rocPRIM Documentation

Release 3.4.1

Advanced Micro Devices, Inc.

Aug 08, 2025

CONTENTS

rocPRIM is a header-only library that provides HIP parallel primitives. The purpose of the library is to ease the maintainability of performant, yet portable GPU-accelerated code on the AMD ROCm platform. rocPRIM is written in HIP and has been optimized for AMD's latest discrete GPUs. For more information refer to *Introduction to rocPRIM*.

The code is open and hosted at: https://github.com/ROCmSoftwarePlatform/rocPRIM

The rocPRIM documentation is structured as follows:

Conceptual

- *Introduction to rocPRIM*
- *Glossary*
- *Performance tuning*
- *Developer guidelines*

API reference

- *Summary of the Operations*
- *Data type support*
- *Device-Wide Operations*
- *Block-Wide Operations*
- *Warp-Level Operations*
- *Thread-Level Operations*
- *Iterators*
- *Intrinsics*
- *Implementing traits for custom types in rocPRIM*

To contribute to the documentation refer to Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

ROCPRIM CONCEPTS

- *Introduction to rocPRIM*
- *Glossary*
- *Developer guidelines*
- *Performance tuning*
- *Implementing traits for custom types in rocPRIM*

1.1 Introduction to rocPRIM

1.1.1 Operations and Sequences

A rocPRIM operation is a computation over a sequence of objects. A rocPRIM operation can return a single value like the `reduce` operation; return another sequence like the `sort` operation; or return multiple sequences like the `partition` operation. The elements of the sequence could be of any type or class, although template specialization allows rocPRIM to optimize the computations over the usual numerical datatypes. Operations accept input in the form of `iterators` that point to a sequence of objects to process, and write output to a mutable `iterator`.

A high level view of the available operations can be found on *Summary of the Operations*. rocPRIM includes a variety of generic operations that are frequently very useful.

Note

Refer to *Data type support* for information on supported datatypes.

1.1.2 Scope

An important property of a rocPRIM operation is its scope, which determines the level of the computing model used for processing the operation. The scope determines which parts of the GPU will cooperate to compute the result. The scope has a direct influence on how the data will be subdivided and processed by the computing units or VALUs. The rocPRIM operation scopes are:

- *Device/Grid* the operation and data will be split and dispatched to all the CUs.
- *Block* The operation should take place within the same block by the same CU.
- *Warp* as above but with a warp and a VALU.
- *Thread* The operation will take place sequentially in the same thread. Thread-wide operations are also called *Utilities* since they coincide with utility functions used on a CPU.

The scope has an impact on how the operation is initiated:

- *Device/Grid* it is a kernel, thus it is dispatched with its own grid/block dimensions.
- *Block/Warp/Thread* it is a function call, and inherits the dimensions of the current kernel.

This also dictates how synchronization should be done to wait for completion:

- *Device/Grid* Synchronization is done via wait lists and queue barriers (`stream`).
- *Block/Warp/Thread* it is in the same control flow of the caller threads. Synchronization is done via memory barriers.

1.2 Glossary

This glossary is to help users understand the basic concepts or terminologies used in the rocPRIM library.

Warp
Refers to a group of threads that execute in single instruction, multiple thread (SIMT) fashion. Also known as wavefronts on AMD GPUs.

Hardware Warp Size
Refers to the number of threads in a warp defined by the hardware. On Nvidia GPUs a warp size is 32, while on AMD GPUs a warp size is 64.

Logical Warp Size
Refers to the number of threads in a warp defined by the user, which can be equal to or less than the size of the hardware warp size.

Lane ID
Refers to the thread identifier within the warp. A logical lane ID refers to the thread identifier in a "logical warp", which can be smaller than a hardware warp size (and can be defined as `lane_id() % WarpSize`).

Warp ID
Refers to the identifier of the hardware/logical warp in a block. Warp ID is guaranteed to be unique among warps.

Block
Refers to a group of threads that are executed on the same compute unit (streaming multiprocessor). These threads can be indexed using 1 Dimension {X}, 2 Dimensions {X, Y} or 3 Dimensions {X, Y, Z}. A block consists of multiple warps.

Tile
Refers to a block in C++AMP/HIPCC nomenclature.

Flat ID
Refers to a flattened identifier of a block (tile) or a thread identifier. Flat ID is a 1D value created from 2D or 3D identifier. For example the flat ID of thread ID (X, Y) in 2D thread block 128x4 (XxY) is `Y * 128 + X`.

1.3 Performance tuning

Algorithms often perform better if their launch parameters (number of blocks, block size, items per thread, etc.) are tailored for the particular architecture they are run on. rocPRIM achieves this by passing structs called configs to algorithms as template parameters. A config struct encapsulates all the information that's needed to run a particular algorithm in the most performant way for a specific device. Default configurations (non custom-defined by users) can be selected by device code at compile time, since the architecture is known, while device algorithms detect at runtime which configuration should be used.

What we call *autotuning* is a method of generating the above-mentioned architecture-optimized default configurations for algorithms. The process to run the autotune is described below, as well as all the templates and scripts used.

1. Configure the project for autotuning. Autotune is an extension on top of the regular benchmarking process and it is enabled with a CMake option `BENCHMARK_CONFIG_TUNING`, which doubles as a C++ macro to determine whether autotuning is enabled.
2. When the project is configured, a large amount of C++ benchmark files are generated with variation in parameters such as block size, items per thread, and method. The files are generated based on a template (`benchmark/benchmark_*.parallel.cpp.in`) and arguments defined in `ConfigAutotuneSettings.cmake`. CMake will automatically detect when the input template changes and will reconfigure the required files as necessary.
3. Compile results in one executable based on all generated files for an algorithm.
4. Run the executable and gather the JSON output files. The generation of output files is triggered by the use of `--benchmark_out_format=json` and `--benchmark_out=<output_file_name>.json` options when running the executable.
5. Convert the benchmark results into a config with `scripts/autotune/create_optimization.py`. This python script injects the optimal configurations into the templates in `scripts/autotune/templates`.

- The option `--out_basedir` can be used to place the output config(s) in a specific path, otherwise the config(s) will be placed in the current directory.

6. If `--out_basedir rocprim/include/device/detail/config` was not used in the previous step, place the generated config(s) from the output path to `rocprim/include/device/detail/config`.

1.3.1 Device-level algorithm dependencies

Due to the modularity of rocPRIM, some device-level algorithms depend on other device-level algorithms. The implication is that when an algorithm is changed, the performance of algorithms that depend on it must be checked as well. This also applies to configuration tuning. Below is a list of device-level algorithm dependencies and additional considerations for the tuning of these algorithms.

- `lower_bound`, `upper_bound`, and `binary_search` depend on `transform`. However, all these algorithms are all tuned separately.
- `merge_sort` has two stages that are tuned separately: `merge_sort_block_sort` and `merge_sort_block_merge`. Since the latter algorithm depends on the sorted block size, the best `merge_sort` configurations are obtained by tuning `merge_sort_block_sort` first, adding the configurations, and then tuning `merge_sort_block_merge`.
- `partition_two_way`, `partition`, `partition_three_way`, `select`, `unique`, and `unique_by_key` all use the same underlying implementation. However, all these algorithms have separate tuning.
- `radix_sort` has three sub-algorithms: `radix_sort_block_sort`, `radix_sort_onesweep`, and a merge sort for small sizes. `radix_sort_block_sort` and `radix_sort_onesweep` are tuned. The threshold for radix sort that determines whether to perform merge sort or onesweep is manually set.
- `segmented_radix_sort` depends on `partition` and `partition_three_way` but does not use the tuned configurations.
- `segmented_reduce` does not depend on `reduce` but uses the same tuned configurations.
- `segmented_scan` does not depend on `scan` but uses the same tuned configurations.
- `run_length_encode` depends on `reduce_by_key`, but uses separate tuned configurations. `run_length_encode_non_trivial_runs` depends on `reduce_by_key` and `select`, but does not use the tuned configurations from `reduce_by_key` and `select`.

1.4 Developer guidelines

1.4.1 Overview

As explained in *Introduction to rocPRIM*, rocPRIM's operations are part of one of four different hierarchical scopes: *Device/Grid*, *Block*, *Warp*, or *Thread*. This division facilitates re-use in the codebase and provides flexibility for users. Additional developer considerations are:

- *Device/Grid*: algorithms called from host code, executed on the entire device. The input size is variable and passed as an argument to the function.
- *Block*: algorithms that are called from device code, executed by one thread block. All threads in a thread block should participate in the function call, and the threads together perform the algorithm. They are defined as structures, to group similar overloads and provide associated types such as the `storage_type` defining shared memory storage requirements. The maximum input size is defined by template arguments. Optionally, an actual size can be defined by `valid_items` overloads.
- *Warp*: algorithms called from device code, executed by one warp. In many ways these algorithms are similar to the block-level algorithms, the key difference is that all threads in a warp collectively perform the algorithm. Through template arguments, a logical warp size can be specified.
- *Thread*: algorithms called from the device and perform work in a single thread without communicating with other threads.

See the contributing guide CONTRIBUTING.md for information on file structure and how test and benchmarks should be implemented.

1.4.2 General rules

Code should be modular, and when possible broader scoped to facilitate reuse. If there is no adverse effect on performance, extract common functionality. The different hierarchies of the API are not only for the user, algorithm implementations use these endpoints as well. For instance, device-level algorithms typically use the block-level algorithms for loading and storing data.

It should be clear from function template parameters whether they are tuning options that do not affect behavior, or are algorithmic parameters that change behavior. For instance, tuning options may be block size, items per thread, or the block-level scan method (`block_scan_algorithm`). An algorithmic parameter could be whether a scan has an initial value, or whether a reduction is inclusive or exclusive. An example of an enumeration that violates this rule is `block_load_method`, where the different members make different orders of the elements.

Between minor ROCm versions, breaking changes in the public API MUST NOT be introduced. Everything in the namespace `rocprim` is considered public API, based on the assumption that a user may in theory depend on it. Pay special attention not to break backward-compatibility, as it can be done in subtle ways. For example, many functions allow user-defined types, which behave differently in many ways from fundamental types. Be defensive in what is placed in the public API as sustaining backward-compatibility is a burden on maintenance. If it is not necessary to be exposed, place it in `rocprim::detail` (or lower) instead. A common additional check is to make sure downstream libraries still compile and execute tests successfully (hipCUB, rocThrust, Tensorflow, and PyTorch).

HIP Graphs are a way to capture all stream-ordered HIP API calls into a graph without executing, and then replaying the graph many times afterwards. Supporting graph capture makes rocPRIM more flexible to use, and all device-level algorithms should strive to allow it. Among other things, one general requirement is that the number of kernel calls and the launch parameters of kernel calls should not depend on input data. If support is not possible for a specific algorithm, the documentation should state this clearly.

1.4.3 Configurations and architecture dispatch

One of the most complex parts of rocPRIM is the mechanism that allows for the user-provided configuration and defaulted automatic configuration of device-level algorithms.

Default and user-specified configuration

As explained in *Performance tuning*, device-level algorithms may be configured by accepting a tuning config. It may be provided by the caller, or defaulted to `default_config`, which selects a suitable default configuration.

The number of threads in a block (the "block size") is a quintessential configuration parameter for kernels. It needs to be known at the host side to launch the kernel and at the device side at compile-time for the generation of algorithmic functions. HIP code is compiled in multiple passes, one pass for the host and one pass for each targeted device architecture. When a kernel is launched on the host, the HIP runtime selects the binary based on the device associated with the HIP stream. Since the configuration, and thus the block size, depends on this device architecture, rocPRIM must have a similar mechanism to infer the architecture of the device based on the the HIP stream.

To facilitate a dispatching mechanism supporting the above requirements, several standardized structures need to be defined for each algorithm, which is outlined in this section. These structures depend on a generalized dispatching mechanism.

The algorithm's configuration struct is defined in `rocprim/device/detail/device_config_helper.hpp`. The reason for putting all configurations in one file is to make the configuration templates simpler (generating configurations is explained *Performance tuning*). The tuning config has the name `ALGO_config`, and no members (unless for backward-compatibility reasons), only template parameters.

The config struct derives from a non-public parameter struct holding the actual parameters. This separation between structs is done to facilitate change without breaking public API.

```
namespace detail
{

struct ALGO_config_params
{
    unsigned int BlockSize;
    unsigned int ItemsPerThread;
};

} // namespace detail

template<unsigned int BlockSize, unsigned int ItemsPerThread>
struct ALGO_config : public detail::ALGO_config_params
{
    constexpr ALGO_config() : detail::ALGO_config_params{BlockSize, ItemsPerThread}
    {}
}
```

In order to accept either `default_config` or `ALGO_config` as the device-level configuration template type and convert it to a parameter instance, a non-public config wrapper is defined in `rocprim/device/device_ALGO_config.hpp`.

```
namespace detail {

// generic struct that instantiates custom configurations
template<typename ALGOConfig, typename>
struct wrapped_ALGO_config
{
```

(continues on next page)

(continued from previous page)

```
    template<target_arch Arch>
    struct architecture_config
    {
        static constexpr ALGO_config_params params = ALGOConfig();
    };
};

// specialized for rocprim::default_config, which instantiates the default_ALGO_config
template<typename Type>
struct wrapped_ALGO_config<default_config, Type>
{
    template<target_arch Arch>
    struct architecture_config
    {
        static constexpr ALGO_config_params params = default_ALGO_config<static_cast
↪<unsigned int>(Arch), Type>();
    };
};

} // namespace detail
```

Selecting the default configuration is done based on the target architecture `target_arch` and typically also on the input types of the algorithm (in the example above, a single type `Type` is used). The `default_ALGO_config` is defined in `rocprim/include/device/detail/config/device_ALGO.hpp`. This file will be generated by the autotuning process, as explained in *Performance tuning*. The files look like this:

```
namespace detail
{

// base configuration in case no specific configuration exists
template<unsigned int arch, typename Type, class enable = void>
struct default_ALGO_config : default_ALGO_config_base<Type>::type
{};

// generated configuration for architecture gfx1030, based on float
template<class Type>
struct default_ALGO_config<
    static_cast<unsigned int>(target_arch::gfx1030),
    Type,
    std::enable_if_t<bool(rocprim::traits::get<value_type>().is_floating_point()) &&␣
↪(sizeof(value_type) <= 4) && (sizeof(value_type) > 2)>>
    : ALGO_config<256, 16>
{};

// many generated configurations..

} // namespace detail
```

It is up to the implementer to specify a suitable and generic base configuration. This base configuration is not placed in the template to make the template simpler. Instead, it is defined in `rocprim/device/detail/device_config_helper.hpp`:

```
namespace detail
{

template<typename Type>
struct default_ALGO_config_base
{
    using type = ALGO_config<256, 4>;
};

} // namespace detail
```

Finally, the kernel is templatized with the `wrapped_ALGO_config` and not the actual configuration parameters. It is done so that the architecture enumeration value (or any dependent configuration parameters) does not appear in the function signature. This prevents a host-side switch statement over the architecture values to select the right kernel to launch. Instead, this selection is done at compile time in device code.

Config dispatch

The default configuration depends on the types of the input values of the algorithm, as well as the device architecture. The device architecture is determined at runtime, based on the HIP stream. At the host side, the configuration parameters are selected at runtime using the following pattern:

```
using config = wrapped_ALGO_config<config, Type>;

detail::target_arch target_arch;
hipError_t          result = host_target_arch(stream, target_arch);
if(result != hipSuccess)
{
    return result;
}
const ALGO_config_params params = dispatch_target_arch<config>(target_arch);
```

In device code the device architecture is known at compile time, and thus the configuration can also be selected at compile time. All that is needed, is the following pattern:

```
constexpr ALGO_CONFIG_PARAMS params = device_params<config>();
```

The `device_params` function selects the configuration based on the predefined compiler macro `__amdgcn_processor__`. In the example, `config` is of type `wrapped_ALGO_config` as in the host example.

1.4.4 Common patterns

There are several patterns throughout rocPRIM's codebase for uniformity and enforcing good practice.

Temporary storage allocation

If a device-level function requires temporary storage, `void* temporary_storage` and `size_t& storage_size` will be the first two parameters. When calling the function with `nullptr` for `temporary_storage`, the function will set `storage_size` to the required number of temporary device memory bytes. If no temporary storage is required under specific circumstances, `storage_size` should be set to a small non-zero value, to prevent the users from having to check before making a zero-sized allocation.

