <int BLOCK_THREADS, int ITEMS_PER_THREAD>
__global__ void BlockSortKernel(int *d_in, int *d_out)
{
    // 专门定义 BlockLoad、BlockStore 和 BlockRadixSort 集合类型
    typedef cub::BlockLoad<
```

(下页继续)

(续上页)

```
        int, BLOCK_THREADS, ITEMS_PER_THREAD, cub::BLOCK_LOAD_TRANSPOSE>
↪BlockLoadT;
    typedef cub::BlockStore<
        int, BLOCK_THREADS, ITEMS_PER_THREAD, cub::BLOCK_STORE_TRANSPOSE>
↪BlockStoreT;
    typedef cub::BlockRadixSort<
        int, BLOCK_THREADS, ITEMS_PER_THREAD> BlockRadixSortT;
    // 为集合操作分配类型安全、可重复使用的共享内存
    __shared__ union {
        typename BlockLoadT::TempStorage       load;
        typename BlockStoreT::TempStorage      store;
        typename BlockRadixSortT::TempStorage  sort;
    } temp_storage;
    // 获取此块在连续键中的段（按线程分块）
    int thread_keys[ITEMS_PER_THREAD];
    int block_offset = blockIdx.x * (BLOCK_THREADS * ITEMS_PER_THREAD);
    BlockLoadT(temp_storage.load).Load(d_in + block_offset, thread_keys);

    __syncthreads();     // 共享内存重用的障碍
    // 多个线程协同对键进行排序
    BlockRadixSortT(temp_storage.sort).Sort(thread_keys);
    __syncthreads();     // 共享内存重用的障碍
    // 存储已排序的段
    BlockStoreT(temp_storage.store).Store(d_out + block_offset, thread_
↪keys);
}

int main()
{
    // 在主程序的其他位置：对块排序操作进行参数化设置并启动该操作
    // 内核中的每个块由 128 个线程组成，每个块对 256 个键的段进行排序
    constexpr int TOTAL_ITEMS = 256;
    constexpr int NUM_BLOCKS = 1;
    constexpr int BLOCK_THREADS = 128;
    constexpr int ITEMS_PER_THREAD = TOTAL_ITEMS / NUM_BLOCKS / BLOCK_
↪THREADS;
    int h_in[TOTAL_ITEMS];
    int h_out[TOTAL_ITEMS];
    printf("The items in array h_in:\n");
    for (int i = 0; i < TOTAL_ITEMS; ++i) {
        h_in[i] = TOTAL_ITEMS - i; // [256, 255, 254, ..., 1]
        printf("h_in[%d]: %d\n", i, h_in[i]);
    }
    int *d_in;
    int *d_out;
    mcMalloc((void**)&d_in, sizeof(int) * TOTAL_ITEMS);
    mcMemcpy(d_in, h_in, sizeof(int) * TOTAL_ITEMS, mcMemcpyHostToDevice);
    mcMalloc((void**)&d_out, sizeof(int) * TOTAL_ITEMS);
    BlockSortKernel<BLOCK_THREADS, ITEMS_PER_THREAD><<<NUM_BLOCKS, BLOCK_
↪THREADS>>>(d_in, d_out);
    mcMemcpy(h_out, d_out, sizeof(int) * TOTAL_ITEMS,
↪mcMemcpyDeviceToHost);
    printf("The items in array h_out:\n");
    for (int i = 0; i < TOTAL_ITEMS; ++i) {
        printf("h_out[%d]: %d\n", i, h_out[i]); // [1, 2, 3, ..., 256]
    }
```

(下页继续)

(续上页)

```
    mcFree(d_in);
    mcFree(d_out);
    return 0;
}
```

在 linux 上编译上述应用程序，可以使用以下命令：

```
export MACA_PATH=your/maca/toolkits/path
export MACA_CLANG_PATH=${MACA_PATH}/mxgpu_llvm/bin
export LD_LIBRARY_PATH=${MACA_PATH}/lib:$LD_LIBRARY_PATH

${MACA_CLANG_PATH}/mxcc myCubApp1.cu -o myCubApp1 -I ${MACA_PATH}/include/
↪cub
```

在这个例子中，线程使用 cub::BlockLoad、cub::BlockRadixSort 和 cub::BlockStore 来共同加载、排序和存储块的输入项段。因为这些操作是并行的，所以每个原语都需要分配共享内存，以便线程进行通信。

一个 CUB 集合的典型使用模式是：针对手头的特定问题设置静态专门化原语，例如，正在排序的数据类型、每个块的线程数、每个线程的键数，可选的算法替代方案等。（CUB 原语也被目标编译架构隐式特化。）在共享内存空间中分配（或别名）一个特化原语的嵌套 TempStorage 类型的实例。指定通信细节（例如，分配 TempStorage 内存）来构造该原语的实例。调用原语实例上的方法。

具体来说，cub::BlockRadixSort 用于对分配到线程块的数据项段进行集体排序。为了提供对设备内存的合并访问，我们配置了 cub::BlockLoad 和 cub::BlockStore 原语，以便使用条带化访问模式（连续的线程同时访问连续的项）访问内存，然后将键转换为跨线程的块状排列。为了在所有三种原语之间重复使用共享内存，线程块会静态分配它们的 TempStorage 类型的联合体。

2 模块

2.1 单指令，多线程（SIMT）原语“集合”

2.1.1 线程级单指令，多线程（SIMT）原语“集合”

2.1.1.1 类（Classes）

class cub::WarpExchange

```
template<
    typename InputT,
    int ITEMS_PER_THREAD,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpExchange< InputT, ITEMS_PER_THREAD, LOGICAL_WARP_THREADS,
↪PTX_ARCH >
```

WarpExchange 类提供了用于重新排列跨 MXMACA 线程束分区数据的集合方法。

模板参数

- T 要交换的数据类型。
- ITEMS_PER_THREAD 分配到每个线程上的项数量。
- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能小于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。必须是二次幂。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值

概述

- 在一个线程束的线程之间，重新排列数据项是很常见的。例如，全局内存的访问更倾向于数据项跨线程进行“条带化”的模式（连续线程访问连续数据项），但大多数线程束范围的操作更倾向于跨线程用“分块”的方式分区（连续数据项属于单个线程）。
- WarpExchange 支持以下类型的数据交换：
  - 在分块排列和条带排列之间进行转换
  - 将排列好的项分散为条带排列

简单示例

下面的代码段说明了将跨 16 个线程分区的 64 个整数项“分块”排列转换为“条带”排列，每个线程有 4 个项。

```
#include <cub/cub.cuh>   // 或 <cub/warp/warp_exchange.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    constexpr int warps_per_block = block_threads / warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 特化 WarpExchange 以适应一个有 16 个线程的虚拟线程束，每个线程拥有 4 个整数项
    using WarpExchangeT =
    cub::WarpExchange<int, items_per_thread, warp_threads>;
    // 为 WarpExchange 类分配共享内存
    __shared__ typename WarpExchangeT::TempStorage temp_storage[warps_per_
↪block];
    // 加载跨线程的条带化数据块
    int thread_data[items_per_thread];
    // ...
    // 将数据以分块排列方式跨线程地进行集体交换
    WarpExchangeT(temp_storage[warp_id]).StripedToBlocked(thread_data,
↪thread_data);
```

假设线程块中跨线程的条带化输入数据集为 {[0，16，32，48]，[1，17，33，49]，…，[15，32，47，63]}。相应的输出线程数据集将会是 {[0，1，2，3]，[4，5，6，7]，[8，9，10，11]，…，[60，61，62，63]}。

class cub::WarpLoad

```
template<
    typename InputT,
    int ITEMS_PER_THREAD,
    WarpLoadAlgorithm ALGORITHM = WARP_LOAD_DIRECT,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpLoad< InputT, ITEMS_PER_THREAD, ALGORITHM, LOGICAL_WARP_
↪THREADS, PTX_ARCH >
```

WarpLoad 类提供了集合数据移动方法，用于将内存中的一段线性数据项加载到一个跨 MXMACA 线程块分块排列中。

模板参数

- InputT 要读入的数据类型（要求其必须可以转换为输入迭代器的值类型）。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::WarpLoadAlgorithm 调优策略。默认值：cub::WARP_LOAD_DIRECT。
- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能小于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。必须是二次幂。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值

概述

- WarpLoad 类提供单个数据移动抽象（Data Movement Abstraction），可以专用于实现不同的 cub::WarpLoadAlgorithm 策略。这有助于针对不同的架构、数据类型、粒度大小等实现不同的性能策略。
- WarpLoad 类可以选择通过不同的数据移动策略进行特殊化：

- cub::WARP_LOAD_DIRECT：分块排列的数据直接从内存中读取。
- cub::WARP_LOAD_STRIPED：条带排列的数据直接从内存中读取。
- cub::WARP_LOAD_VECTORIZE：使用 MXMACA 的内置矢量化加载从内存中直接读取分块排列的数据用作合并优化。
- cub::WARP_LOAD_TRANSPOSE：从内存中直接读取条带排列的数据，然后在本地转换为分块排列。

简单示例

下面的代码段说明了将 64 个整数的线性段加载到跨 16 个线程的“分块”排列中，其中每个线程拥有 4 个连续项。加载是针对 WARP_LOAD_TRANSPOSE 进行优化的，这意味着内存引用会使用 warp-striped 访问模式有效地合并（之后，线程间对项进行了本地重新排序）。

```
#include <cub/cub.cuh>   // 或 <cub/warp/warp_load.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    // 特化 WarpLoad 以适应一个有 16 个线程的虚拟线程束，每个线程拥有 4 个整数项
    using WarpLoadT = WarpLoad<int,
                               items_per_thread,
                               cub::WARP_LOAD_TRANSPOSE,
                               warp_threads>;
    constexpr int warps_in_block = block_threads / warp_threads;
    constexpr int tile_size = items_per_thread * warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 为 WarpLoad 分配共享内存
    __shared__ typename WarpLoadT::TempStorage temp_storage[warps_in_
↪block];
    // 跨线程加载一段分块的连续项
    int thread_data[items_per_thread];
    WarpLoadT(temp_storage[warp_id]).Load(d_data + warp_id * tile_size,
                                          thread_data);
```

假设输入 d_data 为 0，1，2，3，4，5，…，这些线程中第一个逻辑线程束上跨线程的 thread_data 的集合将是：{[0，1，2，3]，[4，5，6，7]，…，[60，61，62，63]}。

class cub::WarpMergeSort

```
template<
    typename KeyT,
    int ITEMS_PER_THREAD,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    typename ValueT = NullType,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpMergeSort< KeyT, ITEMS_PER_THREAD, LOGICAL_WARP_THREADS,
↪ValueT, PTX_ARCH >
```

WarpMergeSort 类提供了使用归并排序的方法对跨 MXMACA 线程束分区的项进行排序。

模板参数

- KeyT 键类型。
- ITEMS_PER_THREAD 每个线程的项数量。

- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能小于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。必须是二次幂。
- ValueT **[可选]** 值类型，默认值：cub::NullType，表示仅按键排序。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

WarpMergeSort 使用具有小于语义的比较函数将项目按升序排列。归并排序可以处理任意类型和比较函数。

简单示例

下面的代码段说明了一种由 64 个整数键组成的类型，这些键跨 16 个线程进行分区，其中每个线程拥有 4 个连续项。

```
#include <cub/cub.cuh>  // 或 <cub/warp/warp_merge_sort.cuh>
struct CustomLess
{
    template <typename DataType>
    __device__ bool operator()(const DataType &lhs, const DataType &rhs)
    {
        return lhs < rhs;
    }
};
__global__ void ExampleKernel(...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    constexpr int warps_per_block = block_threads / warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 特化 WarpMergeSort 以适应一个有 16 个线程的虚拟线程束
    // 每个拥有 4 个整数项
    using WarpMergeSortT =
    cub::WarpMergeSort<int, items_per_thread, warp_threads>;
    // 为 WarpMergeSort 分配共享内存
    __shared__ typename WarpMergeSort::TempStorage temp_storage[warps_per_
↪block];
    // 获取按线程分块的一段连续项
    int thread_keys[items_per_thread];
    // ...
    WarpMergeSort(temp_storage[warp_id]).Sort(thread_keys, CustomLess());
    // ...
}
```

假设线程块中的输入 thread_keys 集合为 {[0，511，1，510]，[2，509，3，508]，[4，507，5，506]，…，[254，257，255，256]}。在这些线程中，对应的输出 thread_keys 将会是 {[0，1，2，3]，[4，5，6，7]，[8，9，10，11]，…，[508，509，510，511]}。

class cub::WarpReduce

```
template<
    typename T,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>
```

(下页继续)

(续上页)

```
class cub::WarpReduce< T, LOGICAL_WARP_THREADS, PTX_ARCH >
```

WarpReduce 类提供了进行并行归约的集体方法，用于计算划分到跨 MXMACA 线程线程束中的项。

模板参数

- T 归约输入/输出元素类型
- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能小于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。必须是二次幂。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 归约（或折叠）使用二元组合运算符从输入元素列表中计算出单个聚合值。
- 支持小于物理线程束大小的“逻辑”线程束（例如，由 8 个线程组成的逻辑线程束）。
- 参与的线程数必须是 LOGICAL_WARP_THREADS 的倍数。

性能注意事项

- 在适用的情况下使用特殊说明（例如，线程束 SHFL 指令）
- 在适用的情况下，在线程束的线程之间使用无同步通信
- 对于大多数类型，不会产生存储体冲突
- 计算效率略高（即具有较低的指令开销）的情况包括：
  - 求和（与通用归约相比）
  - 该架构的线程束大小是 LOGICAL_WARP_THREADS 的整数倍

简单示例

线程束中的每个线程都使用 WarpReduce 类，方法是首先特化 WarpReduce 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了在一个包含 128 个线程的线程块中进行 4 个并发线程束求和归约（每个线程束包含 64 个线程）。

```
#include <cub/cub.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 WarpReduce 以适应 int 类型
    typedef cub::WarpReduce<int> WarpReduce;
    // 为 4 个线程束分配 WarpReduce 共享内存
    __shared__ typename WarpReduce::TempStorage temp_storage[4];
    // 每个线程获取一个输入项
    int thread_data = ...
    // 将线程级的求和结果返回给每个 lane0（线程 0 和 64）
    int warp_id = threadIdx.x / 64;
    int aggregate = WarpReduce(temp_storage[warp_id]).Sum(thread_data);
```

假设线程块上的输入 thread_data 集为 {0，1，2，3，…，127}。线程 0、64 中相应的求和输出将分别为 1520、3568（在其他线程中未定义）。下面的代码段说明了在一个包含 128 个线程的线程块中进行单个线程束求和归约。

```
#include <cub/cub.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 WarpReduce 以适应 int 类型
    typedef cub::WarpReduce<int> WarpReduce;
```

(下页继续)

(续上页)

```
    // 为 1 个线程束分配 WarReduce 共享内存
    __shared__ typename WarpReduce::TempStorage temp_storage;
    ...
    // 只有第一个线程束执行了归约操作
    if (threadIdx.x < 64)
    {
        // 每个线程获取一个输入项
        int thread_data = ...
        // 将线程级的求和结果返回给 lane0
        int aggregate = WarpReduce(temp_storage).Sum(thread_data);
```

假设跨线程束的输入 thread_data 集为 {0，1，2，3，…，63}。线程 0 中对应的求和输出将是 2016（在其他线程中未定义）。

class cub::WarpScan

```
template<
    typename T,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpScan< T, LOGICAL_WARP_THREADS, PTX_ARCH >
```

WarpScan 类提供了一些集合方法，用于在 MXMACA 线程束中并行计算分区的项的前缀扫描。

模板参数

- T 扫描输入/输出元素类型
- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能小于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 给定一个输入元素列表和一个二元的归约算子，前缀扫描生成一个输出列表，其中每个元素被计算为输入列表中元素的归约。前缀和意味着使用加法运算符进行前缀扫描。inclusive 表示第 i 个输出归约包含第 i 个输入。exclusive 表示第 i 个输入未包含在第 i 个输出归约中。
- 支持非交换的扫描运算符
- 支持小于物理线程束大小的“逻辑”线程束（例如，由 8 个线程组成的逻辑线程束）
- 参与的线程数必须是逻辑线程束线程（LOGICAL_WARP_threads）的倍数

性能注意事项

- 在适用的情况下使用特殊说明（例如，线程束 SHFL 指令）
- 在适用情况下在线程束内的线程中使用无同步通信
- 对于大多数类型不存在存储体冲突
- 计算效率略高（即具有较低的指令开销）的情况包括：
  - 求和（与一般扫描相比）
  - 架构的线程束大小是 LOGICAL_WARP_THREADS 的整倍数

简单示例

线程束中的每个线程都使用 WarpScan 类，方法是首先特化 WarpScan 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集体成员函数。下面的代码段说明了一个包含 128 个线程的线程块中四个并发线程束的前缀和（每个线程束包含 64 个线程）。

```
#include <cub/cub.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 WarpScan 以适应 int 类型
    typedef cub::WarpScan<int> WarpScan;
    // 为 4 个线程束分配 WarpScan 类型的共享内存
    __shared__ typename WarpScan::TempStorage temp_storage[4];
    // 每个线程获取一个输入项
    int thread_data = ...
    // 计算线程级的前缀和
    int warp_id = threadIdx.x / 64;
    WarpScan(temp_storage[warp_id]).ExclusiveSum(thread_data, thread_data);
```

假设线程块中跨线程输入的 thread_data 集为 {1，1，1，1，…}。在四个线程束中的每个线程中，对应的输出 thread_data 将为 {0，1，2，3，…63}。下面的代码段说明了一个包含 128 个线程的线程块中单个线程束的前缀和。

```
#include <cub/cub.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 WarpScan 以适应 int 类型
    typedef cub::WarpScan<int> WarpScan;
    // 为一个线程束分配 WarpScan 类型的共享内存
    __shared__ typename WarpScan::TempStorage temp_storage;
    ...
    // 只有第一个线程束执行前缀和
    if (threadIdx.x < 64)
    {
        // 每个线程获取一个输入项
        int thread_data = ...
        // 计算线程级的前缀和
        WarpScan(temp_storage).ExclusiveSum(thread_data, thread_data);
```

假设线程束中跨线程输入的 thread_data 集为 {1，1，1，1，…}。对应的输出 thread_data 将是 {0，1，2，3，…63}。

class cub::WarpStore

```
template<
    typename T,
    int ITEMS_PER_THREAD,
    WarpStoreAlgorithm ALGORITHM = WARP_STORE_DIRECT,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpStore< T, ITEMS_PER_THREAD, ALGORITHM, LOGICAL_WARP_THREADS,
↪ PTX_ARCH >
```

WarpStore 类提供了集体数据移动方法，用于将跨 MXMACA 线程束分区的分块排列的项写入线性内存段。

模板参数

- T 要写入的数据类型。

- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::WarpStoreAlgorithm 调优策略枚举。默认值：cub::WARP_STORE_DIRECT。
- LOGICAL_WARP_THREADS **[可选]** 每个“逻辑”线程束的线程数（可能少于硬件线程束的线程数）。默认值是目标 MXMACA 计算能力的线程束大小。必须是二次幂。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- WarpStore 类提供了一个单一的数据移动抽象，可以实现不同的 cub::WarpStoreAlgorithm 策略。这有助于针对不同的体系结构、数据类型、粒度大小等制定不同的性能策略。
- WarpStore 可以选择性地通过不同的数据移动策略进行特殊化：
  - cub::WARP_STORE_DIRECT：分块排列的数据直接写入内存。
  - cub::WARP_STORE_STRIPED：条带排列的数据直接写入内存。
  - cub::WARP_STORE_VECTORIZE：使用 MXMACA 内置的矢量化存储作为合并优化，将分块排列的数据直接写入内存。
  - cub::WARP_STORE_TRANSPOSE：分块排列被局部地转换成条带排列，然后被写入内存。
- 对于多维块，线程按行主序线性排列。

简单示例

下面的代码段说明了如何在 16 个线程（每个线程拥有 4 个连续项）中将“分块”排列的 64 个整数存储到一个线性内存段中。该存储专用于 WARP_STORE_TRANSPOSE，这意味着项在线程之间进行本地重新排序，以便使用 warp-striped 访问模式有效地合并内存引用。

```
#include <cub/cub.cuh>   //  或 <cub/warp/warp_store.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    // 特化 WarpStore 以适应一个虚拟线程束中的 16 个线程，其中每一个线程拥有 4 个整数项。
    using WarpStoreT = WarpStore<int,
                                 items_per_thread,
                                 cub::WARP_STORE_TRANSPOSE,
                                 warp_threads>;
    constexpr int warps_in_block = block_threads / warp_threads;
    constexpr int tile_size = items_per_thread * warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 为 WarpStore 类型分配共享内存
    __shared__ typename WarpStoreT::TempStorage temp_storage[warps_in_
↪block];
    // 获取跨线程分块的一段连续项
    int thread_data[4];
    ...
    // 将项存储到线性内存中
    WarpStoreT(temp_storage[warp_id]).Store(d_data + warp_id * tile_size,
↪thread_data);
```

假设线程束中跨线程的 thread_data 集为 {[0,1,2,3]，[4,5,6,7]，…，[60,61,62,63]}。输出 d_data 将是 0,1,2,3,4,5，…。

2.1.1.2 函数

cub::ShuffleUp

```
template<int LOGICAL_WARP_THREADS, typename T >
__device__ __forceinline__ T cub::ShuffleUp
(   T                 input,
    int               src_offset,
    int               first_thread,
    unsigned int      member_mask
)
```

对任何数据类型进行 Shuffle-up。每个线程束中线程 i 的输入值由 warp-lanei-src_offset 得到。对于线程 i <src_offset，则将线程自己的输入返回给线程。

模板参数

- LOGICAL_WARP_THREADS 每个“逻辑”线程束的线程数。它的二次方必须 <=64。
- T **[推断]** 输入/输出元素类型

代码段

下面的代码段说明了每个线程从其前置线程的前置线程获得一个双精度值。

```
#include <cub/cub.cuh>   // 或<cub/util_ptx.cuh>
__global__ void ExampleKernel(...)
{
    // 每个线程获取一个输入项
    double thread_data = ...
    // 从当前线程向下两个位置获取项。
    double peer_data = ShuffleUp<64>(thread_data, 2, 0,
↪0xffffffffffffffffull);
```

假设第一组线程束中跨线程输入 thread_data 集为 {1.0，2.0，3.0，4.0，5.0，…，64.0}。对应的输出 peer_data 将是 {1.0，2.0，1.0，2.0，3.0，…，62.0}。5 位 SHFL 掩码用于逻辑上将线程束分割成从 8 位开始的字段

参数

- [in] input 要广播的值
- [in] src_offset 要读取当前位置相对向下的偏移量
- [in] first_thread 逻辑线程束中第一个线程的索引（通常为 0）
- [in] member_mask 参与线程束线程的 64 位掩码

cub::ShuffleDown

```
template<int LOGICAL_WARP_THREADS, typename T >
__device__ __forceinline__ T cub::ShuffleDown
(    T                 input,
     int               src_offset,
     int               last_thread,
     unsigned int      member_mask
)
```

对任何数据类型进行 Shuffle-down。每个线程束中线程 i 的输入值由 warp-lanei-src_offset 得到。对于线程 i >= WARP_THREADS，则将线程自己的输入返回给线程。

模板参数

- LOGICAL_WARP_THREADS 每个“逻辑”线程束的线程数。必须是二次方 <=64。

- T **[推断]** 输入/输出元素类型

代码段

下面的代码段说明了每个线程从后继线程的后继线程获得一个双精度值。

```
#include <cub/cub.cuh>   // 或 <cub/util_ptx.cuh>
__global__ void ExampleKernel(...)
{
    // 每个线程获取一个输入项
    double thread_data = ...
    // 从当前线程向下两个位置获取项。
    double peer_data = ShuffleDown<64>(thread_data, 2, 63,
↪0xffffffffffffffffull);
```

假设第一组线程束中跨线程输入的 thread_data 集为 {1.0,2.0,3.0,4.0,5.0，…, 64.0}。对应的输出 peer_data 将是 {3.0,4.0,5.0,6.0,7.0，…, 64.0}。5 位 SHFL 掩码用于逻辑上将线程束分割成从 8 位开始的字段

参数

- [in] input 要广播的值
- [in] src_offset 要读取当前位置相对向上的偏移量
- [in] last_thread 逻辑线程束中最后一个线程的索引（包含 64 个线程的线程束通常为 63）
- [in] member_mask 参与线程束中线程的 64 位掩码

cub::ShuffleIndex

```
template<int LOGICAL_WARP_THREADS, typename T >
__device__ __forceinline__ T cub::ShuffleIndex
(     T                   input,
      int                 src_lane,
      unsigned int        member_mask
)
```

对任何数据类型进行 Shuffle-broadcast。每个线程束中线程 i 的输入值由 warp-lanei-src_offset 得到。对于 src_lane < 0 或 src_lane >= WARP_THREADS，则将线程自己的输入返回给线程。

模板参数

- LOGICAL_WARP_THREADS 每个“逻辑”线程束的线程数。必须是二次方 <=64。
- T **[推断]** 输入/输出元素类型

代码段

下面的代码段说明了每个线程从线程束中的线程 0 得到一个双精度值。

```
#include <cub/cub.cuh>   // 或 <cub/util_ptx.cuh>
__global__ void ExampleKernel(...)
{
    // 每个线程获取一个输入项
    double thread_data = ...
    // 从线程 0 获取项
    double peer_data = ShuffleIndex<64>(thread_data, 0,
↪0xffffffffffffffffull);
```

假设第一组线程束中跨线程输入的 thread_data 集为 {1.0,2.0,3.0,4.0,5.0，…, 64.0}。对应的输出 peer_data 将是 {1.0,1.0,1.0,1.0,1.0，…, 1.0}。5 位 SHFL 掩码用于逻辑上将线程束分割成从 8 位开始的字段。

参数

- [in] input 要广播的值
- [in] src_lane 线程束中要做广播的线程
- [in] member_mask 参与线程束线程的 64 位掩码

2.1.2 块级别（Block-wide）的单指令，多线程（SIMT）原语“集合”

2.1.2.1 类

class cub::BlockAdjacentDifference

```
template<
    typename T,
    int BLOCK_DIM_X,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockAdjacentDifference< T, BLOCK_DIM_X, BLOCK_DIM_Y, BLOCK_DIM_
↪Z, PTX_ARCH >
```

BlockAdjacentDifference 提供了集合方法，用于计算跨 MXMACA 线程块分区的相邻元素之间的差异。

概述

- BlockAdjacentDifference 计算跨 MXMACA 线程块分区的相邻元素的差异。因为二进制操作可能是不可交换的，所以有两组方法 SubtractLeft 和 SubtractRight。SubtractLeft 方法从当前元素 i 中减去输入序列中左侧元素 i - 1。SubtractRight 方法从右侧元素 i + 1 中减去当前元素 i：

```
int values[4]; // [1, 2, 3, 4]
//...
int subtract_left_result[4];  <-- [  1,  1,  1,  1 ]
int subtract_right_result[4]; <-- [ -1, -1, -1,  4 ]
```

- 对于 SubtractLeft，如果左侧元素越界，输出值不加修改地赋给 input[0]。
- 对于 SubtractRight，如果右侧元素越界，输出值将不加修改地赋给当前输入值。
- examples/block 文件夹下的示例用于说明 BlockReduce 动态共享内存的使用以及如何重新利用相同的内存区域：example_block_reduce_dyn_smem. cu，这个例子对于 BlockAdjacentDifference 所需的存储具有广泛的适用性。

代码段

下面的代码段说明了如何使用 BlockAdjacentDifference 来计算相邻元素之间的左差。

```
#include <cub/cub.cuh>
// 或<cub/block/block_adjacent_difference.cuh>
struct CustomDifference
{
    template <typename DataType>
    __device__ DataType operator()(DataType &lhs, DataType &rhs)
    {
        return lhs - rhs;
    }
};
__global__ void ExampleKernel(...)
```

(下页继续)

(续上页)

```
{
    // 特化 BlockAdjacentDifference 以适应一维线程块
    // 128个 int 类型线程
    using BlockAdjacentDifferenceT =
    cub::BlockAdjacentDifference<int, 128>;
    // 为 BlockDiscontinuity 分配共享的内存
    __shared__ typename BlockAdjacentDifferenceT::TempStorage temp_storage;
    // 获取跨线程分块的一段连续项
    int thread_data[4];
    ...
    // 并行计算 adjacent_difference
    int result[4];
    BlockAdjacentDifferenceT(temp_storage).SubtractLeft(
        result,
        thread_data,
        CustomDifference());
```

假设线程块上跨线程输入的 thread_data 集为 {[4,2,1,1]，[1,1,1,1]，[2,3,3,3]，[3,4,1,4]，…}。在这些线程中对应的输出结果将是 {[4，-2，-1,0]，[0,0,0,0]，[1,1,0,0]，[0,1，-3,3]，…}。

class cub::BlockDiscontinuity

```
template<
    typename T,
    int BLOCK_DIM_X,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockDiscontinuity< T, BLOCK_DIM_X, BLOCK_DIM_Y, BLOCK_DIM_Z,
↪PTX_ARCH >
```

BlockDiscontinuity 类提供了集体方法，用于标记跨 MXMACA 线程块分区的有序项集合中的不连续点。

模板参数

- T 要标记的数据类型。
- BLOCK_DIM_X 在 X 维度上线程块长度。
- BLOCK_DIM_Y **[可选]** 在 Y 维度上线程块长度，默认：1。
- BLOCK_DIM_Z **[可选]** 在 Z 维度上线程块长度，默认：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 一组“头标记”（或“尾标记”）通常用来表示与它们的前身（或后继）不同的相应项。例如，作为分段扫描或归约的一部分，头标记可以方便地划分不相交的数据段。
- 假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中线程 i 拥有 items-per-thread 连续项的第 i 个范围。对于多维线程块，假定采用主行序的线程顺序。

性能注意事项

- 效率随着粒度 ITEMS_PER_THREAD 的增加而提高。性能通常也会提高，直到额外的寄存器压力或共享内存分配大小导致 SM 占用率下降得太低。考虑使用 cub::BlockLoad 的各种变体，以便跨线程有效地收集分块排列的元素。

简单示例

块中的每个线程都使用 BlockDiscontinuity 类，方法是首先特化 BlockDiscontinuity 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集体成员函数。下面的代码段说明了 512 个整数项的头部标记以分块排列的方式分配在 128 个线程中，其中每个线程拥有 4 个连续项。

```
#include <cub/cub.cuh>   // 或 <cub/block/block_discontinuity.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 BlockDiscontinuity 以适应一个包含 128 个 int 型线程的一维线程块
    typedef cub::BlockDiscontinuity<int, 128> BlockDiscontinuity;
    // 为 BlockDiscontinuity 类型分配共享内存
    __shared__ typename BlockDiscontinuity::TempStorage temp_storage;
    // 获取跨线程分块的一段连续项
    int thread_data[4];
    ...
    // 并行计算段中不连续点的头部标志
    int head_flags[4];
    BlockDiscontinuity(temp_storage).FlagHeads(head_flags, thread_data,
↪cub::Inequality());
```

假设跨线程块上的输入 thread_data 集为 {[0,0,1,1,1]，[1,1,1,1]，[2,3,3,3]，[3,4,4,4]，…}。在这些线程中对应的输出 head_flags 将是 {[1,0,1,0]，[0,0,0,0]，[1,1,0,0]，[0,1,0,0]，…}。

性能注意事项

- 大多数类型不存在存储体冲突。

class cub::BlockExchange

```
template<
    typename InputT,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    bool WARP_TIME_SLICING = false,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockExchange< InputT, BLOCK_DIM_X, ITEMS_PER_THREAD, WARP_TIME_
↪SLICING, BLOCK_DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockExchange 类提供了用于重新排列跨 MXMACA 线程块分区的数据的集合方法。

模板参数

- T 要交换的数据类型。
- BLOCK_DIM_X 在 X 维度上线程块长度。
- ITEMS_PER_THREAD 分配到每个线程上的项数量。
- WARP_TIME_SLICING **[可选]** 当设置为 true 时，仅使用足够的共享内存存储单个线程束块数据的值，在多个同步轮次中对块范围的交换进行时间切片。以降低并行性为代价获得更小的内存占用。默认值：false。
- BLOCK_DIM_Y **[可选]** 在 Y 维度上线程块长度，默认：1。
- BLOCK_DIM_Z **[可选]** 在 Z 维度上线程块长度，默认：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 线程块之间重新排列数据项是很常见的。例如，设备可访问内存子系统更喜欢数据项跨线程“跨距”的访问模式（即连续的线程访问连续的数据项），然而大多数块范围的操作更倾向于数据项跨线程的“分块”分区（即连续的数据项属于单个线程）。
- BlockExchange 支持以下类型的数据交换：
  - 在分块和条带化排列之间进行转置
  - 在分块和线程束-条带化排列之间进行转置
  - 将排列好的项分散成分块排列
  - 将排列好的项分散成条带排列
- 对于多维块，线程按行主序线性排列。

简单示例

块中的每个线程都使用 BlockExchange 类，首先特化 BlockExchange 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集体成员函数。下面的代码片段说明了将“分块”排列转换为“条带”排列，512 个整数项目被分割到 128 个线程中，每个线程拥有 4 个项目。

```
#include <cub/cub.cuh>   // 或 <cub/block/block_exchange.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    // 特化 BlockExchange 以适应一个 128 个线程的一维线程块（每个线程拥有 4 个整数项）

    typedef cub::BlockExchange<int, 128, 4> BlockExchange;
    // 为 BlockExchange 类型分配共享内存
    __shared__ typename BlockExchange::TempStorage temp_storage;
    // 加载跨线程的条带化数据块
    int thread_data[4];
    cub::LoadDirectStriped<128>(threadIdx.x, d_data, thread_data);
    // 跨线程地集体交换分块排列的数据
    BlockExchange(temp_storage).StripedToBlocked(thread_data);
```

假设跨线程块上的条带输入 thread_data 集为 {[0,128,256,384]，[1,129,257,385]，…，[127,255,383,511]}。在这些线程中对应的输出 thread_data 将是 {[0,1,2,3]，[4,5,6,7]，[8,9,10,11]，…，[508,509,510,511]}。

性能注意事项

特定于设备的正确填充，保证了大多数类型不存在存储体冲突。

class cub::BlockHistogram

```
template<
    typename T,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    int BINS,
    BlockHistogramAlgorithm ALGORITHM = BLOCK_HISTO_SORT,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockHistogram< T, BLOCK_DIM_X, ITEMS_PER_THREAD, BINS,
↪ALGORITHM, BLOCK_DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockHistogram 类提供了集合方法，用于根据跨 MXMACA 线程块分区的数据样本来构建块级别（Block-Wide）的直方图。

模板参数

- T 直方图表示的样本类型，必须可转换为整数 bin 标识符。
- BLOCK_DIM_X 在 X 维度上线程块长度。
- ITEMS_PER_THREAD 每个线程的项数量。
- BINS 直方图中 bins 的数量。
- ALGORITHM **[可选]** cub::BlockHistogramAlgorithm 枚举器，指定要使用的底层算法。默认值：cub::BLOCK_HISTO_SORT。
- BLOCK_DIM_Y **[可选]** 在 Y 维度上线程块长度，默认：1。
- BLOCK_DIM_Z **[可选]** 在 Z 维度上线程块长度，默认：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 直方图统计属于每个不相交类别（称为 bin）的观测值数量。
- T 类型必须隐式地可转换为整数类型。
- BlockHistogram 期望每个积分输入 input[i] 值满足 0 <= input[i] < BINS。超出此范围的值将导致未定义行为。
- 可对 BlockHistogram 进行可选择地特殊化，以使用不同的算法：
  - cub::BLOCK_HISTO_SORT。先排序后微分。
  - cub::BLOCK_HISTO_ATOMIC。使用原子加法直接更新字节计数。

性能注意事项

效率随着粒度 ITEMS_PER_THREAD 的增加而提高。性能通常也会提高，直到额外的寄存器压力或共享内存分配大小导致 SM 占用率下降得太低。考虑使用 cub::BlockLoad 的各种变体，以便跨线程有效地收集分块排列的元素。

简单示例

每个块中的线程都使用 BlockHistogram 类，首先特化 BlockHistogram 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了一个包含 512 个整数样本的 256-bin 直方图，这些样本分布在 128 个线程中，其中每个线程拥有 4 个样本。