Common functionality in the `detail::temp_storage` namespace is used to calculate the required storage on the first function call and assign pointers in the second function call. The below example allocates and assigns a temporary array of ten integers.

```
hipError_t function(void* temporary_storage, size_t& storage_size)
{
    int* d_tmp{};

    // if temporary_storage is nullptr, sets storage_size to the required size
    // else, assigns the pointer d_tmp
    const hipError_t partition_result = detail::temp_storage::partition(
        temporary_storage,
        storage_size,
        detail::temp_storage::make_linear_partition(
            detail::temp_storage::ptr_aligned_array(&d_tmp, 10)));
    if(partition_result != hipSuccess || temporary_storage == nullptr)
    {
        return partition_result;
    }

    // perform the function with temporary memory
    return function_impl(d_tmp);
}
```

Reusing shared memory

Shared memory reuse in a kernel is facilitated by placing multiple `storage_type` declarations in a union.

```
using block_load_t = block_load<T, block_size>;
using block_scan_t = block_scan<T, block_size>;
using block_store_t = block_store<T, block_size>;

ROCPRIM_SHARED_MEMORY union
{
    typename block_load_t::storage_type  load;
    typename block_scan_t::storage_type  scan;
    typename block_store_t::storage_type store;
} storage;

T value;
block_load_t().load(input, value, storage.load);

syncthreads();

block_scan_t().scan(value, storage.scan);

syncthreads();

block_store_t().store(output, value, storage.store);
```

Partial block idiom

Since thread blocks have uniform sizes, bounds checking is necessary to prevent out-of-bounds loads and stores. Applying a check to every loaded and stored value may become a performance bottleneck. A typical solution is to have a block-wide check, whether a per-item check is necessary. A simple example is below.

```
// slow, adds a check for every stored item in each block
const unsigned int thread_id = detail::block_thread_id<0>();
const unsigned int block_id  = detail::block_id<0>();
const auto num_valid_in_last_block = input_size - block_offset;
block_store_t().store(
    output,
    values,
    num_valid_in_last_block,
    storage);

// fast, adds a check only for incomplete blocks (which can only be the last block)
constexpr unsigned int items_per_block = BlockSize * ItemsPerThread;
const bool is_incomplete_block = block_id == (input_size / items_per_block);
if(is_incomplete_block)
{
    block_store_t().store(
        output,
        values,
        num_valid_in_last_block,
        storage);
}
else
{
    block_store_t().store(
        output,
        values,
        storage);
}
```

Large indices

Typically, each thread handles a fixed amount of elements and HIP limits how many threads can be in a single launch. This means there is a hard limit to the number of elements that can be handled in a single kernel call. Special attention must be paid to how input sizes beyond this limit are handled. This is commonly handled by launching multiple kernels in a loop and combining results.

Naming of device-level functions

Typically, multiple overloads of device-level functions exist, that call into a common implementation. Below is an example of this pattern and what the naming should look like

```
BEGIN_ROCPRIM_NAMESPACE

namespace detail
{

ROCPRIM_KERNEL reduce_kernel(...)
{
    // reduce_kernel_impl defined in rocprim/device/detail/device_reduce.hpp
    reduce_kernel_impl(...);
}

template<bool HasInitialValue>
```

(continues on next page)

(continued from previous page)

```
hipError_t reduce_impl(...)
{
    reduce_kernel<<<...>>>(...);
}

} // namespace detail

// default reduce
hipError_t reduce(...)
{
    return detail::reduce_impl<false>(...);
}

// reduce overload with initial value
hipError_t reduce(...)
{
    return detail::reduce_impl<true>(...);
}

END_ROCPRIM_NAMESPACE
```

Synchronous debugging

All device-level functions have as a last parameter `bool debug_synchronous`, which defaults to `false`. This parameter toggles synchronization after kernel launches for debugging purposes. Typically, additional debugging information is printed as well.

Items per thread

Most device functions operate on a fixed number of elements and are templatized based on the element type. These functions will have an `unsigned int ItemsPerThread` template parameter, which specifies how many elements each thread should process. The main purpose of this parameter is to tune the performance of such a function. As different types are of different sizes, it is likely that there is no single `ItemsPerThread` value that gives good performance for types of all sizes. The `ItemsPerThread` value often directly influences register usage of a kernel, which influences the kernel's occupancy.

Kernel launch bounds

To guide the code generation process, it is possible to specify the maximum block size for a kernel with `__launch_bounds__()`. Since most kernels are templatized based on a configuration, a common pattern is the following:

```
template<class Config>
ROCPRIM_KERNEL __launch_bounds__(device_params<Config>().block_size) void kernel(...)
{}
```

Pitfalls and common mistakes

HIP code is compiled in multiple passes: one for the host and one for each targeted device architecture. As such, host code is agnostic of device architecture, and should be designed as such. Only with a `hipStream` can the device be inferred and can certain properties be obtained. Since device code is compiled for a specific architecture, it can contain compile-time optimizations for specific architectures. Note that AMD GPUs have a warp size of 32 or 64, and unless specialized, algorithms should work for both warp sizes.

All variables with the `__shared__` memory space specifier should either be in a function with the `__global__` (`ROCPRIM_KERNEL`) execution space specifier or in a function with the `__device__` (`ROCPRIM_DEVICE`) execution space specifier marked with `__forceinline__` (`ROCPRIM_FORCE_INLINE`). The reason for this is that without forcing the inlining of the function, the compiler may choose not to optimize shared memory allocations, leading to exceeding the limit dictated by hardware.

CHAPTER

TWO

ROCPRIM API REFERENCE

2.1 Summary of the Operations

2.1.1 Basics

- `transform` applies a function to each element of the sequence, equivalent to the functional operation `map`
- `select` takes the first N ` elements of the sequence satisfying a condition (via a selection mask or a predicate function)
- `unique` returns unique elements within a sequence
- `histogram` generates a summary of the statistical distribution of the sequence

2.1.2 Aggregation

- `reduce` traverses the sequence while accumulating some data, equivalent to the functional operation `fold_left`
- `scan` is the cumulative version of `reduce` which returns the sequence of the intermediate values taken by the accumulator

2.1.3 Differentiation

- `adjacent_difference` computes the difference between the current element and the previous or next one in the sequence
- `discontinuity` detects value change between the current element and the previous or next one in the sequence

2.1.4 Rearrangement

- `sort` rearranges the sequence by sorting it. It could be according to a comparison operator or a value using a radix approach
- `partial_sort` rearranges the sequence by sorting it up to and including a given index, according to a comparison operator.
- `nth_element` places the nth element in its sorted position, with elements less-than before, and greater after, according to a comparison operator.
- `exchange` rearranges the elements according to a different stride configuration which is equivalent to a tensor axis transposition
- `shuffle` rotates the elements

2.1.5 Partition/Merge

- `partition` divides the sequence into two or more sequences according to a predicate while preserving some ordering properties
- `merge` merges two ordered sequences into one while preserving the order

2.1.6 Data Movement

- `store` stores the sequence to a continuous memory zone. There are variations to use an optimized path or to specify how to store the sequence to better fit the access patterns of the CUs.
- `load` the complementary operations of the above ones.
- `memcpy` copies bytes between device sources and destinations

2.1.7 Sequence Search

- `find_first_of` searches for the first occurrence of any of the provided elements.
- `adjacent_find` searches a given sequence for the first occurence of two consecutive equal elements.
- `search` searches for the first occurrence of the sequence.
- `search_n` searches for the first occurrence of a sequence of count elements all equal to value.
- `find_end` searches for the last occurrence of the sequence.

2.1.8 Other operations

- `run_length_encode` generates a compact representation of a sequence
- `binary_search` finds for each element the index of an element with the same value in another sequence (which has to be sorted)
- `config` selects a kernel's grid/block dimensions to tune the operation to a GPU

2.2 Data type support

The following table shows the supported input and output datatypes.

Table 2.1: Supported Input/Output Types

Type	Support
int8	
int16	
int32	
int64	
float8	
float16	
bfloat16	
tensorfloat32	
float32	
float64	

2.3 Device-Wide Operations

- *Configuring the Kernels*
- *Transform*
- *Unique*
- *Sort*
- *Merge*
- *Partition*
- *Run Length Encode*
- *Scan*
- *Search N*
- *Select*
- *Reduce*
- *Adjacent Difference*
- *Adjacent Find*
- *Binary Search*
- *Histogram*
- *DeviceCopy*
- *Memcpy*
- *Nth Element*
- *Partial Sort*
- *Find first of*
- *Find end*
- *Search*

2.3.1 Configuring the Kernels

A kernel config is a way to select the grid/block dimensions, but also how the data will be fetched and stored (the algorithms used for `load` and `store`) for the operations using them (such as `select`).

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **kernel_config** : public rocprim::detail::kernel_config_params

Configuration of particular kernels launched by device-level operation.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.

Subclassed by rocprim::detail::default_radix_sort_block_sort_config< arch, key_type, value_type, enable >

Setting the configuration is important to better tune the kernel to a given GPU model. `rocPRIM` uses a placeholder type to let the macros select the default configuration for the GPU model

struct **default_config**

Special type used to show that the given device-level operation will be executed with optimal configuration dependent on types of the function's parameters and the target device architecture specified by ROCPRIM_TARGET_ARCH. Algorithms supporting dynamic dispatch will ignore ROCPRIM_TARGET_ARCH and launch using optimal configuration based on the target architecture derived from the stream.

Warning

To provide information about the GPU you're targeting, you have to set `ROCPRIM_TARGET_ARCH`.

If the target is not supported by `rocPRIM`, the templates will use the configuration for the model `900`.

If `ROCPRIM_TARGET_TARGET` is not defined, it defaults to `0`, which is not supported by `rocPRIM` and thus the configurations will be for the model `900`.

2.3.2 Transform

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **transform_config** : public rocprim::detail::transform_config_params

Configuration for the device-level transform operation.

Template Parameters

- **BlockSize** – Number of threads in a block.
- **ItemsPerThread** – Number of items processed by each thread.
- **SizeLimit** – Limit on the number of items for a single kernel launch.

Subclassed by rocprim::detail::default_transform_config< arch, data_type, enable >

transform

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **UnaryFunction**>
inline hipError_t **rocprim::transform**(*InputIterator* input, *OutputIterator* output, const size_t size, *UnaryFunction* transform_op, const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel transform primitive for device level.

transform function performs a device-wide transformation operation using unary `transform_op` operator.

Overview

- Ranges specified by `input` and `output` must have at least `size` elements.

Example

In this example a device-level transform operation is performed on an array of integer values (`shorts` are transformed into `ints`).

```
#include <rocprim/rocprim.hpp>

// custom transform function
auto transform_op =
    [] __device__ (int a) -> int
    {
        return a + 5;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
short * input;        // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output;         // empty array of 8 elements

// perform transform
rocprim::transform(
    input, output, input_size, transform_op
);
// output: [6, 7, 8, 9, 10, 11, 12, 13]
```

Template Parameters

- **Config** – - [optional] configuration of the primitive. It has to be *transform_config* or a class derived from it.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **UnaryFunction** – - type of unary function used for transform.

Parameters

- **input** – **[in]** - iterator to the first element in the range to transform.
- **output** – **[out]** - iterator to the first element in the output range.
- **size** – **[in]** - number of element in the input range.

- **transform_op** – **[in]** - unary operation function object that will be used for transform. The signature of the function should be equivalent to the following: `U f(const T &a);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

template<class **Config** = *default_config*, class **InputIterator1**, class **InputIterator2**, class **OutputIterator**, class **BinaryFunction**>
inline hipError_t `rocprim::`**transform**(*InputIterator1* input1, *InputIterator2* input2, *OutputIterator* output, const size_t size, *BinaryFunction* transform_op, const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel device-level transform primitive for two inputs.

transform function performs a device-wide transformation operation on two input ranges using binary `transform_op` operator.

Overview

- Ranges specified by `input1`, `input2`, and `output` must have at least `size` elements.

Example

In this example a device-level transform operation is performed on two arrays of integer values (element-wise sum is performed).

```
#include <rocprim/rocprim.hpp>

// custom transform function
auto transform_op =
    [] __device__ (int a, int b) -> int
    {
        return a + b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;   // e.g., 8
int* input1;   // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int* input2;   // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int* output;   // empty array of 8 elements

// perform transform
rocprim::transform(
    input1, input2, output, input1.size(), transform_op
);
// output: [2, 4, 6, 8, 10, 12, 14, 16]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *transform_config*.

- **InputIterator1** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **InputIterator2** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for transform.

Parameters

- **input1** – **[in]** - iterator to the first element in the 1st range to transform.
- **input2** – **[in]** - iterator to the first element in the 2nd range to transform.
- **output** – **[out]** - iterator to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **transform_op** – **[in]** - binary operation function object that will be used for transform. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced. Default value is `false`.

2.3.3 Unique

unique

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **UniqueCountOutputIterator**, class **EqualityOp** = ::rocprim::equal_to<typename std::iterator_traits<*InputIterator*>::value_type>>
inline hipError_t **rocprim::unique**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, *UniqueCountOutputIterator* unique_count_output, const size_t size, *EqualityOp* equality_op = *EqualityOp*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Device-level parallel unique primitive.

From given `input` range unique primitive eliminates all but the first element from every consecutive group of equivalent elements and copies them into `output`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Range specified by `input` must have at least `size` elements.
- Range specified by `output` must have at least so many elements, that all selected values can be copied into it.
- Range specified by `unique_count_output` must have at least 1 element.
- By default `InputIterator::value_type`'s equality operator is used to check if elements are equivalent.

Example

In this example a device-level unique operation is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;     // e.g., 8
int * input;           // e.g., [1, 4, 2, 4, 4, 7, 7, 7]
int * output;          // empty array of 8 elements
size_t * output_count; // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::unique(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, output_count,
    input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform unique operation
rocprim::unique(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, output_count,
    input_size
);
// output: [1, 4, 2, 4, 7]
// output_count: 5
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **UniqueCountOutputIterator** – - random-access iterator type of the unique_count_output value used to return number of unique values. It can be a simple pointer type.
- **EqualityOp** – - type of an binary operator used to compare values for equality.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the unique operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.

- **output** – **[out]** - iterator to the first element in the output range.
- **unique_count_output** – **[out]** - iterator to the total number of selected values (length of output).
- **size** – **[in]** - number of element in the input range.
- **equality_op** – **[in]** - [optional] binary function object used to compare input values for equality. The signature of the function should be equivalent to the following: `bool equal_to(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

unique_by_key

template<typename **Config** = *default_config*, typename **KeyIterator**, typename **ValueIterator**, typename **OutputKeyIterator**, typename **OutputValueIterator**, typename **UniqueCountOutputIterator**, typename **EqualityOp** = ::rocprim::equal_to<typename std::iterator_traits<*KeyIterator*>::value_type>>
inline hipError_t **rocprim::unique_by_key**(void *temporary_storage, size_t &storage_size, const *KeyIterator* keys_input, const *ValueIterator* values_input, const *OutputKeyIterator* keys_output, const *OutputValueIterator* values_output, const *UniqueCountOutputIterator* unique_count_output, const size_t size, const *EqualityOp* equality_op = *EqualityOp*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Device-level parallel unique by key primitive.

From given `input` range unique primitive eliminates all but the first element from every consecutive group of equivalent elements and copies them and their corresponding keys into `output`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Ranges specified by `keys_input` and value_input must have at least `size` elements each.
- Ranges specified by `keys_output` and values_output each must have at least so many elements, that all selected values can be copied into them.
- Range specified by `unique_count_output` must have at least 1 element.
- By default `InputIterator::value_type`'s equality operator is used to check if elements are equivalent.

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **KeyIterator** – - random-access iterator type of the input key range. It can be a simple pointer type.
- **ValueIterator** – - random-access iterator type of the input value range. It can be a simple pointer type.

- **OutputKeyIterator** – - random-access iterator type of the output key range. It can be a simple pointer type.
- **OutputValueIterator** – - random-access iterator type of the output value range. It can be a simple pointer type.
- **UniqueCountOutputIterator** – - random-access iterator type of the unique_count_output value used to return number of unique keys and values. It can be a simple pointer type.
- **EqualityOp** – - type of an binary operator used to compare keys for equality.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the unique operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - iterator to the first element in the range to select keys from.
- **values_input** – **[in]** - iterator to the first element in the range of values corresponding to keys
- **keys_output** – **[out]** - iterator to the first element in the output key range.
- **values_output** – **[out]** - iterator to the first element in the output value range.
- **unique_count_output** – **[out]** - iterator to the total number of selected values (length of `output`).
- **size** – **[in]** - number of element in the input range.
- **equality_op** – **[in]** - [optional] binary function object used to compare input values for equality. The signature of the function should be equivalent to the following: `bool equal_to(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

2.3.4 Sort

Configuring the kernel

merge_sort

template<unsigned int **MergeOddevenBlockSize** = 512, unsigned int **SortBlockSize** = *MergeOddevenBlockSize*, unsigned int **SortItemsPerThread** = 1, unsigned int **MergeMergepathPartitionBlockSize** = 128, unsigned int **MergeMergepathBlockSize** = 128, unsigned int **MergeMergepathItemsPerThread** = 4, unsigned int **MinInputSizeMergepath** = (1 << 17) + 70000>
struct **merge_sort_config** : public rocprim::detail::merge_sort_config_params

Configuration of device-level merge primitives.

Template Parameters

- **SortBlockSize** – - block size in the block-sort step
- **SortItemsPerThread** – - ItemsPerThread in the block-sort step
- **MergeOddevenBlockSize** – - block size in the block merge step using oddeven impl (used when input_size < MinInputSizeMergepath)

- **MergeMergepathPartitionBlockSize** – - block size of the partition kernel in the block merge step using mergepath impl
- **MergeMergepathBlockSize** – - block size in the block merge step using mergepath impl
- **MergeMergepathItemsPerThread** – - ItemsPerThread in the block merge step using mergepath impl
- **MinInputSizeMergepath** – - breakpoint of input-size to use mergepath impl for block merge step

radix_sort

template<class **SingleSortConfig** = *default_config*, class **MergeSortConfig** = *default_config*, class **OnesweepConfig** = *default_config*, size_t **MergeSortLimit** = 1024 * 1024>
struct **radix_sort_config**

Configuration of device-level radix sort operation.

One of three algorithms is used: single sort (launches only a single block), merge sort, or Onesweep.

Template Parameters

- **SortSingleConfig** – - Configuration for the single kernel subalgorithm. must be *kernel_config* or *default_config*.
- **MergeSortConfig** – - Configuration for the merge sort subalgorithm. must be *merge_sort_config* or *default_config*. If *merge_sort_config*, the sorted items per block must be a power of two.
- **OnesweepConfig** – - Configuration for the Onesweep subalgorithm. must be radix_sort_onesweep_config or *default_config*.
- **MergeSortLimit** – - The largest number of items for which the merge sort algorithm will be used. Note that below this limit, a different algorithm may be used.

merge_sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*KeysInputIterator*>::value_type>>
inline hipError_t **rocprim::merge_sort**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, const size_t size, *BinaryFunction* compare_function = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel merge sort primitive for device level.

merge_sort function performs a device-wide merge sort of keys. Function sorts input keys based on comparison function.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of temporary_storage in storage_size if temporary_storage in a null pointer.
- Accepts custom compare_functions for sorting across the device.

Stability

merge_sort is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys a and b and a binary boolean operation op such that:

- a precedes b in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that a will precede b as well in the output (ordered) keys.

Example

In this example a device-level ascending merge sort is performed on an array of `float` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
float * input;          // e.g., [0.6, 0.3, 0.65, 0.4, 0.2, 0.08, 1, 0.7]
float * output;         // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::merge_sort(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::merge_sort(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);
// keys_output: [0.08, 0.2, 0.3, 0.4, 0.6, 0.65, 0.7, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *merge_sort_config*.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.

- **compare_function** – **[in]** - binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*KeysInputIterator*>::value_type>>
inline hipError_t `rocprim::merge_sort`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, const size_t size, *BinaryFunction* compare_function = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending merge sort-by-key primitive for device level.

`merge_sort` function performs a device-wide merge sort of (key, value) pairs. Function sorts input pairs based on comparison function.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Accepts custom compare_functions for sorting across the device.

Stability

`merge_sort` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending merge sort is performed where input keys are represented by an array of unsigned integers and input values by an array of `double`s.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
unsigned int * keys_input;  // e.g., [ 6, 3,  5, 4,  1,  8,  2, 7]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
unsigned int * keys_output; // empty array of 8 elements
double * values_output;     // empty array of 8 elements
```

(continues on next page)

(continued from previous page)

```
size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::merge_sort(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::merge_sort(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size
);
// keys_output:   [ 1,  2, 3, 4,  5,  6, 7,  8]
// values_output: [-1, -2, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *merge_sort_config*.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **values_input** – **[in]** - pointer to the first element in the range to sort.
- **values_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **compare_function** – **[in]** - binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const`

`T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.

- **`stream`** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **`debug_synchronous`** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

radix_sort_keys

Ascending Sort

template<class **`Config`** = *default_config*, class **`KeysInputIterator`**, class **`KeysOutputIterator`**, class **`Size`**, class **`Key`** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
hipError_t `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of `float` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
```

(continues on next page)

(continued from previous page)

```
float * input;          // e.g., [0.6, 0.3, 0.65, 0.4, 0.2, 0.08, 1, 0.7]
float * output;         // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);
// keys_output: [0.08, 0.2, 0.3, 0.4, 0.6, 0.65, 0.7, 1]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).

- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *`rocprim::tuple`*`<int16_t&, uint8_t&>`, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>
```

(continues on next page)

(continued from previous page)

```
struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
custom_type * input;    // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0,␣
↪0.2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * output;   // empty array of 8 elements

constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 96;
size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys_output: [{-3, 0.3}, {-1, 0.7}, {0, 0.2}, {0, 0.4}, {2, 0.6}, {2, 0.65},
↪{11, 0.08}, {11, 1.0}]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – Random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – Random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – Integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.

- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
custom_type * input;    // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0,␣
↪0.2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * output;   // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}
);
// keys_output: [{-3, 0.3}, {-1, 0.7}, {0, 0.2}, {0, 0.4}, {2, 0.6}, {2, 0.65},
↪{11, 0.08}, {11, 1.0}]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – Random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – Random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – Integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**>
hipError_t `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type must be an arithmetic type (that is, an integral type or a floating-point type).

- Buffers of `keys` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of `float` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;  // e.g., 8
float * input;      // e.g., [0.6, 0.3, 0.65, 0.4, 0.2, 0.08, 1, 0.7]
float * tmp;        // empty array of 8 elements
// Create double-buffer
rocprim::double_buffer<float> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size
);
// keys.current(): [0.08, 0.2, 0.3, 0.4, 0.6, 0.65, 0.7, 1]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and

function returns without performing the sort operation.

- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *rocprim::tuple*<int16_t&, uint8_t&>, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 96;
size_t input_size;   // e.g., 8
custom_type * input; // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0, 0.
↪2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * tmp;   // empty array of 8 elements
// Create double-buffer
rocprim::double_buffer<custom_type> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);
```

(continues on next page)

(continued from previous page)

```
// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys.current(): [{-3, 0.3}, {-1, 0.7}, {0, 0.2}, {0, 0.4}, {2, 0.6}, {2, 0.
↪65}, {11, 0.08}, {11, 1.0}]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_keys`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort primitive for device level.

`radix_sort_keys` function performs a device-wide radix sort of keys. Function sorts input keys in ascending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *rocprim::tuple* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;   // e.g., 8
custom_type * input; // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0, 0.
↪2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * tmp;   // empty array of 8 elements
// Create double-buffer
```

(continues on next page)

(continued from previous page)

```
rocprim::double_buffer<custom_type> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}
);
// keys.current(): [{-3, 0.3}, {-1, 0.7}, {0, 0.2}, {0, 0.4}, {2, 0.6}, {2, 0.
↪65}, {11, 0.08}, {11, 1.0}]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

Descending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
hipError_t `rocprim::radix_sort_keys_desc`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
int * input;          // e.g., [6, 3, 5, 4, 2, 8, 1, 7]
int * output;         // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);

// allocate temporary storage
```

(continues on next page)

(continued from previous page)

```
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size
);
// keys_output: [8, 7, 6, 5, 4, 3, 2, 1]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::`**radix_sort_keys_desc**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *rocprim::tuple* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *rocprim::tuple*<int16_t&, uint8_t&>, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};
```

(continues on next page)

(continued from previous page)

```
// Prepare input and output (declare pointers, allocate device memory etc.)
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 96;
size_t input_size;      // e.g., 8
custom_type * input;    // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0,␣
↪0.2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * output;   // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys_output: [{11, 1.0}, {11, 0.08}, {2, 0.65}, {2, 0.6}, {0, 0.4}, {0, 0.2},
↪ {-1, 0.7}, {-3, 0.3},]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.

- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_keys_desc`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
custom_type * input;    // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0,␣
↪0.2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * output;   // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, custom_type_decomposer{}
);
// keys_output: [{11, 1.0}, {11, 0.08}, {2, 0.65}, {2, 0.6}, {0, 0.4}, {0, 0.2},
↪ {-1, 0.7}, {-3, 0.3},]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.

- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**>
hipError_t `rocprim::`**radix_sort_keys_desc**(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type must be an arithmetic type (that is, an integral type or a floating-point type).
- Buffers of `keys` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- a precedes b in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that a will precede b as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;  // e.g., 8
int * input;        // e.g., [6, 3, 5, 4, 2, 8, 1, 7]
int * tmp;          // empty array of 8 elements
// Create double-buffer
rocprim::double_buffer<int> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size
);
// keys.current(): [8, 7, 6, 5, 4, 3, 2, 1]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.

- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_keys_desc`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *`rocprim::tuple`*`<int16_t&, uint8_t&>`, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- a precedes b in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that a will precede b as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 96;
size_t input_size;   // e.g., 8
custom_type * input; // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0, 0.
↪2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * tmp;   // empty array of 8 elements
// Create double-buffer
rocprim::double_buffer<custom_type> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}
);
// keys.current(): [{11, 1.0}, {11, 0.08}, {2, 0.65}, {2, 0.6}, {0, 0.4}, {0, 0.
↪2}, {-1, 0.7}, {-3, 0.3},]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Defaults to `0`.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Defaults to the size of the decomposed tuple's bit range.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Size**, class **Decomposer**>
auto **rocprim::radix_sort_keys_desc**(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) ->
std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort primitive for device level.

`radix_sort_keys_desc` function performs a device-wide radix sort of keys. Function sorts input keys in descending order.

Overview

- The contents of both buffers of `keys` may be altered by the sorting function.
- `current()` of `keys` is used as the input.
- The function will update `current()` of `keys` to point to the buffer that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.

- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *rocprim::tuple* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of values of a custom type, using a custom decomposer.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;   // e.g., 8
custom_type * input; // e.g., [{2, 0.6}, {-3, 0.3}, {2, 0.65}, {0, 0.4}, {0, 0.
↪2}, {11, 0.08}, {11, 1}, {-1, 0.7}]
custom_type * tmp;   // empty array of 8 elements
// Create double-buffer
rocprim::double_buffer<custom_type> keys(input, tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}
```

(continues on next page)

(continued from previous page)

```
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, input_size, custom_type_decomposer{}
);
// keys.current(): [{11, 1.0}, {11, 0.08}, {2, 0.65}, {2, 0.6}, {0, 0.4}, {0, 0.
↪2}, {-1, 0.7}, {-3, 0.3},]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

Segmented Ascending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **OffsetIterator**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
inline hipError_t `rocprim::segmented_radix_sort_keys`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, unsigned int size, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort primitive for device level.

`segmented_radix_sort_keys` function performs a device-wide radix sort across multiple, non-overlapping sequences of keys. Function sorts input keys in ascending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`segmented_radix_sort_keys` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed on an array of `float` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
float * input;          // e.g., [0.6, 0.3, 0.65, 0.4, 0.2, 0.08, 1, 0.7]
float * output;         // empty array of 8 elements
unsigned int segments;  // e.g., 3
int * offsets;          // e.g. [0, 2, 3, 8]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size,
    segments, offsets, offsets + 1
);

// allocate temporary storage
```

(continues on next page)

(continued from previous page)

```
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::segmented_radix_sort_keys(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size,
    segments, offsets, offsets + 1
);
// keys_output: [0.3, 0.6, 0.65, 0.08, 0.2, 0.4, 0.7, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or segmented_radix_sort_config.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to storage_size and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of temporary_storage.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range [0; 8 * sizeof(Key)). Default value: 0. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range (begin_bit; 8 * sizeof(Key)]. Default value: * sizeof(Key). Non-default value not supported for floating-point key-types.
- **stream** – **[in]** - [optional] HIP stream object. Default is 0 (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is false.

Returns

hipSuccess (0) after successful sort; otherwise a HIP runtime error of type hipError_t.

Segmented Descending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **OffsetIterator**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
inline hipError_t `rocprim::`**segmented_radix_sort_keys_desc**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, unsigned int size, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort primitive for device level.

`segmented_radix_sort_keys_desc` function performs a device-wide radix sort across multiple, non-overlapping sequences of keys. Function sorts input keys in descending order.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`segmented_radix_sort_keys_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;      // e.g., 8
int * input;            // e.g., [6, 3, 5, 4, 2, 8, 1, 7]
int * output;           // empty array of 8 elements
unsigned int segments;  // e.g., 3
int * offsets;          // e.g. [0, 2, 3, 8]
```

(continues on next page)

(continued from previous page)

```
size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size,
    segments, offsets, offsets + 1
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::segmented_radix_sort_keys_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size,
    segments, offsets, offsets + 1
);
// keys_output: [6, 3, 5, 8, 7, 4, 2, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or `segmented_radix_sort_config`.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.

- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

radix_sort_pairs

Ascending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
hipError_t `rocprim::radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input`, `keys_output`, `values_input` and `values_output` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of unsigned integers and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
unsigned int * keys_input;  // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
unsigned int * keys_output; // empty array of 8 elements
double * values_output;     // empty array of 8 elements

// Keys are in range [0; 8], so we can limit compared bit to bits on indexes
// 0, 1, 2, 3, and 4. In order to do this begin_bit is set to 0 and end_bit
// is set to 5.

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, 0, 5
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, 0, 5
);
// keys_output:   [ 1,  1, 3, 4,  5,  6, 7,  8]
// values_output: [-1, -2, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and

function returns without performing the sort operation.

- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *rocprim::tuple*<int16_t&, uint8_t&>, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_output;  // empty array of 8 elements
double * values_output;     // empty array of 8 elements

// The integer field of the keys is in range 0-11, which can be represented on␣
↪4 bits,
// while for the double member we must specify full bit range [0; 63].␣
↪Therefore begin_bit
// is set to 0 and end_bit is set to 68.
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 68;

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
```

(continues on next page)

(continued from previous page)

```
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys_output:   [{0, 0.2}, {0, 0.3}, {0, 0.4}, {2, 0.6}, {2, 0.65}, {5, 0.7},
↪{11, 0.08}, {11, 1.0}]
// values_output: [-1, 2, 3, -5, -4, 7, -8, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.

- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>
```

(continues on next page)

(continued from previous page)

```
struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_output;  // empty array of 8 elements
double * values_output;     // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}
);
// keys_output:   [{0, 0.2}, {0, 0.3}, {0, 0.4}, {2, 0.6}, {2, 0.65}, {5, 0.7},
↪{11, 0.08}, {11, 1.0}]
// values_output: [-1, 2, 3, -5, -4, 7, -8, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**>
hipError_t **rocprim::radix_sort_pairs**(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.

- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type must be an arithmetic type (that is, an integral type or a floating-point type).
- Buffers of `keys` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of unsigned integers and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
unsigned int * keys_input;  // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
unsigned int * keys_tmp;    // empty array of 8 elements
double*  values_tmp;        // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<unsigned int> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

// Keys are in range [0; 8], so we can limit compared bit to bits on indexes
// 0, 1, 2, 3, and 4. In order to do this begin_bit is set to 0 and end_bit
// is set to 5.

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size,
    0, 5
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size,
```

(continues on next page)

(continued from previous page)

```
    0, 5
);
// keys.current():   [ 1,  1, 3, 4,  5,  6, 7,  8]
// values.current(): [-1, -2, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Value** – value type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *rocprim::tuple* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *rocprim::tuple*<int16_t&, uint8_t&>, `begin_bit`==6 and `end_bit`==12, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
```

(continues on next page)

(continued from previous page)

```
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_tmp;     // empty array of 8 elements
double*  values_tmp;        // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<custom_type> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

// The integer field of the keys is in range 0-11, which can be represented on␣
↪4 bits,
// while for the double member we must specify full bit range [0; 63].␣
↪Therefore begin_bit
// is set to 0 and end_bit is set to 68.
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 68;

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys.current():   [{0, 0.2}, {0, 0.3}, {0, 0.4}, {2, 0.6}, {2, 0.65}, {5, 0.
↪7}, {11, 0.08}, {11, 1.0}]
// values.current(): [-1, 2, 3, -5, -4, 7, -8, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Value** – value type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison. Defaults to the size of the decomposed tuple's bit range.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel ascending radix sort-by-key primitive for device level.

`radix_sort_pairs` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These

references must point to member variables of `Key`, however not every member variable has to be exposed this way.

- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `double`s.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_tmp;     // empty array of 8 elements
double*  values_tmp;        // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<custom_type> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);
```

(continues on next page)

(continued from previous page)

```
// perform sort
rocprim::radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}
);
// keys.current():   [{0, 0.2}, {0, 0.3}, {0, 0.4}, {2, 0.6}, {2, 0.65}, {5, 0.
↪7}, {11, 0.08}, {11, 1.0}]
// values.current(): [-1, 2, 3, -5, -4, 7, -8, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Value** – value type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

Descending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>

hipError_t rocprim::**radix_sort_pairs_desc**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input`, `keys_output`, `values_input` and `values_output` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of integers and input values by an array of `double`s.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;       // e.g., 8
int * keys_input;        // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;   // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
int * keys_output;       // empty array of 8 elements
double * values_output;  // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size
```

(continues on next page)

(continued from previous page)

```
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size
);
// keys_output:   [ 8, 7,  6,  5, 4, 3,  1,  1]
// values_output: [-8, 7, -5, -4, 3, 2, -1, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).

- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>
auto `rocprim::radix_sort_pairs_desc`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *`rocprim::tuple`*<`int16_t&, uint8_t&`>, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_output;  // empty array of 8 elements
double * values_output;     // empty array of 8 elements

// The integer field of the keys is in range 0-11, which can be represented on␣
↪4 bits,
// while for the double member we must specify full bit range [0; 63].␣
↪Therefore begin_bit
// is set to 0 and end_bit is set to 68.
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 68;

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys_output:   [{11, 1.0}, {11, 0.08}, {5, 0.7}, {2, 0.65}, {2, 0.6}, {0, 0.
↪4}, {0, 0.3}, {0, 0.2}]
// values_output: [-2, -1, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** index of the first (least significant) bit used in key comparison.
- **end_bit** – **[in]** past-the-end index (most significant) bit used in key comparison.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **Size**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type, class **Decomposer**>

auto rocprim::**radix_sort_pairs_desc**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};
```

(continues on next page)

(continued from previous page)

```
// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
custom_type * keys_input;   // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4},
↪{0, 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_output;  // empty array of 8 elements
double * values_output;     // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size, custom_type_decomposer{}
);
// keys_output:   [{11, 1.0}, {11, 0.08}, {5, 0.7}, {2, 0.65}, {2, 0.6}, {0, 0.
↪4}, {0, 0.3}, {0, 0.2}]
// values_output: [-2, -1, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **KeysInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **Size** – integral type that represents the problem size.
- **Key** – The value type of the input and output iterators.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** pointer to the first element in the range to sort.
- **keys_output** – **[out]** pointer to the first element in the output range.
- **values_input** – **[in]** pointer to the first element in the range to sort.
- **values_output** – **[out]** pointer to the first element in the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**>
hipError_t `rocprim::`**radix_sort_pairs_desc**(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type must be an arithmetic type (that is, an integral type or a floating-point type).
- Buffers of `keys` must have at least `size` elements.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of integers and input values by an array of `double`s.