```
#include <cub/cub。cuh>   // 或 <cub/block/block_histogram。cuh>
__global__ void ExampleKernel(...)
{
    // 特化 256-bin 的 BlockHistogram 类型以适应 128 个线程 （每个线程有 4 个字符样本 ） 的一维线程块
    typedef cub::BlockHistogram<unsigned char, 128, 4, 256> BlockHistogram;
    // 为 BlockHistogram 分配共享内存
    __shared__ typename BlockHistogram::TempStorage temp_storage;
    // 为整个区块的直方图 bin 计数分配共享内存
    __shared__ unsigned int smem_histogram[256];
    // 每个线程获取输入样本
    unsigned char data[4];
    ...
     // 计算整个块的直方图
    BlockHistogram(temp_storage)。Histogram(data, smem_histogram);
```

性能和使用注意事项

- 所有的输入值必须介于 [0, BINS) 范围内，否则会产生定义外的行为。

- 直方图输出可以在共享内存或设备可访问的内存中构建。

class cub::BlockLoad

```
template<
    typename InputT,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    BlockLoadAlgorithm ALGORITHM = BLOCK_LOAD_DIRECT,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockLoad< InputT, BLOCK_DIM_X, ITEMS_PER_THREAD, ALGORITHM,
↪BLOCK_DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockLoad 类提供了集体数据移动方法，用于在 MXMACA 线程块内将内存中的线性段加载到块排列。

模板参数

- InputT 要读入的数据类型，必须可由输入迭代器值的类型进行转换。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::BlockLoadAlgorithm 调优策略。默认值：cub::BLOCK_LOAD_DIRECT。
- WARP_TIME_SLICING **[可选]** 在任何与负载相关的数据转换过程中，是否只分配一个线程束的共享内存，并在各块线程束之间进行时间分片 (而不是每个线程束都有自己的存储空间)。默认值：false。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- BlockLoad 类提供了一个单一的数据移动抽象，可以专门用于实现不同的 cub::BlockLoadAlgorithm 策略。这有助于针对不同的体系结构、数据类型、粒度大小等制定不同的性能策略。
- BlockLoad 可以通过不同的数据移动策略进行专门化：
  - cub::BLOCK_LOAD_DIRECT：直接从内存中读取分块排列的数据。
  - cub::BLOCK_LOAD_STRIPED：直接从内存中读取条带排列的数据。
  - cub::BLOCK_LOAD_VECTORIZE：使用 MXMACA 内置的矢量化加载，以分块排列的方式直接从内存中读取数据，作为一种强化合并的优化。
  - cub::BLOCK_LOAD_TRANSPOSE：先直接从内存中读取条带排列的数据，然后将其在本地转置为分块排列。
  - cub::BLOCK_LOAD_WARP_TRANSPOSE：先直接从内存中读取 warp-striped 排列的数据，然后将其在本地转置为分块排列。
  - cub::BLOCK_LOAD_WARP_TRANSPOSE_TIMESLICED：先直接从内存中读取 warp-striped 排列的数据，然后线程束逐个进行本地转置，转换为分块排列。
- 对于多维块，线程按行主序进行线性排序。

简单示例

每个块中的线程都使用 BlockLoad 类，首先特化 BlockLoad 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码片段说明了将 512 个整数的线性段加载到 128 个线

程的分块排列中，其中每个线程拥有 4 个连续项。该加载针对 BLOCK_LOAD_WARP_TRANSPOSE 进行了专门处理，这意味着内存引用使用 warp-striped 访问模式进行了有效的合并 (之后，项在线程间进行了本地重新排序)。

```
#include <cub/cub。cuh>   // 或 <cub/block/block_load。cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    // 特化 BlockLoad 类型，用于 128 个线程的一维线程块，每个线程拥有 4 个整数项
    typedef cub::BlockLoad<int, 128, 4, BLOCK_LOAD_WARP_TRANSPOSE>
↪BlockLoad;
    // 为 BlockLoad 分配共享内存
    __shared__ typename BlockLoad::TempStorage temp_storage;
    // 加载在线程之间以分块排列的连续元素段
    int thread_data[4];
    BlockLoad(temp_storage)。Load(d_data, thread_data);
```

假设输入的 d_data 是 0，1，2，3，4，5，…，那么在这些线程中，thread_data 的集合将会是 { [0，1，2，3]，[4，5，6，7]，…，[508，509，510，511] }。

class cub::BlockMergeSort

```
template<
    typename KeyT,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    typename ValueT = NullType,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1>

class cub::BlockMergeSort< KeyT, BLOCK_DIM_X, ITEMS_PER_THREAD, ValueT,
↪BLOCK_DIM_Y, BLOCK_DIM_Z >
```

BlockMergeSort 类提供了归并排序方法，用于对跨 MXMACA 线程块中的分区的项进行排序。

模板参数

- KeyT KeyT 类
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ITEMS_PER_THREAD 每个线程的项数量。
- ValueT **[可选]** ValueT 类，默认值：cub::NullType，表示仅进行键排序。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。

概述

BlockMergeSort 使用小于语义的比较函数，将项按升序排列。归并排序可以处理任意类型和比较函数，但在将算术类型按升序/降序排序时比 BlockRadixSort 慢。

简单示例

每个块中的线程都使用 BlockMergeSort 类，首先特化 BlockMergeSort 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了对 512 个整数键进行排序，这些键被分割到 128 个线程，其中每个线程拥有 4 个连续项。

```
#include <cub/cub。cuh>  // 或<cub/block/block_merge_sort。cuh>
struct CustomLess
{
```

(下页继续)

(续上页)

```
    template <typename DataType>
    __device__ bool operator()(const DataType &lhs, const DataType &rhs)
    {
        return lhs < rhs;
    }
};
__global__ void ExampleKernel(...)
{
    // 特化 BlockMergeSort 以适应包含 128 个线程一维线程块，每个线程拥有 4 个整数项。
    typedef cub::BlockMergeSort<int, 128, 4> BlockMergeSort;
    // 为 BlockMergeSort 分配共享内存
    __shared__ typename BlockMergeSort::TempStorage temp_storage_shuffle;
    // 获取在线程间以分块排列的连续元素段
    int thread_keys[4];
    ...
    BlockMergeSort(temp_storage_shuffle).Sort(thread_keys, CustomLess());
    ...
}
```

假设在这些线程中，输入的 thread_keys 集是 {[0，511，1，510]，[2，509，3，508]，[4，507，5，506]，…，[254，257，255，256]}。那么在这些线程中输出的 thread_keys 将是 {[0，1，2，3]，[4，5，6，7]，[8，9，10，11]，…，[508，509，510，511]}。

class cub::BlockRadixSort

```
template<
    typename KeyT,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    typename ValueT = NullType,
    int RADIX_BITS = 4,
    bool MEMOIZE_OUTER_SCAN = (CUB_PTX_ARCH >= 350) ? true : false,
    BlockScanAlgorithm INNER_SCAN_ALGORITHM = BLOCK_SCAN_WARP_SCANS,
    mcSharedMemConfig SMEM_CONFIG = mcSharedMemBankSizeFourByte,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockRadixSort< KeyT, BLOCK_DIM_X, ITEMS_PER_THREAD, ValueT,
↪RADIX_BITS, MEMOIZE_OUTER_SCAN, INNER_SCAN_ALGORITHM, SMEM_CONFIG, BLOCK_
↪DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockRadixSort 类提供了集合方法，使用基数排序方法，对在 MXMACA 线程块中分区的项进行排序

模板参数

- KeyT KeyT 类型。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ITEMS_PER_THREAD 每个线程的项数量。
- ValueT **[可选]** ValueT 类型，默认值：cub::NullType，表示仅按键排序。
- RADIX_BITS **[可选]** 每个数字位上的基数位数，默认值：4 bits。
- MEMOIZE_OUTER_SCAN **[可选]** 是否缓冲外部扫描部分，减少共享内存读取次数，但代价是增加寄存器压力。

- INNER_SCAN_ALGORITHM **[可选]** cub::BlockScanAlgorithm 算法的选择，默认值：cub::BLOCK_SCAN_WARP_SCANS。
- SMEM_CONFIG **[可选]** 共享存储器组模式，默认值：mcSharedMemBankSizeFourByte。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

基数排序方法按升序排列项目。它依赖于键的位置表示，即每个键由一个有序的符号序列组成 (例如，数字、字符等)。有序序列按指定的最低有效位到最高有效位排序。对于给定的键输入序列和一组指定符号字母表总排序的规则，基数排序方法产生这些键的字典序排序。对于多维块，线程按行主序进行线性排序。

支持的类型

BlockRadixSort 可以对所有内置的 C++ 数值原始类型 (unsigned char、int、double 等) 进行排序，同时也支持 MXMACA 的 __half 半精度浮点类型进行排序。

浮点特殊情况

正零和负零被视为等价，在输出中也将如此处理。对于 NaN 值不做特殊处理；它们在经过任何转换后，这些值都按照它们的位表示进行排序。

位键转换

尽管直接基数排序方法只能应用于无符号整数类型，但 BlockRadixSort 能够通过简单的逐位转换对有符号和浮点型进行排序，从而确保按字典顺序排列。当限制 [begin_bit, end_bit) 范围时，必须考虑位操作转换，因为位操作转换会在位范围截断之前发生。

在写入最终输出缓冲区时，在对键进行排序之前应用的任何转换将会被还原。

特定类型的位转换

为了将输入值转换为可进行基数排序的位表示形式，在排序之前进行以下转换：

- 对于无符号整数值，直接使用键值。
- 对于有符号整数值，符号位取反。
- 对于正浮点值，符号位取反。
- 对于负浮点值，整个键值取反。

不支持降序排序

与 DeviceRadixSort 不同，BlockRadixSort 在执行降序排序时不会反转输入键位。而是它具有特殊逻辑，可以在排序过程中反转键值的顺序。

稳定性

BlockRadixSort 是稳定的。对于浮点数类型，-0.0 和 +0.0 被视为相等，并且在输出顺序与它们在输入的相对顺序相同。

性能注意事项

效率随着粒度 ITEMS_PER_THREAD 的增加而提高。性能通常也会提高，直到额外的寄存器压力或共享内存分配大小导致 SM 占用率下降得太低。考虑使用 cub::BlockLoad 的各种变体，以便跨线程有效地收集分块排列的元素。

简单示例

每个块中的线程都使用 BlockRadixSort 类，首先特化 BlockRadixSort 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码片段说明了 512 个整数键的排序，这些键以分块排列方式划分在 128 个线程中，每个线程拥有 4 个连续项。

```
#include <cub/cub。cuh>   // 或 <cub/block/block_radix_sort。cuh>
__global__ void ExampleKernel(...)
{
    // 特化 BlockMergeSort 以适应包含 128 个线程一维线程块，每个线程拥有 4 个整数项。
    typedef cub::BlockRadixSort<int, 128, 4> BlockRadixSort;
    // 为 BlockRadixSort 分配共享内存
    __shared__ typename BlockRadixSort::TempStorage temp_storage;
    // 获取在线程间以分块排列的连续元素段
    int thread_keys[4];
    ...
    // 以集体的方式对键进行排序
    BlockRadixSort(temp_storage)。Sort(thread_keys);
    ...
```

假设线程块上的输入 thread_keys 集为 {[0，511，1，510]，[2，509，3，508]，[4，507，5，506]，…，[254，257，255，256]}。相应线程中的输出 thread_keys 将会是 {[0，1，2，3]，[4，5，6，7]，[8，9，10，11]，…，[508，509，510，511]}。

class cub::BlockReduce

```
template<
    typename T,
    int BLOCK_DIM_X,
    BlockReduceAlgorithm ALGORITHM = BLOCK_REDUCE_WARP_REDUCTIONS,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockReduce< T, BLOCK_DIM_X, ALGORITHM, BLOCK_DIM_Y, BLOCK_DIM_
↪Z, PTX_ARCH >
```

BlockReduce 类提供了并行归约的集体方法，用于计算在 MXMACA 线程块中分区的项。

模板参数

- T 被归约的数据类型。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ALGORITHM **[可选]** cub::BlockScanAlgorithm 枚举类型，指定要使用的底层算法，默认值：cub::BLOCK_REDUCE_WARP_REDUCTIONS。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 归约 (或折叠) 使用二元组合运算符从输入元素列表中计算出单个聚合结果。
- 对于多维块，线程按行主序进行线性排序。
- BlockReduce 可以通过选择不同的算法来进行专门化，以适应不同的延迟/吞吐量工作负载配置：
  - cub::BLOCK_REDUCE_RAKING_COMMUTATIVE_ONLY. 一种高效的“raking”归约算法，仅支持可交换归约运算符。
  - cub::BLOCK_REDUCE_RAKING。一种高效的“raking”归约算法，支持可交换和不可交换归约运算符。

- cub::BLOCK_REDUCE_WARP_REDUCTIONS。一种快速的“tiled warp-reductions”归约算法，支持可交换和不可交换归约运算符。

性能注意事项

- 效率随着粒度 ITEMS_PER_THREAD 的增加而提高。性能通常也会提高，直到额外的寄存器压力或共享内存分配大小导致 SM 占用率下降得太低。考虑使用 cub::BlockLoad 的各种变体，以便跨线程有效地收集分块排列的元素。
- 非常高效 (只有一个同步屏障)。
- 对于大多数类型，没有存储体冲突。
- 计算效率略高 (即指令开销较低) 的情况包括：
  - 求和运算 (与通用归约相比)
  - BLOCK_THREADS 是架构中线程束大小的倍数
  - 每个线程都有有效的输入 (即完整块 vs. 部分块)

简单示例

每个块中的线程都使用 BlockReduce 类，首先特化 BlockReduce 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了对 512 个整数项进行求和归约，这些项按照指定的块排列方式划分在 128 个线程中，每个线程拥有 4 个连续项。

```
#include <cub/cub.cuh>   // 或 <cub/block/block_reduce.cuh>
__global__ void ExampleKernel(...)
{
    // 特化 BlockReduce 以适应包含 128 个 int 类型线程的一维线程块。
    typedef cub::BlockReduce<int, 128> BlockReduce;
    // 为 BlockReduce 分配共享内存
    __shared__ typename BlockReduce::TempStorage temp_storage;
    // 获取在线程间以分块排列的连续元素段
    int thread_data[4];
    ...
    // 计算线程 0 的块范围求和
    int aggregate = BlockReduce(temp_storage)。Sum(thread_data);
```

class cub::BlockScan

```
template<
    typename T,
    int BLOCK_DIM_X,
    BlockScanAlgorithm ALGORITHM = BLOCK_SCAN_RAKING,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockScan< T, BLOCK_DIM_X, ALGORITHM, BLOCK_DIM_Y, BLOCK_DIM_Z,
↪PTX_ARCH >
```

BlockScan 类提供了集体方法，用于对跨 MXMACA 线程块分区的项进行并行前缀和/前缀扫描。

模板参数

- T 正在扫描的数据类型。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ALGORITHM **[可选]** cub::BlockScanAlgorithm 枚举器，指定要使用的底层算法。默认值：cub::BLOCK_SCAN_RAKING。

- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- 给定一个输入元素列表和一个二元的归约算子，前缀扫描生成一个输出列表，其中每个元素被计算为输入列表中较早出现的元素的归约。前缀和意味着使用加法运算符进行前缀扫描。inclusive 表示第 i 个输出归约包含第 i 个输入。exclusive 表示第 i 个输入未包含在第 i 个输出归约中。
- 对于多维块，线程按行主序进行线性排列。
- BlockScan 可以根据算法进行可选的特殊化，以适应不同的工作负载配置文件：
  - cub::BLOCK_SCAN_RAKING。一种高效 (高吞吐量) 的“raking reduce-then-scan”前缀扫描算法。
  - cub::BLOCK_SCAN_RAKING_MEMOIZE。类似于 cub::BLOCK_SCAN_RAKING，但以额外的寄存器压力为代价，为中间存储提高吞吐量。
  - cub::BLOCK_SCAN_WARP_SCANS。一种快速 (低延迟) 的“tiled warpscans”前缀扫描算法。

性能注意事项

- 效率随着粒度 ITEMS_PER_THREAD 的增加而提高。性能通常也会提高，直到额外的寄存器压力或共享内存分配大小导致 SM 占用率下降得太低。考虑使用 cub::BlockLoad 的各种变体，以便跨线程有效地收集分块排列的元素。
- 在适用时使用特殊指令 (例如，线程束 SHFL)
- 在适用的情况下，在线程束线程之间使用免同步通信
- 调用最小数量的最小块范围同步屏障 (根据算法选择只有一个或两个)
- 大多数类型没有存储体冲突问题
- 计算效率略高 (即指令开销较低) 的情况包括：
  - 前缀和 (Prefix Sum) 变体 (与通用扫描相比)
  - 块中的线程数是体系结构线程束大小的倍数

简单示例

每个块中的线程都使用 BlockScan 类，首先特化 BlockScan 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了一个 512 个整数项的独占式前缀和扫描，这些项在 128 个线程中按照指定的分块排列方式划分，每个线程拥有 4 个连续项。

```
#include <cub/cub。cuh>   // 或 <cub/block/block_scan。cuh>
__global__ void ExampleKernel(...)
{
     //特化 BlockReduce 以适应包含 128 个 int 类型线程的一维线程块。
    typedef cub::BlockScan<int, 128> BlockScan;
    // 为 BlockScan 分配共享内存
    __shared__ typename BlockScan::TempStorage temp_storage;
    // 获取在线程间以分块排列的连续元素段
    int thread_data[4];
    ...
    // 并行计算整个线程块的独占式前缀和
    BlockScan(temp_storage)。ExclusiveSum(thread_data, thread_data);
```

假设在线程块中输入线程数据集为 {[1，1，1，1]，[1，1，1，1]，…，[1，1，1，1]}。那些线程中对应的输出线程数据将是 {[0，1，2，3]，[4，5，6，7]，…，[508，509，510，511]}。

class cub::BlockShuffle

```
template<
    typename T,
    int BLOCK_DIM_X,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockShuffle< T, BLOCK_DIM_X, BLOCK_DIM_Y, BLOCK_DIM_Z, PTX_
↪ARCH >
```

BlockShuffle 类提供了集合方法，用于跨 MXMACA 线程块中分区的数据进行重新排列。

模板参数

- T 要交换的数据类型。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

线程块内的数据项重新排列是很常见的操作。BlockShuffle 抽象化允许线程高效地将数据项进行移动，要么 (a) 向它们的后继线程移动，要么 (b) 向它们的前驱线程移动。

class cub::BlockStore

```
template<
    typename T,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    BlockStoreAlgorithm ALGORITHM = BLOCK_STORE_DIRECT,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockStore< T, BLOCK_DIM_X, ITEMS_PER_THREAD, ALGORITHM, BLOCK_
↪DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockStore 类提供了集体数据传输方法，用于在 MXMACA 线程块内将分块排列的数据项写入线性内存段。

模板参数

- T 要写入的数据类型。
- BLOCK_DIM_X 线程块在 X 维度上的长度。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::BlockStoreAlgorithm 优化策略枚举，默认值：cub::BLOCK_STORE_DIRECT。
- BLOCK_DIM_Y **[可选]** 线程块在 Y 维度上的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 线程块在 Z 维度上的长度，默认值：1。

- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- BlockStore 类提供了一个单一的数据移动抽象的方法，可以特化实现不同的 cub::BlockStoreAlgorithm 策略。这有助于针对不同的体系结构、数据类型、粒度大小等制定不同的性能策略。
- BlockStore 可以通过不同的数据移动策略进行特定化：
  - cub::BLOCK_STORE_DIRECT. 分块排列的数据直接写入内存。
  - cub::BLOCK_STORE_STRIPED. 条带排列的数据直接写入内存。
  - cub::BLOCK_STORE_VECTORIZE. 使用 MXMACA 内置的矢量化存储作为合并优化，将分块排列的数据直接写入内存。
  - cub::BLOCK_STORE_TRANSPOSE. 分块排列被本体地转换成条带排列，然后被写入内存。
  - cub::BLOCK_STORE_WARP_TRANSPOSE. 将分块排列的数据本地转置为 warp-stripes 排列，然后写入内存中。
  - cub::BLOCK_STORE_WARP_TRANSPOSE_TIMESLICED. 将分块排列本地转置为 warp-stripes 排列后写入内存。为了减小共享内存的需求，只分配一个线程束大小的共享内存，并在各个线程束之间进行时间切片。
- 对于多维块，线程以行主序进行线性排列。

简单示例

每个块中的线程都使用 BlockStore 类，首先特化 BlockStore 类型，然后实例化一个带有通信参数的实例，最后调用一个或多个集合成员函数。下面的代码段说明了将 512 个整数的“blocked”排列在 128 个线程上 (每个线程拥有 4 个连续项)，并将它们存储到线性内存段中。该存储专用于 BLOCK_STORE_WARP_TRANSPOSE，这意味着数据项在线程之间进行本地重新排序，以便使用 warp-striped 访问模式来实现内存引用的有效合并。

```
#include <cub/cub。cuh>   // 或 <cub/block/block_store。cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    //特化 BlockStore 以适应包含 128 个线程的一维线程块，每个线程拥有 4 个整数项。
    typedef cub::BlockStore<int, 128, 4, BLOCK_STORE_WARP_TRANSPOSE>
↪BlockStore;
    // 为 BlockStore 分配共享内存
    __shared__ typename BlockStore::TempStorage temp_storage;
    // 获取在线程间以分块排列的连续元素段
    int thread_data[4];
    ...
    // 将项存储到线性内存中
    BlockStore(temp_storage)。Store(d_data, thread_data);
```

假设线程块中的 thread_data 集合为 {[0，1，2，3]，[4，5，6，7]，…，[508，509，510，511]}。则输出 d_data 将是 0，1，2，3，4，5，…。

2.2 设备级原语 (Device-wide primitives)

2.2.1 单问题设备级原语 (Single-problem Device-wide primitives)

2.2.1.1 类

struct cub::DeviceAdjacentDifference

“DeviceAdjacentDifference”提供了设备级并行操作，用于计算设备可访问内存中的相邻元素之间的差值。

概述

- DeviceAdjacentDifference 会计算在 d_input 中的相邻元素的差异。由于二进制运算可以是非交换的，这里有两种设置的方法，分别为 SubtractLeft 和 SubtractRight。SubtractLeft 方法会在当前元素 *i 中减去左侧 *(i - 1) 元素。SubtractRight 方法会在当前 *i 的元素中减去右侧 *(i + 1) 元素：

```
int *d_values; // [1, 2, 3, 4]
//...
int *d_subtract_left_result  <-- [  1,  1,  1,  1 ]
int *d_subtract_right_result <-- [ -1, -1, -1,  4 ]
```

- 对于 SubtractLeft，如果左侧元素越界，迭代器不做修改赋值给 *(result + (i - first))。
- 对于 SubtractRight，如果右侧元素越界，迭代器不做修改赋值给 *(result + (i - first))。

代码段

下面代码段说明了怎样使用 DeviceAdjacentDifference 去计算临近元素的左差值。

```
#include <cub/cub.cuh>
// 或<cub/device/device_adjacent_difference.cuh>
// 声明、分配和初始化设备可访问的指针
int  num_items;       // e.g., 8
int  *d_values;       // e.g., [1, 2, 1, 2, 1, 2, 1, 2]
//...
// 设定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceAdjacentDifference::SubtractLeft(
d_temp_storage, temp_storage_bytes, d_values, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行操作
cub::DeviceAdjacentDifference::SubtractLeft(
d_temp_storage, temp_storage_bytes, d_values, num_items);
// d_values <-- [1, 1, -1, 1, -1, 1, -1, 1]
```

struct cub::DeviceHistogram

DeviceHistogram 提供了设备级并行操作，用于对设备可访问存储中的样本数据序列构建直方图。

概述

直方图统计属于每个不相交类别（称为 bin）的观测值数量。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceHistogram 方法。

struct cub::DeviceMergeSort

DeviceMergeSort 提供了设备级并行操作，用于对设备可访问存储器内的数据项序列进行归并排序。

概述

- DeviceMergeSort 使用一个具有小于语义的比较函数对项进行升序排序。归并排序可以处理任意类型（只要这些类型的值是 [LessThan Comparable] 的模型即可）和比较函数，但在对算术类型数据进行升序/降序排序时，该方法速度会比 DeviceRadixSort 慢。
- 与 RadixSort 的另一个不同之处在于 DeviceMergeSort 可以处理任意的随机访问迭代器，如下所示。

简单示例

下面的代码段说明了一个 thrust 反向迭代器的用法。

```
#include <cub/cub.cuh>  // 或 <cub/device/device_merge_sort.cuh>
struct CustomLess
{
    template <typename DataType>
    __device__ bool operator()(const DataType &lhs, const DataType &rhs)
    {
        return lhs < rhs;
    }
};
// 声明、分配和初始化设备可访问的指针
// 用于数据排序
thrust::device_vector<KeyType> d_keys(num_items);
thrust::device_vector<DataType> d_values(num_items);
// ...
// 初始化迭代器
using KeyIterator = typename thrust::device_vector<KeyType>::iterator;
thrust::reverse_iterator<KeyIterator> reverse_iter(d_keys.end());
// 设定临时设备存储需求
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortPairs(
    nullptr,
    temp_storage_bytes,
    reverse_iter,
    thrust::raw_pointer_cast(d_values.data()),
    num_items,
    CustomLess());
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceMergeSort::SortPairs(
    d_temp_storage,
    temp_storage_bytes,
    reverse_iter,
    thrust::raw_pointer_cast(d_values.data()),
    num_items,
    CustomLess());
```

struct cub::DevicePartition

DevicePartition 提供了设备级并行操作，对设备可访问内存中的数据项序列进行分区。

概述

在一个选择/未选择的项中的一个指定输入序列中，这些操作应用了一个选择标准去构建一个分区的输出序列。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DevicePartition 方法。

struct cub::DeviceRadixSort

DeviceRadixSort 提供设备级并行操作，用于计算设备可访问内存中数据项序列的基数排序。

概述

基数排序法将项按升序（或降序）排序。该算法依赖于键的位置表示，即每个键都由从最低有效位到最高有效位指定的有序符号序列（例如数字、字符等）组成。对于给定的键输入序列和一组指定符号字母表总排序的规则，基数排序法会产生这些键的词典排序。

支持的类型

DeviceRadixSort 可以对所有内置的 C++ 原始数值类型（unsigned char、int、double 等）以及 MXMACA 的 __half 和 __nv_bfloat16 16 位浮点类型进行排序。

浮点特殊情况

- 正零和负零被认为是等价的，在输出中也将如此处理。
- 对 NaN 值不做特殊处理；这些值会在任一转换后根据其位表示进行排序。

变换

虽然直接的基数排序方法只能适用于无符号整形数据类型，但 DeviceRadixSort 可以通过简单的位变换来对有符号和浮点类型的数据进行排序，从而确保键值的字典序。在写入最终输出缓冲区时，任何在排序前对键进行的转换都会被还原。

特定类型的位变换

为了将输入值转化为可进行基数排序的位表示方法，在排序之前需要进行如下转换：

- 对于无符号整型数值，直接使用键值。
- 对于有符号整型数值，符号位会取反。
- 对于正浮点数值，符号位会取反。
- 对于负浮点数值，整个键值会取反。
- 对于浮点类型数值，正零和负零是一种特殊情况，在排序时将被视为等价。

降序排序的位运算变换

如果使用降序排序，则在执行任何特定类型转换后，键值会被取反，得到的键值按升序排序。

稳定性

DeviceRadixSort 是稳定的。对于浮点类型，-0.0 和 +0.0 被认为是相等的，在输出与输入的相对顺序不变。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceRadixSort 方法。

struct cub::DeviceReduce

DeviceReduce 提供设备级并行操作，用于归约设备可访问内存中的数据项序列。

概述

一个归约（或折叠）操作，使用二进制组合运算符从输入元素序列中计算出单个集合。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceReduce 方法。

struct cub::DeviceRunLengthEncode

DeviceRunLengthEncode 提供设备级的并行操作，用于划分设备可访问内存中序列内同值项的 `run`。

概述

运行长度编码会对输入的元素序列进行一个简单压缩表示，使得每个最大连续相同数值数据项的 `run` 被编码为一个数据值，同时附带一个表示该运行中元素数量的计数。

使用注意事项

动态并行。DeviceRunLengthEncode 方法可以在支持 MXMACA 动态平行设备上的内核代码中进行调用。

struct cub::DeviceScan

DeviceScan 提供设备级的并行操作，用于计算设备可访问内存中的数据序列所进行的前缀扫描。

概述

给定输入元素列表和二进制归约运算符，前缀扫描产生一个输出列表，其中每个元素都计算为输入列表中元素的归约结果。前缀和表示具有添加运算符的前缀扫描。inclusive 表示第 i 个输出归约包括第 i 个输入。exclusive 表示第 i 个输入未包括在第 i 个输出归约中。仅需要约 2n 个数据移动（读取 n 个输入，写入 n 个输出），并且通常可以以 `memcpy` 的速度进行。支持原地操作。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceScan 方法。

struct cub::DeviceSelect

DeviceSelect 提供设备级并行操作，用于压缩设备可访问内存中数据项序列的选定项。

概述

这些操作将选择条件应用于指定的输入序列，以有选择地将元素从输入序列复制到一个紧凑的输出序列中。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceSelect 方法。

struct cub::DeviceSpmv

DeviceSpmv 提供设备级并行操作，用于执行稀疏矩阵 * 密集向量乘法 (SpMV) 。

概述

SpMV 计算方法用于执行矩阵-矢量运算 y = A*x + y ，其中：

- A 是一个 mxn 稀疏矩阵，其非零结构是以压缩存储行 (CSR) 格式进行存储（即三个数组：values、row_offsets 和 column_indices ）。
- x 和 y 都是密集向量。

使用注意事项

动态并行。可以在支持 MXMACA 动态平行设备上的内核代码中进行调用 DeviceSpmv 方法。

2.2.1.2 函数

cub::DeviceAdjacentDifference::SubtractLeftCopy

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪DifferenceOpT = cub::Difference>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceAdjacentDifference::SubtractLeftCopy
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      InputIteratorT      d_input,
      OutputIteratorT      d_output,
      std::size_t      num_items,
      DifferenceOpT      difference_op = {},
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

从设备可访问内存中的每个相邻元素对中减去左侧元素。

概述

计算 d_input 中相邻元素的差值。即 *d_input 赋值给 *d_output，并且对于范围 [d_input + 1, d_input + num_items] 内的每个迭代器 i，把 difference_op(*i, *(i-1)) 的结果赋值给 *(d_output + (i - d_input))。注意，如果输入和输出范围在任何方式上重叠，会产生定义外的行为。

代码段

下面代码段说明了如何使用 DeviceAdjacentDifference 计算相邻元素之间的差值。

```
#include <cub/cub.cuh>
// 或<cub/device/device_adjacent_difference.cuh>
struct CustomDifference
{
    template <typename DataType>
    __device__ DataType operator()(DataType &lhs, DataType &rhs)
    {
        return lhs - rhs;
    }
};
// 声明、分配和初始化设备可访问的指针
int  num_items;      // 例如， 8
int  *d_input;       // 例如， [1, 2, 1, 2, 1, 2, 1, 2]
int  *d_output;
...
// 设定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceAdjacentDifference::SubtractLeftCopy(
    d_temp_storage, temp_storage_bytes,
    d_input, d_output,
    num_items, CustomDifference());
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行操作
cub::DeviceAdjacentDifference::SubtractLeftCopy(
    d_temp_storage, temp_storage_bytes,
    d_input, d_output,
    num_items, CustomDifference());
```

(下页继续)

(续上页)

```
// d_input  <-- [1, 2, 1, 2, 1, 2, 1, 2]
// d_output <-- [1, 1, -1, 1, -1, 1, -1, 1]
```

模板参数

- InputIteratorT 输入迭代器模型。x 和 y 是输入迭代器 T 的 value_type 的对象，那么 x - y 是被定义的，并且 InputIteratorT 的 value_type 可以转换为 OutputIteratorT 的 value_types 集合中的类型，同时 x - y 的返回类型也可以转换为 OutputIteratorT 的 value_types 集合中的类型。
- OutputIteratorT 输出迭代器模型。
- DifferenceOpT 其 result_type 可转换为输出迭代器 T 的 value_types 集合中的一个类型。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作
- [in,out] temp_storage_bytes d_temp_storage 分配的大小（以字节为单位）
- [in] d_input 指向输入序列的指针
- [out] d_output 指向输出序列的指针
- [in] num_items 输入序列中的项数
- [in] difference_op 用于计算差值的二进制函数
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 steam 0
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false

cub::DeviceAdjacentDifference::SubtractLeft

```
template<typename RandomAccessIteratorT , typename DifferenceOpT =
↪cub::Difference>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceAdjacentDifference::SubtractLeft
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      RandomAccessIteratorT     d_input,
      std::size_t      num_items,
      DifferenceOpT      difference_op = {},
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

从设备可访问内存中的每个相邻元素对中减去左侧元素。

概述

计算 d_input 中相邻元素的差值。也就是说，对于范围 [d_input + 1, d_input + num_items] 中的每个迭代器 i，把 difference_op(*i, *(i-1)) 的结果赋值给 *(d_output + (i - d_input))。

代码段

下面代码段说明了如何使用 DeviceAdjacentDifference 计算相邻元素之间的差值。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_adjacent_difference.cuh>
struct CustomDifference
{
```

(下页继续)

(续上页)

```
    template <typename DataType>
    __device__ DataType operator()(DataType &lhs, DataType &rhs)
    {
        return lhs - rhs;
    }
};
// 声明、分配和初始化设备可访问的指针
int  num_items;      // 例如， 8
int  *d_data;        // 例如， [1, 2, 1, 2, 1, 2, 1, 2]
...
// 设定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceAdjacentDifference::SubtractLeft(
    d_temp_storage, temp_storage_bytes,
    d_data, num_items, CustomDifference());
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行操作
cub::DeviceAdjacentDifference::SubtractLeft(
    d_temp_storage, temp_storage_bytes,
    d_data, num_items, CustomDifference());
// d_data <-- [1, 1, -1, 1, -1, 1, -1, 1]
```

模板参数

- RandomAccessIteratorT 随机访问迭代器模型。RandomAccessIteratorT 是可变的。如果 x 和 y 是 RandomAccessIteratorT 的 value_type 的对象且 x - y 已定义，那么 x - y 的返回类型应该可以转换为 RandomAccessIteratorT 的 value_type 集合中的类型。
- DifferenceOpT 其 result_type 可转换为 RandomAccessIteratorT 的 value_types 集合中的一种类型。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小（以字节为单位）
- [in,out] d_input 输入序列和结果的指针
- [in] num_items 输入序列中的项数
- [in] difference_op 用于计算差值的二进制函数
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceAdjacentDifference::SubtractRightCopy

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪DifferenceOpT = cub::Difference>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceAdjacentDifference::SubtractRightCopy
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     InputIteratorT      d_input,
```

(下页继续)

(续上页)

```
    OutputIteratorT     d_output,
    std::size_t     num_items,
    DifferenceOpT     difference_op = {},
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

从设备可访问内存中的每个相邻元素对中减去右侧元素。

概述

计算 d_input 中相邻元素的右差值。将 *(d_input + num_items - 1) 赋值给 *(d_output + num_items - 1)，并且对于范围 [d_input, d_input + num_items - 1] 内的每个迭代器 i，difference_op(*i, *(i+1)) 的结果会赋值给 *(d_output+(i-d_input))。注意，如果输入和输出范围在任何方式上重叠，会产生定义外的行为。