```
#include <rocprim/rocprim.hpp>

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;       // e.g., 8
int * keys_input;        // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;   // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
int * keys_tmp;          // empty array of 8 elements
double * values_tmp;     // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<int> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size
);
// keys.current():   [ 8, 7,  6,  5, 4, 3,  1,  1]
// values.current(): [-8, 7, -5, -4, 3, 2, -1, -2]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Value** – value type.
- **Size** – integral type that represents the problem size.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_pairs_desc`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, *Decomposer* decomposer, unsigned int begin_bit, unsigned int end_bit, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.

- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *rocprim::tuple* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.
- `begin_bit` and `end_bit` can be provided to control the radix range that is considered in the decomposed tuple. For example, if the decomposer returns *rocprim::tuple*<`int16_t&, uint8_t&`>, `begin_bit==6` and `end_bit==12`, then the 2 MSBs of the `uint8_t` value and the 4 LSBs of the `int16_t` value are considered for sorting. The range specified by `begin_bit` and `end_bit` must be valid with regards to the sizes of the return tuple's elements.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;        // e.g., 8
custom_type * keys_input; // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4}, {0,
↪ 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;    // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_tmp;   // empty array of 8 elements
double * values_tmp;      // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<custom_type> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

// The integer field of the keys is in range 0-11, which can be represented on␣
↪4 bits,
// while for the double member we must specify full bit range [0; 63].␣
```

(continues on next page)

(continued from previous page)

```
↪Therefore begin_bit
// is set to 0 and end_bit is set to 68.
constexpr unsigned int begin_bit = 0;
constexpr unsigned int end_bit = 68;

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}, begin_bit, end_bit
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}, begin_bit, end_bit
);
// keys.current():   [{11, 1.0}, {11, 0.08}, {5, 0.7}, {2, 0.65}, {2, 0.6}, {0,␣
↪0.4}, {0, 0.3}, {0, 0.2}]
// values.current(): [-2, -1, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.
- **Value** – value type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **begin_bit** – **[in]** [optional] index of the first (least significant) bit used in key comparison. Defaults to `0`.

- **end_bit** – **[in]** [optional] past-the-end index (most significant) bit used in key comparison. Defaults to the size of the decomposed tuple's bit range.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **Key**, class **Value**, class **Size**, class **Decomposer**>
auto `rocprim::radix_sort_pairs_desc`(void *temporary_storage, size_t &storage_size, *double_buffer*<*Key*> &keys, *double_buffer*<*Value*> &values, *Size* size, *Decomposer* decomposer, hipStream_t stream = 0, bool debug_synchronous = false) -> std::enable_if_t<!std::is_convertible<*Decomposer*, unsigned int>::value, hipError_t>

Parallel descending radix sort-by-key primitive for device level.

`radix_sort_pairs_desc` function performs a device-wide radix sort of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of both buffers of `keys` and `values` may be altered by the sorting function.
- `current()` of `keys` and `values` are used as the input.
- The function will update `current()` of `keys` and `values` to point to buffers that contains the output range.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- The function requires small `temporary_storage` as it does not need a temporary buffer of `size` elements.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) can be any trivially copyable type.
- `decomposer` must be a functor that implements `operator()(Key&) const`. This operator must return a *`rocprim::tuple`* that contains one or more reference to value(s) of arithmetic types. These references must point to member variables of `Key`, however not every member variable has to be exposed this way.
- Ranges specified by `keys_input` and `keys_output` must have at least `size` elements.

Stability

`radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of a custom type and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

struct custom_type
{
    int i;
    double d;
};

struct custom_type_decomposer
{
    rocprim::tuple<int&, double&> operator()(custom_type& key) const
    {
        return rocprim::tuple<int&, double&>(key.i, key.d);
    }
};

// Prepare input and tmp (declare pointers, allocate device memory etc.)
size_t input_size;        // e.g., 8
custom_type * keys_input; // e.g., [{2, 0.6}, {0, 0.3}, {2, 0.65}, {0, 0.4}, {0,
↪ 0.2}, {11, 0.08}, {11, 1.0}, {5, 0.7}]
double * values_input;    // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
custom_type * keys_tmp;   // empty array of 8 elements
double * values_tmp;      // empty array of 8 elements
// Create double-buffers
rocprim::double_buffer<custom_type> keys(keys_input, keys_tmp);
rocprim::double_buffer<double> values(values_input, values_tmp);

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, values, input_size, custom_type_decomposer{}
);
// keys.current():   [{11, 1.0}, {11, 0.08}, {5, 0.7}, {2, 0.65}, {2, 0.6}, {0,␣
↪0.4}, {0, 0.3}, {0, 0.2}]
// values.current(): [-2, -1, 2, 3, -4, -5, 7, -8]
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *radix_sort_config*.
- **Key** – key type. Must be an integral type or a floating-point type.

- **Value** – value type.
- **Size** – integral type that represents the problem size.
- **Decomposer** – The type of the decomposer functor.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** reference to the double-buffer of keys, its `current()` contains the input range and will be updated to point to the output range.
- **values** – **[inout]** reference to the double-buffer of values, its `current()` contains the input range and will be updated to point to the output range.
- **size** – **[in]** number of element in the input range.
- **decomposer** – **[in]** decomposer functor that produces a tuple of references from the input key type.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

Segmented Ascending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **OffsetIterator**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
inline hipError_t `rocprim::segmented_radix_sort_pairs`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, unsigned int size, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel ascending radix sort-by-key primitive for device level.

`segmented_radix_sort_pairs` function performs a device-wide radix sort across multiple, non-overlapping sequences of (key, value) pairs. Function sorts input pairs in ascending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).

- Ranges specified by `keys_input`, `keys_output`, `values_input` and `values_output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`segmented_radix_sort_pairs` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example

In this example a device-level ascending radix sort is performed where input keys are represented by an array of unsigned integers and input values by an array of `doubles`.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
unsigned int * keys_input;  // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;      // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
unsigned int * keys_output; // empty array of 8 elements
double * values_output;     // empty array of 8 elements
unsigned int segments;      // e.g., 3
int * offsets;              // e.g. [0, 2, 3, 8]

// Keys are in range [0; 8], so we can limit compared bit to bits on indexes
// 0, 1, 2, 3, and 4. In order to do this begin_bit is set to 0 and end_bit
// is set to 5.

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output, input_size,
    segments, offsets, offsets + 1,
    0, 5
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::segmented_radix_sort_pairs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output, input_size,
```

(continues on next page)

(continued from previous page)

```
    segments, offsets, offsets + 1,
    0, 5
);
// keys_output:   [3,  6,  5,  1,  1, 4, 7,  8]
// values_output: [2, -5, -4, -1, -2, 3, 7, -8]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or `segmented_radix_sort_config`.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **values_input** – **[in]** - pointer to the first element in the range to sort.
- **values_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

Segmented Descending Sort

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **ValuesInputIterator**, class **ValuesOutputIterator**, class **OffsetIterator**, class **Key** = typename std::iterator_traits<*KeysInputIterator*>::value_type>
inline hipError_t `rocprim::`**segmented_radix_sort_pairs_desc**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, *ValuesInputIterator* values_input, *ValuesOutputIterator* values_output, unsigned int size, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel descending radix sort-by-key primitive for device level.

`segmented_radix_sort_pairs_desc` function performs a device-wide radix sort across multiple, non-overlapping sequences of (key, value) pairs. Function sorts input pairs in descending order of keys.

Overview

- The contents of the inputs are not altered by the sorting function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- `Key` type (a `value_type` of `KeysInputIterator` and `KeysOutputIterator`) must be an arithmetic type (that is, an integral type or a floating-point type).
- Ranges specified by `keys_input`, `keys_output`, `values_input` and `values_output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.
- If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

`segmented_radix_sort_pairs_desc` is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that `a` will precede `b` as well in the output (ordered) keys. (ordered) keys.

Example

In this example a device-level descending radix sort is performed where input keys are represented by an array of integers and input values by an array of `double`s.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;       // e.g., 8
int * keys_input;        // e.g., [ 6, 3,  5, 4,  1,  8,  1, 7]
double * values_input;   // e.g., [-5, 2, -4, 3, -1, -8, -2, 7]
int * keys_output;       // empty array of 8 elements
double * values_output;  // empty array of 8 elements
unsigned int segments;   // e.g., 3
int * offsets;           // e.g. [0, 2, 3, 8]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size,
    segments, offsets, offsets + 1
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform sort
rocprim::segmented_radix_sort_pairs_desc(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, values_input, values_output,
    input_size,
    segments, offsets, offsets + 1
);
// keys_output:   [ 6, 3,  5,  8, 7, 4,  1,  1]
// values_output: [-5, 2, -4, -8, 7, 3, -1, -2]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or `segmented_radix_sort_config`.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - pointer to the first element in the range to sort.
- **keys_output** – **[out]** - pointer to the first element in the output range.
- **values_input** – **[in]** - pointer to the first element in the range to sort.
- **values_output** – **[out]** - pointer to the first element in the output range.
- **size** – **[in]** - number of element in the input range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`. Non-default value not supported for floating-point key-types.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`. Non-default value not supported for floating-point key-types.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

2.3.5 Partial Sort

Configuring the kernel

template<class **NthElementConfig**, class **MergeSortConfig** = *default_config*, class **RadixSortConfig** = *default_config*>
struct **partial_sort_config**

Configuration of device-level partial sort.

Template Parameters

- **NthElementConfig** – - configuration of device-level nth element operation. Must be *nth_element_config* or *default_config*.
- **MergeSortConfig** – - configuration of device-level merge sort operation. Must be *merge_sort_config* or *default_config*.
- **RadixSortConfig** – - configuration of device-level radix sort operation. Must be *radix_sort_config* or *default_config*.

partial_sort

Warning

doxygenfunction: Unable to resolve function "rocprim::partial_sort" with arguments (void*, size_t&, KeysIterator, size_t, size_t, BinaryFunction, hipStream_t, bool) in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml. Potential matches:

```
- template<class Config = default_config, class KeysIterator, class BinaryFunction =␣
↪::rocprim::less<typename std::iterator_traits<KeysIterator>::value_type>, class␣
↪Decomposer = ::rocprim::identity_decomposer> hipError_t partial_sort(void␣
↪*temporary_storage, size_t &storage_size, KeysIterator keys, size_t middle, size_t␣
↪size, BinaryFunction compare_function = BinaryFunction(), hipStream_t stream = 0,␣
↪bool debug_synchronous = false, Decomposer decomposer = {})
```

Warning

doxygenfunction: Unable to resolve function "rocprim::partial_sort_copy" with arguments (void*, size_t&, KeysInputIterator, KeysOutputIterator, size_t, size_t, BinaryFunction, hipStream_t, bool) in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml. Potential matches:

```
- template<class Config = default_config, class KeysInputIterator, class␣
↪KeysOutputIterator, class BinaryFunction = ::rocprim::less<typename std::iterator_
↪traits<KeysInputIterator>::value_type>, class Decomposer = ::rocprim::identity_
↪decomposer> hipError_t partial_sort_copy(void *temporary_storage, size_t &storage_
↪size, KeysInputIterator keys_input, KeysOutputIterator keys_output, size_t middle,␣
↪size_t size, BinaryFunction compare_function = BinaryFunction(), hipStream_t stream␣
↪= 0, bool debug_synchronous = false, Decomposer decomposer = Decomposer())
```

2.3.6 Nth Element

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, unsigned int **StopRecursionSize**, unsigned int **NumberOfBuckets**, block_radix_rank_algorithm **RadixRankAlgorithm**>
struct **nth_element_config** : public rocprim::detail::nth_element_config_params

Configuration of device-level nth_element.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread.
- **StopRecursionSize** – the size from where recursion is stopped to do a block sort
- **NumberOfBuckets** – the number of buckets that are used in the algorithm
- **RadixRankAlgorithm** – algorithm for radix rank

nth_element

template<class **Config** = *default_config*, class **KeysIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*KeysIterator*>::value_type>>
inline hipError_t **rocprim::nth_element**(void *temporary_storage, size_t &storage_size, *KeysIterator* keys, size_t nth, size_t size, *BinaryFunction* compare_function = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Rearrange elements smaller than the n-th before and bigger than n-th after the n-th element.

The element at index `n` is set to the element that would be at the n-th position if the input was sorted. Additionally the other elements are rearranged such that for all values of `i` in [keys_output, keys_output + n) and all values of `j` in [keys_output + n, keys_output + size): `comp(*i, *j)` is false. Smaller elements than the n-th will be arranged before, and bigger ones after the n-th element.

Overview

- The contents of the inputs are not altered by the function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Accepts custom compare_functions for nth_element across the device.
- Streams in graph capture mode are not supported

Stability

`nth_element` is **not stable**: it doesn't necessarily preserve the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **not guaranteed** that `a` will precede `b` as well in the output keys.

Example

In this example a device-level nth_element is performed where input keys are represented by an array of unsigned integers.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
size_t nth;                 // e.g., 4
unsigned int * keys;  // e.g., [ 6, 3, 5, 4, 1, 8, 2, 7 ]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::nth_element(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, nth, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);
```

(continues on next page)

(continued from previous page)

```
// perform nth_element
rocprim::nth_element(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys, nth, input_size
);
// possible keys:   [ 1, 3, 4, 2, 5, 8, 7, 6 ]
```

Template Parameters

- **Config** – [optional] configuration of the primitive. It has to be *nth_element_config*.
- **KeysIterator** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **CompareFunction** – [inferred] Type of binary function that accepts two arguments of the type `KeysIterator` and returns a value convertible to bool. Default type is `rocprim::less<>`.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the nth_element rearrangement.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys** – **[inout]** iterator to the input range.
- **nth** – **[in]** The index of the nth_element in the input range.
- **size** – **[in]** number of element in the input range.
- **compare_function** – **[in]** binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The comparator must meet the C++ named requirement Compare. The default value is `BinaryFunction()`.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful rearrangement; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator**, class **KeysOutputIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*KeysInputIterator*>::value_type>>
inline hipError_t `rocprim::nth_element`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *KeysOutputIterator* keys_output, size_t nth, size_t size, *BinaryFunction* compare_function = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Rearrange elements smaller than the n-th before and bigger than n-th after the n-th element.

The element at index `n` is set to the element that would be at the n-th position if the input was sorted. Additionally the other elements are rearranged such that for all values of `i` in [keys_output, keys_output + n) and all values of `j` in [keys_output + n, keys_output + size): `comp(*i, *j)` is false. Smaller elements than the n-th will be arranged before, and bigger ones after the n-th element.

Overview

- The contents of the inputs are not altered by the function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Accepts custom compare_functions for nth_element across the device.
- Streams in graph capture mode are not supported

Stability

`nth_element` is **not stable**: it doesn't necessarily preserve the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **not guaranteed** that `a` will precede `b` as well in the output keys.

Example

In this example a device-level nth_element is performed where input keys are represented by an array of unsigned integers.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
size_t nth;                 // e.g., 4
unsigned int * keys_input;  // e.g., [ 6, 3, 5, 4, 1, 8, 2, 7 ]
unsigned int * keys_output; // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::nth_element(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, nth, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform nth_element
rocprim::nth_element(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, keys_output, nth, input_size
);
// possible keys_output:   [ 1, 3, 4, 2, 5, 8, 7, 6 ]
```

Template Parameters

- **`Config`** – [optional] configuration of the primitive. It has to be *nth_element_config*.
- **`KeysInputIterator`** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.

- **KeysOutputIterator** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **CompareFunction** – [inferred] Type of binary function that accepts two arguments of the type `KeysIterator` and returns a value convertible to bool. Default type is `rocprim::less<>`.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the nth_element rearrangement.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** iterator to the input range.
- **keys_output** – **[out]** iterator to the output range. No overlap at all is allowed between `keys_input` and `keys_output`. `keys_output` should be able to be written and read from for `size` elements.
- **nth** – **[in]** The index of the nth_element in the input range.
- **size** – **[in]** number of element in the input range.
- **compare_function** – **[in]** binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The comparator must meet the C++ named requirement Compare. The default value is `BinaryFunction()`.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful rearrangement; otherwise a HIP runtime error of type `hipError_t`.

2.3.7 Merge

Configuring the kernel

> **Warning**
>
> doxygentypedef: Cannot find typedef "rocprim::merge_config" in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml

merge

template<class **Config** = *default_config*, class **InputIterator1**, class **InputIterator2**, class **OutputIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*InputIterator1*>::value_type>>
inline hipError_t `rocprim::merge`(void *temporary_storage, size_t &storage_size, *InputIterator1* input1, *InputIterator2* input2, *OutputIterator* output, const size_t input1_size, const size_t input2_size, *BinaryFunction* compare_function = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel merge primitive for device level.

`merge` function performs a device-wide merge. Function merges two ordered sets of input values based on comparison function.

Overview

- The contents of the inputs are not altered by the merging function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Accepts custom compare_functions for merging across the device.

Example

In this example a device-level ascending merge is performed on an array of `int` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size1;     // e.g., 4
size_t input_size2;     // e.g., 4
int * input1;           // e.g., [0, 1, 2, 3]
int * input2;           // e.g., [0, 1, 2, 3]
int * output;           // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::merge(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input1, input2, output, input_size1, input_size2
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform merge
rocprim::merge(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input1, input2, output, input_size1, input_size2
);
// output: [0, 0, 1, 1, 2, 2, 3, 3]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or `merge_config`.
- **InputIterator1** – - random-access iterator type of the first input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **InputIterator2** – - random-access iterator type of the second input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input1** – **[in]** - iterator to the first element in the first range to merge.
- **input2** – **[in]** - iterator to the first element in the second range to merge.
- **output** – **[out]** - iterator to the first element in the output range.
- **input1_size** – **[in]** - number of element in the first input range.
- **input2_size** – **[in]** - number of element in the second input range.
- **compare_function** – **[in]** - binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **KeysInputIterator1**, class **KeysInputIterator2**, class **KeysOutputIterator**, class **ValuesInputIterator1**, class **ValuesInputIterator2**, class **ValuesOutputIterator**, class **BinaryFunction** = ::rocprim::less<typename std::iterator_traits<*KeysInputIterator1*>::value_type>>
inline hipError_t **rocprim::merge**(void *temporary_storage, size_t &storage_size, *KeysInputIterator1* keys_input1, *KeysInputIterator2* keys_input2, *KeysOutputIterator* keys_output, *ValuesInputIterator1* values_input1, *ValuesInputIterator2* values_input2, *ValuesOutputIterator* values_output, const size_t input1_size, const size_t input2_size, *BinaryFunction* compare_function = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel merge primitive for device level.

`merge` function performs a device-wide merge of (key, value) pairs. Function merges two ordered sets of input keys and corresponding values based on key comparison function.

Overview

- The contents of the inputs are not altered by the merging function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Accepts custom compare_functions for merging across the device.

Example

In this example a device-level ascending merge is performed on an array of `int` values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
```

(continues on next page)

(continued from previous page)

```
    size_t input_size1;     // e.g., 4
    size_t input_size2;     // e.g., 4
    int * keys_input1;      // e.g., [0, 1, 2, 3]
    int * keys_input2;      // e.g., [0, 1, 2, 3]
    int * keys_output;      // empty array of 8 elements
    int * values_input1;    // e.g., [10, 11, 12, 13]
    int * values_input2;    // e.g., [20, 21, 22, 23]
    int * values_output;    // empty array of 8 elements

    size_t temporary_storage_size_bytes;
    void * temporary_storage_ptr = nullptr;
    // Get required size of the temporary storage
    rocprim::merge(
        temporary_storage_ptr, temporary_storage_size_bytes,
        keys_input1, keys_input2, keys_output,
        values_input1, values_input2, values_output,
//      input_size1, input_size2
    );

    // allocate temporary storage
    hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

    // perform merge
    rocprim::merge(
        temporary_storage_ptr, temporary_storage_size_bytes,
        keys_input1, keys_input2, keys_output,
        values_input1, values_input2, values_output,
//      input_size1, input_size2
    );
    // keys_output: [0, 0, 1, 1, 2, 2, 3, 3]
    // values_output: [10, 20, 11, 21, 12, 22, 13, 23]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or merge_config.
- **KeysInputIterator1** – - random-access iterator type of the first keys input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysInputIterator2** – - random-access iterator type of the second keys input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **KeysOutputIterator** – - random-access iterator type of the keys output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator1** – - random-access iterator type of the first values input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator2** – - random-access iterator type of the second values input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the values output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the sort operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input1** – **[in]** - iterator to the first key in the first range to merge.
- **keys_input2** – **[in]** - iterator to the first key in the second range to merge.
- **keys_output** – **[out]** - iterator to the first key in the output range.
- **values_input1** – **[in]** - iterator to the first value in the first range to merge.
- **values_input2** – **[in]** - iterator to the first value in the second range to merge.
- **values_output** – **[out]** - iterator to the first value in the output range.
- **input1_size** – **[in]** - number of element in the first input range.
- **input2_size** – **[in]** - number of element in the second input range.
- **compare_function** – **[in]** - binary operation function object that will be used for key comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful sort; otherwise a HIP runtime error of type `hipError_t`.

2.3.8 Partition

partition

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **SelectedCountOutputIterator**, class **UnaryPredicate**>
inline hipError_t **rocprim::partition**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, *SelectedCountOutputIterator* selected_count_output, const size_t size, *UnaryPredicate* predicate, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel select primitive for device level using selection predicate.

Performs a device-wide partition using selection predicate. Partition copies the values from `input` to `output` in such a way that all values for which the `predicate` returns `true` precede the elements for which it returns `false`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Range specified by `selected_count_output` must have at least 1 element.
- Relative order is preserved for the elements for which the `predicate` returns `true`. Other elements are copied in reverse order.

Example

In this example a device-level partition operation is performed on an array of integer values, even values are copied before odd values.

```
#include <rocprim/rocprim.hpp>

auto predicate =
    [] __device__ (int a) -> bool
    {
        return (a%2) == 0;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;     // e.g., 8
int * input;           // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output;          // empty array of 8 elements
size_t * output_count; // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::partition(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    output, output_count,
    input_size,
    predicate
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform partition
rocprim::partition(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    output, output_count,
    input_size,
    predicate
);
// output: [2, 4, 6, 8, 7, 5, 3, 1]
// output_count: 4
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **SelectedCountOutputIterator** – - random-access iterator type of the se-

lected_count_output value. It can be a simple pointer type.

- **UnaryPredicate** – - type of a unary selection predicate.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the select operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.
- **output** – **[out]** - iterator to the first element in the output range.
- **selected_count_output** – **[out]** - iterator to the total number of selected values (length of `output`).
- **size** – **[in]** - number of elements in the input range.
- **predicate** – **[in]** - unary function object which returns `true` if the element should be ordered before other elements. The signature of the function should be equivalent to the following: `bool f(const T &a);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

partition_two_way

template<class **Config** = *default_config*, class **InputIterator**, class **SelectedOutputIterator**, class **RejectedOutputIterator**, class **SelectedCountOutputIterator**, class **Predicate**>
inline hipError_t `rocprim::`**partition_two_way**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *SelectedOutputIterator* output_selected, *RejectedOutputIterator* output_rejected, *SelectedCountOutputIterator* selected_count_output, const size_t size, *Predicate* predicate, const hipStream_t stream = 0, const bool debug_synchronous = false)

Two-way parallel select primitive for device level using selection predicate.

Performs a device-wide partition using selection predicate. Partition copies the values from `input` to `output_selected` and `output_rejected` for all values for which the `predicate` returns `true` and `false` respectively.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- The number of elements written to `output_selected` is equal to the number elements in the input for which `predicate` returns `true`.
- The number of elements written to `output_rejected` is equal to the number elements in the input for which `predicate` returns `false`.
- Range specified by `selected_count_output` must have at least 1 element.
- Relative order is preserved.

Example

In this example a device-level two-way partition operation is performed on an array of integer values, even values are copied into the selected output and odd values are copied into rejected output.

```
#include <rocprim/rocprim.hpp>

auto predicate =
    [] __device__ (int a) -> bool
    {
        return (a%2) == 0;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;     // e.g., 8
int * input;           // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * selected_output; // empty array of at least 4 elements
int * rejected_output; // empty array of at least 4 elements
size_t * output_count; // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::partition_two_way(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    selected_output,
    rejected_output,
    selected_output_count,
    input_size,
    predicate
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform partition
rocprim::partition_two_way(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    selected_output,
    rejected_output,
    selected_output_count,
    input_size,
    predicate
);
// output_selected: [2, 4, 6, 8]
// output_rejected: [1, 3, 5, 7]
// selected_output_count: 4
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.

- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **SelectedOutputIterator** – - random-access iterator type of the selected output range. It can be a simple pointer type.
- **RejectedOutputIterator** – - random-access iterator type of the rejected output range. It can be a simple pointer type.
- **SelectedCountOutputIterator** – - random-access iterator type of the selected_count_output value. It can be a simple pointer type.
- **Predicate** – - type of the selection predicate.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the select operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.
- **output_selected** – **[out]** - iterator to the first element in the selected output range.
- **output_rejected** – **[out]** - iterator to the first element in the rejected output range.
- **selected_count_output** – **[out]** - iterator to the total number of selected values (length of `output_selected`).
- **size** – **[in]** - number of elements in the input range.
- **predicate** – **[in]** - the unary selection predicate to select values into the select and reject outputs.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

partition_three_way

template<class **Config** = *default_config*, typename **InputIterator**, typename **FirstOutputIterator**, typename **SecondOutputIterator**, typename **UnselectedOutputIterator**, typename **SelectedCountOutputIterator**, typename **FirstUnaryPredicate**, typename **SecondUnaryPredicate**>
inline hipError_t `rocprim::`**partition_three_way**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *FirstOutputIterator* output_first_part, *SecondOutputIterator* output_second_part, *UnselectedOutputIterator* output_unselected, *SelectedCountOutputIterator* selected_count_output, const size_t size, *FirstUnaryPredicate* select_first_part_op, *SecondUnaryPredicate* select_second_part_op, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel select primitive for device level using two selection predicates.

Performs a device-wide three-way partition using two selection predicates. Partition copies the values from `input` to either `output_first_part` or `output_second_part` or `output_unselected` according to the following criteria: The value is copied to `output_first_part` if the predicate `select_first_part_op` invoked with the value returns `true`. It is copied to `output_second_part` if `select_first_part_op` returns `false` and `select_second_part_op` returns `true`, and it is copied to `output_unselected` otherwise.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Range specified by `selected_count_output` must have at least 2 elements.
- Relative order is preserved for the elements.
- The number of elements written to `output_first_part` is equal to the number of elements in the input for which `select_first_part_op` returned `true`.
- The number of elements written to `output_second_part` is equal to the number of elements in the input for which `select_first_part_op` returned `false` and `select_second_part_op` returned `true`.
- The number of elements written to `output_unselected` is equal to the number of input elements minus the number of elements written to `output_first_part` minus the number of elements written to `output_second_part`.

Example

In this example a device-level three-way partition operation is performed on an array of integer values, even values are copied to the first partition, odd and 3-divisible values are copied to the second partition, and the rest of the values are copied to the unselected partition

```
#include <rocprim/rocprim.hpp>

auto first_predicate =
    [] __device__ (int a) -> bool
    {
        return (a%2) == 0;
    };
auto second_predicate =
    [] __device__ (int a) -> bool
    {
        return (a%3) == 0;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
int * input;                // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output_first_part;    // array of 8 elements
int * output_second_part;   // array of 8 elements
int * output_unselected;    // array of 8 elements
size_t * output_count;      // array of 2 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::partition_three_way(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    output_first_part, output_second_part, output_unselected,
    output_count,
    input_size,
```

(continues on next page)

(continued from previous page)

```
    first_predicate,
    second_predicate
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform partition
rocprim::partition_three_way(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input,
    output_first_part, output_second_part, output_unselected,
    output_count,
    input_size,
    first_predicate,
    second_predicate
);
// elements denoted by '*' were not modified
// output_first_part:  [2, 4, 6, 8, *, *, *, *]
// output_second_part: [3, *, *, *, *, *, *, *]
// output_unselected:  [1, 5, 7, *, *, *, *, *]
// output_count:       [4, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **FirstOutputIterator** – - random-access iterator type of the first output range. It can be a simple pointer type.
- **SecondOutputIterator** – - random-access iterator type of the second output range. It can be a simple pointer type.
- **UnselectedOutputIterator** – - random-access iterator type of the unselected output range. It can be a simple pointer type.
- **SelectedCountOutputIterator** – - random-access iterator type of the selected_count_output value. It can be a simple pointer type.
- **FirstUnaryPredicate** – - type of the first unary selection predicate.
- **SecondUnaryPredicate** – - type of the second unary selection predicate.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the select operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.
- **output_first_part** – **[out]** - iterator to the first element in the first output range.

- **output_second_part** – **[out]** - iterator to the first element in the second output range.
- **output_unselected** – **[out]** - iterator to the first element in the unselected output range.
- **selected_count_output** – **[out]** - iterator to the total number of selected values in `output_first_part` and `output_second_part` respectively.
- **size** – **[in]** - number of elements in the input range.
- **select_first_part_op** – **[in]** - unary function object which returns `true` if the element should be in `output_first_part` range The signature of the function should be equivalent to the following: `bool f(const T &a);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **select_second_part_op** – **[in]** - unary function object which returns `true` if the element should be in `output_second_part` range (given that `select_first_part_op` returned `false`) The signature of the function should be equivalent to the following: `bool f(const T &a);`. The signature does not need to have `const &`, but function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

2.3.9 Run Length Encode

Configuring the kernel

template<typename **ReduceByKeyConfig**, typename **SelectConfig** = *default_config*>

struct **run_length_encode_config**

Configuration of device-level run-length encoding operation.

Template Parameters

- **ReduceByKeyConfig** – - configuration of device-level reduce-by-key operation. Must be *reduce_by_key_config* or *default_config*.
- **SelectConfig** – - configuration of device-level select operation. Must be *select_config* or *default_config*.

run_length_encode

template<typename **Config** = *default_config*, typename **InputIterator**, typename **UniqueOutputIterator**, typename **CountsOutputIterator**, typename **RunsCountOutputIterator**>
inline hipError_t `rocprim::`**run_length_encode**(void *temporary_storage, size_t &storage_size, *InputIterator* input, unsigned int size, *UniqueOutputIterator* unique_output, *CountsOutputIterator* counts_output, *RunsCountOutputIterator* runs_count_output, hipStream_t stream = 0, bool debug_synchronous = false)

Parallel run-length encoding for device level.

run_length_encode function performs a device-wide run-length encoding of runs (groups) of consecutive values. The first value of each run is copied to `unique_output` and the length of the run is written to `counts_output`. The total number of runs is written to `runs_count_output`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Range specified by `input` must have at least `size` elements.
- Range specified by `runs_count_output` must have at least 1 element.
- Ranges specified by `unique_output` and `counts_output` must have at least `*runs_count_output` (i.e. the number of runs) elements.

Example

In this example a device-level run-length encoding operation is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
int * input;                // e.g., [1, 1, 1, 2, 10, 10, 10, 88]
int * unique_output;        // empty array of at least 4 elements
int * counts_output;        // empty array of at least 4 elements
int * runs_count_output;    // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::run_length_encode(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, input_size,
    unique_output, counts_output, runs_count_output
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform encoding
rocprim::run_length_encode(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, input_size,
    unique_output, counts_output, runs_count_output
);
// unique_output:     [1, 2, 10, 88]
// counts_output:     [3, 1,  3,  1]
// runs_count_output: [4]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *run_length_encode_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **UniqueOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **CountsOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

- **RunsCountOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range of values.
- **size** – **[in]** - number of element in the input range.
- **unique_output** – **[out]** - iterator to the first element in the output range of unique values.
- **counts_output** – **[out]** - iterator to the first element in the output range of lenghts.
- **runs_count_output** – **[out]** - iterator to total number of runs.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful operation; otherwise a HIP runtime error of type `hipError_t`.

run_length_encode_non_trivial_runs

template<typename **Config** = *default_config*, typename **InputIterator**, typename **OffsetsOutputIterator**, typename **CountsOutputIterator**, typename **RunsCountOutputIterator**>
inline hipError_t `rocprim::`**run_length_encode_non_trivial_runs**(void *temporary_storage, size_t &storage_size, *InputIterator* input, unsigned int size, *OffsetsOutputIterator* offsets_output, *CountsOutputIterator* counts_output, *RunsCountOutputIterator* runs_count_output, hipStream_t stream = 0, bool debug_synchronous = false)

Parallel run-length encoding of non-trivial runs for device level.

run_length_encode_non_trivial_runs function performs a device-wide run-length encoding of non-trivial runs (groups) of consecutive values (groups of more than one element). The offset of the first value of each non-trivial run is copied to `offsets_output` and the length of the run (the count of elements) is written to `counts_output`. The total number of non-trivial runs is written to `runs_count_output`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Range specified by `input` must have at least `size` elements.
- Range specified by `runs_count_output` must have at least 1 element.
- Ranges specified by `offsets_output` and `counts_output` must have at least `*runs_count_output` (i.e. the number of non-trivial runs) elements.

Example

In this example a device-level run-length encoding of non-trivial runs is performed on an array of integer values.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
int * input;                // e.g., [1, 1, 1, 2, 10, 10, 10, 88]
int * offsets_output;       // empty array of at least 2 elements
int * counts_output;        // empty array of at least 2 elements
int * runs_count_output;    // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::run_length_encode_non_trivial_runs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, input_size,
    offsets_output, counts_output, runs_count_output
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform encoding
rocprim::run_length_encode_non_trivial_runs(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, input_size,
    offsets_output, counts_output, runs_count_output
);
// offsets_output:    [0, 4]
// counts_output:     [3, 3]
// runs_count_output: [2]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *run_length_encode_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OffsetsOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **CountsOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **RunsCountOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.

- **input** – **[in]** - iterator to the first element in the range of values.
- **size** – **[in]** - number of element in the input range.
- **offsets_output** – **[out]** - iterator to the first element in the output range of offsets.
- **counts_output** – **[out]** - iterator to the first element in the output range of lenghts.
- **runs_count_output** – **[out]** - iterator to total number of runs.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful operation; otherwise a HIP runtime error of type `hipError_t`.

2.3.10 Scan

Configuring the kernel

scan

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, ::rocprim::*block_load_method* **BlockLoadMethod**, ::rocprim::*block_store_method* **BlockStoreMethod**, ::rocprim::*block_scan_algorithm* **BlockScanMethod**, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **scan_config** : public rocprim::detail::scan_config_params

Configuration of device-level scan primitives.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.
- **BlockLoadMethod** – - method for loading input values.
- **StoreLoadMethod** – - method for storing values.
- **BlockScanMethod** – - algorithm for block scan.
- **SizeLimit** – - limit on the number of items for a single scan kernel launch.

Subclassed by rocprim::detail::default_scan_config< arch, value_type, enable >

scan_by_key

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, ::rocprim::*block_load_method* **BlockLoadMethod**, ::rocprim::*block_store_method* **BlockStoreMethod**, ::rocprim::*block_scan_algorithm* **BlockScanMethod**, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **scan_by_key_config** : public rocprim::detail::scan_by_key_config_params

Configuration of device-level scan-by-key operation.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.
- **BlockLoadMethod** – - method for loading input values.
- **StoreLoadMethod** – - method for storing values.

- **BlockScanMethod** – - algorithm for block scan.
- **SizeLimit** – - limit on the number of items for a single scan kernel launch.

Subclassed by rocprim::detail::default_scan_by_key_config< arch, key_type, value_type, enable >

scan

inclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>, class **AccType** = typename std::iterator_traits<*InputIterator*>::value_type>
inline hipError_t `rocprim::inclusive_scan`(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const size_t size, *BinaryFunction* scan_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel inclusive scan primitive for device level.

inclusive_scan function performs a device-wide inclusive prefix scan operation using binary `scan_op` operator.

Overview

- Supports non-commutative scan operators. However, a scan operator should be associative.
- When used with non-associative functions (e.g. floating point arithmetic operations):
  - the results may be non-deterministic and/or vary in precision,
  - and bit-wise reproducibility is not guaranteed, that is, results from multiple runs using the same input values on the same device may not be bit-wise identical. If deterministic behavior is required, Use *rocprim::deterministic_inclusive_scan* instead.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` and `output` must have at least `size` elements.
- By default, the input type is used for accumulation. A custom type can be specified using the `AccType` type parameter, see the example below.

Example

In this example a device-level inclusive sum operation is performed on an array of integer values (`shorts` are scanned into `ints`).

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
short * input;        // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output;         // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::inclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
```