代码段

下面代码段说明了如何使用 DeviceAdjacentDifference 计算相邻元素之间的差值。

```
#include <cub/cub.cuh>
// 或<cub/device/device_adjacent_difference.cuh>
struct CustomDifference
{
    template <typename DataType>
    __device__ DataType operator()(DataType &lhs, DataType &rhs)
    {
        return lhs - rhs;
    }
};
// 声明、分配和初始化设备可访问的指针
int  num_items;      // 例如， 8
int  *d_input;       // 例如， [1, 2, 1, 2, 1, 2, 1, 2]
int  *d_output;
..
// 设定临时设备存储需求
void *d_temp_storage = nullptr;
size_t temp_storage_bytes = 0;
cub::DeviceAdjacentDifference::SubtractRightCopy(
    d_temp_storage, temp_storage_bytes,
    d_input, d_output, num_items, CustomDifference());
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行操作
cub::DeviceAdjacentDifference::SubtractRightCopy(
    d_temp_storage, temp_storage_bytes,
    d_input, d_output, num_items, CustomDifference());
// d_input <-- [1, 2, 1, 2, 1, 2, 1, 2]
// d_data  <-- [-1, 1, -1, 1, -1, 1, -1, 2]
```

模板参数

- InputIteratorT 输入迭代器模型。如果 x 和 y 是输入迭代器 T 的 value_type 对象，那么 x - y 是有定义的，并且输入迭代器 T 的 value_type 可以转换为输出迭代器 T 的 value_type 集合中的一个类型，并且 x - y 的返回类型可以转换为输出迭代器 T 的 value_type 集合中的一个类型。
- OutputIteratorT 输出迭代器模型。
- DifferenceOpT 其 result_type 可转换为 RandomAccessIteratorT 的 value_types 集合中的一种类型。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小（以字节为单位）。
- [in] d_input 输入序列的指针。
- [out] d_output 输出序列指针。
- [in] num_items 输入序列中的项数。
- [in] difference_op 用于计算差值的二进制函数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceAdjacentDifference::SubtractRight

```
template<typename RandomAccessIteratorT , typename DifferenceOpT =
↪cub::Difference>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceAdjacentDifference::SubtractRight
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     RandomAccessIteratorT      d_input,
     std::size_t      num_items,
     DifferenceOpT      difference_op = {},
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

从设备可访问内存中的每个相邻元素对中减去右侧元素。

概述

计算 d_input 中相邻元素的右差值。对于范围 [d_input, d_input + num_items - 1) 中的每个迭代器 i，difference_op(*i, *(i+1)) 的结果会赋值给 *(d_input+(i-d_input))。

代码段

下面代码段说明了如何使用 DeviceAdjacentDifference 计算相邻元素之间的差值。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_adjacent_difference.cuh>
// 声明、分配和初始化设备可访问指针
int  num_items;      // 例如， 8
int  *d_data;        // 例如， [1, 2, 1, 2, 1, 2, 1, 2]
...
// 声明、分配和初始化设备可访问的指针
void *d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceAdjacentDifference::SubtractRight(
    d_temp_storage, temp_storage_bytes, d_data, num_items);
// 设定临时设备存储需求
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行操作
cub::DeviceAdjacentDifference::SubtractRight(
    d_temp_storage, temp_storage_bytes, d_data, num_items);
// d_data  <-- [-1, 1, -1, 1, -1, 1, -1, 2]
```

模板参数

- RandomAccessIteratorT 随机访问迭代器模型，RandomAccessIteratorT 是可变的。如果 x 和 y 是 RandomAccessIteratorT 的 value_type 对象，并且 x - y 已经被定义了，那么 x - y 的返回类型应该可以被转换为 RandomAccessIteratorT 的 value_types 集合中的某种类型。
- DifferenceOpT 其 result_type 可转换为 RandomAccessIteratorT 的 value_types 集合中的一种类型。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in,out] d_input 输入序列的指针。
- [in] num_items 输入序列中的项数。
- [in] difference_op 用于计算差异的二元函数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceHistogram::HistogramEven

```
template<typename SampleIteratorT , typename CounterT , typename LevelT ,
↪typename OffsetT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceHistogram::HistogramEven
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     SampleIteratorT      d_samples,
     CounterT *      d_histogram,
     int      num_levels,
     LevelT      lower_level,
     LevelT      upper_level,
     OffsetT      num_samples,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

根据数据样本序列，使用等宽分段方法计算强度直方图。

- 直方图 bin 的数量为 (num_levels - 1)
- 所有的 bin 都拥有相同的取值宽度：(upper_level - lower_level) / (num_levels - 1)
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了使用一系列浮点样本计算 6-bin 直方图的计算过程：

```
#include <cub/cub.cuh>    // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int      num_samples;     // 例如，10
float*   d_samples;       // 例如，[2.2, 6.1, 7.1, 2.9, 3.5, 0.3, 2.9, 2.1,
↪6.1, 999.5]
int*     d_histogram;     // 例如，[ -, -, -, -, -, -]
```

(下页继续)

(续上页)

```
int      num_levels;      // 例如，7        (六个 bin 的七个级别边界值)
float    lower_level;     // 例如，0.0      (最小 bin 的样本值下边界)
float    upper_level;     // 例如，12.0     (最大 bin 的样本值上边界)
...
// 确定临时设备存储的需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::HistogramEven(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level, num_
↪samples);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::HistogramEven(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level, num_
↪samples);
// d_histogram   <-- [1, 5, 0, 3, 0, 0];
```

模板参数

- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型
- LevelT **[推断]** 用于指定边界 (级别) 的类型
- OffsetT **[推断]** 有符号整数类型，用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

输入迭代器的样本值类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 输入序列的指针。
- [out] d_histogram 长度为 num_levels - 1 的直方图计数器输出数组的指针。
- [in] num_levels 划分直方图样本的边界 (级别) 数量。bin 的数量是 num_levels - 1。
- [in] lower_level 最小直方图 bin 的样本值下边界 (包含)。
- [in] upper_level 最大直方图 bin 的样本值上边界 (不包含)。
- [in] num_samples 输入样本的数量。(也就是，d_samples 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::HistogramEven

```
template<typename SampleIteratorT , typename CounterT , typename LevelT ,
↪typename OffsetT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceHistogram::HistogramEven
(    void *     d_temp_storage,
     size_t &     temp_storage_bytes,
```

(下页继续)

(续上页)

```
    SampleIteratorT     d_samples,
    CounterT *      d_histogram,
    int     num_levels,
    LevelT      lower_level,
    LevelT      upper_level,
    OffsetT      num_row_samples,
    OffsetT      num_rows,
    size_t      row_stride_bytes,
    mcStream_t      stream = 0,
    bool      debug_synchronous = false
)
```

根据数据样本序列，使用等宽分段方法计算强度直方图。

- 可以使用 num_row_samples，num_rows，和 row_stride_bytes 参数来指定 d_samples 中的二维感兴趣区域。
- 行跨度必须是样本数据类型大小的整数倍，即必须满足下列公式，(row_stride_bytes % sizeof(SampleT)) == 0。
- 直方图 bin 的数量为 (num_levels - 1)
- 所有的 bin 都拥有相同的取值宽度：(upper_level - lower_level) / (num_levels - 1)
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何使用一个 2x7 浮点样本数组 2x5 感兴趣区域计算一个六 bin 的直方图。

```
#include <cub/cub.cuh>   // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int      num_row_samples;     // 例如，5
int      num_rows;            // 例如，2;
size_t   row_stride_bytes;    // 例如，7 * sizeof(float)
float*   d_samples;           // 例如，[2.2, 6.1, 7.1, 2.9, 3.5,   -, -,
                              //         0.3, 2.9, 2.1, 6.1, 999.5, -, -]
int*     d_histogram;         // 例如，[ -, -, -, -, -, -]
int      num_levels;          // 例如，7       (六个 bin 的七个级别边界值)
float    lower_level;         // 例如，0.0     (最小 bin 的样本值下边界)
float    upper_level;         // 例如，12.0    (最大 bin 的样本值上边界)
...
// 确定临时设备存储的需求
void*    d_temp_storage  = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::HistogramEven(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level,
    num_row_samples, num_rows, row_stride_bytes);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::HistogramEven(d_temp_storage, temp_storage_bytes, d_
↪samples, d_histogram,
    d_samples, d_histogram, num_levels, lower_level, upper_level,
    num_row_samples, num_rows, row_stride_bytes);
// d_histogram   <-- [1, 5, 0, 3, 0, 0];
```

模板参数

- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型
- LevelT **[推断]** 用于指定边界 (级别) 的类型
- OffsetT **[推断]** 有符号整数类型，用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 输入序列的指针。
- [out] d_histogram 长度为 num_levels - 1 的直方图计数器输出数组的指针。
- [in] num_levels 划分直方图样本的边界 (级别) 的数量。bin 的数量是 num_levels - 1。
- [in] lower_level 最小直方图 bin 的样本值下边界 (包含)。
- [in] upper_level 最大直方图 bin 的样本值上边界 (不包含)。
- [in] num_row_samples 感兴趣区域中每行数据的样本数量。
- [in] num_rows 感兴趣区域中的行数。
- [in] row_stride_bytes 感兴趣区域中连续行起始点之间的字节数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::MultiHistogramEven

```
template<int NUM_CHANNELS, int NUM_ACTIVE_CHANNELS, typename
↪SampleIteratorT , typename CounterT , typename LevelT , typename OffsetT
↪>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceHistogram::MultiHistogramEven
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     SampleIteratorT      d_samples,
     CounterT *      d_histogram[NUM_ACTIVE_CHANNELS],
     int      num_levels[NUM_ACTIVE_CHANNELS],
     LevelT      lower_level[NUM_ACTIVE_CHANNELS],
     LevelT      upper_level[NUM_ACTIVE_CHANNELS],
     OffsetT      num_pixels,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

根据多通道“像素”数据样本序列，使用等宽分段方法计算每个通道的强度直方图。

- 输入由一系列像素构成，其中每个像素由 NUM_CHANNELS 个连续的数据样本组成 (例如 RGBA 像素)。
- 在指定的 NUM_CHANNELS 通道中，该函数只会为前 NUM_ACTIVE_CHANNELS 个通道计算直方图 (例如，仅使用 RGBA 像素样本计算 RGB 直方图)。
- 任意通道 i 的直方图 bin 的数量为 num_levels[i] - 1。
- 对于任意通道 i，所有直方图 bin 的取值范围具有相同的取值宽度：(upper_level[i] - lower_level[i]) / (num_levels[i] - 1)

- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何使用一个四通道的 RGBA 像素序列 (每个通道每个像素 8 位) 计算三个 256-bin 值的 RGB 直方图。

```
#include <cub/cub.cuh>    // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int              num_pixels;          // 例如， 5
unsigned char*   d_samples;           // 例如， [(2, 6, 7, 5), (3, 0, 2, 1),
↪(7, 0, 6, 2),
                                     //         (0, 6, 7, 5), (3, 0, 2, 6)]
int*             d_histogram[3];      // 例如， 三个设备指针指向三个设备缓冲区，
                                     //       每个缓冲区都分配了 256 个整数计数器
int              num_levels[3];       // 例如， {257, 257, 257};
unsigned int     lower_level[3];      // 例如， {0, 0, 0};
unsigned int     upper_level[3];      // 例如， {256, 256, 256};
...
// 确定临时设备存储的需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::MultiHistogramEven<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level, num_
↪pixels);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::MultiHistogramEven<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level, num_
↪pixels);
// d_histogram   <-- [ [1, 0, 1, 2, 0, 0, 0, 1, 0, 0, 0, ..., 0],
//                   [0, 3, 0, 0, 0, 0, 2, 0, 0, 0, 0, ..., 0],
//                   [0, 0, 2, 0, 0, 0, 1, 2, 0, 0, 0, ..., 0] ]
```

模板参数

- NUM_CHANNELS 输入数据中交错的通道数 (数量可能大于参与直方图计算的通道数)。
- NUM_ACTIVE_CHANNELS **[推断]** 参与直方图计算的通道数
- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型
- LevelT **[推断]** 用于指定边界 (级别) 的类型
- OffsetT **[推断]** 有符号整数类型用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

输入迭代器的样本值类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 多通道输入数据样本序列的指针。来自不同通道的样本被假定为交错的 (例如，一个 32 位像素数组，该数组的每个像素由四个 RGBA 8 位样本组成)。

- [out] d_histogram 直方图计数器输出数组的指针，每个参与的通道只有一个这样的指针。对于 channeli，d_histogram[i] 的分配长度应为 num_levels[i] - 1。
- [in] num_levels 在每个参与的通道中用于划分直方图样本的边界 (级别) 的数量。也就是说 channeli 的 bin 数量为 num_levels[i] - 1。
- [in] lower_level 在每个参与的通道中最小直方图 bin 的样本值下边界 (包含)。
- [in] upper_level 在每个参与的通道中最大直方图 bin 的样本值上边界 (不包含)。
- [in] num_pixels 多通道像素的数量 (即，the length of d_samples / NUM_CHANNELS)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::MultiHistogramEven

```
template<int NUM_CHANNELS, int NUM_ACTIVE_CHANNELS, typename
↪SampleIteratorT , typename CounterT , typename LevelT , typename OffsetT
↪>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceHistogram::MultiHistogramEven
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      SampleIteratorT     d_samples,
      CounterT *      d_histogram[NUM_ACTIVE_CHANNELS],
      int      num_levels[NUM_ACTIVE_CHANNELS],
      LevelT      lower_level[NUM_ACTIVE_CHANNELS],
      LevelT      upper_level[NUM_ACTIVE_CHANNELS],
      OffsetT      num_row_pixels,
      OffsetT      num_rows,
      size_t      row_stride_bytes,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

根据多通道“像素”数据样本序列，使用等宽分段方法计算每个通道的强度直方图。

- 输入由一系列像素构成，其中每个像素包含 NUM_CHANNELS 连续数据样本的记录 (例如，一个 RGBA 像素)。
- 在指定的 NUM_CHANNELS 中，该函数只会为前 NUM_ACTIVE_CHANNELS 个通道计算直方图 (例如，仅使用 RGBA 像素样本中计算 RGB 直方图)。
- 可以使用 num_row_samples，num_rows，和 row_stride_bytes 参数来指定 d_samples 中的二维感兴趣区域。
- 行跨度必须是样本数据类型大小的整数倍，即必须满足下列公式。(row_stride_bytes % sizeof(SampleT)) == 0。
- 任意通道 i 的直方图 bin 的数量为 num_levels[i] - 1。
- 对于任意通道 i，所有直方图 bin 的取值范围具有相同的取值宽度：(upper_level[i] - lower_level[i]) / (num_levels[i] - 1)
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何使用一个 2x4 四通道的 RGBA 像素序列 (每个通道每个像素 8 位)2x3 感兴趣区域计算三个 256 个箱的 RGB 直方图。

```
#include <cub/cub.cuh>   // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int              num_row_pixels;     // 例如， 3
int              num_rows;           // 例如， 2
size_t           row_stride_bytes;   // 例如， 4 * sizeof(unsigned char) *
↪NUM_CHANNELS
unsigned char*   d_samples;          // 例如， [(2, 6, 7, 5), (3, 0, 2, 1),
↪(7, 0, 6, 2), (-, -, -, -),
                                     //         (0, 6, 7, 5), (3, 0, 2, 6),
↪(1, 1, 1, 1), (-, -, -, -)]
int*             d_histogram[3];     // 例如， 三个设备指针指向三个设备缓冲区，
                                     //         每个缓冲区都分配了 256 个整数计数器
int              num_levels[3];      // 例如， {257, 257, 257};
unsigned int     lower_level[3];     // 例如， {0, 0, 0};
unsigned int     upper_level[3];     // 例如， {256, 256, 256};
...
// 确定临时设备存储的需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::MultiHistogramEven<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level,
    num_row_pixels, num_rows, row_stride_bytes);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::MultiHistogramEven<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, lower_level, upper_level,
    num_row_pixels, num_rows, row_stride_bytes);
// d_histogram   <-- [ [1, 1, 1, 2, 0, 0, 0, 1, 0, 0, 0, ..., 0],
//                   [0, 4, 0, 0, 0, 0, 2, 0, 0, 0, 0, ..., 0],
//                   [0, 1, 2, 0, 0, 0, 1, 2, 0, 0, 0, ..., 0] ]
```

模板参数

- NUM_CHANNELS 输入数据中交错的通道数 (数量可能大于参与直方图计算的通道数)。
- NUM_ACTIVE_CHANNELS **[推断]** 参与直方图计算的通道数
- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型
- LevelT **[推断]** 用于指定边界 (级别) 的类型
- OffsetT **[推断]** 有符号整数类型用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

输入迭代器的样本值类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 多通道输入数据样本序列的指针。来自不同通道的样本被假定为交错的 (例如，一个 32 位像素数组，该数组的每个像素由四个 RGBA 8 位样本组成)。
- [out] d_histogram 直方图计数器输出数组的指针，每个参与的通道只有一个这样的指针。对于任意通道 i，d_histogram[i] 的分配长度应为 num_levels[i] - 1。

- [in] num_levels 在每个参与的通道中用于划分直方图样本的边界 (级别) 的数量。也就是说任意通道 i 的 bin 数量为 num_levels[i] - 1。
- [in] lower_level 在每个参与的通道中最小直方图 bin 的样本值下边界 (包含)。
- [in] upper_level 在每个参与的通道中最大直方图 bin 的样本值上边界 (不包含)。
- [in] num_row_pixels 感兴趣区域中每行数据的样本数量。
- [in] num_rows 感兴趣区域中的行数。
- [in] row_stride_bytes 感兴趣区域中连续行起始点之间的字节数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::HistogramRange

```
template<typename SampleIteratorT , typename CounterT , typename LevelT ,
↪typename OffsetT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceHistogram::HistogramRange
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     SampleIteratorT      d_samples,
     CounterT *      d_histogram,
     int      num_levels,
     LevelT *      d_levels,
     OffsetT      num_samples,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用指定的 bin 边界级别以一系列数据样本为依据计算强度直方图。

- 直方图 bin 的数量是 (num_levels - 1)
- bini 的取值范围是 [level[i], level[i+1])
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何使用一系列浮点样本计算一个 6-bin 直方图。

```
#include <cub/cub.cuh>   // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int      num_samples;     // 例如，10
float*   d_samples;       // 例如，[2.2, 6.0, 7.1, 2.9, 3.5, 0.3, 2.9, 2.0,
↪6.1, 999.5]
int*     d_histogram;     // 例如，[ -, -, -, -, -, -]
int      num_levels       // 例如，7 (seven level boundaries for six bins)
float*   d_levels;        // 例如，[0.0, 2.0, 4.0, 6.0, 8.0, 12.0, 16.0]
...
// 确定临时设备存储的需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::HistogramRange(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_samples);
```

(下页继续)

(续上页)

```
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::HistogramRange(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_samples);
// d_histogram   <-- [1, 5, 0, 3, 0, 0];
```

模板参数

- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型。
- LevelT **[推断]** 用于指定边界 (级别) 的类型
- OffsetT **[推断]** 有符号整数类型用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

输入迭代器的样本值类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 输入序列的指针。
- [out] d_histogram 长度为 num_levels - 1 直方图计数器输出数组的指针。
- [in] num_levels 用于划分直方图样本的边界 (级别) 的数量。这意味着 bin 的数量是 num_levels - 1。
- [in] d_levels 边界 (级别) 数组的指针。连续边界对定义了 bin 的范围：样本值下边界是包含的，样本值上边界是不包含的。
- [in] num_samples 感兴趣区域中每行的数据样本数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::HistogramRange

```
template<typename SampleIteratorT , typename CounterT , typename LevelT ,
↪typename OffsetT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceHistogram::HistogramRange
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      SampleIteratorT      d_samples,
      CounterT *      d_histogram,
      int      num_levels,
      LevelT *      d_levels,
      OffsetT      num_row_samples,
      OffsetT      num_rows,
      size_t      row_stride_bytes,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用指定的 bin 边界级别以一系列数据样本为依据计算强度直方图。

- 可以使用 num_row_samples，num_rows，和 row_stride_bytes 参数来指定 d_samples 中的二维感兴趣区域。
- 行跨度必须是样本数据类型大小的整数倍，即必须满足下列公式，(row_stride_bytes % sizeof(SampleT)) == 0。
- 直方图 bin 的数量为 (num_levels - 1)。
- 所有 bini 都拥有相同的取值宽度：(upper_level - lower_level) / (num_levels - 1)。
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面代码段说明了如何使用一个 2x7 浮点样本数组 2x5 感兴趣区域计算一个 6-bin 的直方图。

```
#include <cub/cub.cuh>   // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int      num_row_samples;     // 例如，5
int      num_rows;            // 例如，2;
int      row_stride_bytes;    // 例如，7 * sizeof(float)
float*   d_samples;           // 例如，[2.2, 6.0, 7.1, 2.9, 3.5,  -, -,
                              //       , 0.3, 2.9, 2.0, 6.1, 999.5, -, -]
int*     d_histogram;         // 例如，[ -, -, -, -, -, -]
int      num_levels           // 例如，7 (seven level boundaries for six
↪bins)
float    *d_levels;           // 例如，[0.0, 2.0, 4.0, 6.0, 8.0, 12.0, 16.0]
...
// 确定临时设备存储的需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::HistogramRange(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, d_levels,
    num_row_samples, num_rows, row_stride_bytes);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::HistogramRange(d_temp_storage, temp_storage_bytes,
    d_samples, d_histogram, num_levels, d_levels,
    num_row_samples, num_rows, row_stride_bytes);
// d_histogram   <-- [1, 5, 0, 3, 0, 0];
```

模板参数

- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型。(可能是一个简单的指针类型)
- CounterT **[推断]** 直方图 bin 计数器的整数类型。
- LevelT **[推断]** 用于指定边界 (级别) 的类型。
- OffsetT **[推断]** 有符号整数类型用于序列偏移、列表长度、指针差异等等。(建议使用 32 位值作为偏移量/长度/等等。例如，在 64 位内存模式下 int 通常比 size_t 具有更好的性能。)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当值为 NULL 时，所需分配的空间大小将被写入 temp_storage_bytes，并且不会执行其余任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 指向输入序列的指针。
- [out] d_histogram 长度为 num_levels - 1 的直方图计数器输出数组的指针。

- [in] num_levels 用于划分直方图样本的边界 (级别) 的数量。这意味着 bin 的数量是 num_levels - 1。
- [in] d_levels 边界 (级别) 数组的指针。连续的双线性边界定义了 bin 的范围：样本值下边界是包含的，样本值上边界是不包含的。
- [in] num_row_samples 感兴趣区域中每行数据的样本数量。
- [in] num_rows 感兴趣区域中的行数。
- [in] row_stride_bytes 感兴趣区域中连续行起始点之间的字节数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 用于决定是否在每次内核启动后同步流以检查错误。启用后可能会显著导致运行减速。默认值为 false。

cub::DeviceHistogram::MultiHistogramRange

```
template<int NUM_CHANNELS, int NUM_ACTIVE_CHANNELS, typename
↪SampleIteratorT , typename CounterT , typename LevelT , typename OffsetT
↪>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceHistogram::MultiHistogramRange
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      SampleIteratorT      d_samples,
      CounterT *      d_histogram[NUM_ACTIVE_CHANNELS],
      int      num_levels[NUM_ACTIVE_CHANNELS],
      LevelT *      d_levels[NUM_ACTIVE_CHANNELS],
      OffsetT      num_pixels,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

根据指定的分箱边界值，从多通道“像素”数据样本序列计算每个通道的强度直方图。

- 输入是一个像素结构组成的序列，其中每一个像素都包含 NUM_CHANNELS 个连续的数据样本 (例如，RGBA piexel)
- 在所指定的 NUM_CHANNELS 中，该函数仅会计算前 NUM_ACTIVE_CHANNELS 个通道的直方图 (例如，从 RGBA 像素样本计算 RGB 直方图)
- 在任一通道 i 中，直方图的 bin 数为 num_levels[i]-1。
- 在任一通道 i 中，直方图的每一个 bin 都拥有相同的宽度。bin 宽度的计算方式为:(upper_level[i] - lower_level[i]) / (num_levels[i] - 1)
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面示例代码用于计算来自 RGBA 像素序列 (每像素每通道 8 位) 的三个 4-bin RGB 直方图。

```
#include <cub/cub.cuh>    // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int             num_pixels;        // 例如，5
unsigned char   *d_samples;        // 例如，[(2, 6, 7, 5),(3, 0, 2, 1),(7, 0,
↪6, 2),
                                   //       , (0, 6, 7, 5),(3, 0, 2, 6)]
unsigned int    *d_histogram[3];   // 例如，[[ -, -, -, -],[ -, -, -, -],[ -,
↪-, -, -]];
```

(下页继续)

(续上页)

```
int            num_levels[3];     // 例如，{5, 5, 5};
unsigned int   *d_levels[3];      // 例如，[ [0, 2, 4, 6, 8],
                                  //         [0, 2, 4, 6, 8],
                                  //         [0, 2, 4, 6, 8] ];
...
// 确定临时设备的存储要求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::MultiHistogramRange<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_pixels);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::MultiHistogramRange<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_pixels);
// d_histogram   <-- [ [1, 3, 0, 1],
//                     [3, 0, 0, 2],
//                     [0, 2, 0, 3] ]
```

模板参数

- NUM_CHANNELS 输入数据中的通道数量 (可能大于参与直方图计算的通道数)。
- NUM_ACTIVE_CHANNELS **[推断]** 参与直方图计算的通道数
- SampleIteratorT **[推断]** 用于读取输入样本的随机访问输入迭代器类型 (可能只是一个简单的指针类型)
- CounterT **[推断]** 记录直方图 bin 数量的整型类型
- LevelT **[推断]** 用于指定边界 (等级) 的类型
- OffsetT **[推断]** 有符号整数类型，用于序列偏移量、列表长度、指针差异等等的。(考虑使用 32 位作为偏移量/长度等。例如，在 64bit 内存模式下，int 类型通常比 size_t 类型具有更好的性能。)

输入迭代器的样本值类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 指向多通道输入数据样本序列的指针。不同通道的样本被认为是交替存储的 (比如，一个 32 位像素的数组，每个像素都是由四个 8 位 RGBA 样本组成)
- [out] d_histogram 指向直方图计数输出数组的指针，数组中的每个数都代表一个通道。对于任意通道 i，d_histogram[i] 分配的长度应该是 num_levels[i]-1。
- [in] num_levels 表示在每个活动通道中用于划分直方图样本的边界 (级别) 数量。对于任意通道 i 的 bin 数为 num_levels[i]-1。
- [in] d_levels 指向边界 (级别) 数组的指针。每个活动通道都有一个边界 (级别) 数组。区间范围是由连续的边界配对定义的：区间左边较小值是包含在内，而区间右边最大值是不包含在内。
- [in] num_pixels 表示多通道像素图像中像素点的数量 (比如，长度 d_samples/NUM_CHANNELS)。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步执行流来检查错误，可能会导致显著的性能下降。默认为 false。

cub::DeviceHistogram::MultiHistogramRange

```
template<int NUM_CHANNELS, int NUM_ACTIVE_CHANNELS, typename
↪SampleIteratorT , typename CounterT , typename LevelT , typename OffsetT
↪>
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceHistogram::MultiHistogramRange
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     SampleIteratorT      d_samples,
     CounterT *      d_histogram[NUM_ACTIVE_CHANNELS],
     int      num_levels[NUM_ACTIVE_CHANNELS],
     LevelT *      d_levels[NUM_ACTIVE_CHANNELS],
     OffsetT      num_row_pixels,
     OffsetT      num_rows,
     size_t      row_stride_bytes,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

从多通道像素图数据样本序列中，根据指定的分箱边界值来计算每个通道的强度直方图。

- 输入是一个像素结构组成的序列，其中每一个像素都包含 NUM_CHANNELS 个连续的数据样本 (比如，RGBA piexel)
- 根据指定 NUM_CHANNELS 的值，函数将只会计算前 NUM_ACTIVE_CHANNELS 个通道的直方图 (比如，RGB 是 RGBA 的直方图)
- 可以使用 num_row_sample，num_rows，和 row_stride_bytes 参数来指定 d_samples 中二维感兴趣区域
- 行之间的跨度必须是一个样本数据类型大小的整数倍，即 (row_stride_bytes % sizeof(SampleT)) == 0。
- 对于任一通道 i 直方图的 bin 数量是 num_levels[i]- 1。
- 对于任一通道 i 直方图的 bins 的取值范围都拥有相同的宽度: (upper_level[i] - lower_level[i]) / (num_levels[i] - 1)
- 当 d_temp_storage 值为 NULL，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面代码片段说明了如何从一个二维 2x4 数组中的 2x3 感兴趣区域内计算三个 4-binRGB 直方图，该数组包含四通道的 RGBA 像素 (每个像素的每个通道使用 8 位表示)。

```
#include <cub/cub.cuh>   // 或<cub/device/device_histogram.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int              num_row_pixels;     // 例如，3
int              num_rows;           // 例如，2
size_t           row_stride_bytes;   // 例如，4 * sizeof(unsigned char) *
↪NUM_CHANNELS
unsigned char*   d_samples;          // 例如，[(2, 6, 7, 5),(3, 0, 2, 1),(1,
↪ 1, 1, 1),(-, -, -, -),
                                     //       , (7, 0, 6, 2),(0, 6, 7, 5),(3,
↪ 0, 2, 6),(-, -, -, -)]
int*             d_histogram[3];     // 例如，[[ -, -, -, -],[ -, -, -, -],
↪[ -, -, -, -]];
int              num_levels[3];      // 例如，{5, 5, 5};
```

(下页继续)

(续上页)

```
unsigned int*    d_levels[3];         // 例如，[ [0, 2, 4, 6, 8],
                                      //          [0, 2, 4, 6, 8],
                                      //          [0, 2, 4, 6, 8] ];
...
// 确定临时设备的存储需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceHistogram::MultiHistogramRange<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_row_pixels, num_rows,
↪ row_stride_bytes);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 计算直方图
cub::DeviceHistogram::MultiHistogramRange<4, 3>(d_temp_storage, temp_
↪storage_bytes,
    d_samples, d_histogram, num_levels, d_levels, num_row_pixels, num_rows,
↪ row_stride_bytes);
// d_histogram   <-- [ [2, 3, 0, 1],
//                     [3, 0, 0, 2],
//                     [1, 2, 0, 3] ]
```

模板参数

- NUM_CHANNELS 输入数据通道的数量 (可能大于参与直方图计算的通道数)
- NUM_ACTIVE_CHANNELS **[推测]** 参与直方图计算的通道数
- SampleIteratorT **[推测]** 用于读取输入样本的随机访问输入迭代器类型 (可能只是一个简单的指针类型)
- CounterT **[推测]** 整数类型，用于记录直方图 bin 数量
- LevelT **[推测]** 用于指定边界 (级别) 的类型
- OffsetT **[推测]** 有符号整数类型，用于表示序列偏移量、列表长度、指针差异等等。(考虑使用 32 位值作为偏移量、长度等。例如，在 64 位内存模式下，int 类型通常比 size_t 类型具有更好的性能。)

输入迭代器的样本值类型。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_samples 指向多通道输入数据样本序列的指针。不同通道的样本被认为是交替存储的 (比如，一个 32 位像素的数组，每个像素都是由四个 8 位 RGBA 样本组成)
- [out] d_histogram 指向直方图计数输出数组的指针，数组中的每个数都代表一个通道。对于任一通道 i，d_histogram[i] 分配的长度应该是 num_levels[i]-1。
- [in] num_levels 表示在每个活动通道中用于划分直方图样本的边界 (级别) 数量。对于任一通道 i 的 bin 数为 num_levels[i]-1。
- [in] d_levels 指向边界 (级别) 数组的指针。每个活动通道都有一个边界 (级别) 数组。bin 范围是由连续的边界配对定义的: 区间左边较小值是包含在内，而区间右边最大值是不包含在内。
- [in] num_row_pixels 感兴趣区域中每行的多通道像素数量
- [in] num_rows 感兴趣区域中的行数
- [in] row_stride_bytes 感兴趣区域中相邻行起始点之间相隔字节的数量
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。可能会导致显著的性能下降，默认为 false。

cub::DeviceMergeSort::SortPairs

```
template<typename KeyIteratorT , typename ValueIteratorT , typename
↪OffsetT , typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::SortPairs
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      KeyIteratorT      d_keys,
      ValueIteratorT      d_items,
      OffsetT      num_items,
      CompareOpT      compare_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用归并排序算法进行排序。

SortPairs 无法保证稳定性。即，假设在位置 i 和 j 上的值彼此之间没有大小差异，排序无法保证两个元素的相对顺序会被保留。

代码段

下面代码段用于说明对包含 int 类型键值对的设备向量进行排序。

```
#include <cub/cub.cuh>
// 或<cub/device/device_merge_sort.cuh>
// 为数据排序声明，分配和初始化设备访问指针
//
int  num_items;          // 例如，7
int  *d_keys;            // 例如，[8, 6, 6, 5, 3, 0, 9]
int  *d_values;          // 例如，[0, 1, 2, 3, 4, 5, 6]
...
// 初始化比较器
CustomOpT custom_op;
// 确定临时设备的存储要求
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序算法
cub::DeviceMergeSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 6, 8, 9]
// d_values    <-- [5, 4, 3, 2, 1, 0, 6]
```

模板参数

- KeyIteratorT 随机访问迭代器。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- ValueIteratorT 随机访问迭代器，并且 ValueIteratorT 是可变的。
- OffsetT 用于表示全局偏移量的整数类型。

- CompareOpT 可调用对象类型，包含严格弱序的有符号布尔类型 operator()(KeyT lhs，KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in,out] d_keys 指向未排序输入键序列的指针
- [in,out] d_items 指向未排序输入值序列的指针
- [in] num_items 参与排序的项的数量
- [in] compare_op 比较函数对象。如果第一个参数排在第二个参数前，则返回 true
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceMergeSort::SortPairsCopy

```
template<typename KeyInputIteratorT , typename ValueInputIteratorT ,
↪typename KeyIteratorT , typename ValueIteratorT , typename OffsetT ,
↪typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::SortPairsCopy
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      KeyInputIteratorT      d_input_keys,
      ValueInputIteratorT      d_input_items,
      KeyIteratorT      d_output_keys,
      ValueIteratorT      d_output_items,
      OffsetT      num_items,
      CompareOpT      compare_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用归并排序算法排序项。

- SortPairsCopy 无法保证稳定性。即，假设在位置 i 和 j 上的值彼此之间没有大小差异，排序无法保证两个元素的相对顺序会被保留。
- 输入数组的 d_in_keys 和 d_input_items 无法被修改。
- 请注意确保输入和输出数据的范围没有重叠，否则会产生定义外的行为。

代码段

下面代码段用于说明对一个包含 int 类型键值对的设备向量进行排序。

```
#include <cub/cub.cuh>
// 或<cub/device/device_merge_sort.cuh>
// 声明，分配，和初始化用于输入样本和输出直方图的设备可访问指针
//
int  num_items;        // 例如，7
int  *d_keys;          // 例如，[8, 6, 6, 5, 3, 0, 9]
int  *d_values;        // 例如，[0, 1, 2, 3, 4, 5, 6]
...
// 初始化比较器
CustomOpT custom_op;
```

(下页继续)

(续上页)

```
// 确定临时设备的存储要求
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortPairsCopy(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceMergeSort::SortPairsCopy(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 6, 8, 9]
// d_values    <-- [5, 4, 3, 2, 1, 0, 6]
```

模板参数

- KeyInputIteratorT 随机访问迭代器模型。KeyInputIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- ValueInputIteratorT 随机访问迭代器模型。
- KeyIteratorT 随机访问迭代器模型。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 模型。即其 value_type 的排序关系是符合 LessThan Comparable 要求的严格弱序。
- ValueIteratorT 随机访问迭代器模型，并且 ValueIteratorT 是可变的。
- OffsetT 用于表示全局偏移量的整数类型
- CompareOpT CompareOpT 是可调用的对象类型。包含严格弱序的有符号布尔类型 operator()(KeyT lhs, KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_input_keys 指向未排序输入键序列的指针
- [in] d_input_items 指向未排序输入值序列的指针
- [out] d_output_keys 指向已排序输出键序列的指针
- [out] d_output_items 指向已排序输出值序列的指针
- [in] num_items 参与排序的项的数量
- [in] compare_op 比较函数对象。如果第一个参数排在第二个参数前，则返回 true
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceMergeSort::SortKeys