(continues on next page)

(continued from previous page)

```
    input, output, input_size, rocprim::plus<int>()
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan
rocprim::inclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, rocprim::plus<int>()
);
// output: [1, 3, 6, 10, 15, 21, 28, 36]
```

The same example as above, but now a custom accumulator type is specified.

```
#include <rocprim/rocprim.hpp>

size_t input_size;
short * input;
int * output;

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;

rocprim::inclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, rocprim::plus<int>()
);

hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// Use type parameter to set custom accumulator type
rocprim::inclusive_scan<rocprim::default_config,
                        short*,
                        int*,
                        rocprim::plus<int>,
                        int>(temporary_storage_ptr,
                             temporary_storage_size_bytes,
                             input_iterator,
                             output,
                             input_size,
                             rocprim::plus<int>());
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.

- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>, where T is a value_type of InputIterator.
- **AccType** – - accumulator type used to propagate the scanned values. Default type is value type of the input iterator.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to storage_size and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of temporary_storage.
- **input** – **[in]** - iterator to the first element in the range to scan.
- **output** – **[out]** - iterator to the first element in the output range. It can be same as input.
- **size** – **[in]** - number of element in the input range.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: T f(const T &a, const T &b); . The signature does not need to have const &, but function object must not modify the objects passed to it. Default is BinaryFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is 0 (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is false.

Returns

hipSuccess (0) after successful scan; otherwise a HIP runtime error of type hipError_t.

exclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **InitValueType**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>, class **AccType** = detail::input_type_t<*InitValueType*>>
inline hipError_t **rocprim::exclusive_scan**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const *InitValueType* initial_value, const size_t size, *BinaryFunction* scan_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel exclusive scan primitive for device level.

exclusive_scan function performs a device-wide exclusive prefix scan operation using binary scan_op operator.

Overview

- Supports non-commutative scan operators. However, a scan operator should be associative.
- When used with non-associative functions (e.g. floating point arithmetic operations):
  - the results may be non-deterministic and/or vary in precision,
  - and bit-wise reproducibility is not guaranteed, that is, results from multiple runs using the same input values on the same device may not be bit-wise identical. If deterministic behavior is required, Use *rocprim::deterministic_exclusive_scan* instead.
- Returns the required size of temporary_storage in storage_size if temporary_storage in a null pointer.
- Ranges specified by input and output must have at least size elements.

Example

In this example a device-level exclusive min-scan operation is performed on an array of integer values (`shorts` are scanned into `ints`) using custom operator.

```
#include <rocprim/rocprim.hpp>

// custom scan function
auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
short * input;        // e.g., [4, 7, 6, 2, 5, 1, 3, 8]
int * output;         // empty array of 8 elements
int start_value;      // e.g., 9

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::exclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, start_value, input_size, min_op
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan
rocprim::exclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, start_value, input_size, min_op
);
// output: [9, 4, 4, 4, 2, 2, 1, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **InitValueType** – - type of the initial value.
- **BinaryFunction** – - type of binary function used for scan. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.
- **AccType** – - accumulator type used to propagate the scanned values. Default type is 'InitValueType', unless it's '*rocprim::future_value*'. Then it will be the wrapped input type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to scan.
- **output** – **[out]** - iterator to the first element in the output range. It can be same as `input`.
- **initial_value** – **[in]** - initial value to start the scan. A rocpim::future_value may be passed to use a value that will be later computed.
- **size** – **[in]** - number of element in the input range.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

Returns

`hipSuccess` (`0`) after successful scan; otherwise a HIP runtime error of type `hipError_t`.

deterministic, inclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>, class **AccType** = typename std::iterator_traits<*InputIterator*>::value_type>
inline hipError_t `rocprim::deterministic_inclusive_scan`(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const size_t size, *BinaryFunction* scan_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Bitwise-reproducible parallel inclusive scan primitive for device level.

This function behaves the same as *`inclusive_scan()`*, except that unlike *`inclusive_scan()`*, it provides run-to-run deterministic behavior for non-associative scan operators like floating point arithmetic operations. Refer to the documentation for *rocprim::inclusive_scan* for a detailed description of this function.

deterministic, exclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **InitValueType**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>, class **AccType** = detail::input_type_t<*InitValueType*>>
inline hipError_t `rocprim::deterministic_exclusive_scan`(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const *InitValueType* initial_value, const size_t size, *BinaryFunction* scan_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Bitwise-reproducible parallel exclusive scan primitive for device level.

This function behaves the same as *exclusive_scan()*, except that unlike *exclusive_scan()*, it provides run-to-run deterministic behavior for non-associative scan operators like floating point arithmetic operations. Refer to the documentation for *rocprim::exclusive_scan* for a detailed description of this function.

segmented, inclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **OffsetIterator**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>>
inline hipError_t `rocprim::`**segmented_inclusive_scan**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, *BinaryFunction* scan_op = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel segmented inclusive scan primitive for device level.

segmented_inclusive_scan function performs a device-wide inclusive scan operation across multiple sequences from `input` using binary `scan_op` operator.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` and `output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.

Example

In this example a device-level segmented inclusive min-scan operation is performed on an array of integer values (`short`s are scanned into `int`s) using custom operator.

```
#include <rocprim/rocprim.hpp>

// custom scan function
auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
short * input;        // e.g., [4, 7, 6, 2, 5, 1, 3, 8]
int   * output;       // empty array of 8 elements
size_t segments;      // e.g., 3
int * offsets;        // e.g. [0, 2, 4, 8]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_inclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
```

(continues on next page)

(continued from previous page)

```
    input, output, segments, offsets, offsets + 1, min_op
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan
rocprim::inclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, segments, offsets, offsets + 1, min_op
);
// output: [4, 4, 6, 2, 5, 1, 1, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for scan operation. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to scan.
- **output** – **[out]** - iterator to the first element in the output range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

Returns

`hipSuccess` (`0`) after successful scan; otherwise a HIP runtime error of type `hipError_t`.

segmented, exclusive

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **OffsetIterator**, class **InitValueType**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>>
inline hipError_t **rocprim::segmented_exclusive_scan**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, const *InitValueType* initial_value, *BinaryFunction* scan_op = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel segmented exclusive scan primitive for device level.

segmented_exclusive_scan function performs a device-wide exclusive scan operation across multiple sequences from `input` using binary `scan_op` operator.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` and `output` must have at least `size` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.

Example

In this example a device-level segmented exclusive min-scan operation is performed on an array of integer values (`short`s are scanned into `int`s) using custom operator.

```
#include <rocprim/rocprim.hpp>

// custom scan function
auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
int start_value;      // e.g., 9
short * input;        // e.g., [4, 7, 6, 2, 5, 1, 3, 8]
int   * output;       // empty array of 8 elements
size_t segments;      // e.g., 3
int * offsets;        // e.g. [0, 2, 4, 8]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_exclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, segments, offsets, offsets + 1
```

(continues on next page)

(continued from previous page)

```
    start_value, min_op
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan
rocprim::exclusive_scan(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, segments, offsets, offsets + 1
    start_value, min_op
);
// output: [9, 4, 9, 6, 9, 5, 1, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ RandomAccessIterator concept. It can be a simple pointer type.
- **InitValueType** – - type of the initial value.
- **BinaryFunction** – - type of binary function used for scan operation. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to scan.
- **output** – **[out]** - iterator to the first element in the output range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **initial_value** – **[in]** - initial value to start the scan.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).

- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

Returns

`hipSuccess` (`0`) after successful scan; otherwise a HIP runtime error of type `hipError_t`.

scan_by_key

inclusive

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **ValuesOutputIterator**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>, typename **AccType** = typename std::iterator_traits<*ValuesInputIterator*>::value_type>

inline hipError_t `rocprim::inclusive_scan_by_key`(void *const temporary_storage, size_t &storage_size, const *KeysInputIterator* keys_input, const *ValuesInputIterator* values_input, const *ValuesOutputIterator* values_output, const size_t size, const *BinaryFunction* scan_op = *BinaryFunction*(), const *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel inclusive scan-by-key primitive for device level.

inclusive_scan_by_key function performs a device-wide inclusive prefix scan-by-key operation using binary `scan_op` operator.

Overview

- Supports non-commutative scan operators. However, a scan operator should be associative.
- When used with non-associative functions (e.g. floating point arithmetic operations):
  - the results may be non-deterministic and/or vary in precision,
  - and bit-wise reproducibility is not guaranteed, that is, results from multiple runs using the same input values on the same device may not be bit-wise identical. If deterministic behavior is required, Use *rocprim::deterministic_inclusive_scan_by_key* instead.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `keys_input`, `values_input`, and `values_output` must have at least `size` elements.

Example

In this example a device-level inclusive sum-by-key operation is performed on an array of integer values (`short`s are scanned into `int`s).

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;           // e.g., 8
int *   keys_input;    // e.g., [1, 1, 2, 2, 3, 3, 3, 5]
short * values_input;  // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
```

(continues on next page)

(continued from previous page)

```
int *   values_output; // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::inclusive_scan_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input,
    values_output, size,
    rocprim::plus<int>()
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan-by-key
rocprim::inclusive_scan_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input,
    values_output, size,
    rocprim::plus<int>()
);
// values_output: [1, 3, 3, 7, 5, 11, 18, 8]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_by_key_config*.
- **KeysInputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for scan. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.
- **KeyCompareFunction** – - type of binary function used to determine keys equality. Default type is `rocprim::equal_to<T>`, where `T` is a `value_type` of `KeysInputIterator`.
- **AccType** – - accumulator type used to propagate the scanned values. Default type is value type of the input iterator.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - iterator to the first element in the range of keys.
- **values_input** – **[in]** - iterator to the first element in the range of values to scan.

- **values_output** – **[out]** - iterator to the first element in the output value range.
- **size** – **[in]** - number of element in the input range.
- **scan_op** – **[in]** - binary operation function object that will be used for scanning input values. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. Default is BinaryFunction().
- **key_compare_op** – **[in]** - binary operation function object that will be used to determine keys equality. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. Default is KeyCompareFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful scan; otherwise a HIP runtime error of type `hipError_t`.

exclusive

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **ValuesOutputIterator**, typename **InitialValueType**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>, typename **AccType** = detail::input_type_t<*InitialValueType*>>
inline hipError_t **rocprim::exclusive_scan_by_key**(void *const temporary_storage, size_t &storage_size, const *KeysInputIterator* keys_input, const *ValuesInputIterator* values_input, const *ValuesOutputIterator* values_output, const *InitialValueType* initial_value, const size_t size, const *BinaryFunction* scan_op = *BinaryFunction*(), const *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel exclusive scan-by-key primitive for device level.

inclusive_scan_by_key function performs a device-wide exclusive prefix scan-by-key operation using binary `scan_op` operator.

Overview

- Supports non-commutative scan operators. However, a scan operator should be associative.
- When used with non-associative functions (e.g. floating point arithmetic operations):
  - the results may be non-deterministic and/or vary in precision,
  - and bit-wise reproducibility is not guaranteed, that is, results from multiple runs using the same input values on the same device may not be bit-wise identical. If deterministic behavior is required, Use *rocprim::deterministic_exclusive_scan_by_key* instead.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `keys_input`, `values_input`, and `values_output` must have at least `size` elements.

Example

In this example a device-level inclusive sum-by-key operation is performed on an array of integer values (`shorts` are scanned into `ints`).

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;           // e.g., 8
int *   keys_input;    // e.g., [1, 1, 1, 2, 2, 3, 3, 4]
short * values_input;  // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int start_value;       // e.g., 9
int *   values_output; // empty array of 8 elements

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::exclusive_scan_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input,
    values_output, start_value,
    size,rocprim::plus<int>()
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform scan-by-key
rocprim::exclusive_scan_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input,
    values_output, start_value,
    size,rocprim::plus<int>()
);
// values_output: [9, 10, 12, 9, 13, 9, 15, 9]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *scan_by_key_config*.
- **KeysInputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **ValuesOutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **InitValueType** – - type of the initial value.
- **BinaryFunction** – - type of binary function used for scan. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.
- **KeyCompareFunction** – - type of binary function used to determine keys equality. Default type is `rocprim::equal_to<T>`, where `T` is a `value_type` of `KeysInputIterator`.

- **AccType** – - accumulator type used to propagate the scanned values. Default type is 'InitValueType', unless it's '*rocprim::future_value*'. Then it will be the wrapped input type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - iterator to the first element in the range of keys.
- **values_input** – **[in]** - iterator to the first element in the range of values to scan.
- **values_output** – **[out]** - iterator to the first element in the output value range.
- **initial_value** – **[in]** - initial value to start the scan. A rocpim::future_value may be passed to use a value that will be later computed.
- **size** – **[in]** - number of element in the input range.
- **scan_op** – **[in]** - binary operation function object that will be used for scanning input values. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. Default is BinaryFunction().
- **key_compare_op** – **[in]** - binary operation function object that will be used to determine keys equality. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. Default is KeyCompareFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful scan; otherwise a HIP runtime error of type `hipError_t`.

deterministic, inclusive

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **ValuesOutputIterator**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>, typename **AccType** = typename std::iterator_traits<*ValuesInputIterator*>::value_type>
inline hipError_t `rocprim::deterministic_inclusive_scan_by_key`(void *const temporary_storage, size_t &storage_size, const *KeysInputIterator* keys_input, const *ValuesInputIterator* values_input, const *ValuesOutputIterator* values_output, const size_t size, const *BinaryFunction* scan_op = *BinaryFunction*(), const *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Bitwise-reproducible parallel inclusive scan-by-key primitive for device level.

This function behaves the same as *inclusive_scan_by_key()*, except that unlike *inclusive_scan_by_key()*, it provides run-to-run deterministic behavior for non-associative scan operators like floating point arithmetic operations. Refer to the documentation for *rocprim::inclusive_scan_by_key* for a detailed description of this function.

deterministic, exclusive

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **ValuesOutputIterator**, typename **InitialValueType**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>, typename **AccType** = detail::input_type_t<*InitialValueType*>>

inline hipError_t **rocprim::deterministic_exclusive_scan_by_key**(void *const temporary_storage, size_t &storage_size, const *KeysInputIterator* keys_input, const *ValuesInputIterator* values_input, const *ValuesOutputIterator* values_output, const *InitialValueType* initial_value, const size_t size, const *BinaryFunction* scan_op = *BinaryFunction*(), const *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), const hipStream_t stream = 0, const bool debug_synchronous = false)

Bitwise-reproducible parallel exclusive scan-by-key primitive for device level.

This function behaves the same as *exclusive_scan_by_key()*, except that unlike *exclusive_scan_by_key()*, it provides run-to-run deterministic behavior for non-associative scan operators like floating point arithmetic operations. Refer to the documentation for *rocprim::exclusive_scan_by_key* for a detailed description of this function.

2.3.11 Search N

Configuring the kernel

Warning

doxygenstruct: Cannot find class "rocprim::search_n" in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml

search_n

Warning

doxygenfunction: Unable to resolve function "rocprim::search_n" with arguments (void*const, size_t&, InputIterator, OutputIterator, const size_t, const size_t, const typename std::iterator_traits<InputIterator>::value_type*, const BinaryPredicate, const hipStream_t, const bool) in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml. Potential matches:

```
- template<class Config = default_config, class InputIterator, class OutputIterator,␣
↪class BinaryPredicate = rocprim::equal_to<typename std::iterator_traits
```

```
↪<InputIterator>::value_type>> hipError_t search_n(void *temporary_storage, size_t &
↪storage_size, InputIterator input, OutputIterator output, const size_t size, const␣
↪size_t count, const typename std::iterator_traits<InputIterator>::value_type *value,
↪ const BinaryPredicate binary_predicate = BinaryPredicate(), const hipStream_t␣
↪stream = static_cast<hipStream_t>(0), const bool debug_synchronous = false)
```

2.3.12 Select

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, ::rocprim::*block_load_method* **KeyBlockLoadMethod** = ::rocprim::*block_load_method*::*block_load_transpose*, ::rocprim::*block_load_method* **ValueBlockLoadMethod** = ::rocprim::*block_load_method*::*block_load_transpose*, ::rocprim::*block_load_method* **FlagBlockLoadMethod** = ::rocprim::*block_load_method*::*block_load_transpose*, ::rocprim::*block_scan_algorithm* **BlockScanMethod** = ::rocprim::*block_scan_algorithm*::*using_warp_scan*, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **select_config** : public rocprim::detail::partition_config_params

Configuration of device-level partition and select operation.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.
- **KeyBlockLoadMethod** – - method for loading input keys.
- **ValueBlockLoadMethod** – - method for loading input values.
- **FlagBlockLoadMethod** – - method for loading flag values.
- **BlockScanMethod** – - algorithm for block scan.
- **SizeLimit** – - limit on the number of items for a single select kernel launch.

Subclassed by rocprim::detail::default_partition_flag_config< arch, data_type, enable >, rocprim::detail::default_partition_predicate_config< arch, data_type, enable >, rocprim::detail::default_partition_three_way_config< arch, data_type, enable >, rocprim::detail::default_partition_two_way_flag_config< arch, data_type, enable >, rocprim::detail::default_partition_two_way_predicate_config< arch, data_type, enable >, rocprim::detail::default_select_flag_config< arch, data_type, enable >, rocprim::detail::default_select_predicate_config< arch, data_type, enable >, rocprim::detail::default_select_predicated_flag_config< arch, data_type, flag_type, enable >, rocprim::detail::default_select_unique_by_key_config< arch, key_type, value_type, enable >, rocprim::detail::default_select_unique_config< arch, data_type, enable >

select

template<class **Config** = *default_config*, class **InputIterator**, class **FlagIterator**, class **OutputIterator**, class **SelectedCountOutputIterator**>
inline hipError_t **rocprim::select**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *FlagIterator* flags, *OutputIterator* output, *SelectedCountOutputIterator* selected_count_output, const size_t size, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel select primitive for device level using range of flags.

Performs a device-wide selection based on input `flags`. If a value from `input` should be selected and copied into `output` range the corresponding item from `flags` range should be set to such value that can be implicitly converted to `true` (`bool` type).

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` and `flags` must have at least `size` elements.
- Range specified by `output` must have at least so many elements, that all positively flagged values can be copied into it.
- Range specified by `selected_count_output` must have at least 1 element.
- Values of `flag` range should be implicitly convertible to `bool` type.

Example

In this example a device-level select operation is performed on an array of integer values with array of `char`s used as flags.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;     // e.g., 8
int * input;           // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
char * flags;          // e.g., [0, 1, 1, 0, 0, 1, 0, 1]
int * output;          // empty array of 8 elements
size_t * output_count; // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::select(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, flags,
    output, output_count,
    input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform selection
rocprim::select(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, flags,
    output, output_count,
    input_size
);
// output: [2, 3, 6, 8]
// output_count: 4
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **FlagIterator** – - random-access iterator type of the flag range. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **SelectedCountOutputIterator** – - random-access iterator type of the selected_count_output value. It can be a simple pointer type.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the select operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.
- **flags** – **[in]** - iterator to the selection flag corresponding to the first element from `input` range.
- **output** – **[out]** - iterator to the first element in the output range.
- **selected_count_output** – **[out]** - iterator to the total number of selected values (length of `output`).
- **size** – **[in]** - number of element in the input range.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **SelectedCountOutputIterator**, class **UnaryPredicate**>
inline hipError_t **rocprim::select**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, *SelectedCountOutputIterator* selected_count_output, const size_t size, *UnaryPredicate* predicate, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel select primitive for device level using selection operator.

Performs a device-wide selection using selection operator. If a value `x` from `input` should be selected and copied into `output` range, then `predicate(x)` has to return `true`.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Range specified by `input` must have at least `size` elements.
- Range specified by `output` must have at least so many elements, that all selected values can be copied into it.
- Range specified by `selected_count_output` must have at least 1 element.

Example

In this example a device-level select operation is performed on an array of integer values. Only even values are selected.

```
#include <rocprim/rocprim.hpp>

auto predicate =
    [] __device__ (int a) -> bool
    {
        return (a % 2) == 0;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;     // e.g., 8
int * input;           // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output;          // empty array of 8 elements
size_t * output_count; // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::select(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, output_count,
    predicate, input_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform selection
rocprim::select(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, output_count,
    predicate, input_size
);
// output: [2, 4, 6, 8]
// output_count: 4
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *select_config*.
- **InputIterator** – - random-access iterator type of the input range. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. It can be a simple pointer type.
- **SelectedCountOutputIterator** – - random-access iterator type of the selected_count_output value. It can be a simple pointer type.
- **UnaryPredicate** – - type of a unary selection predicate.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the select operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to select values from.
- **output** – **[out]** - iterator to the first element in the output range.
- **selected_count_output** – **[out]** - iterator to the total number of selected values (length of `output`).
- **size** – **[in]** - number of element in the input range.
- **predicate** – **[in]** - unary function object that will be used for selecting values. The predicate must meet the C++ named requirement `BinaryPredicate` :
  - The result of applying the predicate must be convertible to bool
  - The predicate must accept const object arguments, with the same behavior regardless of whether its arguments are const or non-const. In practice, the signature of the function should be equivalent to the following: `bool f(const T &a);`. The signature does not need to have `const &`, but the function object must not modify the object passed to it.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

2.3.13 Reduce

Configuring the kernel

reduce

template<unsigned int **BlockSize** = 256, unsigned int **ItemsPerThread** = 8, ::rocprim::*block_reduce_algorithm* **BlockReduceMethod** = ::rocprim::*block_reduce_algorithm*::*default_algorithm*, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **reduce_config** : public rocprim::detail::reduce_config_params

Configuration of device-level reduce primitives.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.
- **BlockReduceMethod** – - algorithm for block reduce.
- **SizeLimit** – - limit on the number of items reduced by a single launch

Subclassed by rocprim::detail::default_reduce_config< arch, key_type, enable >, rocprim::detail::default_segmented_reduce_config< arch, key_type, enable >

reduce_by_key

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, *block_load_method* **LoadKeysMethod** = *block_load_method*::*block_load_transpose*, *block_load_method* **LoadValuesMethod** = *block_load_method*::*block_load_transpose*, *block_scan_algorithm* **ScanAlgorithm** = *block_scan_algorithm*::*using_warp_scan*, unsigned int **TilesPerBlock** = 1, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>

struct **reduce_by_key_config** : public rocprim::detail::reduce_by_key_config_params

Configuration of device-level reduce-by-key operation.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread per tile.
- **LoadKeysMethod** – method of loading keys
- **LoadValuesMethod** – method of loading values
- **ScanAlgorithm** – block level scan algorithm to use
- **TilesPerBlock** – number of tiles (`BlockSize` * `ItemsPerThread` items) to process per block. This parameter is only here for legacy purposes. Its no longer used.
- **SizeLimit** – limit on the number of items for a single reduce_by_key kernel launch.

reduce

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **InitValueType**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>>
inline hipError_t **rocprim::reduce**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const *InitValueType* initial_value, const size_t size, *BinaryFunction* reduce_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel reduction primitive for device level.

reduce function performs a device-wide reduction operation using binary `reduce_op` operator.

Overview

- Does not support non-commutative reduction operators. Reduction operator should also be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` must have at least `size` elements, while `output` only needs one element.
- By default, the input type is used for accumulation. A custom type can be specified using *rocprim::transform_iterator*, see the example below.

Example

In this example a device-level min-reduction operation is performed on an array of integer values (`short`s are reduced into `int`s) using custom operator.

```
#include <rocprim/rocprim.hpp>

// custom reduce function
auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };
```

(continues on next page)

(continued from previous page)

```
// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
short * input;        // e.g., [4, 7, 6, 2, 5, 1, 3, 8]
int * output;         // empty array of 1 element
int start_value;      // e.g., 9

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, start_value, input_size, min_op
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform reduce
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, start_value, input_size, min_op
);
// output: [1]
```

The same example as above, but now a custom accumulator type is specified.

```
#include <rocprim/rocprim.hpp>

auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };

size_t input_size;
short * input;
int * output;
int start_value;

// Use a transform iterator to specifiy a custom accumulator type
auto input_iterator = rocprim::make_transform_iterator(
    input, [] __device__ (T in) { return static_cast<int>(in); });

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Use the transform iterator
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input_iterator, output, start_value, input_size, min_op
);
```

(continues on next page)

(continued from previous page)

```
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input_iterator, output, start_value, input_size, min_op
);
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *reduce_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **InitValueType** – - type of the initial value.
- **BinaryFunction** – - type of binary function used for reduction. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to reduce.
- **output** – **[out]** - iterator to the first element in the output range. It can be same as `input`.
- **initial_value** – **[in]** - initial value to start the reduction.
- **size** – **[in]** - number of element in the input range.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduction. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

Returns

`hipSuccess` (`0`) after successful reduction; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>>
inline hipError_t `rocprim::reduce`(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, const size_t size, *BinaryFunction* reduce_op = *BinaryFunction*(), const hipStream_t stream = 0, bool debug_synchronous = false)

Parallel reduce primitive for device level.

reduce function performs a device-wide reduction operation using binary `reduce_op` operator.

Overview

- Does not support non-commutative reduction operators. Reduction operator should also be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` must have at least `size` elements, while `output` only needs one element.
- By default, the input type is used for accumulation. A custom type can be specified using *rocprim::transform_iterator*, see the example below.

Example

In this example a device-level sum operation is performed on an array of integer values (`short`s are reduced into `int`s).

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;    // e.g., 8
short * input;        // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int * output;         // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, rocprim::plus<int>()
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform reduce
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, input_size, rocprim::plus<int>()
);
// output: [36]
```

The same example as above, but now a custom accumulator type is specified.

```
#include <rocprim/rocprim.hpp>

size_t input_size;
short * input;
int * output;

// Use a transform iterator to specifiy a custom accumulator type
auto input_iterator = rocprim::make_transform_iterator(
```

(continues on next page)

(continued from previous page)

```
    input, [] __device__ (T in) { return static_cast<int>(in); });

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Use the transform iterator
rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input_iterator, output, start_value, input_size, rocprim::plus<int>()
);

hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

rocprim::reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input_iterator, output, start_value, input_size, rocprim::plus<int>()
);
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *reduce_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for reduction. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to reduce.
- **output** – **[out]** - iterator to the first element in the output range. It can be same as `input`.
- **size** – **[in]** - number of element in the input range.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduction. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. Default is BinaryFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful reduction; otherwise a HIP runtime error of type `hipError_t`.

segmented_reduce

template<class **Config** = *default_config*, class **InputIterator**, class **OutputIterator**, class **OffsetIterator**, class **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*InputIterator*>::value_type>, class **InitValueType** = typename std::iterator_traits<*InputIterator*>::value_type>
inline hipError_t **rocprim::segmented_reduce**(void *temporary_storage, size_t &storage_size, *InputIterator* input, *OutputIterator* output, unsigned int segments, *OffsetIterator* begin_offsets, *OffsetIterator* end_offsets, *BinaryFunction* reduce_op = *BinaryFunction*(), *InitValueType* initial_value = *InitValueType*(), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel segmented reduction primitive for device level.

segmented_reduce function performs a device-wide reduction operation across multiple sequences using binary `reduce_op` operator.

Overview

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `input` must have at least `size` elements, `output` must have `segments` elements.
- Ranges specified by `begin_offsets` and `end_offsets` must have at least `segments` elements. They may use the same sequence `offsets` of at least `segments + 1` elements: `offsets` for `begin_offsets` and `offsets + 1` for `end_offsets`.

Example

In this example a device-level segmented min-reduction operation is performed on an array of integer values (`short`s are reduced into `int`s) using custom operator.

```
#include <rocprim/rocprim.hpp>

// custom reduce function
auto min_op =
    [] __device__ (int a, int b) -> int
    {
        return a < b ? a : b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int segments;   // e.g., 3
short * input;           // e.g., [4, 7, 6, 2, 5, 1, 3, 8]
int * output;            // empty array of 3 elements
int * offsets;           // e.g. [0, 2, 3, 8]
int init_value;          // e.g., 9

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::segmented_reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output,
    segments, offsets, offsets + 1,
```

(continues on next page)

(continued from previous page)

```
    min_op, init_value
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform segmented reduction
rocprim::segmented_reduce(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output,
    segments, offsets, offsets + 1,
    min_op, init_value
);
// output: [4, 6, 1]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *reduce_config*.
- **InputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **OffsetIterator** – - random-access iterator type of segment offsets. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for reduction. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `InputIterator`.
- **InitValueType** – - type of the initial value.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** - iterator to the first element in the range to reduce.
- **output** – **[out]** - iterator to the first element in the output range.
- **segments** – **[in]** - number of segments in the input range.
- **begin_offsets** – **[in]** - iterator to the first element in the range of beginning offsets.
- **end_offsets** – **[in]** - iterator to the first element in the range of ending offsets.
- **initial_value** – **[in]** - initial value to start the reduction.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduction. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The default value is `BinaryFunction()`.
- **stream** – **[in]** - [optional] HIP stream object. The default is `0` (default stream).

- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

Returns

`hipSuccess` (`0`) after successful reduction; otherwise a HIP runtime error of type `hipError_t`.

reduce_by_key

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **UniqueOutputIterator**, typename **AggregatesOutputIterator**, typename **UniqueCountOutputIterator**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>>
inline hipError_t **rocprim::reduce_by_key**(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *ValuesInputIterator* values_input, const size_t size, *UniqueOutputIterator* unique_output, *AggregatesOutputIterator* aggregates_output, *UniqueCountOutputIterator* unique_count_output, *BinaryFunction* reduce_op = *BinaryFunction*(), *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel reduce-by-key primitive for device level.

reduce_by_key function performs a device-wide reduction operation on groups of consecutive values having the same key using binary `reduce_op` operator. The first key of each group is copied to `unique_output` and the reduction of the group is written to `aggregates_output`. The total number of groups is written to `unique_count_output`.

Overview

- Supports non-commutative reduction operators. However, a reduction operator should be associative.
- When used with non-associative functions (e.g. floating point arithmetic operations):
  - the results may be non-deterministic and/or vary in precision,
  - and bit-wise reproducibility is not guaranteed, that is, results from multiple runs using the same input values on the same device may not be bit-wise identical. If deterministic behavior is required, Use *rocprim::deterministic_reduce_by_key* instead.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.
- Ranges specified by `keys_input` and `values_input` must have at least `size` elements.
- Range specified by `unique_count_output` must have at least 1 element.
- Ranges specified by `unique_output` and `aggregates_output` must have at least *`unique_count_output` (i.e. the number of unique keys) elements.

Example

In this example a device-level sum operation is performed on an array of integer values and integer keys.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t input_size;          // e.g., 8
int * keys_input;           // e.g., [1, 1, 1, 2, 10, 10, 10, 88]
```