```
template<typename KeyIteratorT , typename OffsetT , typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::SortKeys
(    void *     d_temp_storage,
     std::size_t &     temp_storage_bytes,
```

(下页继续)

(续上页)

```
    KeyIteratorT     d_keys,
    OffsetT     num_items,
    CompareOpT     compare_op,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用归并排序方法进行排序。

SortKeys 无法保证稳定性。即，假设在位置 i 和 j 上的值彼此之间没有大小差异，那么排序不保证两个元素的相对顺序会被保留。

代码段

下面代码片段说明了对一个包含 int 类型键值对的设备向量进行排序。

```
#include <cub/cub.cuh>
// 或<cub/device/device_merge_sort.cuh>
// 声明、分配和初始化用于数据排序的设备访问指针
//
int  num_items;        // 例如， 7
int  *d_keys;          // 例如， [8, 6, 7, 5, 3, 0, 9]
...
// 初始化比较器
CustomOpT custom_op;
// 确定临时设备存储的需求
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortKeys(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceMergeSort::SortKeys(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 7, 8, 9]
```

模板参数

- KeyIteratorT 随机访问迭代器模型。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- OffsetT 整数类型，用于表示全局偏移量。
- CompareOpT 可调用对象类型，包含严格弱序的有符号布尔类型 operator()(KeyT lhs, KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in,out] d_keys 指向未排序输入键序列的指针
- [in] num_items 参与排序的项的数量
- [in] compare_op 比较函数对象。如果第一个参数在第二个参数之前排序，则返回 true。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceMergeSort::SortKeysCopy

```
template<typename KeyInputIteratorT , typename KeyIteratorT , typename
↪OffsetT , typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::SortKeysCopy
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      KeyInputIteratorT      d_input_keys,
      KeyIteratorT      d_output_keys,
      OffsetT      num_items,
      CompareOpT      compare_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用归并排序方法进行排序。

- SortKeysCopy 并不保证是稳定的。即，假设在位置 i 和 j 上的值彼此之间没有大小差异，那么排序不保证两个元素的相对顺序会被保留。
- 输入数组 d_input_keys 不会被修改。
- 请注意确保输入和输出数据的范围没有重叠，否则产生定义外的行为。

代码段

下面代码段说明了对一个包含 int 类型键值对的设备向量进行排序。

```
#include <cub/cub.cuh>
// 或<cub/device/device_merge_sort.cuh>
// 声明、分配和初始化用于数据排序的设备访问指针
//
int  num_items;        // 例如，7
int  *d_keys;          // 例如，[8, 6, 7, 5, 3, 0, 9]
...
// 初始化比较器
CustomOpT custom_op;
// 确定临时设备的存储要求
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortKeysCopy(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceMergeSort::SortKeysCopy(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 7, 8, 9]
```

模板参数

- KeyInputIteratorT 随机访问迭代器模型。其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。

- KeyIteratorT 随机访问迭代器模型。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- OffsetT 整数类型，用于表示全局偏移量。
- CompareOpT 可调用对象类型，包含严格弱序的有符号布尔类型 operator()(KeyT lhs, KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将把所需的分配大小写入 temp_storage_bytes，并且不执行任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_input_keys 指向未排序输入键 (input keys) 的输入序列的指针。
- [out] d_output_keys 指向已排序输入键 (input keys) 的输出序列的指针。
- [in] num_items 参与排序的项的数量。
- [in] compare_op 比较函数对象。如果第一个参数在第二个参数之前排序，则返回 true。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceMergeSort::StableSortPairs

```
template<typename KeyIteratorT , typename ValueIteratorT , typename
↪OffsetT , typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::StableSortPairs
(     void *      d_temp_storage,
      std::size_t &       temp_storage_bytes,
      KeyIteratorT      d_keys,
      ValueIteratorT       d_items,
      OffsetT     num_items,
      CompareOpT      compare_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用归并排序算法进行排序。

StableSortPairs 具有稳定性：其保留了没有大小差异元素之间的相对顺序。也就是说，如果 x 排在 y 前面，且这两个元素是等价的 (既不是 x < y，也不是 y < x)，那么稳定排序结果仍然是 x 在 y 之前。

代码段

下面代码片段说明了对一个包含 int 类型键值对的设备向量进行排序。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_merge_sort.cuh>
// 声明、分配和初始化用于数据排序的设备访问指针
//
int  num_items;        // 例如，7
int  *d_keys;          // 例如，[8, 6, 6, 5, 3, 0, 9]
int  *d_values;        // 例如，[0, 1, 2, 3, 4, 5, 6]
...
// 初始化比较器
CustomOpT custom_op;
// 确定临时设备存储的需求
void *d_temp_storage = nullptr;
```

(下页继续)

(续上页)

```
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceMergeSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys, d_values, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 6, 8, 9]
// d_values    <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyIteratorT 随机访问迭代器。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- ValueIteratorT 随机访问迭代器，并且 ValueIteratorT 是可变的。
- OffsetT 用于表示全局偏移量的整数类型
- CompareOpT 可调用对象类型，包含严格弱序的有符号布尔类型 operator()(KeyT lhs，KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将所需的分配大小写入 temp_storage_bytes，并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in,out] d_keys 指向未排序输入键序列的指针。
- [in,out] d_items 指向未排序输入值序列的指针。
- [in] num_items 要排序的项的数量。
- [in] compare_op 比较函数对象，如果第一个参数在第二个参数之前，则返回 true。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceMergeSort::StableSortKeys

```
template<typename KeyIteratorT , typename OffsetT , typename CompareOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceMergeSort::StableSortKeys
(    void *      d_temp_storage,
     std::size_t &       temp_storage_bytes,
     KeyIteratorT      d_keys,
     OffsetT       num_items,
     CompareOpT      compare_op,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用归并排序方法对项进行排序。

StableSortPairs 具有稳定性：其保留了相等元素之间的相对顺序。也就是说，如果 x 在 y 之前，且这两个元素是等价的 (既不是 x < y，也不是 y < x)，那么稳定排序的后置条件是 x 仍然在 y 之前。

代码段

下面代码片段说明了对一个设备向量中的整数键进行排序。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_merge_sort.cuh>
// 声明、分配和初始化用于数据排序的设备访问指针
//
int  num_items;        // 例如，7
int  *d_keys;          // 例如，[8, 6, 7, 5, 3, 0, 9]
...
// 初始化比较器
CustomOpT custom_op;
// 确定临时设备存储的需求。
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// 分配临时存储空间。
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceMergeSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes,
    d_keys, num_items, custom_op);
// d_keys      <-- [0, 3, 5, 6, 7, 8, 9]
```

模板参数

- KeyIteratorT 随机访问迭代器模型。KeyIteratorT 是可变的，并且其 value_type 是符合 LessThan Comparable 的模型。即其 value_type 的排序关系符合 LessThan Comparable 要求的严格弱序。
- OffsetT 用于表示全局偏移量的整数类型。
- CompareOpT 可调用对象类型，包含严格弱序的有符号布尔类型 operator()(KeyT lhs，KeyT rhs)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将所需的分配大小写入 temp_storage_bytes，并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in,out] d_keys 指向未排序输入键序列的指针。
- [in] num_items 要排序的项的数量。
- [in] compare_op 比较函数对象，如果第一个参数在第二个参数之前，则返回 true。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DevicePartition::Flagged

```
template<typename InputIteratorT , typename FlagIterator , typename
↪OutputIteratorT , typename NumSelectedIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DevicePartition::Flagged
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
```

(下页继续)

(续上页)

```
    FlagIterator     d_flags,
    OutputIteratorT     d_out,
    NumSelectedIteratorT     d_num_selected_out,
    int     num_items,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用 d_flags 序列将 d_in 中的对应项分割成一个分区序列 d_out。将复制到的第一个分区中的项的总数写入 d_num_selected_out。

- d_flags 的值类型必须能够被转换为布尔类型（例如，bool、char、int 等）。
- 选定项的副本被压缩到 d_out 中，并保持它们原始的相对顺序，然而未选定项的副本被压缩到 d_out 的末尾，且以相反的顺序排列。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码段说明了如何从一个 int 类型的设备向量中压缩选定项。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_partition.cuh>
// 声明、分配和初始化用于设备访问的指针
// 输入数据、标志参数以及输出结果
int  num_items;               // 例如，8
int  *d_in;                   // 例如，[1, 2, 3, 4, 5, 6, 7, 8]
char *d_flags;                // 例如，[1, 0, 0, 1, 0, 1, 1, 0]
int  *d_out;                  // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int  *d_num_selected_out;     // 例如，[ ]
...
// 确定临时设备存储的需求。
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DevicePartition::Flagged(
    d_temp_storage, temp_storage_bytes,
    d_in, d_flags, d_out, d_num_selected_out, num_items);
// 分配临时存储空间。
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行选择操作
cub::DevicePartition::Flagged(
    d_temp_storage, temp_storage_bytes,
    d_in, d_flags, d_out, d_num_selected_out, num_items);
// d_out                 <-- [1, 4, 6, 7, 8, 5, 3, 2]
// d_num_selected_out    <-- [4]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型（可以是简单的指针类型）
- FlagIterator **[推断]** 用于读取选择标志的随机访问输入迭代器类型（可以是简单的指针类型）
- OutputIteratorT **[推断]** 用于写入输出项的随机访问输出迭代器类型（可以是简单的指针类型）
- NumSelectedIteratorT **[推断]** 用于记录所选项数目的输出迭代器类型（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将所需的分配大小写入 temp_storage_bytes，并且不执行任何操作。

- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_in 指向数据项输入序列的指针。
- [in] d_flags 指向选择标志输入序列的指针。
- [out] d_out 指向分区数据项输出序列的指针。
- [out] d_num_selected_out 指向所选项总数的输出指针（即未选择分区的偏移量）。
- [in] num_items 需要从中选择的项的总数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DevicePartition::If

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪NumSelectedIteratorT , typename SelectOp >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DevicePartition::If
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      NumSelectedIteratorT      d_num_selected_out,
      int      num_items,
      SelectOp      select_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用 select_op 函数对象将 d_in 中对应的元素拆分成一个分区序列 d_out。将复制到第一个分区中的项目总数写入 d_num_selected_out。

- 选定项的副本被压缩到 d_out 中，并保持它们原始的相对顺序，然而未选定项的副本被压缩到 d_out 的末尾，且以相反的顺序排列。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码段说明了从一个 int 类型的设备向量中压缩选定项。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_partition.cuh>
// 选择小于某个条件值的函数对象类型
struct LessThan
{
    int compare;
    CUB_RUNTIME_FUNCTION __forceinline__
    explicit LessThan(int compare) : compare(compare) {}
    CUB_RUNTIME_FUNCTION __forceinline__
    bool operator()(const int &a) const
    {
        return (a < compare);
    }
};
```

(下页继续)

(续上页)

```
// 声明、分配和初始化用于设备访问的指针
// 输入和输出
int      num_items;              // 例如，8
int      *d_in;                  // 例如，[0, 2, 3, 9, 5, 2, 81, 8]
int      *d_out;                 // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int      *d_num_selected_out;    // 例如，[ ]
LessThan select_op(7);
...
// 确定临时设备存储的需求。
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DevicePartition::If(
    d_temp_storage, temp_storage_bytes,
    d_in, d_out, d_num_selected_out, num_items, select_op);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行选择操作
cub::DevicePartition::If(
    d_temp_storage, temp_storage_bytes,
    d_in, d_out, d_num_selected_out, num_items, select_op);
// d_out                 <-- [0, 2, 3, 5, 2, 8, 81, 9]
// d_num_selected_out    <-- [5]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型（可以是简单的指针类型）
- OutputIteratorT **[推断]** 用于写入输出项的随机访问输出迭代器类型（可以是简单的指针类型）
- NumSelectedIteratorT **[推断]** 用于记录选择的项数量的输出迭代器类型（可以是简单的指针类型）
- SelectOp **[推断]** 选择函数对象类型，其包含成员函数 bool operator()(const T &a)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将把所需的分配大小写入 temp_storage_bytes，并且不执行任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向分割数据项输出序列的指针。
- [out] d_num_selected_out 指向输出的选定项总数的指针（即未选定分区的偏移量）。
- [in] num_items 选择的项的总数。
- [in] select_op 一元选择运算符。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DevicePartition::If

```
template<typename InputIteratorT , typename FirstOutputIteratorT ,
↪typename SecondOutputIteratorT , typename UnselectedOutputIteratorT ,
↪typename NumSelectedIteratorT , typename SelectFirstPartOp , typename
↪SelectSecondPartOp >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DevicePartition::If
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      FirstOutputIteratorT      d_first_part_out,
      SecondOutputIteratorT      d_second_part_out,
      UnselectedOutputIteratorT      d_unselected_out,
      NumSelectedIteratorT      d_num_selected_out,
      int      num_items,
      SelectFirstPartOp      select_first_part_op,
      SelectSecondPartOp      select_second_part_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用两个函数对象将 d_in 中的相应项分割为三个分区序列：d_first_part_out、d_second_part_out 和 d_unselected_out。将复制到第一个分区的项的总数写入 d_num_selected_out[0]，同时复制到第二个分区的项的总数写入 d_num_selected_out[1]。

- 由 select_first_part_op 选择的项的副本被压缩到 d_first_part_out，并保持它们原始的相对顺序。
- 由 select_second_part_op 选择的项的副本将被压缩到 d_second_part_out 中，并保持它们原始的相对顺序。
- 未选中项的副本将被压缩并以相反的顺序复制到 d_unselected_out 中。

代码段

下面代码段说明了这个算法如何将输入向量分成小、中、大三个部分，从而使得项的相对顺序不变。将任何不超过 6 的值视为小值，任何超过 50 的值将视为大值。因为用于定义小值部分与大值部分之间的值不匹配，所以隐含了中值。

这些定义将值空间分为三个类别。由于该算法提供了稳定的分区，所以想要保留它们在输入向量中出现的顺序可行的。

由于每个类别中的项数事先未知，我们需要每个类别的三个输出数组，每个数组包含 num_items 个元素。为了减少内存需求，我们可以将两个类别的输出存储合并在一起。

由于每个值恰好属于一个类别，因此分别将“大”值和“中”值添加到共享输出向量的头部和尾部是安全的。为了将项添加到输出数组的尾部，我们可以使用 thrust::reverse_iterator。

```
#include <cub/cub.cuh>
// 或<cub/device/device_partition.cuh>
// 选择小于某个条件的值的函数对象类型
struct LessThan
{
    int compare;
    CUB_RUNTIME_FUNCTION __forceinline__
    explicit LessThan(int compare) : compare(compare) {}
    CUB_RUNTIME_FUNCTION __forceinline__
    bool operator()(const int &a) const
    {
        return a < compare;
    }
};
// 选择大于某个条件的值的函数对象类型
```

(下页继续)

(续上页)

```
struct GreaterThan
{
    int compare;
    CUB_RUNTIME_FUNCTION __forceinline__
    explicit GreaterThan(int compare) : compare(compare) {}
    CUB_RUNTIME_FUNCTION __forceinline__
    bool operator()(const int &a) const
    {
        return a > compare;
    }
};
// 声明、分配和初始化用于设备访问的指针
// 输入和输出
int      num_items;                    // 例如，8
int      *d_in;                        // 例如，[0, 2, 3, 9, 5, 2, 81, 8]
int      *d_large_and_unselected_out; // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int      *d_small_out;                 // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int      *d_num_selected_out;          // 例如，[ , ]
thrust::reverse_iterator<T> unselected_out(d_large_and_unselected_out +
↪num_items);
LessThan small_items_selector(7);
GreaterThan large_items_selector(50);
...
// 确定临时设备存储需求
void *d_temp_storage = nullptr;
std::size_t temp_storage_bytes = 0;
cub::DevicePartition::If(
    d_temp_storage, temp_storage_bytes,
    d_in, d_large_and_medium_out, d_small_out, unselected_out,
    d_num_selected_out, num_items,
    large_items_selector, small_items_selector);
// 分配临时存储空间。
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行选择操作。
cub::DevicePartition::If(
    d_temp_storage, temp_storage_bytes,
    d_in, d_large_and_medium_out, d_small_out, unselected_out,
    d_num_selected_out, num_items,
    large_items_selector, small_items_selector);
// d_large_and_unselected_out  <-- [ 81,  ,  ,  ,  ,  , 8, 9 ]
// d_small_out                <-- [  0, 2, 3, 5, 2,  ,  ,  ]
// d_num_selected_out         <-- [  1, 5 ]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项（可以是简单的指针类型）
- FirstOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入由第一个运算符选择的输出项（可以是简单的指针类型）
- SecondOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入由第二个运算符选择的输出项（可以是简单的指针类型）
- UnselectedOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入未被选择项（可以是简单的指针类型）
- NumSelectedIteratorT **[推断]** 输出迭代器类型，用于记录选择的项数量（可以是简单的指针类型）
- SelectFirstPartOp **[推断]** 选择函数对象类型，其包含成员函数 bool operator()(const T &a)

- SelectSecondPartOp **[推断]** 选择函数对象类型，其包含成员函数 bool operator()(const T &a)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，将把所需的分配大小写入 temp_storage_bytes，并且不执行任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_in 指向数据项输入序列的指针。
- [out] d_first_part_out 指向由 select_first_part_op 选择的数据项输出序列的指针。
- [out] d_second_part_out 指向由 select_second_part_op 选择的数据项输出序列的指针。
- [out] d_unselected_out 指向未选中数据项的输出序列的指针。
- [out] d_num_selected_out 指向输出数组的指针。数组包含两个元素，分别是 select_first_part_op 和 select_second_part_op。其中，由 select_first_part_op 选择的项的总数存储在 d_num_selected_out[0] 中，由 select_second_part_op 选择的项的总数存储在 d_num_selected_out[1] 中。
- [in] num_items 要从中选择的项的总数。
- [in] select_first_part_op 一元选择运算符，用于选择 d_first_part_out。
- [in] select_second_part_op 一元选择运算符，用于选择 d_second_part_out。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceRadixSort::SortPairs

```
template<typename KeyT , typename ValueT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceRadixSort::SortPairs
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      const ValueT *      d_values_in,
      ValueT *      d_values_out,
      NumItemsT      num_items,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键-值对按升序排序。(需要约 2N 的辅助存储空间)

- 排序操作不会改变输入数据的内容。
- 必须使用指向连续内存的指针；目前不支持迭代器。
- 不支持原地操作。所提供的范围之间不得有任何重叠：
  - [d_keys_in, d_keys_in + num_items)
  - [d_keys_out, d_keys_out + num_items)
  - [d_values_in, d_values_in + num_items)
  - [d_values_out, d_values_out + num_items)

- 可以指定不同的键位的可选位子集 [begin_bit, end_bit)。这可以减少整体排序开销，相应地提高性能。
- 排序操作需要分配 O(N+P) 大小的临时设备存储空间，其中 N 是输入的长度，P 是设备上的流式多处理器数量。对于使用仅 O(P) 大小的临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码段说明了对带有相关联 int 值的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针。
int  num_items;          // 例如，7
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[        ...        ]
int  *d_values_in;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，[        ...        ]
...
// 确定临时设备存储的需求。
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out, num_items);
// 分配临时存储空间。
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out, num_items);
// d_keys_out           <-- [0, 3, 5, 6, 7, 8, 9]
// d_values_out         <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyT **[推断]** KeyT 类型
- ValueT **[推断]** ValueT 类型
- NumItemsT **[推断]** num_items 的类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 NULL 时，将把所需的分配大小写入 temp_storage_bytes，且不执行任何工作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)。
- [in] d_keys_in 指向排序前的键数据的输入数据指针。
- [out] d_keys_out 指向排序后的键数据的输出序列指针。
- [in] d_values_in 指向与关联值项相关联的输入序列的指针。
- [out] d_keys_out 指向重新排序后与关联值项相关联的输出序列的指针。
- [in] num_items 要排序的项的数量。
- [in] begin_bit **[可选]** 用于键比较所需的最低有效位索引（包含最低有效位）。
- [in] end_bit **[可选]** 用于键比较所需的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortPairs

```
template<typename KeyT , typename ValueT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceRadixSort::SortPairs
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      NumItemsT      num_items,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键-值对按升序排序（需要约 N 的辅助存储空间）。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 在每个键值对的两个缓冲区中的内容可能会被排序操作修改。
- 不支持原地操作。所提供的任何范围之间不能有重叠：
  - [d_keys.Current(), d_keys.Current() + num_items)
  - [d_keys.Alternate(), d_keys.Alternate() + num_items)
  - [d_values.Current(), d_values.Current() + num_items)
  - [d_values.Alternate(), d_values.Alternate() + num_items)
- 完成后，排序操作将更新每个 DoubleBuffer 包装器内的“current”指示器，以指示哪个缓冲区现在包含已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 可以指定一个可选的键位子集 [begin_bit, end_bit)。这可以减少整体排序开销，并相应地提高性能。
- 此操作需要相对较小的 O(P) 大小的临时设备存储分配，其中 P 是设备上的流式多处理器数量（相对于输入大小 N 而言，通常是一个很小的常数）。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面代码段说明了对带有相关联 int 值的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于对数据进行排序的设备可访问指针
int  num_items;          // 例如，7
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[        ...        ]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[        ...        ]
...
// 创建一组 DoubleBuffers 来包装设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时设备存储需求
```

(下页继续)

(续上页)

```
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes, d_keys,
↪ d_values, num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes, d_keys,
↪ d_values, num_items);
// d_keys.Current()      <-- [0, 3, 5, 6, 7, 8, 9]
// d_values.Current()    <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyT **[推断]** KeyT 类型
- ValueT **[推断]** ValueT 类型
- NumItemsT **[推断]** num_items 的类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入键，并在返回后更新指向已排序的输出键
- [in,out] d_values 值的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入值，并在返回后更新指向已排序的输出值
- [in] num_items 要排序的项的数量
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceRadixSort::SortPairsDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      const ValueT *      d_values_in,
      ValueT *      d_values_out,
      NumItemsT      num_items,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
```

(下页继续)

(续上页)

```
    bool      debug_synchronous = false
)
```

将键值对降序排列。(需要约 2N 辅助存储器)。

- 排序操作不会改变输入数据的内容。
- 必须使用指向连续内存的指针。目前不支持迭代器。
- 不支持原地操作。所提供的任何范围之间不得有重叠：
  - [d_keys_in, d_keys_in + num_items)
  - [d_keys_out, d_keys_out + num_items)
  - [d_values_in, d_values_in + num_items)
  - [d_values_out, d_values_out + num_items)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提高相应的性能。
- 这个操作需要为设备分配 O(N+P) 的临时存储空间，其中 N 是输入数据的长度，P 是设备上流式多处理器的数量。对于使用仅 O(P) 大小的临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。
- 性能
- 功能类似于 DeviceRadixSort::SortPairs。

代码段

下面代码段说明了如何对一个以 int 为键值对的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[        ...        ]
int  *d_values_in;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，[        ...        ]
...
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortPairsDescending(d_temp_storage, temp_storage_
↪bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortPairsDescending(d_temp_storage, temp_storage_
↪bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out, num_items);
// d_keys_out            <-- [9, 8, 7, 6, 5, 3, 0]
// d_values_out          <-- [6, 0, 2, 1, 3, 4, 5]
```

模板参数

- KeyT **[推断]** KeyT 类型
- ValueT **[推断]** ValueT 类型

- NumItemsT **[推断]** num_items 的类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_keys_in 指向排序前的键数据的输入数据的指针
- [out] d_keys_out 指向排序后的键数据的输出序列的指针
- [in] d_values_in 指向关联值项的相应输入序列的指针
- [out] d_keys_out 指向排序后关联值项的相应输出序列的指针
- [in] num_items 要排序的项的数量
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceRadixSort::SortPairsDescending
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     DoubleBuffer< ValueT > &      d_values,
     NumItemsT      num_items,
     int      begin_bit = 0,
     int      end_bit = sizeof(KeyT) * 8,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键值对降序排列。(需要约 N 的辅助存储空间)。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变两个缓冲区中的内容。
- 不支持原地操作。所提供的范围之间不能有任何重叠：
  - [d_keys.Current(), d_keys.Current() + num_items)
  - [d_keys.Alternate(), d_keys.Alternate() + num_items)
  - [d_values.Current(), d_values.Current() + num_items)
  - [d_values.Alternate(), d_values.Alternate() + num_items)
- 排序操作完成后，将更新每个双缓冲区包装器中的“current”指示器，以指示哪个缓冲区现在包含排序后的输出序列 (取决于指定的键位数和目标设备架构)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit]。这可以减少总体排序开销，并提升相应的性能。

- 此操作需要相对较小的临时设备存储分配，其复杂度为 O(P)，其中 P 是设备上流式多处理器的数量 (通常与输入大小 N 相比是一个较小的常数)
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

性能

功能类似于 DeviceRadixSort::SortPairs。

代码段

下面代码段说明了如何对一个以 int 为键值对的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[        ...        ]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[        ...        ]
...
// 创建一组 DoubleBuffers 来包装设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortPairsDescending(d_temp_storage, temp_storage_
↪bytes, d_keys, d_values, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortPairsDescending(d_temp_storage, temp_storage_
↪bytes, d_keys, d_values, num_items);
// d_keys.Current()      <-- [9, 8, 7, 6, 5, 3, 0]
// d_values.Current()    <-- [6, 0, 2, 1, 3, 4, 5]
```

模板参数

- KeyT **[推断]** KeyT 类型
- ValueT **[推断]** ValueT 类型
- NumItemsT **[推断]** num_items 的类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入键，并在返回后更新指向已排序的输出键
- [in,out] d_values 值的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入值，并在返回后更新指向已排序的输出值
- [in] num_items 要排序的项的个数
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortKeys

```
template<typename KeyT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceRadixSort::SortKeys
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      NumItemsT      num_items,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键按升序排序。(需要约 2N 的辅助存储空间)

- 排序操作不会改变输入数据的内容。
- 必须使用指向连续内存的指针。目前不支持迭代器。
- 不支持原地操作。所提供的范围之间不能有任何重叠：
  - [d_keys_in, d_keys_in + num_items)
  - [d_keys_out, d_keys_out + num_items)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 这个操作需要为设备分配 O(N+P) 的临时存储空间，其中 N 是输入数据的长度，P 是设备上流式多处理器的数量。对于使用仅 O(P) 大小的临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码段说明了如何对键为 int 的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[        ...        ]
...
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_keys_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_keys_out, num_items);
// d_keys_out            <-- [0, 3, 5, 6, 7, 8, 9]
```

模板参数

- KeyT **[推断]** KeyT 类型
- NumItemsT **[推断]** num_items 的类型
- NumItemsT **[推断]** num_items 的类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_keys_in 指向排序前的键数据的输入数据的指针
- [out] d_keys_out 指向排序后的键数据的输出序列的指针
- [in] num_items 要排序的项的数量
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortKeys

```
template<typename KeyT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceRadixSort::SortKeys
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     NumItemsT      num_items,
     int      begin_bit = 0,
     int      end_bit = sizeof(KeyT) * 8,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键按升序排序。(需要约 N 的辅助存储空间)。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变两个缓冲区的内容。
- 不支持原地操作。所提供的范围之间不能有任何重叠：
  - [d_keys.Current(), d_keys.Current() + num_items)
  - [d_keys.Alternate(), d_keys.Alternate() + num_items)
- 排序操作完成后，将更新每个双缓冲区包装器中的“current”指示器，以指示哪个缓冲区现在包含排序后的输出序列 (取决于指定的键位数和目标设备架构)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit]。这可以减少总体排序开销，并提升相应的性能。
- 此操作需要相对较小的临时设备存储分配，其复杂度为 O(P)，其中 P 是设备上流多处理器的数量 (通常与输入大小 N 相比是一个较小的常数)

- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码段说明了如何对键为 int 的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[        ...        ]
...
// 创建一个 DoubleBuffer 对象来包装这对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys,
↪num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortKeys(d_temp_storage, temp_storage_bytes, d_keys,
↪num_items);
// d_keys.Current()      <-- [0, 3, 5, 6, 7, 8, 9]
```

模板参数

- KeyT **[推断]** KeyT 类型
- NumItemsT **[推断]** num_items 类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入键，并在返回后指向已排序的输出键
- [in] num_items 要排序的项的数量
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortKeysDescending

```
template<typename KeyT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceRadixSort::SortKeysDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
```

(下页继续)

(续上页)

```
    const KeyT *     d_keys_in,
    KeyT *     d_keys_out,
    NumItemsT     num_items,
    int     begin_bit = 0,
    int     end_bit = sizeof(KeyT) * 8,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

将键按降序排序。(需要约 2N 的辅助存储空间)。

- 排序操作不会改变输入数据的内容。
- 必须使用指向连续内存的指针。目前不支持迭代器。
- 不支持原地操作。所提供的范围之间不能有任何重叠：
  - [d_keys_in, d_keys_in + num_items)
  - [d_keys_out, d_keys_out + num_items)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 这个操作需要为设备分配 O(N+P) 的临时存储空间，其中 N 是输入数据的长度，P 是设备上流式多处理器的数量。对于使用仅 O(P) 大小的临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

性能

- [out] d_keys_out

功能类似于 DeviceRadixSort::SortKeys。

代码段

下面代码段说明了如何对键为 int 的设备向量进行排序。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[        ...        ]
...
// 创建一个 DoubleBuffer 对象来包装这对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeysDescending(d_temp_storage, temp_storage_
↪bytes, d_keys_in, d_keys_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortKeysDescending(d_temp_storage, temp_storage_
↪bytes, d_keys_in, d_keys_out, num_items);
// d_keys_out            <-- [9, 8, 7, 6, 5, 3, 0]
```

模板参数

- KeyT **[推断]** KeyT 类型

- NumItemsT **[推断]** num_items 类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_keys_in 指向排序前的键数据的输入数据的指针
- [out] d_keys_out 指向排序后的键数据的输出序列的指针
- [in] num_items 要排序的项的数量
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceRadixSort::SortKeysDescending

```
template<typename KeyT , typename NumItemsT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceRadixSort::SortKeysDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      NumItemsT      num_items,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键按降序排序。(需要约 N 的辅助存储空间)。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变两个缓冲区的内容。
- 不支持原地操作。所提供的范围之间不能有任何重叠：
  - [d_keys.Current(), d_keys.Current() + num_items)
  - [d_keys.Alternate(), d_keys.Alternate() + num_items)
- 排序操作完成后，将更新每个双缓冲区包装器中的“current”指示器，以指示哪个缓冲区现在包含排序后的输出序列 (取决于指定的键位数和目标设备架构)
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit]。这可以减少总体排序开销，并提升相应的性能。
- 此操作需要分配的临时设备存储空间相对较小，其复杂度为 O(P)，其中 P 是设备上流式多处理器的数量 (通常与输入大小 N 相比是一个较小的常数)
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

性能

功能类似于 DeviceRadixSort::SortKeys。

代码段

下面的代码段说明了如何对键为 int 的设备向量进行排序。

```
#include <cub/cub.cuh>   //或<cub/device/device_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针
int  num_items;          // 例如，7
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[        ...        ]
...
// 创建一个 DoubleBuffer 对象来包装这对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeysDescending(d_temp_storage, temp_storage_
↪bytes, d_keys, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排序操作
cub::DeviceRadixSort::SortKeysDescending(d_temp_storage, temp_storage_
↪bytes, d_keys, num_items);
// d_keys.Current()      <-- [9, 8, 7, 6, 5, 3, 0]
```

模板参数

- KeyT **[推断]** KeyT 类型
- NumItemsT **[推断]** num_items 类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入键，在返回后更新指向已排序的输出键
- [in] num_items 要排序的项的个数
- [in] begin_bit **[可选]** 键比较所需的最低有效位索引 (含最低有效位)
- [in] end_bit **[可选]** 键比较所需的最高有效位索引 (不包含最高有效位)(例如，sizeof(unsigned int) * 8)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::Reduce

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪ReductionOpT , typename T >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::Reduce
(    void *     d_temp_storage,
     size_t &     temp_storage_bytes,
```

(下页继续)

(续上页)

```
    InputIteratorT     d_in,
    OutputIteratorT     d_out,
    int     num_items,
    ReductionOpT     reduction_op,
    T     init,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用指定的二进制 reduction_op 函数对象和初始值 init 计算设备范围内的归约。

- 不支持非交换的二元归约运算符。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对设备上的 int 类型数据元素进行用户定义的最小值归约。

```
#include <cub/cub.cuh>   // 或<cub/device/device_radix_sort.cuh>
// CustomMin 函数对象
struct CustomMin
{
    template <typename T>
    __device__ __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针
int             num_items;  // 例如，7
int             *d_in;      // 例如，[8, 6, 7, 5, 3, 0, 9]
int             *d_out;     // 例如，[-]
CustomMin       min_op;
int             init;       // 例如，INT_MAX
...
// 确定临时设备存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::Reduce(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items, min_op, init);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行归约运算
cub::DeviceReduce::Reduce(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items, min_op, init);
// d_out <-- [0]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录减少总量（可以是简单的指针类型）
- ReductionOpT **[推断]** 二元归约函数对象类型，其具有成员函数 T operator()(const T &a, const T &b)
- T **[推断]** 可转换为 InputIteratorT 值类型的数据元素类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] reduction_op 二元归约函数对象。
- [in] init 归约的初始值。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::Sum

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::Sum
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用加法运算符 (+) 计算设备范围内的总和。

- 将 0 作为归约操作的初始值。
- 不支持非交换的加法运算符。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对 int 类型数据元素的设备向量进行求和归约的过程。

```
#include <cub/cub.cuh>    // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;       // 例如，7
int  *d_in;           // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_out;          // 例如，[-]
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items);
```

(下页继续)

(续上页)

```
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行求和归约
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items);
// d_out <-- [38]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录减少总量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::Min

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::Min
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用小于运算符 (<) 计算设备范围内的最小值。

- 使用 std::numeric_limits<T>::max() 作为归约的初始值。
- 不支持非交换的小于运算符 (<)。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面的代码示例说明了对 int 类型数据元素的设备向量进行最小值归约的过程。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;      // 例如，7
int  *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // 例如，[-]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::Min(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行最小值归约
cub::DeviceReduce::Min(d_temp_storage, temp_storage_bytes, d_in, d_out,
↪num_items);
// d_out <-- [0]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录减少总量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::ArgMin

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::ArgMin
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OutputIteratorT      d_out,
     int      num_items,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用小于运算符 (<) 来找到设备范围内的第一个最小值，并返回该项的索引。

- d_out 的输出值类型是 cub::KeyValuePair <int, T>(假设 d_in 的值类型为 T)
- 最小值被写入 d_out.value，而其在输入数组中的偏移量被写入 d_out.key。
- 对于长度为零的输入，将生成元组 {1, std::numeric_limits<T>::max()}。

- 不支持非交换的 < 运算符。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对 int 数据元素的设备向量进行最小值索引归约的过程。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int                      num_items;      // 例如，7
int                      *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
KeyValuePair<int, int>   *d_out;         // 例如，[{-,-}]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::ArgMin(d_temp_storage, temp_storage_bytes, d_in, d_
↪argmin, num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行最小值索引归约
cub::DeviceReduce::ArgMin(d_temp_storage, temp_storage_bytes, d_in, d_
↪argmin, num_items);
// d_out <-- [{5, 0}]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型（某种类型 T），用于读取输入项（可以是简单的指针类型）
- OutputIteratorT **[推断]** 输出迭代器类型（具有值类型 cub::KeyValuePair<int, T>），用于记录减少总量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::Max

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::Max
(    void *     d_temp_storage,
     size_t &     temp_storage_bytes,
```

(下页继续)

(续上页)

```
    InputIteratorT     d_in,
    OutputIteratorT     d_out,
    int     num_items,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用大于运算符 (>) 计算设备范围内的最大值。

- 将 std::numeric_limits<T>::lowest() 作为归约的初始值。
- 不支持非交换的大于运算符 (>)。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对 int 数据元素的设备向量进行最大值归约的过程。

```
#include <cub/cub.cuh>   // 或<cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;      // 例如，7
int  *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // 例如，[-]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::Max(d_temp_storage, temp_storage_bytes, d_in, d_max,
↪num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行最大值归约
cub::DeviceReduce::Max(d_temp_storage, temp_storage_bytes, d_in, d_max,
↪num_items);
// d_out <-- [9]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录减少总量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes 为 d_temp_storage 分配以字节位单位大小的空间
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::ArgMax

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceReduce::ArgMax
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用大于运算符 (>) 来找到设备范围内的第一个最大值，并返回该项的索引。

- d_out 的输出值类型是 cub::KeyValuePair <int, T>(假设 d_in 的值类型为 T)
- 最大值将被写入 d_out.value，其在输入数组中的偏移量将被写入 d_out.key。
- 对于长度为零的输入，将生成 {1, std::numeric_limits<T>::lowest()} 元组。
- 不支持非交换的大于运算符 (>)。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对 int 数据元素的设备向量进行最大值索引归约的过程。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_reduce.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int                      num_items;      // 例如，7
int                      *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
KeyValuePair<int, int>   *d_out;         // 例如，[{-,-}]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceReduce::ArgMax(d_temp_storage, temp_storage_bytes, d_in, d_
↪argmax, num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行最大索引值归约
cub::DeviceReduce::ArgMax(d_temp_storage, temp_storage_bytes, d_in, d_
↪argmax, num_items);
// d_out <-- [{6, 9}]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型（某种类型 T），用于读取输入项（可以是简单的指针类型）
- OutputIteratorT **[推断]** 输出迭代器类型（具有值类型 cub::KeyValuePair<int, T>），用于记录归约后的聚合结果（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。

- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针。
- [out] d_out 指向输出集合的指针。
- [in] num_items 输入项的总数。(即，d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误。同时启动配置会输出到控制台。默认为 false。

cub::DeviceReduce::ReduceByKey

```
template<typename KeysInputIteratorT , typename UniqueOutputIteratorT ,
 ↪typename ValuesInputIteratorT , typename AggregatesOutputIteratorT ,
 ↪typename NumRunsOutputIteratorT , typename ReductionOpT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
 ↪cub::DeviceReduce::ReduceByKey
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      KeysInputIteratorT      d_keys_in,
      UniqueOutputIteratorT      d_unique_out,
      ValuesInputIteratorT      d_values_in,
      AggregatesOutputIteratorT      d_aggregates_out,
      NumRunsOutputIteratorT      d_num_runs_out,
      ReductionOpT      reduction_op,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

对值进行分段归约，分段由相同键的相应 run 划分。

- 该操作使用指定的二元函数对象 reduction_op 在 d_values_in 中计算分段归约。这些段是由 d_keys_in 中相应键的 run 识别的，其中 run 是连续相同键的最大范围。对于遇到的第 i 个 run，该 run 的第一个键和对应的聚合值被分别写入 d_unique_out[i] 和 d_aggregates_out[i] 中。遇到的总 run 数被写入 d_num_runs_out 中。
- == 等式运算符用于确定键是否相等。
- 为同一 GPU 设备上的伪关联归约（例如，浮点类型的加法）提供“run-to-run”的确定性。然而，由于 CUB 可以针对不同的架构采用不同的分块大小，伪关联归约的结果在一个设备与具有不同计算能力的另一设备之间可能不一致。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码片段说明了根据相邻的整数键值序列对整数值进行分段归约的示例。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_reduce.cuh>
// CustomMin 函数对象
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
```

(下页继续)

(续上页)

```
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针
int           num_items;          // 例如，8
int           *d_keys_in;         // 例如，[0, 2, 2, 9, 5, 5, 5, 8]
int           *d_values_in;       // 例如，[0, 7, 1, 6, 2, 5, 3, 4]
int           *d_unique_out;      // 例如，[-, -, -, -, -, -, -, -]
int           *d_aggregates_out;  // 例如，[-, -, -, -, -, -, -, -]
int           *d_num_runs_out;    // 例如，[-]
CustomMin     reduction_op;
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceReduce::ReduceByKey(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_unique_out, d_values_in, d_aggregates_out, d_num_runs_out,
↪reduction_op, num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行按键进行归约
cub::DeviceReduce::ReduceByKey(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_unique_out, d_values_in, d_aggregates_out, d_num_runs_out,
↪reduction_op, num_items);
// d_unique_out      <-- [0, 2, 9, 5, 8]
// d_aggregates_out  <-- [0, 1, 6, 2, 4]
// d_num_runs_out    <-- [5]
```

模板参数

- KeysInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入键（可以是简单的指针类型）
- UniqueOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入唯一输出键（可以是简单的指针类型）
- ValuesInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入值（可以是简单的指针类型）
- AggregatesOutputIterator **[推断]** 随机访问输出迭代器类型，用于写入输出值聚合（可以是简单的指针类型）
- NumRunsOutputIteratorT **[推断]** 输出迭代器类型，用于记录遇到的运行次数（可以是简单的指针类型）
- ReductionOpT **[推断]** 二元归约函数对象类型，具有成员函数 T operator()(const T &a, const T &b) 。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_keys_in 指向输入键序列的指针。
- [out] d_unique_out 指向输出唯一键序列（每次运行一个键）的指针。
- [in] d_values_in 指向输入序列的对应值的指针。
- [out] d_aggregates_out 指向值聚合的输出序列（每次运行一个聚合）的指针。
- [out] d_num_runs_out 指向遇到的总运行次数的指针。（即，d_unique_out 的长度）
- [in] reduction_op 二进制归约函数。
- [in] num_items 关联键值对的总数。（即，d_in_keys 和 d_in_values 的长度）

- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceRunLengthEncode::Encode

```
template<typename InputIteratorT , typename UniqueOutputIteratorT ,
 ↪typename LengthsOutputIteratorT , typename NumRunsOutputIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
 ↪cub::DeviceRunLengthEncode::Encode
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      UniqueOutputIteratorT      d_unique_out,
      LengthsOutputIteratorT      d_counts_out,
      NumRunsOutputIteratorT      d_num_runs_out,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

计算序列 d_in 的运行长度编码。

- 对于遇到的第 i 个运行，运行的第一个键和其长度分别写入 d_unique_out[i] 和 d_counts_out[i]。
- 遇到的运行总数被写入 d_num_runs_out。
- 使用 == 等式运算符来确定值是否相等。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码示例说明了对一系列 int 值进行运行长度编码的过程。

```
#include <cub/cub.cuh>    // 或 <cub/device/device_run_length_encode.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int            num_items;          // 例如，8
int            *d_in;              // 例如，[0, 2, 2, 9, 5, 5, 5, 8]
int            *d_unique_out;      // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int            *d_counts_out;      // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int            *d_num_runs_out;    // 例如，[ ]
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceRunLengthEncode::Encode(d_temp_storage, temp_storage_bytes, d_
 ↪in, d_unique_out, d_counts_out, d_num_runs_out, num_items);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行编码
cub::DeviceRunLengthEncode::Encode(d_temp_storage, temp_storage_bytes, d_
 ↪in, d_unique_out, d_counts_out, d_num_runs_out, num_items);
// d_unique_out      <-- [0, 2, 9, 5, 8]
// d_counts_out      <-- [1, 2, 1, 3, 1]
// d_num_runs_out    <-- [5]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可能是一个简单的指针类型)
- UniqueOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入唯一输出项 (可能是一个简单的指针类型)
- LengthsOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入唯一输出计数 (可能是一个简单的指针类型)
- NumRunsOutputIteratorT **[推断]** 输出迭代器类型，用于记录遇到的运行次数 (可能是一个简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向键输入序列的指针
- [out] d_unique_out 指向唯一键输出序列的指针 (每次运行一个键)
- [out] d_counts_out 指向运行长度输出序列的指针 (每次运行一个计数)
- [out] d_num_runs_out 指向运行总数的指针
- [in] num_items 关联的键值对总数 (d_in_keys 和 d_in_values 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceRunLengthEncode::NonTrivialRuns

```
template<typename InputIteratorT , typename OffsetsOutputIteratorT ,
 ↪typename LengthsOutputIteratorT , typename NumRunsOutputIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
 ↪cub::DeviceRunLengthEncode::NonTrivialRuns
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OffsetsOutputIteratorT      d_offsets_out,
     LengthsOutputIteratorT      d_lengths_out,
     NumRunsOutputIteratorT      d_num_runs_out,
     int      num_items,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

枚举序列 d_in 中相同值键的所有非三维运行（长度 > 1）的起始偏移量和长度。

- 对于第 i 次非三维运行，运行的起始偏移量及其长度分别写入 d_offsets_out[i] 和 d_lengths_out[i]。
- 遇到的运行总数将写入 d_num_runs_out。
- == 等式运算符用于确定值是否相等。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面的代码片段说明了 int 值序列中非三维运行的标识。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_run_length_encode.cuh>
//声明、分配和初始化设备可访问的输入和输出指针。
int           num_items;          // 例如，8
int           *d_in;              // 例如，[0, 2, 2, 9, 5, 5, 5, 8]
int           *d_offsets_out;     // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int           *d_lengths_out;     // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int           *d_num_runs_out;    // 例如，[ ]
...
// 确定临时存储设备需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceRunLengthEncode::NonTrivialRuns(d_temp_storage, temp_storage_
↪bytes, d_in, d_offsets_out, d_lengths_out, d_num_runs_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行代码
cub::DeviceRunLengthEncode::NonTrivialRuns(d_temp_storage, temp_storage_
↪bytes, d_in, d_offsets_out, d_lengths_out, d_num_runs_out, num_items);
// d_offsets_out         <-- [1, 4]
// d_lengths_out         <-- [2, 3]
// d_num_runs_out        <-- [2]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项 (可能是简单的指针类型)
- OffsetsOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入运行偏移值 (可能是简单的指针类型)
- LengthsOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入运行长度值 (可能是简单的指针类型)
- NumRunsOutputIteratorT **[推断]** 输出迭代器类型，用于记录遇到的运行次数 (可能是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针
- [out] d_offsets_out 指向运行偏移输出序列的指针 (每非三维运行一个偏移)
- [out] d_lengths_out 指向运行长度输出序列的指针 (每非三维运行一个计数)
- [out] d_num_runs_out 指向运行总数的指针 (即 d_offsets_out 的长度)
- [in] num_items 关联的键值对总数 (即 d_in_keys 和 d_in_values 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::ExclusiveSum

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::ExclusiveSum
(    void *      d_temp_storage,
```

(下页继续)

(续上页)

```
    size_t &     temp_storage_bytes,
    InputIteratorT     d_in,
    OutputIteratorT     d_out,
    int     num_items,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

计算设备范围的排除前缀总和。初始值为 0，并分配给 *d_out。

- 支持非交换和运算符。
- 伪关联运算符的结果不具有确定性 (例如，添加浮点类型)。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面代码片段说明了 int 设备向量的排除前缀和。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_scan.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针。
int  num_items;      // 例如，7
int  *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // 例如，[ ,  ,  ,  ,  ,  ,  ]
...
// 确定临时存储设备需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行代码
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, num_items);
// d_out s<-- [0, 8, 14, 21, 26, 29, 29]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描输入 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入扫描输出 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 数据项输入序列的随机访问迭代器
- [out] d_out 数据项输出序列的随机访问迭代器
- [in] num_items 输入项总数 (d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::ExclusiveScan

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪ScanOpT , typename InitValueT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::ExclusiveScan
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      ScanOpT      scan_op,
      InitValueT      init_value,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用指定的二进制函数对象 scan_op 计算设备范围内的排除前缀扫描。初始值为 init_value 值，并赋值给 *d_out。

- 支持非交换扫描运算符。
- 伪关联运算符的结果不具有确定性 (例如，添加浮点类型)。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码片段说明了 int 设备向量的排除前缀最小扫描。

```
#include <cub/cub.cuh>    // 或<cub/device/device_scan.cuh>
#include <climits>        // 用于 INT_MAX
// CustomMin 函数
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针。
int           num_items;      // 例如，7
int           *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int           *d_out;         // 例如，[ ,  ,  ,  ,  ,  ,  ]
CustomMin     min_op;
...
// 为排除前缀扫描确定临时存储分配需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveScan(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, min_op, (int) INT_MAX, num_items);
// 为排除前缀扫描分配内存
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排除前缀扫描
cub::DeviceScan::ExclusiveScan(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, min_op, (int) INT_MAX, num_items);
// d_out <-- [2147483647, 8, 6, 6, 5, 3, 0]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描输入 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入扫描输出 (可以是简单的指针类型)
- ScanOp **[推断]** 二元扫描函数对象类型，具有成员函数 T operator()(const T &a, const T &b)
- InitValueT **[推断]** 使用 init_value 数据类型的二元扫描函数对象类型，具有成员函数 T operator()(const T &a, const T &b)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 数据项输入序列的随机访问迭代器
- [out] d_out 数据项输出序列的随机访问迭代器
- [in] scan_op 二进制扫描函数对象
- [in] init_value 为排除扫描设定种子的初始值 (并分配给 *d_out)
- [in] num_items 输入项总数 (d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::ExclusiveScan

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪ScanOpT , typename InitValueT , typename InitValueIterT = InitValueT*>
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::ExclusiveScan
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      ScanOpT      scan_op,
      FutureValue< InitValueT, InitValueIterT >      init_value,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 指向数据项输出序列的指针
- [in] scan_op 二进制扫描函数对象
- [in] init_value 为排除扫描设定种子的初始值 (并分配给 *d_out)
- [in] num_items 输入项总数 (d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::InclusiveSum

```
template<typename InputIteratorT , typename OutputIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::InclusiveSum
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OutputIteratorT      d_out,
     int      num_items,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

计算设备范围的包含前缀和。

- 支持非交换和运算符。
- 伪关联运算符 (例如，添加浮点类型) 的结果不具有确定性。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码片段说明了 int 设备向量的包含前缀总和。

```
#include <cub/cub.cuh>    // 或<cub/device/device_scan.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针。
int  num_items;      // 例如，7
int  *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // 例如，[ ,  ,  ,  ,  ,  ,  ]
...
//为包含前缀和确定临时存储空间
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, num_items);
// 为包含前缀和分配空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行包含前缀和
cub::DeviceScan::InclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, num_items);
// d_out <-- [8, 14, 21, 26, 29, 29, 38]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描输入 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入扫描输出 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 数据项输入序列的随机访问迭代器
- [out] d_out 数据项输出序列的随机访问迭代器
- [in] num_items 输入项总数 (即 d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::InclusiveScan

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪ScanOpT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::InclusiveScan
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      ScanOpT      scan_op,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用指定的二进制函数对象 scan_op 计算设备范围的包含前缀扫描。

- 支持非交换扫描运算符。
- 伪关联运算符的结果不具有确定性 (例如，添加浮点类型)。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

下面代码片段说明了 int 设备向量的包含前缀最小扫描。

```
#include <cub/cub.cuh>   // 或<cub/device/device_scan.cuh>
#include <climits>       // 用于 INT_MAX
// CustomMin 函数对象
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针。
int          num_items;      // 例如，7
int          *d_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int          *d_out;         // 例如，[ ,  ,  ,  ,  ,  ,  ]
CustomMin    min_op;
...
// 为包含前缀和确定临时存储空间
void *d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceScan::InclusiveScan(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, min_op, num_items);
// 为包含前缀和分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行包含前缀和
cub::DeviceScan::InclusiveScan(d_temp_storage, temp_storage_bytes, d_in, d_
↪out, min_op, num_items);
// d_out <-- [8, 6, 6, 5, 3, 0, 0]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描输入 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入扫描输出 (可以是简单的指针类型)
- ScanOp **[推断]** 二元扫描函数对象类型，拥有成员函数 T operator()(const T &a, const T &b)。

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_in 数据项输入序列的随机访问迭代器
- [out] d_out 数据项输出序列的随机访问迭代器
- [in] scan_op 二元扫描函数对象
- [in] num_items 输入项总数 (d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::ExclusiveSumByKey

```
template<typename KeysInputIteratorT , typename ValuesInputIteratorT ,
↪typename ValuesOutputIteratorT , typename EqualityOpT = Equality>
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::ExclusiveSumByKey
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      KeysInputIteratorT      d_keys_in,
      ValuesInputIteratorT      d_values_in,
      ValuesOutputIteratorT      d_values_out,
      int      num_items,
      EqualityOpT      equality_op = EqualityOpT(),
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

计算设备范围的排除前缀按键和，其键的相等性由 equality_op 定义。初始值为 0，并分配给 d_values_out 中每个段的开头。

- 支持非交换和运算符。
- 伪关联运算符的结果不具有确定性 (例如，添加浮点类型)。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码片段

下面代码片段说明了 int 设备向量的排除前缀按键和。

```
#include <cub/cub.cuh>    // 或<cub/device/device_scan.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针。
int num_items;       // 例如，7
int *d_keys_in;      // 例如，[0, 0, 1, 1, 1, 2, 2]
int *d_values_in;    // 例如，[8, 6, 7, 5, 3, 0, 9]
int *d_values_out;   // 例如，[ ,  ,  ,  ,  ,  ,  ]
```

(下页继续)

(续上页)

```
...
// 确定临时存储空间
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveSumByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排除前缀和
cub::DeviceScan::ExclusiveSumByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, num_items);
// d_values_out <-- [0, 8, 0, 7, 12, 0, 0]
```

模板参数

- KeysInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描键输入（可以是简单的指针类型）
- ValuesInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取扫描值输入（可以是简单的指针类型）
- ValuesOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入扫描值输出（可以是简单的指针类型）
- EqualityOpT **[推断]** 对于定义键相等性的二元运算，具有成员函数 T operator()(const T &a, const T &b) 的函数对象类型

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为空指针时，所需的分配大小将被写入 temp_storage_bytes ，并且不会执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配的大小 (以字节为单位)
- [in] d_keys_in 关键项输入序列的随机访问输入迭代器
- [in] d_values_in 对值项的输入序列的随机访问输入迭代器
- [out] d_values_out 值项输出序列的随机访问输出迭代器
- [in] num_items 输入项的总数（即 d_keys_in 和 d_values_in 的长度）
- [in] equality_op 定义键相等的二进制函数对象。默认值为 cub：：Equal（）。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步数据流用以检查是否有错误，可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::ExclusiveScanByKey

```
template<typename KeysInputIteratorT , typename ValuesInputIteratorT ,
↪typename ValuesOutputIteratorT , typename ScanOpT , typename InitValueT ,
↪ typename EqualityOpT = Equality>
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::ExclusiveScanByKey
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      KeysInputIteratorT      d_keys_in,
      ValuesInputIteratorT      d_values_in,
      ValuesOutputIteratorT      d_values_out,
      ScanOpT      scan_op,
      InitValueT      init_value,
```

(下页继续)

(续上页)

```
    int     num_items,
    EqualityOpT     equality_op = EqualityOpT(),
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用指定的二进制函数对象 can_op 计算设备范围的排除按键前缀和。键相等由 equality_op 定义。将 init_value 值作为初始值，并分配给 d_values_out 中每个段的开头。

- 支持非交换扫描运算符。
- 伪关联运算符的结果不具有确定性 (例如，添加浮点类型)。伪关联运算符的结果可能因运行而异。
- 当 d_temp_storage 为 NULL 时，不执行任何操作，并将所需的分配大小返回在 temp_storage_bytes 中。

代码段

以下代码段说明了对 int 设备向量进行排除前缀最小扫描 (exclusive prefix min-scan-by-key) 的过程。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_scan.cuh>
#include <climits>       // 使用其中的 INT_MAX
// 定义 CustomMin 函数对象
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 定义 CustomEqual 函数对象
struct CustomEqual
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return a == b;
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针
int           num_items;      // 例如，7
int           *d_keys_in;     // 例如，[0, 0, 1, 1, 1, 2, 2]
int           *d_values_in;   // 例如，[8, 6, 7, 5, 3, 0, 9]
int           *d_values_out;  // 例如，[ ,  ,  ,  ,  ,  ,  ]
CustomMin     min_op;
CustomEqual   equality_op;
...
// 为排除前缀扫描 (exclusive prefix scan) 确定临时设备存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveScanByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, min_op, (int) INT_MAX, num_items,
↪equality_op);
// 为排除前缀扫描分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行排除前缀最小扫描
cub::DeviceScan::ExclusiveScanByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, min_op, (int) INT_MAX, num_items,
↪equality_op);
```

(下页继续)

(续上页)

```
// d_values_out <-- [2147483647, 8, 2147483647, 7, 5, 2147483647, 0]
```

模板参数

- KeysInputIteratorT **[推断]** 用于读取扫描键输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesInputIteratorT **[推断]** 用于读取扫描值输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesOutputIteratorT **[推断]** 用于写入扫描值输出的随机访问输出迭代器类型（可以是一个简单的指针类型）
- ScanOp **[推断]** 具有成员函数 T operator() 的二进制扫描函数对象 (Binary scan functor) 类型 ()(const T &a, const T &b)
- InitValueT **[推断]** 在具有成员函数 T operator() 的二进制扫描函数对象类型中使用的 init_value 的类型 (const T &a, const T &b)
- EqualityOpT **[推断]** 用于明确键相等性的二进制操作中具有成员函数 T operator() 的函数对象类型 (const T &a, const T &b)

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入参数 temp_storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 参数 d_temp_storage 分配中字节大小的引用
- [in] d_keys_in 具备键项输入序列的随机访问输入迭代器
- [in] d_values_in 具备值项输入序列的随机访问输入迭代器
- [out] d_values_out 具备值项输出序列的随机访问输出迭代器
- [in] scan_op 二进制扫描函数对象
- [in] init_value 用于执行排除扫描操作的初始值（并被分配给 d_values_out 中每个段的开头位置）
- [in] num_items 输入项的总数（也可以说是 d_keys_in 和 d_values_in 的长度）
- [in] equality_op 定义了键相等性的二元函数对象。默认情况下, 使用的函数对象是 cub::Equality()。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::InclusiveSumByKey

```
template<typename KeysInputIteratorT , typename ValuesInputIteratorT ,
↪typename ValuesOutputIteratorT , typename EqualityOpT = Equality>
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::InclusiveSumByKey
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      KeysInputIteratorT      d_keys_in,
      ValuesInputIteratorT      d_values_in,
      ValuesOutputIteratorT      d_values_out,
      int      num_items,
      EqualityOpT      equality_op = EqualityOpT(),
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

计算设备范围的包含前缀键值求和（inclusive prefix sum-by-key），其中键的相等性由 equality_op 定义。

- 支持非交换和运算符。
- 当涉及到伪关联运算符（例如，浮点类型的加法）时，结果是不确定的。伪关联运算符的结果可能会在每次运行时有所不同。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量进行包含前缀键值求和 (inclusive prefix sum-by-key) 的过程。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_scan.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int num_items;       // 例如，7
int *d_keys_in;      // 例如，[0, 0, 1, 1, 1, 2, 2]
int *d_values_in;    // 例如，[8, 6, 7, 5, 3, 0, 9]
int *d_values_out;   // 例如，[ ,  ,  ,  ,  ,  ,  ]
...
// 确定包含前缀键值求和算法的临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceScan::InclusiveSumByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, num_items);
// 为包含前缀键值求和算法分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行包含前缀键值求和算法
cub::DeviceScan::InclusiveSumByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, num_items);
// d_out <-- [8, 14, 7, 12, 15, 0, 9]
```

模板参数

- KeysInputIteratorT **[推断]** 用于读取扫描键输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesInputIteratorT **[推断]** 用于读取扫描值输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesOutputIteratorT **[推断]** 用于写入扫描值输出的随机访问输出迭代器类型（可以是一个简单的指针类型）
- EqualityOpT **[推断]** 用于明确键相等性的二进制操作中具有成员函数 T operator() 的函数对象类型 (const T &a, const T &b)

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入 temp_storage_bytes 并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配中字节大小的引用
- [in] d_keys_in 具备键项输入序列的随机访问输入迭代器
- [in] d_values_in 具备值项输入序列的随机访问输入迭代器
- [out] d_values_out 具备值项输出序列的随机访问输出迭代器
- [in] num_items 输入项的总数（也可以说是 d_keys_in 和 d_values_in 的长度）
- [in] equality_op 定义了键相等性的二元函数对象。默认情况下，使用的函数对象是 cub::Equality()。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。

- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceScan::InclusiveScanByKey

```
template<typename KeysInputIteratorT , typename ValuesInputIteratorT ,
↪typename ValuesOutputIteratorT , typename ScanOpT , typename EqualityOpT
↪= Equality>
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceScan::InclusiveScanByKey
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     KeysInputIteratorT      d_keys_in,
     ValuesInputIteratorT      d_values_in,
     ValuesOutputIteratorT      d_values_out,
     ScanOpT      scan_op,
     int      num_items,
     EqualityOpT      equality_op = EqualityOpT(),
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用指定的二进制扫描 _op 函数计算设备范围内的包含前缀扫描。键的相等性由 equality_op 定义。

- 支持非交换扫描运算符。
- 当涉及到伪关联运算符（例如，浮点类型的加法）时，结果是不确定的。伪关联运算符的结果可能会因运行的不同而不同。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量进行包含前缀最小扫描的过程。

```
#include <cub/cub.cuh>   // 或  <cub/device/device_scan.cuh>
#include <climits>       // 使用其中的 INT_MAX
// 定义 CustomMin 函数对象
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 定义 CustomEqual 函数对象
struct CustomEqual
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return a == b;
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针
int          num_items;      // 例如，7
int          *d_keys_in;     // 例如，[0, 0, 1, 1, 1, 2, 2]
```

(下页继续)

(续上页)

```
int          *d_values_in;   // 例如，[8, 6, 7, 5, 3, 0, 9]
int          *d_values_out;  // 例如，[ ,  ,  ,  ,  ,  ,  ]
CustomMin    min_op;
CustomEqual  equality_op;
...
// 确定包含前缀扫描的临时设备存储需求
void *d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceScan::InclusiveScanByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, min_op, num_items, equality_op);
// 为包含前缀扫描分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行包含前缀最小扫描
cub::DeviceScan::InclusiveScanByKey(d_temp_storage, temp_storage_bytes, d_
↪keys_in, d_values_in, d_values_out, min_op, num_items, equality_op);
// d_out <-- [8, 6, 7, 5, 3, 0, 0]
```

模板参数

- KeysInputIteratorT **[推断]** 用于读取扫描键输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesInputIteratorT **[推断]** 用于读取扫描值输入的随机访问输入迭代器类型（可以是一个简单的指针类型）
- ValuesOutputIteratorT **[推断]** 用于写入扫描值输出的随机访问输出迭代器类型（可以是一个简单的指针类型）
- ScanOp **[推断]** 拥有成员函数 T operator() 的二进制扫描函数对象类型 (const T &a, const T &b)
- EqualityOpT **[推断]** 用于定义键相等性比较的二元操作中具有成员函数 T operator() 的函数对象类型 (const T &a, const T &b)

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入 temp_storage_bytes 并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配中字节大小的引用
- [in] d_keys_in 具备键项输入序列的随机访问输入迭代器
- [in] d_values_in 具备值项输入序列的随机访问输入迭代器
- [out] d_values_out 具备值项输出序列的随机访问输出迭代器
- [in] scan_op 二进制扫描函数对象
- [in] num_items 输入项的总数（也可以说是 d_keys_in 和 d_values_in 的长度）
- [in] equality_op 定义了键相等性的二元函数对象。默认情况下，使用的函数对象是 cub::Equality()。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceSelect::Flagged

```
template<typename InputIteratorT , typename FlagIterator , typename
↪OutputIteratorT , typename NumSelectedIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DeviceSelect::Flagged
```

(下页继续)

(续上页)

```
(    void *     d_temp_storage,
     size_t &     temp_storage_bytes,
     InputIteratorT     d_in,
     FlagIterator     d_flags,
     OutputIteratorT     d_out,
     NumSelectedIteratorT     d_num_selected_out,
     int     num_items,
     mcStream_t     stream = 0,
     bool     debug_synchronous = false
)
```

使用 d_flags 序列有选择性地将对应的项从 d_in 复制到 d_out 中。所选中的项目的总数将写入 d_num_selected_out。

- d_flags 的值类型必须能够转换为布尔类型（例如，bool、char、int 等）。
- 所选项的副本被压缩到 d_out 中，并保持其原始的相对顺序。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量中所选项目的压缩过程。

```
#include <cub/cub.cuh>        // 或 <cub/device/device_select.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;               // 例如，8
int  *d_in;                   // 例如，[1, 2, 3, 4, 5, 6, 7, 8]
char *d_flags;                // 例如，[1, 0, 0, 1, 0, 1, 1, 0]
int  *d_out;                  // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int  *d_num_selected_out;     // 例如，[ ]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSelect::Flagged(d_temp_storage, temp_storage_bytes, d_in, d_
↪flags, d_out, d_num_selected_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行所选项
cub::DeviceSelect::Flagged(d_temp_storage, temp_storage_bytes, d_in, d_
↪flags, d_out, d_num_selected_out, num_items);
// d_out                 <-- [1, 4, 6, 7]
// d_num_selected_out    <-- [4]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型（可以是一个简单的指针类型）
- FlagIterator **[推断]** 用于读取选择标志的随机访问输入迭代器类型（可以是一个简单的指针类型）
- OutputIteratorT **[推断]** 用于写入所选项的随机访问输出迭代器类型（可以是一个简单的指针类型）
- NumSelectedIteratorT **[推断]** 用于记录所选项目数量的输出迭代器类型（可以是一个简单的指针类型）

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入 temp_storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes d_temp_storage 分配中字节大小的引用

- [in] d_in 指向数据项输入序列的指针
- [in] d_flags 选择标志输入序列的指针
- [out] d_out 选定的数据项输出序列的指针
- [out] d_num_selected_out 指向输出所选项目总数的指针 (即 d_out 的长度)
- [in] num_items 输入项的总数 (即 d_in 的长度)
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认情况下，使用的流是 stream0。
- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceSelect::If

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪NumSelectedIteratorT , typename SelectOp >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t cub::DeviceSelect::If
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      NumSelectedIteratorT      d_num_selected_out,
      int      num_items,
      SelectOp      select_op,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用 select_op 函数对象将 d_in 中的项有选择地复制到 d_out 中。所选项目的总数被写入到 d_num_selected_out 中。

- 所选项的副本被压缩到 d_out 中，并保持其原始的相对顺序。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量中所选项目的压缩过程。

```
#include <cub/cub.cuh>    // 或 <cub/device/device_select.cuh>
// 用于选择小于某个条件的值的函数对象类型
struct LessThan
{
    int compare;
    CUB_RUNTIME_FUNCTION __forceinline__
    LessThan(int compare) : compare(compare) {}
    CUB_RUNTIME_FUNCTION __forceinline__
    bool operator()(const int &a) const {
        return (a < compare);
    }
};
// 声明、分配和初始化设备可访问的输入和输出指针
int      num_items;              // 例如，8
int      *d_in;                  // 例如，[0, 2, 3, 9, 5, 2, 81, 8]
int      *d_out;                 // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int      *d_num_selected_out;    // 例如，[ ]
LessThan select_op(7);
```

(下页继续)

(续上页)

```
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceSelect::If(d_temp_storage, temp_storage_bytes, d_in, d_out, d_
↪num_selected_out, num_items, select_op);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行所选项
cub::DeviceSelect::If(d_temp_storage, temp_storage_bytes, d_in, d_out, d_
↪num_selected_out, num_items, select_op);
// d_out                  <-- [0, 2, 3, 5, 2]
// d_num_selected_out     <-- [5]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型（可以是一个简单的指针类型）
- OutputIteratorT **[推断]** 用于写入选定项的随机访问输出迭代器类型（可以是一个简单的指针类型）
- NumSelectedIteratorT **[推断]** 用于记录所选项目数量的输出迭代器类型（可以是一个简单的指针类型）
- SelectOp **[推断]** 选择具有成员函数 bool operator()(const T& a) 的运算符类型

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入 temp_storage_bytes 并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配中字节大小的引用
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 选定的数据项输出序列的指针
- [out] d_num_selected_out 指向输出所选项目总数的指针 (即 d_out 的长度)
- [in] num_items 输入项的总数 (即 d_in 的长度)
- [in] select_op 一元选择运算符
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceSelect::Unique

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪NumSelectedIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DeviceSelect::Unique
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      NumSelectedIteratorT      d_num_selected_out,
      int      num_items,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

给定的一个包含连续相等键值运行的输入序列 d_in，只有每个运行的第一个键被有选择性地复制到 d_out 中。所选项的总数被写入到 d_num_selected_out 中。

- 使用 == 等式运算符来判断键值是否相等。
- 所选项的副本被压缩到 d_out 中，并保持其原始的相对顺序。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量中所选项目的压缩过程。

```
#include <cub/cub.cuh>        // 或 <cub/device/device_select.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;               // 例如，8
int  *d_in;                   // 例如，[0, 2, 2, 9, 5, 5, 5, 8]
int  *d_out;                  // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int  *d_num_selected_out;     // 例如，[ ]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSelect::Unique(d_temp_storage, temp_storage_bytes, d_in, d_out,
 ↪d_num_selected_out, num_items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行所选项
cub::DeviceSelect::Unique(d_temp_storage, temp_storage_bytes, d_in, d_out,
 ↪d_num_selected_out, num_items);
// d_out                 <-- [0, 2, 9, 5, 8]
// d_num_selected_out    <-- [5]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型（可以是一个简单的指针类型）
- OutputIteratorT **[推断]** 用于写入选定项的随机访问输出迭代器类型（可以是一个简单的指针类型）
- NumSelectedIteratorT **[推断]** 用于记录所选项目数量的输出迭代器类型（可以是一个简单的指针类型）

参数

- [in] d_temp_storage 可供设备访问的临时储存空间分配。当参数值为 NULL 时，将所需的储存空间大小写入 temp_storage_bytes 并且不执行任何操作。
- [in,out] temp_storage_bytes d_temp_storage 分配中字节大小的引用
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 选定的数据项输出序列的指针
- [out] d_num_selected_out 指向输出所选项目总数的指针（即 d_out 的长度）
- [in] num_items 输入项的总数（即 d_in 的长度）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 对于每次内核启动之后是否进行流同步以检查错误。这可能会导致显著的性能下降。默认值为 false。

cub::DeviceSelect::UniqueByKey

```
template<typename KeyInputIteratorT , typename ValueInputIteratorT ,
↪typename KeyOutputIteratorT , typename ValueOutputIteratorT , typename
↪NumSelectedIteratorT >
CUB_RUNTIME_FUNCTION static __forceinline__ mcError_t
↪cub::DeviceSelect::UniqueByKey
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     KeyInputIteratorT      d_keys_in,
     ValueInputIteratorT      d_values_in,
     KeyOutputIteratorT      d_keys_out,
     ValueOutputIteratorT      d_values_out,
     NumSelectedIteratorT      d_num_selected_out,
     int      num_items,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

给定一次包含连续相等键值运行的输入序列 d_keys_in 和 d_values_in，只有每个运行的第一个键和对应的值被有选择性地复制到 d_keys_out 和 d_values_out 中。所选项的总数被写入到 d_num_selected_out 中。

- 使用 == 等式运算符来判断键值是否相等。
- 所选项的副本被压缩到 d_out 中，并保持其原始的相对顺序。
- 当 d_temp_storage 为 NULL 时，不会执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

以下代码段说明了对 int 设备向量中所选项目的压缩过程。

```
#include <cub/cub.cuh>        // 或 <cub/device/device_select.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;               // 例如，8
int  *d_keys_in;              // 例如，[0, 2, 2, 9, 5, 5, 5, 8]
int  *d_values_in;            // 例如，[1, 2, 3, 4, 5, 6, 7, 8]
int  *d_keys_out;             // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int  *d_values_out;           // 例如，[ ,  ,  ,  ,  ,  ,  ,  ]
int  *d_num_selected_out;     // 例如，[ ]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSelect::UniqueByKey(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_values_in, d_keys_out, d_values_out, d_num_selected_out, num_
↪items);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行所选项
cub::DeviceSelect::UniqueByKey(d_temp_storage, temp_storage_bytes, d_keys_
↪in, d_values_in, d_keys_out, d_values_out, d_num_selected_out, num_
↪items);
// d_keys_out            <-- [0, 2, 9, 5, 8]
// d_values_out          <-- [1, 2, 4, 5, 8]
// d_num_selected_out    <-- [5]
```

模板参数

- KeyInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入键（可以是简单的指针类型）
- ValueInputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入值（可以是简单的指针类型）
- KeyOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入选定键（可以是简单的指针类型）
- ValueOutputIteratorT **[推断]** 随机访问输出迭代器类型，用于写入选定值（可以是简单的指针类型）
- NumSelectedIteratorT **[推断]** 输出迭代器类型，用于记录所选项目数量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 指针，指向键的输入序列
- [in] d_values_in 指针，指向值的输入序列
- [out] d_keys_out 指针，指向所选键的输出序列
- [out] d_keys_out 指针，指向所选值的输出序列
- [out] d_num_selected_out 指针，指向所选项目总数（即 d_keys_out 或 d_values_out 的长度）
- [in] num_items 输入项目总数（即 d_keys_in 或 d_values_in 的长度）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。检查错误可能使启动明显变慢。默认值为 false。

cub::DeviceSpmv::CsrMV

```
template<typename ValueT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSpmv::CsrMV
(     void *      d_temp_storage,
      size_t &        temp_storage_bytes,
      const ValueT *      d_values,
      const int *     d_row_offsets,
      const int *     d_column_indices,
      const ValueT *      d_vector_x,
      ValueT *      d_vector_y,
      int     num_rows,
      int     num_cols,
      int     num_nonzeros,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

这个函数执行矩阵向量运算 y = A*x。

代码段

下面的代码段说明了 9x9 CSR 格式的矩阵 A 上的稀疏矩阵向量乘法（SpMV），该矩阵（24 个非零）表示 3x3 格点。

```
#include <cub/cub.cuh>    // 或<cub/device/device_spmv.cuh>
//为输入矩阵 A、输入向量 x 和输出向量 y
//声明、分配和初始化设备可访问指针
int     num_rows = 9;
int     num_cols = 9;
```

(下页继续)

(续上页)

```
int    num_nonzeros = 24;
float* d_values;  // 例如, [1, 1, 1, 1, 1, 1, 1, 1,
                  //     1, 1, 1, 1, 1, 1, 1, 1,
                  //     1, 1, 1, 1, 1, 1, 1, 1]
int*   d_column_indices; // 例如, [1, 3, 0, 2, 4, 1, 5, 0,
                         //     4, 6, 1, 3, 5, 7, 2, 4,
                         //     8, 3, 7, 4, 6, 8, 5, 7]
int*   d_row_offsets;    // 例如, [0, 2, 5, 7, 10, 14, 17, 19, 22, 24]
float* d_vector_x;       // 例如, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
float* d_vector_y;       // 例如, [ ,  ,  ,  ,  ,  ,  ,  ,  ,  ]
...
//确定临时设备存储需求
void*    d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSpmv::CsrMV(d_temp_storage, temp_storage_bytes, d_values,
    d_row_offsets, d_column_indices, d_vector_x, d_vector_y,
    num_rows, num_cols, num_nonzeros);
//分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 运行 SpMV
cub::DeviceSpmv::CsrMV(d_temp_storage, temp_storage_bytes, d_values,
    d_row_offsets, d_column_indices, d_vector_x, d_vector_y,
    num_rows, num_cols, num_nonzeros);
// d_vector_y <-- [2, 3, 2, 3, 4, 3, 2, 3, 2]
```

模板参数

- ValueT **[推断]** 矩阵和向量的值类型（例如，/p float, /p double 等）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_values 指针，指向矩阵 A 中相应非零元素的值数组。
- [in] d_row_offsets 指针，指向由 m + 1 个偏移量组成的数组，这些偏移量指示了矩阵每一行非零元素在 d_column_indices 和 d_values 中的起始位置（最后一项等于 num_nonzeros）。
- [in] d_column_indices 指针，指向矩阵 A 中非零元素的列索引数组（索引从 0 开始）。
- [in] d_vector_x 指针，指向密集输入向量 x（长度为 num_cols 的数组）
- [out] d_vector_y 指针，指向密集输出向量 y（长度为 num_rows 的数组）
- [in] num_rows 矩阵 A 的行数。
- [in] num_cols 矩阵 A 的列数。
- [in] num_nonzeros 矩阵 A 的非零元素的个数。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。检查错误可能使启动明显变慢。默认值为 false。

2.2.2 分段问题（批处理）（Segmented-problem (batch)）

2.2.2.1 类（Classes）

cub::DeviceSegmentedRadixSort

DeviceSegmentedRadixSort 提供设备范围的并行操作，用于在设备可访问内存中对多个非重叠数据序列进行批量基数排序。

概述

基数排序算法根据键的位置将项目按升序（或降序）排列。基数排序算法依赖于键的一个有序符号（例如数字、字符等）序列，这些符号从最低有效位到最高有效位按顺序排列。对于给定的键输入序列和一组规则来指定符号字母表的总排序顺序，基数排序算法将产生相应键的字典排序。DeviceSegmentedRadixSort 使用与 DeviceRadixSort 相同的实现方法。

使用注意事项

动态并行。可以在支持 MXMACA 动态并行的设备的内核代码中调用 DeviceSegmentedRadixSort 方法。

cub::DeviceSegmentedReduce

DeviceSegmentedReduce 提供设备范围的并行操作，用于在设备可访问内存中对多个数据项序列进行归约计算。

概述

归约（或折叠）使用二元组合算子从输入元素序列中计算出单个聚合值。

使用注意事项

动态并行。可以在支持 MXMACA 动态并行的设备的内核代码中调用 DeviceSegmentedRadixSort 方法。

cub::DeviceSegmentedSort

DeviceSegmentedRadixSort 提供设备范围的并行操作，用于在设备可访问内存中对多个非重叠数据项序列进行批量基数排序。

概述

该算法将项目按升序（或降序）排列。底层排序算法未定义。根据段的大小，它可能是基数排序、归并排序或其他排序。因此，不应该对底层实现做任何假设。

DeviceSegmentedRadixSort 的不同之处

DeviceSegmentedRadixSort 优化了大规模段（数万个或更多项）的排序。然而，有些域产生的分段大小范围很广。DeviceSegmentedSort 将段分成大小组，并为每个组专门设计排序算法。这种方法可以更好地利用资源，以应对段大小不平衡问题或使段大小适中（最多数千个项）。该算法更复杂，由多个内核组成。这导致编译时间较长且二进制文件大小也更大了。

支持的类型

该算法必须满足底层算法的限制。基数排序的使用限制了支持的类型。因此，DeviceSegmentedSort 可以对所有内置的 C++ 数字基本类型（unsigned char, int, double 等）以及 MXMACA 的 __half 和 __nv_bfloat16 16 位浮点类型进行排序。

简单示例

```
#include <cub/cub.cuh>
// 或者<cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
```

(下页继续)

(续上页)

```
int  *d_keys_out;         // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;        // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;       // 例如，[-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
// d_values_out          <-- [1, 2, 0, 5, 4, 3, 6]
```

2.2.2.2 函数（Functions）

cub::DeviceSegmentedRadixSort::SortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortPairs
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     const KeyT *      d_keys_in,
     KeyT *      d_keys_out,
     const ValueT *      d_values_in,
     ValueT *      d_values_out,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     int      begin_bit = 0,
     int      end_bit = sizeof(KeyT) * 8,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键值对段按升序排序。（需要约 2N 的辅助存储空间）

- 排序操作不会改变输入数据的内容
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- 可以指定区分键的位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提高相应的性能。
- 该操作需要为设备分配临时存储空间，复杂度为 O(N+P)，其中 N 为输入长度，P 为设备上运行的流多处理器数量。如果希望只使用 O(P) 临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。

- 当 d_temp_storage 为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段 (其中一个为零长度段) 的整型键值对进行批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针。
int  num_items;           // 例如，7
int  num_segments;        // 例如，3
int  *d_offsets;          // 例如，[0, 3, 3, 7]
int  *d_keys_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;         // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;        // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;       // 例如，[-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortPairs(d_temp_storage, temp_storage_
↪bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortPairs(d_temp_storage, temp_storage_
↪bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
// d_values_out          <-- [1, 2, 0, 5, 4, 3, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段起始偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段结束偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问指针，指向要排序的输入键数据
- [out] d_keys_out 设备可访问指针，指向排序后的键数据输出序列
- [in] d_values_in 设备可访问指针，指向相应输入值序列
- [out] d_values_out 设备可访问指针，指向相应重新排序的输出值序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素

- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 数据段为空。
- [in] begin_bit **[可选]** 进行键比较所需的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较所需的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。将启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortPairs
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键值对段按升序排序。（需要约 N 辅助存储器）

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变每对缓冲区内的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- 可以指定一个可选位范围 [begin_bit, end_bit) 来区分键位。这可以减少总体排序开销，并提高相应的性能。
- 此操作需要分配相对较小的临时设备存储空间，复杂度为 O(P)，其中 P 是设备上的流多处理器数量（通常相对于输入长度 N 是一个小常数）。
- 当 d_temp_storage 为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针。
int  num_items;          // 例如，7
```

(下页继续)

(续上页)

```
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffer 实例来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortPairs(d_temp_storage, temp_storage_
↪bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortPairs(d_temp_storage, temp_storage_
↪bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
// d_values.Current()    <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段起始偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段结束偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入键，并且在返回时更新缓冲区指向已排序的输出键
- [in,out] d_values 值的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入值，并且在返回时更新缓冲区指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 数据段为空。
- [in] begin_bit **[可选]** 进行键比较所需的最低有效位索引（包含最低有效位）

- [in] end_bit **[可选]** 进行键比较所需的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。将启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortPairsDescending
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     const KeyT *      d_keys_in,
     KeyT *      d_keys_out,
     const ValueT *      d_values_in,
     ValueT *      d_values_out,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     int      begin_bit = 0,
     int      end_bit = sizeof(KeyT) * 8,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键值对段按降序排序。（需要约 2N 的辅助存储空间）。

- 排序操作不会改变输入数据的内容
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- 可以指定一个可选位范围 [begin_bit, end_bit) 来区分键位。这可以减少总体排序开销，并提高相应的性能。
- 此操作需要分配临时设备存储空间，复杂度为 O(N+P)，其中 N 是输入的长度，P 是设备上流多处理器的数量。对于仅使用 O(P) 临时存储的排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针。
int  num_items;           // 例如，7
int  num_segments;        // 例如，3
int  *d_offsets;          // 例如，[0, 3, 3, 7]
int  *d_keys_in;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;         // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;        // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;       // 例如，[-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
```

(下页继续)

(续上页)

```
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortPairsDescending(d_temp_storage, temp_
↪storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortPairsDescending(d_temp_storage, temp_
↪storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [8, 7, 6, 9, 5, 3, 0]
// d_values_out          <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段起始偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段结束偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问指针，指向要排序的输入键数据
- [out] d_keys_out 设备可访问指针，指向排序后的键数据输出序列
- [in] d_values_in 设备可访问指针，指向相应输入值序列
- [out] d_values_out 设备可访问指针，指向相应重新排序的输出值序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 数据段为空。
- [in] begin_bit **[可选]** 进行键比较所需的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 键比较所需的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。将启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortPairsDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键值对段按降序排序。(需要约 N 的辅助存储空间)。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”(包含待排序的输入数据)。
- 排序操作可能会改变每对缓冲区内的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构)。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1)。
- 可以指定一个可选位范围 [begin_bit, end_bit) 来区分键位。这可以减少总体排序开销，并提高相应的性能。
- 此操作需要相对较小的 O(P) 临时设备存储分配，其中 P 是设备上的流多处理器数量（通常相对于输入长度 N 是一个小常数)。
- 当 d_temp_storage 为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化用于排序数据的设备可访问指针。
int  num_items;           // 例如，7
int  num_segments;        // 例如，3
int  *d_offsets;          // 例如，[0, 3, 3, 7]
int  *d_key_buf;          // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;      // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;        // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;    // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffer 实例来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
```

(下页继续)

(续上页)

```
cub::DeviceSegmentedRadixSort::SortPairsDescending(d_temp_storage, temp_
↪storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortPairsDescending(d_temp_storage, temp_
↪storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
// d_values.Current()    <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段起始偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段结束偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键的双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键，返回时更新为指向已排序的输出键
- [in,out] d_values 值的双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] begin_bit **[可选]** 进行键比较的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortKeys
```

(下页继续)

(续上页)

```
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按升序排序。（需要约 2N 的辅助存储空间）

- 排序操作不会改变输入数据的内容
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 当输入连续的段序列时，可以使用一个单一的序列 segment_offsets（长度为 num_segments+1）来为 d_begin_offsets 和 d_end_offsets 参数提供别名（其中后者被指定为 segment_offsets+1）。
- 这个操作需要分配临时的设备存储空间，其复杂度为 O(N+P)，其中 N 是输入数据的长度，P 是设备上的流多处理器的数量。对于只使用 O(P) 复杂度的临时存储空间进行排序，请参见下方使用 DoubleBuffer 封装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对三段整型键（其中一段为零长度段）的批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;           // 例如， 7
int  num_segments;        // 例如， 3
int  *d_offsets;          // 例如， [0, 3, 3, 7]
int  *d_keys_in;          // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;         // 例如， [-, -, -, -, -, -, -]
...
//确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage, temp_storage_bytes,
↪ d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage, temp_storage_bytes,
↪ d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型

- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问指针，指向要排序的输入键数据
- [out] d_keys_out 设备可访问指针，指向已排序键数据输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] begin_bit **[可选]** 进行键比较的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
 ↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
 ↪cub::DeviceSegmentedRadixSort::SortKeys
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     int      begin_bit = 0,
     int      end_bit = sizeof(KeyT) * 8,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键段按升序排序。（需要约 N 的辅助存储空间）

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变两个缓冲区的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。

- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 这个操作需要分配一个相对较小的临时设备存储空间，其复杂度为 O(P)，其中 P 是设备上的流多处理器的数量（通常相对于输入大小 N 而言是一个较小的常数）。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键进行批量排序。

```
#include <cub/cub.cuh>   // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_key_buf;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如， [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 实例来包装这一对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage, temp_storage_bytes,
↪ d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage, temp_storage_bytes,
↪ d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键的双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键，返回时更新为指向已排序的输出键
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素

- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] begin_bit **[可选]** 进行键比较的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortKeysDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按将序排序。（需要约 2N 的辅助存储空间）

- 排序操作不会改变输入数据的内容
- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 这个操作需要分配临时设备存储空间，其复杂度为 O(N+P)，其中 N 是输入的长度，P 是设备上的流多处理器数量。若要使用仅 O(P) 临时存储进行排序，请参见下面使用 DoubleBuffer 包装器的排序接口。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键进行批量排序。

```
#include <cub/cub.cuh>   // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;          // 例如，  7
int  num_segments;       // 例如，  3
int  *d_offsets;         // 例如，  [0, 3, 3, 7]
int  *d_keys_in;         // 例如，  [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，  [-, -, -, -, -, -, -]
```

(下页继续)

(续上页)

```
...
// 创建一个 DoubleBuffer 实例来包装这一对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortKeysDescending(d_temp_storage, temp_
↪storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortKeysDescending(d_temp_storage, temp_
↪storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问指针，指向要排序的输入键数据
- [out] d_keys_out 设备可访问指针，指向已排序键数据输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] begin_bit **[可选]** 进行键比较的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedRadixSort::SortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪cub::DeviceSegmentedRadixSort::SortKeysDescending
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      int      begin_bit = 0,
      int      end_bit = sizeof(KeyT) * 8,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按将序排序。（需要约 N 的辅助存储空间）

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会改变两个缓冲区的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 可以指定区分关键位的可选位范围 [begin_bit, end_bit)。这可以减少总体排序开销，并提升相应的性能。
- 这个操作需要分配一个相对较小的临时设备存储空间，其复杂度为 O(P)，其中 P 是设备上的流多处理器的数量（通常相对于输入大小 N 而言是一个较小的常数）。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键进行批量排序。

```
#include <cub/cub.cuh>    // 或<cub/device/device_segmented_radix_sort.cuh>
// 声明、分配和初始化设备可访问的输入和输出指针
int  num_items;           // 例如，  7
int  num_segments;        // 例如，  3
int  *d_offsets;          // 例如，  [0, 3, 3, 7]
int  *d_key_buf;          // 例如，  [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;      // 例如，  [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 实例来包装这一对设备指针
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortKeysDescending(d_temp_storage, temp_
↪storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
```

(下页继续)

(续上页)

```
// 执行排序操作
cub::DeviceSegmentedRadixSort::SortKeysDescending(d_temp_storage, temp_
↪storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键的双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键，返回时更新为指向已排序的输出键
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] begin_bit **[可选]** 进行键比较的最低有效位索引（包含最低有效位）
- [in] end_bit **[可选]** 进行键比较的最高有效位索引（不包含最高有效位）（例如，sizeof(unsigned int) * 8）
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::Reduce

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪BeginOffsetIteratorT , typename EndOffsetIteratorT , typename
↪ReductionOp , typename T >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::Reduce
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OutputIteratorT      d_out,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     ReductionOp      reduction_op,
     T      initial_value,
     mcStream_t      stream = 0,
```

(下页继续)

(续上页)

```
    bool      debug_synchronous = false
)
```

使用指定的二元 reduction_op 函数对象计算设备范围内的分段归约结果。

- 不支持不可交换的二元归约运算符。
- 在同一 GPU 设备上，对于伪关联性归约（例如，浮点类型的加法），每次运行相同的归约操作时，它会产生相同的结果。然而，因为 CUB 可以针对不同架构使用不同大小的块。那么在不同计算能力的不同设备上，伪关联性归约的结果可能不同。
- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备上的整型数据元素向量进行自定义最小值归约的过程。

```
#include <cub/cub.cuh>   // 或<cub/device/device_radix_sort.cuh>
// 定义 CustomMin 结构体
struct CustomMin
{
    template <typename T>
    CUB_RUNTIME_FUNCTION __forceinline__
    T operator()(const T &a, const T &b) const {
        return (b < a) ? b : a;
    }
};
// 声明、分配并初始化用于输入和输出的设备可访问指针
int          num_segments;   // 例如， 3
int          *d_offsets;     // 例如， [0, 3, 3, 7]
int          *d_in;          // 例如， [8, 6, 7, 5, 3, 0, 9]
int          *d_out;         // 例如， [-, -, -]
CustomMin    min_op;
int          initial_value;          // 例如， INT_MAX
...
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::Reduce(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1, min_op, initial_value);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行归约操作
cub::DeviceSegmentedReduce::Reduce(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1, min_op, initial_value);
// d_out <-- [6, INT_MAX, 0]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项（可以是简单的指针类型）
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录归约聚集（可以是简单的指针类型）
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）

- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）
- ReductionOp **[推断]** 具有成员函数 T operator()（const T &a, const T &b）的二元归约函数对象类型
- T **[推断]** 可转换为 InputIteratorT 值类型的数据元素类型

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_in 指针，指向输入数据项序列
- [out] d_out 指针，指向输出聚合结果
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] reduction_op 二元归约函数对象
- [in] initial_value 每个线段的归约初始值
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::Sum

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪BeginOffsetIteratorT , typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::Sum
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OutputIteratorT      d_out,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用加法运算符 (+) 计算设备范围内的分段和。

- 使用 0 作为每个线段归约的初始值。
- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 不支持非交换的 + 运算符。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备上的整型数据元素向量进行求和归约的过程。

```
#include <cub/cub.cuh>   // 或<cub/device/device_radix_sort.cuh>
// 声明、分配并初始化用于输入和输出的设备可访问指针
int num_segments;   // 例如， 3
int *d_offsets;     // 例如， [0, 3, 3, 7]
int *d_in;          // 例如， [8, 6, 7, 5, 3, 0, 9]
int *d_out;         // 例如， [-, -, -]
...
// 确定设备临时存储要求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::Sum(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行求和归约操作
cub::DeviceSegmentedReduce::Sum(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// d_out <-- [21, 0, 17]
```

模板参数

- InputIteratorT **[推断]** 随机访问输入迭代器类型，用于读取输入项（可以是简单的指针类型）
- OutputIteratorT **[推断]** 输出迭代器类型，用于记录归约聚集（可以是简单的指针类型）
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_in 指针，指向输入数据项序列
- [out] d_out 指针，指向输出聚合结果
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::Min

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪BeginOffsetIteratorT , typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::Min
```

(下页继续)

(续上页)

```
(    void *      d_temp_storage,
     size_t &      temp_storage_bytes,
     InputIteratorT      d_in,
     OutputIteratorT      d_out,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

使用小于运算符 (<) 计算设备范围内的分段最小值。

- 使用 std::numeric_limits<T>::max() 作为每个线段归约的初始值。
- 当输入连续的段序列时，可以使用单一序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 的别名（其中后者被指定为 segment_offsets+1）。
- 不支持不可交换的 < 运算符。
- 当 d_temp_storage 为 NULL 时，不会执行任何的操作并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备 int 型数据元素的最小归约。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问输入和输出数据的指针
int num_segments;    // 例如 3
int *d_offsets;      // 例如 [0, 3, 3, 7]
int *d_in;           // 例如 [8, 6, 7, 5, 3, 0, 9]
int *d_out;          // 例如 [-, -, -]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::Min(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行最小归约
cub::DeviceSegmentedReduce::Min(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// d_out <-- [6, INT_MAX, 0]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型 (可以是简单的指针类型)
- OutputIteratorT **[推断]** 用于记录归约聚合的输出迭代器类型 (可以是简单的指针类型)
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 指向输出聚合的指针
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::ArgMin

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪BeginOffsetIteratorT , typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::ArgMin
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用小于运算符 (<) 查找每个段中的第一个设备范围的最小值，并返回该项的段内索引。

- d_out 的输出值类型为 cub::KeyValuePair <int, T> (假设 d_in 的值类型为 T)
- 第 i 段的最小值写入 d_out[i] ，值和它的偏移量写入 d_out[i].key
- 为长度是零的输入生成 {1, std::numeric_limits<T>::max()} 元组
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets+1)
- 不支持非交换的 < 运算符
- 当 d_temp_storage 为 NULL 时，将不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备 int 型数据元素的最小索引归约。

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问输入和输出数据的指针
int                      num_segments;   // 例如 3
int                      *d_offsets;     // 例如 [0, 3, 3, 7]
int                      *d_in;          // 例如 [8, 6, 7, 5, 3, 0, 9]
```

(下页继续)

(续上页)

```
KeyValuePair<int, int>    *d_out;          // 例如 [{-,-}, {-,-}, {-,-}]
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::ArgMin(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行最小索引归约
cub::DeviceSegmentedReduce::ArgMin(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1);
// d_out <-- [{1,6}, {1,INT_MAX}, {2,0}]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型 (类型为 T)(可能是一个简单的指针类型)
- OutputIteratorT **[推断]** 用于记录归约聚合的输出迭代器类型 (具有值类型 KeyValuePair<int, T>)(可以是一个简单的指针类型)
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 指向输出聚合的指针
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器。其中 d_begin_offsets[i] 是第 i 个数据段在 d_keys_ 和 d_values_ 中的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器。其中 d_end_offsets[i] - 1 是第 i 个数据段在 d_keys_ 和 d_values_ 中的最后一个元素。如果 d_end_offsets[i] - 1 <= d_begin_offsets[i]，则认为第 i 个段为空
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::Max

```
template<typename InputIteratorT , typename OutputIteratorT , typename
↪BeginOffsetIteratorT , typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::Max
(    void *      d_temp_storage,
```

(下页继续)

(续上页)

```
    size_t &     temp_storage_bytes,
    InputIteratorT     d_in,
    OutputIteratorT     d_out,
    int     num_segments,
    BeginOffsetIteratorT     d_begin_offsets,
    EndOffsetIteratorT     d_end_offsets,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

使用大于运算符 (>) 计算设备范围的分段最大值。

- 使用 std::numeric_limits<T>::lowest() 作为归约的初始值。
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets+1)。
- 不支持非交换的 > 运算符。
- 当 d_temp_storage 为 NULL 时，将不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备 int 型数据元素的最大归约

```
#include <cub/cub.cuh>   // 或 <cub/device/device_radix_sort.cuh>
// 声明、分配和初始化设备可访问输入和输出数据的指针
int num_segments;    // 例如 3
int *d_offsets;      // 例如 [0, 3, 3, 7]
int *d_in;           // 例如 [8, 6, 7, 5, 3, 0, 9]
int *d_out;          // 例如 [-, -, -]
...
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::Max(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行最大归约
cub::DeviceSegmentedReduce::Max(d_temp_storage, temp_storage_bytes, d_in,
↪d_out,
    num_segments, d_offsets, d_offsets + 1);
// d_out <-- [8, INT_MIN, 9]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型 (可能是一个简单的指针类型)
- OutputIteratorT **[推断]** 用于记录归约聚合的输出迭代器类型 (可以是一个简单的指针类型)
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 指向输出聚合的指针
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

cub::DeviceSegmentedReduce::ArgMax

```
template<typename InputIteratorT , typename OutputIteratorT , typename
 ↪BeginOffsetIteratorT , typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t cub::DeviceSegmentedReduce::ArgMax
(     void *      d_temp_storage,
      size_t &      temp_storage_bytes,
      InputIteratorT      d_in,
      OutputIteratorT      d_out,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

使用大于运算符 (>) 查找每个段中的第一个设备范围最大值，并返回该项的段内索引。

- d_out 的输出值类型为 cub::KeyValuePair <int, T> (假设 d_in 的值类型为 T)
- 第 i 段的最大值写入 d_out[i] ，值和它的偏移量写入 d_out[i].key
- 为零长度输入生成 {1, std::numeric_limits<T>::max()} 元组
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets+1)
- 不支持非交换的 > 运算符
- 当 d_temp_storage 为 NULL 时，将不执行任何操作，并且在 temp_storage_bytes 中返回所需内存分配。

代码段

下面的代码段说明了对设备 int 型数据元素的最大索引归约的过程。

```
#include <cub/cub.cuh>    // 或 <cub/device/device_reduce.cuh>
// 声明、分配和初始化设备可访问输入和输出数据的指针
int                       num_segments;    // 例如 3
int                       *d_offsets;      // 例如 [0, 3, 3, 7]
int                       *d_in;           // 例如 [8, 6, 7, 5, 3, 0, 9]
```

(下页继续)

(续上页)

```
KeyValuePair<int, int>    *d_out;          // 例如 [{-,-}, {-,-}, {-,-}]
...
// 确定临时设备存储需求
void      *d_temp_storage = NULL;
size_t    temp_storage_bytes = 0;
cub::DeviceSegmentedReduce::ArgMax(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行最大索引归约
cub::DeviceSegmentedReduce::ArgMax(d_temp_storage, temp_storage_bytes, d_
↪in, d_out,
    num_segments, d_offsets, d_offsets + 1);
// d_out <-- [{0,8}, {1,INT_MIN}, {3,9}]
```

模板参数

- InputIteratorT **[推断]** 用于读取输入项的随机访问输入迭代器类型 (类型为 T)(可能是一个简单的指针类型)
- OutputIteratorT **[推断]** 用于记录归约聚合的输出迭代器类型 (具有值类型 KeyValuePair<int, T>)(可以是一个简单的指针类型)
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_in 指向数据项输入序列的指针
- [out] d_out 指向输出聚合的指针
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::SortKeys
(    void *      d_temp_storage,
```

(下页继续)

(续上页)

```
    std::size_t &      temp_storage_bytes,
    const KeyT *      d_keys_in,
    KeyT *      d_keys_out,
    int      num_items,
    int      num_segments,
    BeginOffsetIteratorT      d_begin_offsets,
    EndOffsetIteratorT      d_end_offsets,
    mcStream_t      stream = 0,
    bool      debug_synchronous = false
)
```

按升序对键所在的段进行排序。需要使用大小约为 num_items + 2*num_segments 的辅助存储空间。

- 排序操作不会改变输入数据的内容。
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets+1)。
- SortKeys 不能保证稳定。也就是说，如果 i 和 j 是相等的：i 不小于 j，j 也不小于 i。此时不能保证这两个元素的相对顺序在排序中被保留。

代码段

下面的代码段说明了对三个段 (其中一个长度为零的段) 的 int 键进行分批排序。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 数据排序指针
int  num_items;          // 例如 7
int  num_segments;       // 例如 3
int  *d_offsets;         // 例如 [0, 3, 3, 7]
int  *d_keys_in;         // 例如 [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如 [-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
void   *d_temp_storage = NULL;
size_t  temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortKeys(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortKeys(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配空间大小写入 temp_Storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_keys_in 设备可访问的指针，指向要排序键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向排序后的键数据输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪DeviceSegmentedSort::SortKeysDescending
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

对键的段按降序进行排序。需要使用大小约为 num_items + 2*num_segments 的辅助存储空间。

- 该排序操作不会更改输入数据的内容。
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets+1)。
- SortKeysDescending 不能保证稳定。也就是说，如果 i 和 j 是相等的：i 不小于 j，j 也不小于 i。此时不能保证这两个元素的相对顺序在排序中被保留。

代码段

下面的代码段说明了对三个段 (其中一个长度为零的段) 的 int 键进行分批排序。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于排序数据
int  num_items;          // 例如 7
```

(下页继续)

(续上页)

```
int  num_segments;        // 例如 3
int  *d_offsets;          // 例如 [0, 3, 3, 7]
int  *d_keys_in;          // 例如 [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;         // 例如 [-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
void    *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行排序操作
cub::DeviceSegmentedSort::SortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配空间大小写入 temp_Storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问的指针，指向要排序键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向排序后的键数据输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::SortKeys
```

(下页继续)

(续上页)

```
(    void *     d_temp_storage,
     std::size_t &     temp_storage_bytes,
     DoubleBuffer< KeyT > &     d_keys,
     int     num_items,
     int     num_segments,
     BeginOffsetIteratorT     d_begin_offsets,
     EndOffsetIteratorT     d_end_offsets,
     mcStream_t     stream = 0,
     bool     debug_synchronous = false
)
```

对键的段按升序进行排序。需要使用大小约为 2*num_segments 的辅助存储空间。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会更改两个缓冲区的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets + 1)。
- SortKeys 不能保证稳定。也就是说，如果 i 和 j 是相等的：i 不小于 j，j 也不小于 i。此时不能保证这两个元素的相对顺序在排序中被保留。

代码段

下面的代码段说明了对三个段 (其中一个长度为零的段) 的 int 键进行分批排序。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 数据排序指针
int  num_items;          // 例如 7
int  num_segments;       // 例如 3
int  *d_offsets;         // 例如 [0, 3, 3, 7]
int  *d_key_buf;         // 例如 [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如 [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 来封装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortKeys(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 进行排序操作
cub::DeviceSegmentedSort::SortKeys(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型

- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是一个简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配空间大小写入 temp_Storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in,out] d_keys 值的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器。其中 d_begin_offsets[i] 是第 i 个数据段在 d_keys_ 和 d_values_ 中的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器。其中 d_end_offsets[i] - 1 是第 i 个数据段在 d_keys_ 和 d_values_ 中的最后一个元素。如果 d_end_offsets[i] - 1 <= d_begin_offsets[i]，则认为第 i 个段为空
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
 ↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
 ↪DeviceSegmentedSort::SortKeysDescending
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键段按降序排序。需要使用大小约为 2*num_segments 的辅助存储空间。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 排序操作可能会更改两个缓冲区的内容。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入是连续的段序列时，可以使用单个序列段偏移量 segment_offsets (长度为 num_segments + 1) 同时为 d_begin_offsets 和 d_end_offsets 参数进行别名处理 (其中后者指定为 segment_offsets + 1)。
- SortKeys 不能保证稳定。也就是说，如果 i 和 j 是相等的：i 不小于 j ，j 也不小于 i 。此时不能保证这两个元素的相对顺序在排序中被保留。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针，用于
// 数据排序
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_key_buf;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如， [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 来封装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定临时设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in,out] d_keys 键的双缓冲区，“current”设备可访问缓冲区包含未排序的输入键值，并且在返回时更新为排序后的输出键
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::StableSortKeys
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按升序排序。大约需要 num_items + 2*num_segments 大小的辅助存储空间。

- 排序操作不会改变输入数据的内容。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- StableSortKeys 是稳定的：它保留了等价元素的相对排序。也就是说，如果 x 在 y 之前且这两个元素是等价的（既不是 x < y 也不是 y < x），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_keys_in;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如， [-, -, -, -, -, -, -]
...
// 确定临时设备存储需求
void    *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 并且不执行任何操作
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_keys_in 设备可访问的指针，指向要排序键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向排序后的键数据输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪DeviceSegmentedSort::StableSortKeysDescending
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按降序排序。大约需要 num_items + 2*num_segments 大小的辅助存储空间。

- 排序操作不会改变输入数据的内容。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- StableSortKeys 是稳定的：它保留了等价元素的相对排序。也就是说，如果 x 在 y 之前且这两个元素是等价的（既不是 x < y 也不是 y < x），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_keys_in;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如， [-, -, -, -, -, -, -]
...
// 确定临时的设备存储需求
void    *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys_in, d_keys_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out           <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 且不做任何处理
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_keys_in 设备可访问的指针，指向要排序的键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向键数据排序后的输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成 (comprise) 排序数据的段的数量
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortKeys

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::StableSortKeys
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键段按升序排序。大约需要 2*num_segments 大小的辅助存储空间。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 两个缓冲区的内容都可能因排序操作而改变。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- StableSortKeys 是稳定的：它保留了等价元素的相对排序。也就是说，如果 x 在 y 之前且这两个元素是等价的（既不是 x < y 也不是 y < x），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_key_buf;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如， [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 来封装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortKeys(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 且不做任何处理 (no work is done)
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in,out] d_keys 键的双缓冲区，其“current”设备可访问缓冲区包含未排序的输入键，并且在返回时更新为指向已排序的输出键
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortKeysDescending

```
template<typename KeyT , typename BeginOffsetIteratorT , typename
↪EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪DeviceSegmentedSort::StableSortKeysDescending
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键段按降序排序。大约需要 2*num_segments 大小的辅助存储空间。

- 排序操作使用一对由 DoubleBuffer 结构管理键缓冲区，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 两个缓冲区的内容都可能因排序操作而改变。
- 在排序操作完成后，DoubleBuffer 的包装器会更新“current”指示器，指示其中哪个缓冲区现在包含了已排序的输出序列（这取决于指定的键位数和目标设备架构）。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。

- StableSortKeys 是稳定的：它保留了等价元素的相对排序。也就是说，如果 x 在 y 之前且这两个元素是等价的（既不是 x < y 也不是 y < x），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如， 7
int  num_segments;       // 例如， 3
int  *d_offsets;         // 例如， [0, 3, 3, 7]
int  *d_key_buf;         // 例如， [8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如， [-, -, -, -, -, -, -]
...
// 创建一个 DoubleBuffer 来封装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortKeysDescending(
    d_temp_storage, temp_storage_bytes, d_keys,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
```

模板参数

- KeyT **[推断]** 键类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 且不做任何处理
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in,out] d_keys 键的双缓冲区，其“当前”设备可访问缓冲区包含未排序的输入键，并且在返回时更新为指向已排序的输出键
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。

- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::SortPairs
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      const KeyT *      d_keys_in,
      KeyT *      d_keys_out,
      const ValueT *      d_values_in,
      ValueT *      d_values_out,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键值对段按升序排序。大约需要 2*num_items + 2*num_segments 大小的辅助存储空间。

- 排序操作不会改变输入数据的内容。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- SortPairs 不能保证稳定。也就是说，假设 i 和 j 是等价的：他们都不小于彼此，则这两个元素的相对顺序并不能保证排序后得到保留。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）和相关的 int 值向量进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或  <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如，  7
int  num_segments;       // 例如，  3
int  *d_offsets;         // 例如，  [0, 3, 3, 7]
int  *d_keys_in;         // 例如，  [8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，  [-, -, -, -, -, -, -]
int  *d_values_in;       // 例如，  [0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，  [-, -, -, -, -, -, -]
...
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
```

(下页继续)

(续上页)

```
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
// d_values_out          <-- [1, 2, 0, 5, 4, 3, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 用于读取段起始偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)
- EndOffsetIteratorT **[推断]** 用于读取段结束偏移量的随机访问输入迭代器类型 (可以是简单的指针类型)

参数

- [in] d_temp_storage 设备可访问的临时存储分配。当为 nullptr 时，所需的分配大小将写入 temp_storage_bytes 且不做任何处理
- [in,out] temp_storage_bytes 为 d_temp_storage 分配的空间大小，以字节为单位
- [in] d_keys_in 设备可访问的指针，指向要排序的键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向键数据排序后的输出序列
- [in] d_values_in 设备可访问的指针，指向相关值项的相应有序输入序列
- [out] d_values_out 设备可访问的指针，指向相关值项的相应有序输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
 ↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
 ↪DeviceSegmentedSort::SortPairsDescending
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     const KeyT *      d_keys_in,
     KeyT *      d_keys_out,
     const ValueT *      d_values_in,
     ValueT *      d_values_out,
     int      num_items,
```

(下页继续)

(续上页)

```
    int     num_segments,
    BeginOffsetIteratorT     d_begin_offsets,
    EndOffsetIteratorT     d_end_offsets,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

将键值对段按降序排序。大约需要 2*num_items + 2*num_segments 大小的辅助存储空间。

- 排序操作不会改变输入数据的内容。
- 当输入一个连续的段序列时，可以使用单个序列 segment_offsets（长度为 num_segments+1）同时作为 d_begin_offsets 和 d_end_offsets 参数的别名（后者指定为 segment_offsets+1）。
- SortPairs 不能保证稳定。也就是说，假设 i 和 j 是等价的：他们都不小于彼此，则这两个元素的相对顺序并不能保证排序后得到保留。

代码段

下面的代码段说明了对三个整型键段（其中一个零长度段）和相关的 int 值向量进行分段排序的过程。

```
#include <cub/cub.cuh>
// 或<cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问指针
// 用于数据排序
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，[-, -, -, -, -, -, -]
...
// 确定临时的设备存储需求
void    *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortPairsDescending(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortPairsDescending(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out           <-- [8, 7, 6, 9, 5, 3, 0]
// d_values_out         <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问的指针，指向要排序的键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向键数据排序后的输出序列
- [in] d_values_in 设备可访问的指针，指向相应的关联值项输入序列
- [out] d_values_out 设备可访问的指针，指向相应有序的关联值项输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::SortPairs
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键值对分段后按升序排序。大约需要 2*num_segments 辅助存储空间。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 每对中两个缓冲区的内容都可能因排序操作而改变。
- 完成后，排序操作将更新每个 DoubleBuffer 包装器中的“current”指示器，以引用两个缓冲区中包含排序后的输出序列的缓冲区（这是指定的关键位数量和目标设备架构的函数）。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1 ）可以别名为单个段序列 segment_offsets （长度为 num_segments+1 ）。
- SortPairs 不能保证稳定。也就是说，假设 i 和 j 是等价的：两者都不小于另一者。这两个元素的相对顺序并不能保证通过排序得到保留。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针
// 用于排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffers 来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortPairs(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
// d_values.Current()    <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键值双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键值，返回时更新为指向已排序的输出键值
- [in,out] d_values 值的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。

- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::SortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
 ↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
 ↪DeviceSegmentedSort::SortPairsDescending
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

将键值对分段后按降序排序。大约需要 2*num_segments 辅助存储空间。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 每对中两个缓冲区的内容都可能因排序操作而改变。
- 完成后，排序操作将更新每个 DoubleBuffer 包装器中的“current”指示器，以引用两个缓冲区中包含排序后的输出序列的缓冲区（这是指定的关键位数量和目标设备架构的函数）。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1）可以别名为单个段序列 segment_offsets （长度为 num_segments+1 ）。
- SortPairsDescending 不能保证稳定。也就是说，假设 i 和 j 是等价的：两者都不小于另一者。这两个元素的相对顺序并不能保证通过排序得到保留。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针用于
// 排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffers 来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
```

(下页继续)

(续上页)

```
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::SortPairsDescending(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::SortPairsDescending(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
// d_values.Current()    <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键值双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键值，返回时更新为指向已排序的输出键值
- [in,out] d_values 值的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
 ↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::StableSortPairs
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     const KeyT *      d_keys_in,
     KeyT *      d_keys_out,
```

(下页继续)

(续上页)

```
    const ValueT *     d_values_in,
    ValueT *     d_values_out,
    int     num_items,
    int     num_segments,
    BeginOffsetIteratorT     d_begin_offsets,
    EndOffsetIteratorT     d_end_offsets,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

将键值对分段后按升序排序。大约需要 2*num_items + 2*num_segments 辅助存储空间。

- 输入数据的内容不会被排序操作改变。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1 ）可以别名为单个段序列 segment_offsets （长度为 num_segments+1 ）。
- StableSortPairs 是稳定的：它保留了等价元素的相对顺序。也就是说，如果 x 和 y 是 x 在 y 之前的元素，并且这两个元素相等（x < y 和 y < x 都不发生），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针
// 用于排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，[-, -, -, -, -, -, -]
...
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [6, 7, 8, 0, 3, 5, 9]
// d_values_out          <-- [1, 2, 0, 5, 4, 3, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）

- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问的指针，指向要排序的键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向键数据排序后的输出序列
- [in] d_values_in 设备可访问的指针，指向相应的关联值项输入序列
- [out] d_values_out 设备可访问的指针，指向相应有序的关联值项输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
↪DeviceSegmentedSort::StableSortPairsDescending
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     const KeyT *      d_keys_in,
     KeyT *      d_keys_out,
     const ValueT *      d_values_in,
     ValueT *      d_values_out,
     int      num_items,
     int      num_segments,
     BeginOffsetIteratorT      d_begin_offsets,
     EndOffsetIteratorT      d_end_offsets,
     mcStream_t      stream = 0,
     bool      debug_synchronous = false
)
```

将键值对分段后按降序排序。大约需要 2*num_items + 2*num_segments 辅助存储空间。

- 输入数据的内容不会被排序操作改变。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1 ）可以别名为单个段序列 segment_offsets （长度为 num_segments+1 ）。
- StableSortPairsDescending 是稳定的：它保留了等价元素的相对顺序。也就是说，如果 x 和 y 是 x 在 y 之前的元素，并且这两个元素相等（x < y 和 y < x 都不发生），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针
// 用于排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_keys_in;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_keys_out;        // 例如，[-, -, -, -, -, -, -]
int  *d_values_in;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_values_out;      // 例如，[-, -, -, -, -, -, -]
...
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortPairsDescending(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortPairsDescending(
    d_temp_storage, temp_storage_bytes,
    d_keys_in, d_keys_out, d_values_in, d_values_out,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys_out            <-- [8, 7, 6, 9, 5, 3, 0]
// d_values_out          <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in] d_keys_in 设备可访问的指针，指向要排序的键数据的输入数据
- [out] d_keys_out 设备可访问的指针，指向键数据排序后的输出序列
- [in] d_values_in 设备可访问的指针，指向相应的关联值项输入序列
- [out] d_values_out 设备可访问的指针，指向相应有序的关联值项输出序列
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。

- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortPairs

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t DeviceSegmentedSort::StableSortPairs
(     void *      d_temp_storage,
      std::size_t &      temp_storage_bytes,
      DoubleBuffer< KeyT > &      d_keys,
      DoubleBuffer< ValueT > &      d_values,
      int      num_items,
      int      num_segments,
      BeginOffsetIteratorT      d_begin_offsets,
      EndOffsetIteratorT      d_end_offsets,
      mcStream_t      stream = 0,
      bool      debug_synchronous = false
)
```

对键值对的片段进行升序排序。大约需要 2*num_segments 辅助存储空间。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 每对中两个缓冲区的内容都可能因排序操作而改变。
- 完成后，排序操作将更新每个 DoubleBuffer 包装器中的“current”指示器，以引用两个缓冲区中包含排序后的输出序列的缓冲区（这是指定的关键位数量和目标设备架构的函数）。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1）可以别名为单个段序列 segment_offsets （长度为 num_segments+1）。
- StableSortPairs 是稳定的：它保留了等价元素的相对顺序。也就是说，如果 x 和 y 是 x 在 y 之前的元素，并且这两个元素相等（x < y 和 y < x 都不发生），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针
// 用于排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffers 来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
```

(下页继续)

(续上页)

```
cub::DeviceSegmentedSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortPairs(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [6, 7, 8, 0, 3, 5, 9]
// d_values.Current()    <-- [5, 4, 3, 1, 2, 0, 6]
```

模板参数

- KeyT **[推断]** 键类型
- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间分配。当为 nullptr 时，所需的分配大小将被写入 temp_storage_bytes ，不会执行任何操作。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键值双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键值，返回时更新为指向已排序的输出键值
- [in,out] d_values 值的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，这样 d_begin_offsets[i] 就是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素。
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，这样 d_end_offsets[i]-1 就是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

DeviceSegmentedSort::StableSortPairsDescending

```
template<typename KeyT , typename ValueT , typename BeginOffsetIteratorT ,
 ↪typename EndOffsetIteratorT >
static CUB_RUNTIME_FUNCTION mcError_t
 ↪DeviceSegmentedSort::StableSortPairsDescending
(    void *      d_temp_storage,
     std::size_t &      temp_storage_bytes,
     DoubleBuffer< KeyT > &      d_keys,
     DoubleBuffer< ValueT > &      d_values,
```

(下页继续)

(续上页)

```
    int     num_items,
    int     num_segments,
    BeginOffsetIteratorT     d_begin_offsets,
    EndOffsetIteratorT     d_end_offsets,
    mcStream_t     stream = 0,
    bool     debug_synchronous = false
)
```

将键值对分段后按降序排序。大约需要 2*num_segments 辅助存储空间。

- 排序操作使用一对键缓冲区和相应的关联值缓冲区。每对缓冲区都由一个 DoubleBuffer 结构管理，该结构指示哪个缓冲区为“current”（包含待排序的输入数据）。
- 每对中两个缓冲区的内容都可能因排序操作而改变。
- 完成后，排序操作将更新每个 DoubleBuffer 包装器中的“current”指示器，以引用两个缓冲区中包含排序后的输出序列的缓冲区（这是指定的关键位数量和目标设备架构的函数）。
- 当输入是连续的段序列时，d_begin_offsets 和 d_end_offsets 参数（后者指定为 segment_offsets+1）可以别名为单个段序列 segment_offsets （长度为 num_segments+1）。
- StableSortPairsDescending 是稳定的：它保留了等价元素的相对顺序。也就是说，如果 x 和 y 是 x 在 y 之前的元素，并且这两个元素相等（x < y 和 y < x 都不发生），那么排序后 x 仍然在 y 之前。

代码段

下面的代码段说明了如何对具有三个段（其中一个为零长度段）的整型键值对进行批量排序。

```
#include <cub/cub.cuh>
// 或等效地 <cub/device/device_segmented_sort.cuh>
// 声明、分配和初始化设备可访问的指针
// 用于排序数据
int  num_items;          // 例如，7
int  num_segments;       // 例如，3
int  *d_offsets;         // 例如，[0, 3, 3, 7]
int  *d_key_buf;         // 例如，[8, 6, 7, 5, 3, 0, 9]
int  *d_key_alt_buf;     // 例如，[-, -, -, -, -, -, -]
int  *d_value_buf;       // 例如，[0, 1, 2, 3, 4, 5, 6]
int  *d_value_alt_buf;   // 例如，[-, -, -, -, -, -, -]
...
// 创建一组 DoubleBuffers 来包装设备指针对
cub::DoubleBuffer<int> d_keys(d_key_buf, d_key_alt_buf);
cub::DoubleBuffer<int> d_values(d_value_buf, d_value_alt_buf);
// 确定临时的设备存储需求
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceSegmentedSort::StableSortPairsDescending(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// 分配临时存储空间
mcMalloc(&d_temp_storage, temp_storage_bytes);
// 执行排序操作
cub::DeviceSegmentedSort::StableSortPairsDescending(
    d_temp_storage, temp_storage_bytes, d_keys, d_values,
    num_items, num_segments, d_offsets, d_offsets + 1);
// d_keys.Current()      <-- [8, 7, 6, 9, 5, 3, 0]
// d_values.Current()    <-- [0, 2, 1, 6, 3, 4, 5]
```

模板参数

- KeyT **[推断]** 键类型

- ValueT **[推断]** 值类型
- BeginOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段首偏移量（可以是简单的指针类型）
- EndOffsetIteratorT **[推断]** 随机访问输入迭代器类型，用于读取段尾偏移量（可以是简单的指针类型）

参数

- [in] d_temp_storage 设备可访问的临时存储空间的分配值。当该参数为 NULL 时，不执行任何操作并且在 temp_storage_bytes 中返回所需内存分配。
- [in,out] temp_storage_bytes 以字节为单位的 d_temp_storage 分配大小的引用
- [in,out] d_keys 键值双缓冲区的引用，其“current”设备可访问的缓冲区包含未排序的输入键值，返回时更新为指向已排序的输出键值
- [in,out] d_values 值的双缓冲区，其“current”设备可访问的缓冲区包含未排序的输入值，返回时更新为指向已排序的输出值
- [in] num_items 要排序的项目总数（跨所有分段）
- [in] num_segments 组成排序数据的段数
- [in] d_begin_offsets 随机访问长度为 num_segments 的起始偏移量序列的输入迭代器，使得 d_begin_offsets[i] 是 d_keys_* 和 d_values_* 中第 i 个数据段的第一个元素
- [in] d_end_offsets 随机访问长度为 num_segments 的结束偏移量序列的输入迭代器，使得 d_end_offsets[i]-1 是 d_keys_* 和 d_values_* 中第 i 个数据段的最后一个元素。如果 d_end_offsets[i]-1 <= d_begin_offsets[i]，则认为第 i 个数据段为空。
- [in] stream **[可选]** 用于启动内核的 MXMACA 流。默认值为 stream0。
- [in] debug_synchronous **[可选]** 是否在每次内核启动后同步该流以检查错误。并把启动配置输出到控制台。默认值为 false。

2.3 工具 (Utilities)

2.3.1 高级迭代器 (Fancy iterators)

2.3.1.1 类 (Classes)

class cub::ArgIndexInputIterator

```
template<
    typename InputIteratorT,
    typename OffsetT = ptrdiff_t,
    typename OutputValueT = cub::detail::value_t<InputIteratorT>>

class cub::ArgIndexInputIterator< InputIteratorT, OffsetT, OutputValueT >
```

随机访问输入包装器，用于将引用值与其对应的索引配对（形成 KeyValuePair 元组）。

概述

- ArgIndexInputIteratorT 包裹着一个 InputIteratorT 类型的随机存取输入迭代器 itr。引用 ArgIndex-InputIteratorTat 偏移量 i 会产生一个 KeyValuePair 值，其 key 字段为 i，value 字段为 itr[i]。
- 可用于任何数据类型。
- 可在主机和设备函数内部和之间构建、操作和交换。封装的主机内存只能在主机上取消引用，封装的设备内存只能在设备上取消引用。

- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 ArgIndexInputIteratorT 来取消引用一个双数组

```
#include <cub/cub.cuh>   //或等效的 <cub/iterator/arg_index_input_iterator.
↪cuh>
// 声明、分配和初始化设备数组
double *d_in;         // 例如，[8.0, 6.0, 7.0, 5.0, 3.0, 0.0, 9.0]
// 创建迭代包装器
cub::ArgIndexInputIterator<double*> itr(d_in);
// 在设备代码中：
typedef typename cub::ArgIndexInputIterator<double*>::value_type Tuple;
Tuple item_offset_pair.key = *itr;
printf("%f @ %d\n",
    item_offset_pair.value,
    item_offset_pair.key);   // 8.0 @ 0
itr = itr + 6;
item_offset_pair.key = *itr;
printf("%f @ %d\n",
    item_offset_pair.value,
    item_offset_pair.key);   // 9.0 @ 6
```

模板参数

- InputIteratorT 已封装输入迭代器的值类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）
- OutputValueT <offset,value> 元组的配对值类型（默认：输入迭代器的值类型）

class cub::CacheModifiedInputIterator

```
template<
    CacheLoadModifier MODIFIER,
    typename ValueType,
    typename OffsetT = ptrdiff_t>

class cub::CacheModifiedInputIterator< MODIFIER, ValueType, OffsetT >
```

一个随机访问输入包装器，用于使用 PTX 缓存加载修改器以取消引用数组值。

概述

- CacheModifiedInputIterator 是一个随机访问输入迭代器，它封装了一个 ValueType* 类型的本地设备指针。对 ValueType 的引用是通过读取被 MODIFIER 修改过的负载中的 ValueType 值实现的。
- 可用于通过 PTX 缓存载入修改器（如“LOAD_LDG”、“LOAD_CG”、“LOAD_CA”、“LOAD_CS”、“LOAD_CV”等）从内存中载入任何数据类型。
- 可在主机和设备函数内部和之间构建、操作和交换，但只能在设备函数内部取消引用。
- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 CacheModifiedInputIterator，通过“ldg”PTX 加载修改器（即通过纹理缓存加载数值）来取消引用设备的 double 数组。

```
#include <cub/cub.cuh>   // 或等效的 <cub/iterator/cache_modified_input_
↪iterator.cuh>
// 声明、分配和初始化设备数组
double *d_in;            // 例如，[8.0, 6.0, 7.0, 5.0, 3.0, 0.0, 9.0]
// 创建迭代包装器
cub::CacheModifiedInputIterator<cub::LOAD_LDG, double> itr(d_in);
// 在设备代码中：
printf("%f\n", itr[0]);  // 8.0
printf("%f\n", itr[1]);  // 6.0
printf("%f\n", itr[6]);  // 9.0
```

模板参数

- CacheLoadModifier 访问数据时使用的 cub::CacheLoadModifier
- ValueType 迭代器的值类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）

class cub::CacheModifiedOutputIterator

```
template<
    CacheStoreModifier MODIFIER,
    typename ValueType,
    typename OffsetT = ptrdiff_t>

class cub::CacheModifiedOutputIterator< MODIFIER, ValueType, OffsetT >
```

一个随机访问输出包装器，用于使用 PTX 缓存修改器存储数组值。

概述

- CacheModifiedOutputIterator 是一个随机访问输出迭代器，它封装了一个 ValueType* 类型的本地设备指针。通过 MODIFIER 修改的存储写入 ValueType 值来实现 ValueType 引用。
- 可用于使用 PTX 缓存存储修改器（如“STORE_WB”、“STORE_CG”、“STORE_CS”、“STORE_WT”等）将任何数据类型存储到内存中。
- 可在主机和设备函数内部和之间构建、操作和交换，但只能在设备函数内部取消引用。
- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 CacheModifiedOutputIterator，使用“wt”PTX 加载修改器（即通过写入系统内存）来取消引用设备的双倍数组。

```
#include <cub/cub.cuh>   // 或等效的 <cub/iterator/cache_modified_output_
↪iterator.cuh>
// 声明、分配和初始化设备数组
double *d_out;               // 例如，[, , , , , , ]
// 创建迭代包装器
cub::CacheModifiedOutputIterator<cub::STORE_WT, double> itr(d_out);
// 在设备代码中：
itr[0]  = 8.0;
itr[1]  = 66.0;
itr[55] = 24.0;
```

使用注意事项

只能在设备代码中被取消引用

模板参数

- CacheStoreModifier 访问数据时使用的 cub::CacheStoreModifier
- ValueType 迭代器的值类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）

class cub::ConstantInputIterator

```
template<
    typename ValueType,
    typename OffsetT = ptrdiff_t>

class cub::ConstantInputIterator< ValueType, OffsetT >
```

随机访问输入生成器，用于取消引用同源值序列。

概述

- 对 ConstantInputIteratorTiterator 的读取引用总是返回所提供的 ValueType 类型的常量。
- 可用于任何数据类型。
- 可在主机和设备函数内部和之间构建、操作、取消引用和交换。
- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 ConstantInputIteratorT 来引用同质双倍数序列。

```
#include <cub/cub.cuh>   // 或等效的 <cub/iterator/constant_input_iterator.
↪cuh>
cub::ConstantInputIterator<double> itr(5.0);
printf("%f\n", itr[0]);      // 5.0
printf("%f\n", itr[1]);      // 5.0
printf("%f\n", itr[2]);      // 5.0
printf("%f\n", itr[50]);     // 5.0
```

模板参数

- ValueType 迭代器的值类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）

class cub::CountingInputIterator

```
template<
    typename ValueType,
    typename OffsetT = ptrdiff_t>

class cub::CountingInputIterator< ValueType, OffsetT >
```

一个随机访问输入生成器，用于取消引用递增整数值序列。

概述

- 将 CountingInputIteratorT 初始化为某个整数基数后，偏移量处的读取引用返回基数 + 偏移量的值。
- 可在主机和设备函数内部和之间构建、操作、取消引用和交换。
- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 CountingInputIteratorT 来取消引用一个递增整数序列。

```
#include <cub/cub.cuh>   // 或等效的 <cub/iterator/counting_input_iterator.
↪cuh>
cub::CountingInputIterator<int> itr(5);
printf("%d\n", itr[0]);       // 5
printf("%d\n", itr[1]);       // 6
printf("%d\n", itr[2]);       // 7
printf("%d\n", itr[50]);      // 55
```

模板参数

- ValueType 迭代器的值类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）

class cub::DiscardOutputIterator

```
template<
    typename OffsetT = ptrdiff_t>

class cub::DiscardOutputIterator< OffsetT >
```

丢弃迭代器。

class cub::TransformInputIterator

```
template<
    typename ValueType,
    typename ConversionOp,
    typename InputIteratorT,
    typename OffsetT = ptrdiff_t>

class cub::TransformInputIterator< ValueType, ConversionOp, InputIteratorT,
↪ OffsetT >
```

一个随机访问输入包装器，用于转换取消引用的值。

概述

- TransformInputIteratorT 包装了 ConversionOp 类型的一元转换函子和 InputIteratorT 类型的随机访问输入迭代器，使用前者从后者生成 ValueType 类型的引用。
- 可用于任何数据类型。
- 可在主机和设备函数内部和之间构建、操作和交换。封装的主机内存只能在主机上取消引用，封装的设备内存只能在设备上取消引用。
- 与 Thrust API v1.7 或更新的版本兼容。

代码段

下面的代码段说明了如何使用 TransformInputIteratorT 来取消引用整数数组，将数值增加三倍并转换为双精度浮点数。

```
#include <cub/cub.cuh>   // 或等效的 <cub/iterator/transform_input_iterator.
↪cuh>
// 用于将整数值增加三倍并转换为双精度值的函子
```

(下页继续)

(续上页)

```
struct TripleDoubler
{
    __host__ __device__ __forceinline__
    double operator()(const int &a) const {
        return double(a * 3);
    }
};
// 声明、分配和初始化设备数组
int *d_in;                    // 例如，[8, 6, 7, 5, 3, 0, 9]
TripleDoubler conversion_op;
// 创建迭代包装器
cub::TransformInputIterator<double, TripleDoubler, int*> itr(d_in,
↪conversion_op);
// 在设备代码中：
printf("%f\n", itr[0]);  // 24.0
printf("%f\n", itr[1]);  // 18.0
printf("%f\n", itr[6]);  // 27.0
```

模板参数

- ValueType 迭代器的值类型
- ConversionOp 一元函数类型，用于将 InputType 类型的对象映射到 ValueType 类型。必须有成员 ValueType operator()(const InputType &datum)。
- InputIteratorT 已封装的输入迭代器类型
- OffsetT 迭代器的差分类型（默认值：ptrdiff_t）

2.3.1.2 枚举

enum cub::CacheLoadModifier

用于内存加载操作的缓存修改器枚举。

- LOAD_DEFAULT 默认值（无修饰符）
- LOAD_CA 各级缓存
- LOAD_CG 全局级别的缓存
- LOAD_CS 缓存流（可能被访问一次）
- LOAD_CV 缓存为易失性的（包括缓存的系统行）
- LOAD_LDG 缓存为纹理
- LOAD_VOLATILE 易失性（任何内存空间）

enum cub::CacheStoreModifier

用于内存存储操作的缓存修改器枚举。

- STORE_DEFAULT 默认值（无修饰符）
- STORE_WB 缓存回写所有一致性级别
- STORE_CG 全局缓存
- STORE_CS 缓存流（可能被访问一次）
- STORE_WT 缓存写入（到系统内存）

- STORE_VOLATILE 易失性共享（任何内存空间）

2.3.1.3 函数

cub::LoadDirectBlocked

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到跨线程块的块排列中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的每线程连续项目。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可以是简单指针类型）。

参数

- [in] linear_tid 调用线程的合适一维线程标识符（例如，二维线程块的标识符为（threadIdx.y * blockDim.x）+ linear_tid）
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 加载数据

cub::LoadDirectBlocked

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将项目的线性段加载到整个线程块的分块排列中，并按范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的每线程连续项目。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可以是简单指针类型）。

参数

- [in] linear_tid 调用线程的合适一维线程标识符（例如，二维线程块的标识符为（threadIdx.y * blockDim.x）+ linear_tid）

- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 加载数据
- [in] valid_items 要加载的有效项目数

cub::LoadDirectBlocked

```
template<typename InputT , typename DefaultT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(    int     linear_tid,
     InputIteratorT     block_itr,
     InputT(&)     items[ITEMS_PER_THREAD],
     int     valid_items,
     DefaultT     oob_default
)
```

将项目的线性段加载到整个线程块的分块排列中，通过范围进行保护，并对出界元素进行回退分配。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的每线程连续项目。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可以是简单指针类型）。

参数

- [in] linear_tid 调用线程的合适一维线程标识符（例如，二维线程块的标识符为（threadIdx.y * blockDim.x）+ linear_tid）
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 加载数据
- [in] valid_items 要加载的有效项目数
- [in] oob_default 分配越界项的默认值

cub::LoadDirectBlockedVectorized

```
template<typename T , int ITEMS_PER_THREAD>
__device__ __forceinline__ void cub::LoadDirectBlockedVectorized
(    int     linear_tid,
     T *     block_ptr,
     T(&)    items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到跨线程块的块排列中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的每线程连续项目。对于多维线程块，假定顺序为行主序。

输入偏移（block_ptr + block_offset）必须与四项目对齐

以下情况将阻止矢量化，加载将退回到 cub::BLOCK_LOAD_DIRECT：

- ITEMS_PER_THREAD 为奇数

- 数据类型 T 不是内置原语或 MXMACA 向量类型（如 short、int2、double、float2 等）。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。

参数

- [in] linear_tid 调用线程的合适一维线程标识符（例如，二维线程块的标识符为（threadIdx.y * blockDim.x）+ linear_tid）
- [in] block_ptr 输入指针，用于加载
- [out] items 加载数据

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到线程块上的条带排列中。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项 (i), (i + block-threads), ⋯, (i + (block-threads*(items-per-thread-1)))。对于多维线程块，假定行主线程排序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 调用线程的合适一维线程标识符（例如，二维线程块的标识符为（threadIdx.y * blockDim.x）+ linear_tid）
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

在线程块上以条状排列方式装入一段线性项目，并按范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项 (i)、(i + block-threads)、…、(i + (block-threads*(items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 用于输入的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据
- [in] valid_items 需要加载的有效项目数量

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , typename DefaultT , int
↪ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(     int     linear_tid,
      InputIteratorT     block_itr,
      InputT(&)     items[ITEMS_PER_THREAD],
      int     valid_items,
      DefaultT     oob_default
)
```

将项目的线性段加载到跨线程块的条带排列中，按范围保护，并对越界元素进行回退赋值。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 用于输入的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据
- [in] valid_items 需要加载的有效项目数量
- [in] oob_default 为越界元素分配默认值

cub::LoadDirectWarpStriped

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到跨线程块的线程束条带排列中。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是架构的线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 用于输入的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据

cub::LoadDirectWarpStriped

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将项目的线性段加载到跨线程块的线程束条带排列中，按范围保护。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是架构的线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 用于输入的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)

- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据
- [in] valid_items 需要加载的有效项目数量

cub::LoadDirectWarpStriped

```
template<typename InputT , typename DefaultT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items,
      DefaultT      oob_default
)
```

将项目的线性段加载到跨线程块的线程束条带排列中，按范围保护，并对越界元素进行回退赋值。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是架构的线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 用于输入的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于加载的线程块的基本输入迭代器
- [out] items 要加载的数据
- [in] valid_items 需要加载的有效项目数量
- [in] oob_default 为越界元素分配默认值

cub::StoreDirectBlocked

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectBlocked
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD]
)
```

将跨线程块的分块排列的项目存储到项目的线性段中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [in] items 要存储的数据

cub::StoreDirectBlocked

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectBlocked
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将跨线程块的分块排列的项目存储到项目的线性段中，按范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项数

cub::StoreDirectBlockedVectorized

```
template<typename T , int ITEMS_PER_THREAD>
__device__ __forceinline__ void cub::StoreDirectBlockedVectorized
(     int      linear_tid,
      T *      block_ptr,
      T(&)     items[ITEMS_PER_THREAD]
)
```

将跨线程块的分块排列的项目存储到项目的线性段中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

输出偏移量 (block_ptr + block_offset) 必须四项对齐，这是 mcMalloc() 返回的默认起始偏移量。

以下条件将阻止向量化，并且存储将返回到 cub::BLOCK_STORE_DIRECT: - ITEMS_PER_THREAD 是奇数 - 数据类型 T 不是内置的原语或 MXMACA 向量类型 (例如，short, int2, double, float2, 等等)

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_ptr 用于存储的输入指针
- [in] items 要存储的数据

cub::StoreDirectStriped

```
template<int BLOCK_THREADS, typename T , int ITEMS_PER_THREAD, typename
↪OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD]
)
```

将跨线程块的条带排列的数据存储到项目的线性段中。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 用于输出的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [in] items 要存储的数据

cub::StoreDirectStriped

```
template<int BLOCK_THREADS, typename T , int ITEMS_PER_THREAD, typename
↪OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将跨线程块的条带排列的数据存储到项目的线性段中，按范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 用于输出的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项数

cub::StoreDirectWarpStriped

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectWarpStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD]
)
```

将跨线程块的线程束条带排列的数据存储到项目的线性段中。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是架构的线程束大小的倍数。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 用于输出的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [out] items 要加载的数据

cub::StoreDirectWarpStriped

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectWarpStriped
(    int     linear_tid,
     OutputIteratorT     block_itr,
     T(&)     items[ITEMS_PER_THREAD],
     int     valid_items
)
```

将跨线程块的线程束条带排列的数据存储到项目的线性段中，按范围保护。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数量必须是架构的线程束大小的倍数。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 用于输出的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 用于二维线程块)
- [in] block_itr 用于存储的线程块的基本输出迭代器
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项数

cub::ThreadLoad

```
template<CacheLoadModifier MODIFIER, typename InputIteratorT >
__device__ __forceinline__ cub::detail::value_t<InputIteratorT>
 ↪cub::ThreadLoad (InputIteratorT     itr)
```

使用 cub::CacheLoadModifier 缓存修饰符的线程实用程序，该程序用于读取内存。可用于加载任何数据类型。

示例

```
#include <cub/cub.cuh>   // 或 <cub/thread/thread_load.cuh>
// 通过使用 cache-global 修饰符进行 32 位加载：
int *d_in;
int val = cub::ThreadLoad<cub::LOAD_CA>(d_in + threadIdx.x);
// 通过使用 default 修饰符进行 16 位加载
short *d_in;
short val = cub::ThreadLoad<cub::LOAD_DEFAULT>(d_in + threadIdx.x);
// 通过使用 cache-volatile 修饰符进行 256 位加载
double4 *d_in;
double4 val = cub::ThreadLoad<cub::LOAD_CV>(d_in + threadIdx.x);
// 通过使用 cache-streaming 修饰符进行 96 位加载
struct TestFoo { bool a; short b; };
TestFoo *d_struct;
TestFoo val = cub::ThreadLoad<cub::LOAD_CS>(d_in + threadIdx.x);
```

模板参数

- MODIFIER **[推断]** CacheLoadModifier 枚举
- InputIteratorT **[推断]** 输入迭代器类型 (可以是简单的指针类型)

cub::ThreadStore

```
template<CacheStoreModifier MODIFIER, typename OutputIteratorT , typename
↪T >
__device__ __forceinline__ void cub::ThreadStore
(    OutputIteratorT     itr,
     T      val
)
```

使用 cub::CacheStoreModifier 缓存修饰符的线程实用程序，该程序用于写入内存。可用于存储任何数据类型。

Example

```
#include <cub/cub.cuh>   // 或 <cub/thread/thread_store.cuh>
// 通过使用 cache-global 修饰符进行 32 位存储：
int *d_out;
int val;
cub::ThreadStore<cub::STORE_CG>(d_out + threadIdx.x, val);
// 通过使用 default 修饰符进行 16 位存储
short *d_out;
short val;
cub::ThreadStore<cub::STORE_DEFAULT>(d_out + threadIdx.x, val);
// 通过使用 write-through 修饰符进行 256 位存储
double4 *d_out;
double4 val;
cub::ThreadStore<cub::STORE_WT>(d_out + threadIdx.x, val);
// 通过使用 cache-streaming 缓存修饰符进行 96 位存储
struct TestFoo { bool a; short b; };
TestFoo *d_struct;
TestFoo val;
cub::ThreadStore<cub::STORE_CS>(d_out + threadIdx.x, val);
```

模板参数

- MODIFIER **[推断]** CacheStoreModifier 枚举
- InputIteratorT **[推断]** 输出迭代器类型 (可以是简单的指针类型)
- T **[推断]** 输出值的数据类型

2.3.2 线程和线程块 I/O(Thread and thread block I/O)

2.3.2.1 类 (Classes)

class cub::BlockLoad

```
template<
    typename InputT,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    BlockLoadAlgorithm ALGORITHM = BLOCK_LOAD_DIRECT,
```

(下页继续)

(续上页)

```
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockLoad< InputT, BLOCK_DIM_X, ITEMS_PER_THREAD, ALGORITHM,
↪BLOCK_DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockLoad 类提供了集体数据移动方法，用于将内存中项目的线性段加载到跨越 MXMACA 线程块的分块排列中。

模板参数

- InputT 要读取的数据类型，必须可以从输入迭代器的值（value）类型进行转换。
- BLOCK_DIM_X 在 X 维度上线程中线程块长度。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::BlockLoadAlgorithm 调优策略。默认值为 cub::BLOCK_LOAD_DIRECT。
- WARP_TIME_SLICING **[可选]** 在任何与负载相关的数据转置过程中，是否应该只分配一个线程束大小的共享内存，并在 block-warps 之间进行时间片分配（而不是每个线程束都有自己的存储空间），默认值：false。
- BLOCK_DIM_Y **[可选]** 在 Y 维度上的线程块长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 在 Z 维度上的线程块长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- BlockLoad 类提供了一个单一的数据移动抽象方法，可以专门用于实现不同的 cub::BlockLoadAlgorithm 策略。这有助于为不同的架构、数据类型、粒度大小等提供不同的性能策略。
- BlockLoad 可以通过不同的数据移动策略进行选择性地定义：
  - cub::BLOCK_LOAD_DIRECT. 按照分块排列的数据直接从内存中读取。
  - cub::BLOCK_LOAD_STRIPED. 按照条带排列的数据直接从内存中读取。
  - cub::BLOCK_LOAD_VECTORIZE. 使用 MXMACA 的内置矢量化加载直接从内存中读取分块排列的数据，以实现协同优化。
  - cub::BLOCK_LOAD_TRANSPOSE. 按照条带排列的数据直接从内存中读取，然后在本地转换成分块排列。
  - cub::BLOCK_LOAD_WARP_TRANSPOSE. 按照线程束条带排列的数据直接从内存中读取，然后在本地转换成分块排列。
  - cub::BLOCK_LOAD_WARP_TRANSPOSE_TIMESLICED. 按照线程束条带排列的数据直接从内存中读取，然后逐个线程束地将其在本地转换为分块排列。
- 对于多维线程块，线程按行主序进行线性排列。

简单示例

块中的每个线程都使用 BlockLoad 类，首先定义 BlockLoad 类型，然后实例化一个含通信参数的实例，最后调用一个或多个集体成员函数。下面的代码段说明了如何将一个包含 512 个整数的线性段加载到一个跨 128 个线程的“分块”排列中，其中每个线程拥有 4 个连续项。其中负载是专门为 BLOCK_LOAD_WARP_TRANSPOSE 设计的，这意味着内存通过使用线程束条带（warp-striped）访问模式有效地合并（之后在线程之间本地重新排序）。

```
#include <cub/cub.cuh>   // 或 <cub/block/block_load.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    // 为 128 个线程的一维线程块指定 BlockLoad 类，其中每个线程包含 4 个整数项
    typedef cub::BlockLoad<int, 128, 4, BLOCK_LOAD_WARP_TRANSPOSE>
↪BlockLoad;
    // 为 BlockLoad 分配共享内存
    __shared__ typename BlockLoad::TempStorage temp_storage;
    // 加载跨线程分块的连续项目段
    int thread_data[4];
    BlockLoad(temp_storage).Load(d_data, thread_data);
```

假设输入 d_data 为 0, 1, 2, 3, 4, 5, ⋯，则线程块中的 thread_data 集合为 { [0,1,2,3], [4,5,6,7], ⋯, [508,509,510,511] }。

class cub::BlockStore

```
template<
    typename T,
    int BLOCK_DIM_X,
    int ITEMS_PER_THREAD,
    BlockStoreAlgorithm ALGORITHM = BLOCK_STORE_DIRECT,
    int BLOCK_DIM_Y = 1,
    int BLOCK_DIM_Z = 1,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::BlockStore< T, BLOCK_DIM_X, ITEMS_PER_THREAD, ALGORITHM, BLOCK_
↪DIM_Y, BLOCK_DIM_Z, PTX_ARCH >
```

BlockStore 类提供了集体数据的移动方法，用于将跨 MXMACA 线程块分区的分块项排列写入到内存的线性段。

模板参数

- T 要写入的数据的类型。
- BLOCK_DIM_X 在 X 维度上线程中线程块的长度。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::BlockStoreAlgorithm 调优策略枚举，默认：cub::BLOCK_STORE_DIRECT。
- BLOCK_DIM_Y **[可选]** 在 Y 维度上线程中线程块的长度，默认值：1。
- BLOCK_DIM_Z **[可选]** 在 Z 维度上线程中线程块的长度，默认值：1。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- BlockStore 类提供了一个单一的数据移动抽象方法，可以专门用于实现不同的 cub::BlockStoreAlgorithm 策略。这有助于为不同的体系架构、数据类型、粒度大小等制定不同的性能策略。
- BlockStore 可以根据不同的数据移动策略来选择定义：
  - cub::BLOCK_STORE_DIRECT. 将一个分块的数据排列直接写入到内存中。
  - cub::BLOCK_STORE_STRIPED. 将一个条带化的数据排列直接写入到内存中。
  - cub::BLOCK_STORE_VECTORIZE. 通过使用 MXMACA 内置的矢量化存储技术，将一个块状的数据排列直接写入内存，以实现一种合并优化。

  - cub::BLOCK_STORE_TRANSPOSE. 将一个分块的排列局部转换成一个条带排列，然后写入内存。
  - cub::BLOCK_STORE_WARP_TRANSPOSE. 一个分块的排列局部转换成一个线程束条带的排列，然后被写入内存。
  - cub::BLOCK_STORE_WARP_TRANSPOSE_TIMESLICED. 将一个分块的排列局部转换成一个线程束条带的排列，然后写入内存。为了减少其对共享内存的需求，只提供了一个线程束的共享内存，然后在线程束之间进行时间分割。

- 对于多维线程块，线程按行主序进行线性排列。

简单示例

块中的每个线程首先定义并使用 BlockStore 类，然后实例化一个带通信参数的实例，最后调用一个或多个集体成员函数。下面的代码段说明了将跨 128 个线程（每个线程拥有 4 个连续项）的 512 个整数的“分块”排列存储到一个线性内存段中。其中存储是专门用于 BLOCK_STORE_WARP_TRANSPOSE 的，这意味着项目将在线程之间被本地重新排序，这样就可以使用线程束条带访问模式有效地合并内存引用。

```
#include <cub/cub.cuh>   // 或 <cub/block/block_store.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    // 为 128 个线程的一维线程块指定 BlockStore，其中每个线程包含 4 个整数项
    typedef cub::BlockStore<int, 128, 4, BLOCK_STORE_WARP_TRANSPOSE>
↪BlockStore;
    // 为块存储区分配共享内存
    __shared__ typename BlockStore::TempStorage temp_storage;
    // 获取跨线程的连续分块项目段
    int thread_data[4];
    ...
    // 将项目存储到线性内存中
    BlockStore(temp_storage).Store(d_data, thread_data);
```

假设线程块中的 thread_data 集合为 { [0,1,2,3], [4,5,6,7], ⋯, [508,509,510,511] }。那么输出 d_data 为 0, 1, 2, 3, 4, 5, ⋯。

class cub::WarpLoad

```
template<
    typename InputT,
    int ITEMS_PER_THREAD,
    WarpLoadAlgorithm ALGORITHM = WARP_LOAD_DIRECT,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>

class cub::WarpLoad< InputT, ITEMS_PER_THREAD, ALGORITHM, LOGICAL_WARP_
↪THREADS, PTX_ARCH >
```

WarpLoad 类提供了集体数据移动方法，用于将内存中的线性段加载到跨 MXMACA 线程块的分块排列中。

模板参数

- InputT 要读取的数据类型，必须从输入迭代器的数据类型进行转换。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::WarpLoadAlgorithm 调优策略，默认：cub::WARP_LOAD_DIRECT。
- LOGICAL_WARP_THREADS **[可选]** 每个“logical”线程束的线程数目 (可能少于硬件线程束的线程数量)。默认值是对标 MXMACA 算力的线程束大小，且必须是二的幂。

- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- WarpLoad 类提供了一个单一的数据移动抽象方法，可以专门用于实现不同的 cub::WarpLoadAlgorithm 策略，这有利于针对不同体系架构、数据类型、粒度大小等的提供不同性能策略。
- WarpLoad 可以根据不同的数据移动策略来选择定义：
  - cub::WARP_LOAD_DIRECT. 直接从内存中读取一个分块的数据排列。
  - cub::WARP_LOAD_STRIPED. 直接从内存中读取数据的条带化排列。
  - cub::WARP_LOAD_VECTORIZE. 将 MXMACA 的内置向量化负载作为合并优化策略，直接从内存中读取分块的数据排列。
  - cub::WARP_LOAD_TRANSPOSE. 直接从内存中读取数据的条带排列，然后本地转置成一个分块的排列。

简单示例

下面的代码段说明了将一个包含 64 个整数的线性段加载到一个跨 16 个线程的“分块”排列中，其中负载是专门针对 WARP_LOAD_TRANSPOSE 的，这意味着内存引用使用线程束条带访问模式有效地合并（在线程之间本地重新排序之后）。

```
#include <cub/cub.cuh>   // 或 <cub/warp/warp_load.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    // 用 WarpLoad 定义 1 个包含 16 个线程的线程束
    using WarpLoadT = WarpLoad<int,
                               items_per_thread,
                               cub::WARP_LOAD_TRANSPOSE,
                               warp_threads>;
    constexpr int warps_in_block = block_threads / warp_threads;
    constexpr int tile_size = items_per_thread * warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 为 WarpLoad 分配共享内存
    __shared__ typename WarpLoadT::TempStorage temp_storage[warps_in_
↪block];
    // 加载跨线程分块的连续项目段
    int thread_data[items_per_thread];
    WarpLoadT(temp_storage[warp_id]).Load(d_data + warp_id * tile_size,
                                          thread_data);
```

假设输入 d_data 为 0, 1, 2, 3, 4, 5, …，则跨线程中第一个逻辑线程束的 thread_data 集为: { [0,1,2,3], [4,5,6,7], …, [60,61,62,63] }。

class cub::WarpStore

```
template<
    typename T,
    int ITEMS_PER_THREAD,
    WarpStoreAlgorithm ALGORITHM = WARP_STORE_DIRECT,
    int LOGICAL_WARP_THREADS = CUB_PTX_WARP_THREADS,
    int PTX_ARCH = CUB_PTX_ARCH>
```

(下页继续)

(续上页)

```
class cub::WarpStore< T, ITEMS_PER_THREAD, ALGORITHM, LOGICAL_WARP_THREADS,
↪ PTX_ARCH >
```

WarpStore 类提供集体数据移动方法，用于将跨 MXMACA 线程束的分块排列写入到内存的线性段。

模板参数

- T 要写入的数据的类型。
- ITEMS_PER_THREAD 分配到每个线程上的连续项数量。
- ALGORITHM **[可选]** cub::WarpStoreAlgorithm 调优策略枚举，默认：cub::WARP_STORE_DIRECT。
- LOGICAL_WARP_THREADS **[可选]** 每个“logical”线程束的线程数目 (可能少于硬件线程束的线程数量)。默认值对标 MXMACA 算力的线程束大小，且必须是二的幂。
- PTX_ARCH **[可选]** 为特化此集合的 PTX 计算能力，按照 MACA_ARCH 宏进行格式化。这有助于从主机端确定给定设备上集合的存储需求。默认值：当前编译过程中 MACA_ARCH 的值。

概述

- WarpStore 类提供了一个单一的数据移动抽象方法，可以专门用于实现不同的 cub::WarpStoreAlgorithm 策略。这有利于针对不同体系架构、数据类型、粒度大小等的提供不同性能策略。
- WarpStore 可以根据不同的数据移动策略来选择定义：
  - cub::WARP_STORE_DIRECT. 将一个分块的数据排列直接写入到内存中。
  - cub::WARP_STORE_STRIPED. 将数据的条带化排列直接写入到内存中。
  - cub::WARP_STORE_VECTORIZE. 将 MXMACA 内置的向量化存储作为合并优化策略，把分块的数据排列直接写入内存中。
  - cub::WARP_STORE_TRANSPOSE. 将一个分块的排列局部转换成一个条带排列，然后写入内存。
- 对于多维线程块，线程按行主序进行线性排列。

简单示例

下面的代码段说明了如何将跨 16 个线程（每个线程拥有 4 个连续项）的 64 个整数的“分块”排列存储到一个线性内存段中。其中存储空间是专门为 WARP_STORE_TRANSPOSE 设计的，这意味着项目将在线程之间被本地重新排序，以便将使用线程束条带访问模式有效地合并内存引用。

```
#include <cub/cub.cuh>   // 或 <cub/warp/warp_store.cuh>
__global__ void ExampleKernel(int *d_data, ...)
{
    constexpr int warp_threads = 16;
    constexpr int block_threads = 256;
    constexpr int items_per_thread = 4;
    //WarpStore 定义 1 个包含 16 个线程的虚拟线程束，其中每个线程包含 4 个整数项
    using WarpStoreT = WarpStore<int,
                                 items_per_thread,
                                 cub::WARP_STORE_TRANSPOSE,
                                 warp_threads>;
    constexpr int warps_in_block = block_threads / warp_threads;
    constexpr int tile_size = items_per_thread * warp_threads;
    const int warp_id = static_cast<int>(threadIdx.x) / warp_threads;
    // 为 WarpStore 分配共享内存
    __shared__ typename WarpStoreT::TempStorage temp_storage[warps_in_
↪block];
    // 获取跨线程的连续分块项目段
    int thread_data[4];
```

(下页继续)

(续上页)

```
    ...
    // 将项目存储到线性内存中
    WarpStoreT(temp_storage[warp_id]).Store(d_data + warp_id * tile_size,
↪thread_data);
```

假设跨线程束线程的 thread_data 集为 { [0,1,2,3], [4,5,6,7], ⋯, [60,61,62,63] }，则输出 d_data 为 0, 1, 2, 3, 4, 5, ⋯。

2.3.2.2 枚举

enum cub::CacheLoadModifier

针对内存加载操作的缓存修饰符的枚举。

- LOAD_DEFAULT 默认值（无修饰符）
- LOAD_CA 所有级别的缓存
- LOAD_CG 全局级别缓存
- LOAD_CS 缓存流（可能要访问一次）
- LOAD_CV 不确定的缓存（包括缓存的系统行）
- LOAD_LDG 纹理缓存
- LOAD_VOLATILE 不确定性（任何内存空间）

enum cub::CacheStoreModifier

内存存储操作的缓存修饰符的枚举。

- STORE_DEFAULT 默认值（无修饰符）
- STORE_WB 回写所有一致的级别的缓存
- STORE_CG 全局级别缓存
- STORE_CS 缓存流（可能要访问一次）
- STORE_WT 缓存连续写入（到系统内存）
- STORE_VOLATILE 不确定性（任何内存空间）

2.3.2.3 函数

cub::LoadDirectBlocked

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(    int     linear_tid,
     InputIteratorT     block_itr,
     InputT(&)     items[ITEMS_PER_THREAD]
)
```

将线性项目段载入到跨线程块的分块排列中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据

cub::LoadDirectBlocked

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将线性项目加载到贯穿线程块的分块排列中，按范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数

cub::LoadDirectBlocked

```
template<typename InputT , typename DefaultT , int ITEMS_PER_THREAD,
 ↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectBlocked
(     int      linear_tid,
      InputIteratorT      block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items,
      DefaultT      oob_default
)
```

将项目的线性部分加载到一个跨线程块的分块排列中，按范围保护，并对出界元素进行回退赋值。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序程顺序。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数
- [in] oob_default 分配越界元素的默认值

cub::LoadDirectBlockedVectorized

```
template<typename T , int ITEMS_PER_THREAD>
__device__ __forceinline__ void cub::LoadDirectBlockedVectorized
(     int      linear_tid,
      T *      block_ptr,
      T(&)     items[ITEMS_PER_THREAD]
)
```

将线性项目段载入到跨线程块的分块排列中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定为行-主线程顺序。

输入偏移量 (block_ptr + block_offset) 必须是四项对齐的。

以下条件将防止矢量化，而且加载将回退到 cub::BLOCK_LOAD_DIRECT：

- ITEMS_PER_THREAD 为奇数
- 数据类型 T 不是原始内置的或者 MXMACA 向量类型 (例如，short, int2, double, float2, etc.)

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_ptr 用来加载的输入指针
- [out] items 要加载的数据

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(    int     linear_tid,
     InputIteratorT     block_itr,
     InputT(&)     items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到跨线程块的条带排列中。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(    int     linear_tid,
     InputIteratorT     block_itr,
     InputT(&)     items[ITEMS_PER_THREAD],
     int     valid_items
)
```

将项目的线性部分加载到一个跨线程块的分块排列中，按范围保护

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数

cub::LoadDirectStriped

```
template<int BLOCK_THREADS, typename InputT , typename DefaultT , int
↪ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectStriped
(     int     linear_tid,
      InputIteratorT     block_itr,
      InputT(&)     items[ITEMS_PER_THREAD],
      int     valid_items,
      DefaultT     oob_default
)
```

将项目的线性段加载到贯穿线程块的条带排列中，按范围保护，并将越界元素进行回退分配。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有项目 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数
- [in] oob_default 分配越界元素的默认值

cub::LoadDirectWarpStriped

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(     int     linear_tid,
      InputIteratorT     block_itr,
      InputT(&)     items[ITEMS_PER_THREAD]
)
```

将项目的线性段加载到跨线程块的线程束条带排列中。

假设跨线程块元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是架构中线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型（可能是一个简单的指针类型）。

参数

- [in] linear_tid 用于调用线程的一个合适的一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输入迭代器，用于从中加载
- [out] items 要加载的数据

cub::LoadDirectWarpStriped

```
template<typename InputT , int ITEMS_PER_THREAD, typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(     int      linear_tid,
      InputIteratorT     block_itr,
      InputT(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将线性项目段载入到跨分块线程块的线程束条带排列中，按范围保护。

假设跨线程块元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是结构中线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 用于加载的线程块的基输入迭代器
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数

ub::LoadDirectWarpStriped

```
template<typename InputT , typename DefaultT , int ITEMS_PER_THREAD,
↪typename InputIteratorT >
__device__ __forceinline__ void cub::LoadDirectWarpStriped
(    int     linear_tid,
     InputIteratorT     block_itr,
     InputT(&)     items[ITEMS_PER_THREAD],
     int     valid_items,
     DefaultT     oob_default
)
```

将项的线性段加载到跨线程块的线程束条带排列中，受范围保护，并在超出范围的项上进行回退赋值。

假设项按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续项，每个线程分别拥有项 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是结构中线程束大小的倍数。

模板参数

- T **[推断]** 要加载的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- InputIteratorT **[推断]** 输入的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid 对于二维的线程块)
- [in] block_itr 用于加载的线程块的基输入迭代器
- [out] items 要加载的数据
- [in] valid_items 要加载的有效项目数
- [in] oob_default 分配越界项的默认值

cub::StoreDirectBlocked

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectBlocked
(    int     linear_tid,
     OutputIteratorT     block_itr,
     T(&)     items[ITEMS_PER_THREAD]
)
```

将跨线程块的分块排列的项存储到项的线性段中

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [in] items 要存储的数据

cub::StoreDirectBlocked

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectBlocked
(    int     linear_tid,
     OutputIteratorT     block_itr,
     T(&)     items[ITEMS_PER_THREAD],
     int     valid_items
)
```

将跨线程块的分块排列的项目存储到项目的线性段中，由范围保护

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假定顺序为行主序。

模板参数

- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型（可以是简单的指针类型）。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项目数

cub::StoreDirectBlockedVectorized

```
template<typename T , int ITEMS_PER_THREAD>
__device__ __forceinline__ void cub::StoreDirectBlockedVectorized
(    int     linear_tid,
     T *     block_ptr,
     T(&)    items[ITEMS_PER_THREAD]
)
```

将跨线程块的分块排列的项目存储到项目的线性段中。

假设 (block-threads*items-per-thread) 个项在线程块上按分块排列，其中 thread[i] 拥有第 i 个范围的 items-per-thread 个连续元素。对于多维线程块，假设行主线程顺序。

输出偏移量 (block_ptr + block_offset) 必须是四项对齐，返回的默认起始偏移量 mcMalloc()

以下条件将阻止矢量化，存储将退回到 cub::BLOCK_STORE_DIRECT: - ITEMS_PER_THREAD 为奇数 - 数据类型 T 不是内置的原语或 MXMACA 向量类型 (例如，short, int2, double, float2, etc.)

模板参数

- T **[推断]** 要存储的数据类型

- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_ptr 用于存储的输入指示器
- [in] items 要储存的数据

cub::StoreDirectStriped

```
template<int BLOCK_THREADS, typename T , int ITEMS_PER_THREAD, typename
↪OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD]
)
```

将跨线程块的条带排列的数据存储到项目的线性段中。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有元素 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 以线程为单位的线程块大小
- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [in] items 要存储的数据

cub::StoreDirectStriped

```
template<int BLOCK_THREADS, typename T , int ITEMS_PER_THREAD, typename
↪OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将跨线程块的条带排列的数据存储到项目的线性段中，由范围保护。

假设 (block-threads*items-per-thread) 个项在线程块上呈条带排列，其中 thread[i] 拥有元素 (i)，(i + block-threads)，…，(i + (block-threads * (items-per-thread-1)))。对于多维线程块，假定顺序为行主序。

模板参数

- BLOCK_THREADS 线程中的线程块大小
- T **[推断]** 要存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项的数量

cub::StoreDirectWarpStriped

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectWarpStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD]
)
```

将跨线程块的线程束条带排列的数据存储到项目的线性段中。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续元素，每个线程分别拥有元素 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是结构中线程束大小的倍数。

模板参数

- T **[推断]** 存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可能是一个简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [out] items 待加载数据

cub::StoreDirectWarpStriped

```
template<typename T , int ITEMS_PER_THREAD, typename OutputIteratorT >
__device__ __forceinline__ void cub::StoreDirectWarpStriped
(     int      linear_tid,
      OutputIteratorT      block_itr,
      T(&)      items[ITEMS_PER_THREAD],
      int      valid_items
)
```

将跨线程块的线程束条带排列的数据存储到项目的线性段中，由范围保护。

假设跨线程元素按线程束条带排列，其中 warp[i] 拥有第 i 个范围的 (warp-threadsitems-per-thread) 个连续项目，每个线程分别拥有项目 (i)，(i + warp-threads)，…，(i + (warp-threads(items-per-thread-1)))。

使用注意事项

线程块中的线程数必须是结构中线程束大小的倍数。

模板参数

- T **[推断]** 存储的数据类型。
- ITEMS_PER_THREAD **[推断]** 分配到每个线程上的连续项数量。
- OutputIteratorT **[推断]** 输出的随机访问迭代器类型 (可以是简单的指针类型)。

参数

- [in] linear_tid 调用线程的合适一维线程标识符 (例如，(threadIdx.y * blockDim.x) + linear_tid for 2D thread blocks)
- [in] block_itr 线程块的基本输出迭代器，用于存储
- [in] items 要存储的数据
- [in] valid_items 要写入的有效项的数量

cub::ThreadLoad

```
template<CacheLoadModifier MODIFIER, typename InputIteratorT >
__device__ __forceinline__ cub::detail::value_t<InputIteratorT>
↪cub::ThreadLoad (InputIteratorT     itr)
```

使用 cub::CacheLoadModifier 读取内存的线程实用程序。可用于加载任何数据类型。

例如

```
#include <cub/cub.cuh>   // 或 <cub/thread/thread_load.cuh>
// 32位加载使用 cache-global 修饰符:
int *d_in;
int val = cub::ThreadLoad<cub::LOAD_CA>(d_in + threadIdx.x);
// 16位加载使用默认修饰符
short *d_in;
short val = cub::ThreadLoad<cub::LOAD_DEFAULT>(d_in + threadIdx.x);
// 256位加载使用缓存易失修饰符
double4 *d_in;
double4 val = cub::ThreadLoad<cub::LOAD_CV>(d_in + threadIdx.x);
// 96位加载使用缓存流修饰符
struct TestFoo { bool a; short b; };
TestFoo *d_struct;
TestFoo val = cub::ThreadLoad<cub::LOAD_CS>(d_in + threadIdx.x);
```

模板参数

- MODIFIER **[推断]** CacheLoadModifier 枚举
- InputIteratorT **[推断]** 输入迭代器类型 (可以是简单的指针类型)

cub::ThreadStore

```
template<CacheStoreModifier MODIFIER, typename OutputIteratorT , typename
↪T >
__device__ __forceinline__ void cub::ThreadStore
(     OutputIteratorT     itr,
      T      val
)
```

使用 cub::CacheStoreModifier 缓存修饰符编写内存的线程实用程序。可以用于存储任何数据类型。

实例

```
#include <cub/cub.cuh>   // 或 <cub/thread/thread_store.cuh>
// 32位加载使用 cache-global 修饰符:
int *d_out;
int val;
cub::ThreadStore<cub::STORE_CG>(d_out + threadIdx.x, val);
// 16位加载使用默认修饰符
short *d_out;
short val;
cub::ThreadStore<cub::STORE_DEFAULT>(d_out + threadIdx.x, val);
// 256位加载使用缓存易失修饰符
double4 *d_out;
double4 val;
cub::ThreadStore<cub::STORE_WT>(d_out + threadIdx.x, val);
// 96位加载使用缓存流修饰符
struct TestFoo { bool a; short b; };
TestFoo *d_struct;
TestFoo val;
cub::ThreadStore<cub::STORE_CS>(d_out + threadIdx.x, val);
```

模板参数

- MODIFIER **[推断]** CacheStoreModifier 枚举
- InputIteratorT **[推断]** 输出迭代器类型 (可能是简单的指针类型)
- T **[推断]** 输出值的数据类型

2.3.3 设备、内核和存储管理

2.3.3.1 类

struct cub::CachingDeviceAllocator

一个简单的缓存分配器，用于设备内存分配。

概述

该分配器是线程安全和流安全的，能够管理多个设备上缓存分配。它的表现如下：

- 来自分配器的分配值与 active_stream 相关联。释放后，分配将立即在分配期间与其关联的流中重用，并且当提交给 active_stream 的所有先前工作完成后，该分配可在其他流中重用。
- 内存分配按 bin 大小分类并缓存。一个给定大小的新分配请求只会考虑对应 bin 中的缓存分配。
- 根据构造过程中提供的增长因子 bin_growth 对进程进行几何限制。较大的 bin 缓存中未使用的设备分配不会被归类到较小的 bin 中的分配请求重用。
- 分配请求如低于 (bin_growth ^ min_bin) 将向上取为 (bin_growth ^ min_bin)。

- 分配值如高于 (bin_growth ^ max_bin) 不会舍入到最近的 bin 中，并且在解除它们时被释放，而不是返回到 bin 缓存中。
- 如果给定设备上缓存分配的总存储空间超过 max_cached_bytes，那么在解除分配给该设备的内存时，只会简单地释放这些内存，而不会返回到它们的 bin-cache 中。

例如，默认构造的 CachingDeviceAllocator 被配置为：

- bin_growth = 8
- min_bin = 3
- max_bin = 7
- max_cached_bytes = 6MB - 1B

它划定了 5 个存储器大小:512B、4KB、32KB、256KB 和 2MB，并设置每个设备的最大缓存字节为 6,291,455

struct cub::SwitchDevice

RAII 助手（RAII helper），在构造时保存当前设备并切换到指定设备，在销毁时切换到保存的设备。

struct cub::KernelConfig

内核调度配置

struct cub::ChainedPolicy

```
template<
    int PTX_VERSION,
    typename PolicyT,
    typename PrevPolicyT>

struct cub::ChainedPolicy< PTX_VERSION, PolicyT, PrevPolicyT >
```

用于分派到策略链的帮助程序（Helper）

struct cub::ChainedPolicy

```
template<
    int PTX_VERSION,
    typename PolicyT>

struct cub::ChainedPolicy< PTX_VERSION, PolicyT, PolicyT >
```

分配到策略链的助手 (链末端专业化（end-of-chain specialization）)

2.3.3.2 宏

```
#define CubDebug(e)   CUB_NS_QUALIFIER::Debug((mcError_t) (e), __FILE__, __
↪LINE__)
```

调试宏。

```
#define CubDebugExit(e)    if (CUB_NS_QUALIFIER::Debug((mcError_t) (e), __
↪FILE__, __LINE__)) { exit(1); }
```

用 exit 调试宏。

```
#define _CubLog(format,...)    printf(format,__VA_ARGS__);
```

printf 语句的日志宏。

2.3.3.3 功能

cub:: 调试

```
__host__ __device__ __forceinline__ mcError_t cub:: 调试
(    mcError_t     error,
     const char *     filename,
     int     line
)
```

CUB 错误报告宏 (向 stderr 打印错误信息)。

如果定义了 CUB_STDERR 并且 error 不是 mcSuccess，则将相应的错误消息连同提供的上下文一起打印到 stderr(或设备代码中的 stdout)。

返回内容

MXMACA 错误。

cub::CurrentDevice

```
CUB_RUNTIME_FUNCTION int cub::CurrentDevice()
```

返回当前设备，如果发生错误则返回-1。

cub::DeviceCountUncached()

```
CUB_RUNTIME_FUNCTION int cub::DeviceCountUncached()
```

返回可用 MXMACA 设备的数量，如果发生错误则返回-1。

cub::DeviceCount()

```
CUB_RUNTIME_FUNCTION int cub::DeviceCount()
```

返回可用 MXMACA 设备的数目。

注解:

- 这个函数可能会在内部缓存结果。
- 这个函数是线程安全的。

---

cub::PtxVersionUncached

```
CUB_RUNTIME_FUNCTION mcError_t cub::PtxVersionUncached (int &     ptx_
↪version)
```

获取将在当前设备上使用的 PTX 版本 (major * 100 + minor * 10)。

cub::PtxVersionUncached

```
__host__ mcError_t cub::PtxVersionUncached
(     int &     ptx_version,
      int     device
)
```

获取设备上使用的 PTX 版本 (major * 100 + minor * 10)。

cub::PtxVersion

```
__host__ mcError_t cub::PtxVersion
(     int &     ptx_version,
      int     device
)
```

获取设备上使用的 PTX 版本 (major * 100 + minor * 10)。

注解:

- 这个函数可以在内部缓存结果。
- 这个函数是线程安全的。

cub::PtxVersion

```
CUB_RUNTIME_FUNCTION mcError_t cub::PtxVersion (int &     ptx_version)
```

获取将在当前设备上使用的 PTX 版本 (major * 100 + minor * 10)。

注解:

- 这个函数可能在内部缓存结果。
- 这个函数是线程安全的。

cub::SmVersionUncached

```
CUB_RUNTIME_FUNCTION mcError_t cub::SmVersionUncached
(     int &     sm_version,
      int     device = CurrentDevice()
)
```

获取设备的 SM 版本信息 (major * 100 + minor * 10)。

cub::SmVersion

```
CUB_RUNTIME_FUNCTION mcError_t cub::SmVersion
(     int &      sm_version,
      int      device = CurrentDevice()
)
```

获取设备的 SM 版本信息 (major * 100 + minor * 10)。

注解:

- 这个函数可能会在内部缓存结果。
- 这个函数是线程安全的。

cub::SyncStream

```
CUB_RUNTIME_FUNCTION mcError_t cub::SyncStream (mcStream_t      stream)
```

同步指定的流。

cub::MaxSmOccupancy

```
template<typename KernelPtr >
CUB_RUNTIME_FUNCTION mcError_t cub::MaxSmOccupancy
(     int &      max_sm_occupancy,
      KernelPtr      kernel_ptr,
      int      block_threads,
      int      dynamic_smem_bytes = 0
)
```

计算在当前设备上执行给定内核函数指针 kernel_ptr 的最大 SM 占用率，每个线程块有 block_threads 个线程。

代码段

下面的代码片段说明了 maxsmocupancy 函数的使用。

```
#include <cub/cub.cuh>   // 或 <cub/util_device.cuh>
template <typename T>
__global__ void ExampleKernel()
{
    // 为块扫描分配共享内存
    __shared__ volatile T buffer[4096];
    ...
}
    ...
// 确定实例内核对无符号字符的占用率
int max_sm_occupancy;
MaxSmOccupancy(max_sm_occupancy, ExampleKernel<unsigned char>, 64);
```

参数

- [out] max_sm_occupancy 单个 SM 上可以驻留的线程块的最大数目
- [in] kernel_ptr 计算 SM 占用率的内核指针

- [in] block_threads 每个线程块的线程数
- [in] dynamic_smem_bytes 以字节为单位的动态分配的共享内存。默认值为 0。

2.4 GridModule

2.4.1 类

2.4.1.1 cub::GridBarrier

GridBarrier 实现了 MXMACA 网格中线程块之间的软件全局屏障。

2.4.1.2 cub::GridBarrierLifetime

GridBarrierLifetime 扩展 GridBarrier，提供合作所需的临时设备存储的生命周期管理。

使用 RAII（资源获取即初始化）来管理生命周期，即在析构函数被调用时回收设备资源。

2.4.1.3 cub::GridEvenShare

```
template<
    typename OffsetT>

struct cub::GridEvenShare< OffsetT >
```

GridEvenShare 是一个描述符实用程序，用于以“均匀共享”的方式在 MXMACA 线程块之间分配输入。每个线程块得到大致相同的输入块的数量。

概述

每个线程块被分配一个连续的输入块序列。为了帮助保持对齐并消除除了最后一个线程块之外的所有受保护负载的开销，gridvenshare 将三种不同数量的工作中的一种分配给给定的线程块:“大”、“正常”或“最后”。“大”的工作负载是比“正常”的调度粒度更大的任务。如果输入不是调度粒度大小的偶数倍，则最后一个线程块的“最后一个”工作单元可能是部分满的。在调用子网格之前，父线程通常会构造一个 griddevenshare 的实例。该实例可以传递给子线程块，子线程块可以使用 BlockInit() 初始化它们的每个线程块的偏移量。

2.4.1.4 cub::GridQueue

```
template<
    typename OffsetT>

class cub::GridQueue< OffsetT >
```

GridQueue 是一个用于动态队列管理的描述符实用程序。

概述

GridQueue 描述符提供了用于“filling”或“draining”全局共享向量的抽象概念。

“filling”GridQueue 的工作原理是以原子方式添加到零初始化的计数器中，为调用线程返回唯一偏移量以写入其项。GridQueue 维护总的“fill-size”。在将要填充的内核实例之前，主机或内核实例必须使用 GridQueue::ResetFill 重置 fill 计数器。

同样，“draining”GridQueue 的工作原理是以原子方式递增零初始化的计数器，为调用线程读取其项返回唯一偏移量。线程可以安全地耗尽内存，直到超出数组的逻辑填充数大小。在将要填充的内核实例之前，主机或内核实例必须使用 GridQueue::ResetDrain 或 GridQueue::FillAndResetDrain 重置 drain 计数器。（对于现有数据的动态工作分布，对应的 fill-size 只是数组中项的数量。）

迭代工作管理可以简单地用一对交替使用的工作缓冲器实现，每个缓冲器都有一组相关联的填充和耗尽 GridQueue 描述符。

模版参数

OffsetT 全局偏移量的有符号整型

2.4.2 枚举

2.4.2.1 enum cub::GridMappingStrategy

cub::GridMappingStrategy 列举了将设备范围内的常量大小的数据块映射到 MXMACA 线程块网格上的替代策略。

GRID_MAPPING_RAKE

一种‘raking’访问模式，在这种模式中，分配给每个线程块的输入数据块（tiles）以网格的大小为步长进行分离。

概述

输入被均匀地划分为 p 个段，其中 p 是常数，大致对应于可能驻留在目标设备上的线程块的数目。每个片段由连续的 tile 组成，其中 tile 是一个小的、固定大小的输入单元，在线程块终止或获得更多工作之前被处理完成。内核调用 p 个线程块，每个线程块以块大小的增量迭代地消耗 n/p 个项目的段。

GRID_MAPPING_STRIP_MINE

一种 strip mining 访问模式，在这种模式中，分配给每个线程块的输入数据块以网格的大小为步长进行分离。

概述

输入值被均匀地划分为 p 个集合，其中 p 是常数，大致对应于可能驻留在目标设备上的线程块的数目。每个集合由由 stride tiles 分隔的数据 tiles 组成，其中 tile 是一个小的、固定大小的输入单元，在线程块终止或获得更多工作之前被处理完成。内核调用 p 个线程块，每个线程块以块大小的增量迭代地消耗 n/p 个项目的段。

GRID_MAPPING_DYNAMIC

动态的基于队列（queue-based）的策略，用于将输入块分配给线程块。

概述

输入被视为一个由线程块网格动态使用的队列。工作将自动以块的形式从队列中取出，其中块是在线程块终止或获得更多工作之前要处理完成的输入单元网格大小 p 是恒定的，与目标设备上可能存在的线程块数量大致对应。