(continues on next page)

(continued from previous page)

```
int * values_input;         // e.g., [1, 2, 3, 4,  5,  6,  7,  8]
int * unique_output;        // empty array of at least 4 elements
int * aggregates_output;    // empty array of at least 4 elements
int * unique_count_output;  // empty array of 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::reduce_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input, input_size,
    unique_output, aggregates_output, unique_count_output
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform reduction
rocprim::reduce_by_key(
    temporary_storage_ptr, temporary_storage_size_bytes,
    keys_input, values_input, input_size,
    unique_output, aggregates_output, unique_count_output
);
// unique_output:       [1, 2, 10, 88]
// aggregates_output:   [6, 4, 18,  8]
// unique_count_output: [4]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *reduce_by_key_config*.
- **KeysInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **ValuesInputIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **UniqueOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **AggregatesOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **UniqueCountOutputIterator** – - random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – - type of binary function used for reduction. Default type is `rocprim::plus<T>`, where `T` is a `value_type` of `ValuesInputIterator`.
- **KeyCompareFunction** – - type of binary function used to determine keys equality. Default type is `rocprim::equal_to<T>`, where `T` is a `value_type` of `KeysInputIterator`.

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size`

and function returns without performing the reduction operation.

- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **keys_input** – **[in]** - iterator to the first element in the range of keys.
- **values_input** – **[in]** - iterator to the first element in the range of values to reduce.
- **size** – **[in]** - number of element in the input range.
- **unique_output** – **[out]** - iterator to the first element in the output range of unique keys.
- **aggregates_output** – **[out]** - iterator to the first element in the output range of reductions.
- **unique_count_output** – **[out]** - iterator to total number of groups.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduction. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it and must not have any side effects since the function may be called on uninitalized data. Default is BinaryFunction().
- **key_compare_op** – **[in]** - binary operation function object that will be used to determine key equality. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it and must not have any side effects since the function may be called on uninitalized data. Default is KeyCompareFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful reduction; otherwise a HIP runtime error of type `hipError_t`.

deterministic

template<typename **Config** = *default_config*, typename **KeysInputIterator**, typename **ValuesInputIterator**, typename **UniqueOutputIterator**, typename **AggregatesOutputIterator**, typename **UniqueCountOutputIterator**, typename **BinaryFunction** = ::rocprim::plus<typename std::iterator_traits<*ValuesInputIterator*>::value_type>, typename **KeyCompareFunction** = ::rocprim::equal_to<typename std::iterator_traits<*KeysInputIterator*>::value_type>>
inline hipError_t `rocprim::deterministic_reduce_by_key`(void *temporary_storage, size_t &storage_size, *KeysInputIterator* keys_input, *ValuesInputIterator* values_input, const size_t size, *UniqueOutputIterator* unique_output, *AggregatesOutputIterator* aggregates_output, *UniqueCountOutputIterator* unique_count_output, *BinaryFunction* reduce_op = *BinaryFunction*(), *KeyCompareFunction* key_compare_op = *KeyCompareFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Bitwise-reproducible parallel reduce-by-key primitive for device level.

This function behaves the same as *reduce_by_key()*, except that unlike *reduce_by_key()*, it provides run-to-run deterministic behavior for non-associative scan operators like floating point arithmetic operations. Refer to the documentation for *rocprim::reduce_by_key* for a detailed description of this function.

2.3.14 Adjacent Difference

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, *block_load_method* **BlockLoadMethod** = *block_load_method*::*block_load_transpose*, *block_store_method* **BlockStoreMethod** = *block_store_method*::*block_store_transpose*, unsigned int **SizeLimit** = std::numeric_limits<unsigned int>::max()>
struct **adjacent_difference_config** : public rocprim::detail::adjacent_difference_config_params

Configuration of device-level adjacent difference primitives.

Template Parameters

- **BlockSize** – - number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread.
- **BlockLoadMethod** – - method for loading input values.
- **BlockStoreMethod** – - method for storing values.
- **SizeLimit** – - limit on the number of items for a single adjacent difference kernel launch.

Subclassed by rocprim::detail::default_adjacent_difference_config< arch, value_type, enable >, rocprim::detail::default_adjacent_difference_inplace_config< arch, value_type, enable >

left

template<typename **Config** = *default_config*, typename **InputIt**, typename **OutputIt**, typename **BinaryFunction** = ::rocprim::minus<>>
hipError_t **rocprim::adjacent_difference**(void *const temporary_storage, std::size_t &storage_size, const *InputIt* input, const *OutputIt* output, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the left item.

Copies the first item to the output then performs calls the supplied operator with each pair of neighboring elements and writes its result to the location of the second element. Equivalent to the following code

```
output[0] = input[0];
for(std::size_t int i = 1; i < size; ++i)
{
    output[i] = op(input[i], input[i - 1]);
}
```

Example

In this example a device-level adjacent_difference operation is performed on integer values.

```
#include <rocprim/rocprim.hpp> //or <rocprim/device/device_adjacent_difference.
↪hpp>

// custom binary function
auto binary_op =
    [] __device__ (int a, int b) -> int
    {
        return a - b;
```

(continues on next page)

(continued from previous page)

```
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
std::size_t size; // e.g., 8
int* input1; // e.g., [8, 7, 6, 5, 4, 3, 2, 1]
int* output; // empty array of 8 elements

std::size_t temporary_storage_size_bytes;
void* temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::adjacent_difference(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, size, binary_op
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform adjacent difference
rocprim::adjacent_difference(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, size, binary_op
);
// output: [8, 1, 1, 1, 1, 1, 1, 1]
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIt** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **input** – iterator to the input range
- **output** – iterator to the output range, must not have any overlap with input.
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)

- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) after successful scan, otherwise the HIP runtime error of type `hipError_t`

left, inplace

template<typename **Config** = *default_config*, typename **InputIt**, typename **BinaryFunction** = ::rocprim::minus<>>
hipError_t `rocprim::adjacent_difference_inplace`(void *const temporary_storage, std::size_t &storage_size, const *InputIt* values, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the left item in place.

Copies the first item to the output then performs calls the supplied operator with each pair of neighboring elements and writes its result to the location of the second element. Equivalent to the following code

```
for(std::size_t int i = size - 1; i > 0; --i)
{
    input[i] = op(input[i], input[i - 1]);
}
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the value range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **values** – iterator to the range values, will be overwritten with the results
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)
- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) after successful scan, otherwise the HIP runtime error of type `hipError_t`

left, aliased

template<typename **Config** = *default_config*, typename **InputIt**, typename **OutputIt**, typename **BinaryFunction** = ::rocprim::minus<>>
hipError_t `rocprim::adjacent_difference_inplace`(void *const temporary_storage, std::size_t &storage_size, const *InputIt* input, const *OutputIt* output, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the left item.

Note

This function has to perform an extra copy due to (potentially) writing its values in-place. If it is known that `input` and `output` don't overlap then adjacent_difference should be preferred as it avoids this extra copy.

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the value range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIt** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **input** – iterator to the range values
- **output** – iterator to the output range. Allowed to point to the same elements as `input`. Only complete overlap or no overlap at all is allowed between `input` and `output`. In other words writing to `output[i]` is only allowed to overwrite `input[i]`, any other element must not be changed.
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)

- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) on success, otherwise the HIP runtime error of type `hipError_t`

right

template<typename **Config** = *default_config*, typename **InputIt**, typename **OutputIt**, typename **BinaryFunction** = ::rocprim::minus<>>
hipError_t `rocprim::adjacent_difference_right`(void *const temporary_storage, std::size_t &storage_size, const *InputIt* input, const *OutputIt* output, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the right item.

Copies the last item to the output then performs calls the supplied operator with each pair of neighboring elements and writes its result to the location of the first element. Equivalent to the following code

```
output[size - 1] = input[size - 1];
for(std::size_t int i = 0; i < size - 1; ++i)
{
    output[i] = op(input[i], input[i + 1]);
}
```

Example

In this example a device-level adjacent_difference operation is performed on integer values.

```
#include <rocprim/rocprim.hpp> //or <rocprim/device/device_adjacent_difference.
↪hpp>

// custom binary function
auto binary_op =
    [] __device__ (int a, int b) -> int
    {
        return a - b;
    };

// Prepare input and output (declare pointers, allocate device memory etc.)
std::size_t size; // e.g., 8
int* input1; // e.g., [1, 2, 3, 4, 5, 6, 7, 8]
int* output; // empty array of 8 elements

std::size_t temporary_storage_size_bytes;
void* temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::adjacent_difference_right(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, size, binary_op
```

(continues on next page)

(continued from previous page)

```
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform adjacent difference
rocprim::adjacent_difference_right(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, output, size, binary_op
);
// output: [1, 1, 1, 1, 1, 1, 1, 8]
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIt** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **input** – iterator to the input range
- **output** – iterator to the output range, must not have any overlap with input.
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)
- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) after successful scan, otherwise the HIP runtime error of type `hipError_t`

right, inplace

template<typename **Config** = *default_config*, typename **InputIt**, typename **BinaryFunction** = ::rocprim::minus<>>

hipError_t rocprim::**adjacent_difference_right_inplace**(void *const temporary_storage, std::size_t &storage_size, const *InputIt* values, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the right item in place.

Copies the last item to the output then performs calls the supplied operator with each pair of neighboring elements and writes its result to the location of the first element. Equivalent to the following code

```
for(std::size_t int i = 0; i < size - 1; --i)
{
    input[i] = op(input[i], input[i + 1]);
}
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the value range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **values** – iterator to the range values, will be overwritten with the results
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)
- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) after successful scan, otherwise the HIP runtime error of type `hipError_t`

right, aliased

template<typename **Config** = *default_config*, typename **InputIt**, typename **OutputIt**, typename **BinaryFunction** = ::rocprim::minus<>>

hipError_t rocprim::**adjacent_difference_right_inplace**(void *const temporary_storage, std::size_t &storage_size, const *InputIt* input, const *OutputIt* output, const std::size_t size, const *BinaryFunction* op = *BinaryFunction*{}, const hipStream_t stream = 0, const bool debug_synchronous = false)

Parallel primitive for applying a binary operation across pairs of consecutive elements in device accessible memory. Writes the output to the position of the right item.

Note

This function has to perform an extra copy due to (potentially) writing its values in-place. If it is known that `input` and `output` don't overlap then adjacent_difference_right should be preferred as it avoids this extra copy.

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *adjacent_difference_config*.
- **InputIt** – [inferred] random-access iterator type of the value range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIt** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] binary operation function object that will be applied to consecutive items. The signature of the function should be equivalent to the following: `U f(const T1& a, const T2& b)`. The signature does not need to have `const &`, but function object must not modify the object passed to it

Parameters

- **temporary_storage** – pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the scan operation
- **storage_size** – reference to a size (in bytes) of `temporary_storage`
- **input** – iterator to the range values, will be overwritten with the results
- **output** – iterator to the output range. Allowed to point to the same elements as `input`. Only complete overlap or no overlap at all is allowed between `input` and `output`. In other words writing to `output[i]` is only allowed to overwrite `input[i]`, any other element must not be changed.
- **size** – number of items in the input
- **op** – [optional] the binary operation to apply
- **stream** – [optional] HIP stream object. Default is `0` (the default stream)
- **debug_synchronous** – [optional] If true, synchronization after every kernel launch is forced in order to check for errors and extra debugging info is printed to the standard output. Default value is `false`

Returns

`hipSuccess` (0) on success, otherwise the HIP runtime error of type `hipError_t`

2.3.15 Adjacent Find

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**>

struct **adjacent_find_config** : public rocprim::detail::adjacent_find_config_params

Configuration of device-level adjacent_find.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread.

Subclassed by rocprim::detail::default_adjacent_find_config< arch, input_type, enable >

adjacent_find

Warning

doxygenfunction: Unable to resolve function "rocprim::adjacent_find" with arguments (void*const, std::size_t&, InputIteratorType, OutputIteratorType, const std::size_t, const BinaryPred, const hipStream_t, const bool) in doxygen xml output for project "rocPRIM" from directory: /home/docs/checkouts/readthedocs.org/user_builds/advanced-micro-devices-rocprim/checkouts/docs-6.4.3/docs/doxygen/xml. Potential matches:

```
- template<typename Config = default_config, typename InputIterator, typename␣
↪OutputIterator, typename BinaryPred = ::rocprim::equal_to<typename std::iterator_
↪traits<InputIterator>::value_type>> hipError_t adjacent_find(void *const temporary_
↪storage, std::size_t &storage_size, InputIterator input, OutputIterator output,␣
↪const std::size_t size, BinaryPred op = BinaryPred{}, const hipStream_t stream = 0,␣
↪const bool debug_synchronous = false)
```

2.3.16 Binary Search

template<class **Config** = *default_config*, class **HaystackIterator**, class **NeedlesIterator**, class **OutputIterator**, class **CompareFunction** = ::rocprim::less<>>

inline hipError_t `rocprim::`**binary_search**(void *temporary_storage, size_t &storage_size, *HaystackIterator* haystack, *NeedlesIterator* needles, *OutputIterator* output, size_t haystack_size, size_t needles_size, *CompareFunction* compare_op = *CompareFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Parallel primitive for performing a binary search (on a sorted range) of a given input.

The `binary_search` function determines for each element of a given input if it's present in a given ordered range `haystack`. It uses the search function `detail::binary_search_op` which in turn uses a binary operator `compare_op` for comparing the input values with the haystack ones.

Overview

- When a null pointer is passed as `temporary_storage`, the required allocation size (in bytes) is written to `storage_size` and the function returns without performing the search operation.

Example

In this example a device-level binary search on a haystack of integer values is performed on an input array of integer values too.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.).
size_t          haystack_size;    // e.g. 10
int *           haystack;         // e.g. {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
size_t          needles_size;     // e.g. 8
int *           needles;          // e.g. {0, 2, 12, 4, 14, 6, 8, 10}
compare_op_type compare_op;       // e.g. compare_op_type = rocprim::less<int>
size_t *        output;           // empty array of needles_size elements

// Get required size of the temporary storage.
void * temporary_storage = nullptr;
size_t temporary_storage_bytes;
rocprim::binary_search<config>(temporary_storage,
                               temporary_storage_bytes,
                               haystack,
                               needles,
                               output,
                               haystack_size,
                               needles_size,
                               compare_op,
                               stream,
                               debug_synchronous);

// Allocate temporary storage.
hipMalloc(&temporary_storage, temporary_storage_bytes);

// Perform binary search.
rocprim::binary_search<config>(temporary_storage,
                               temporary_storage_bytes,
                               haystack,
                               needles,
                               output,
                               haystack_size,
                               needles_size,
                               compare_op,
                               stream,
                               debug_synchronous);

// output = {1, 1, 0, 1, 0, 1, 1, 0}
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or binary_search_config.
- **HaystackIterator** – - [inferred] Random-access iterator type of the search range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **NeedlesIterator** – - [inferred] Random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type. Elements of the type pointed by it must be comparable to elements of the type pointed by

HaystackIterator as either operand of compare_op.

- **OutputIterator** – - [inferred] Random-access iterator type of the output range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **CompareFunction** – - [inferred] Type of binary function that accepts two arguments of the types pointed by HaystackIterator and NeedlesIterator, and returns a value convertible to bool. Default type is rocprim::less<>.

Parameters

- **temporary_storage** – **[in]** - Pointer to a device-accessible temporary storage.
- **storage_size** – **[inout]** - Reference to the size (in bytes) of temporary_storage.
- **haystack** – **[in]** - Iterator to the first element in the search range. Elements of this range must be sorted.
- **needles** – **[in]** - Iterator to the first element in the range of values to search for on haystack.
- **output** – **[out]** - Iterator to the first element in the output range of boolean values.
- **haystack_size** – **[in]** - Number of elements in the search range haystack.
- **needles_size** – **[in]** - Number of elements in the input range needles.
- **compare_op** – **[in]** - Binary operation function object that is used to compare values. The signature of the function should be equivalent to the following: bool f(const T &a, const U &b);. It does not need to have const &, but the function object must not modify the objects passed to it. Default is CompareFunction().
- **stream** – **[in]** - [optional] HIP stream object. Default is 0 (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors.

Returns

hipSuccess (0) after a successful search; otherwise a HIP runtime error of type hipError_t.

2.3.17 Histogram

Configuring the kernel

template<class **HistogramConfig**, unsigned int **MaxGridSize** = 1024, unsigned int **SharedImplMaxBins** = 2048, unsigned int **SharedImplHistograms** = 3>
struct **histogram_config** : public rocprim::detail::histogram_config_params

Configuration of device-level histogram operation.

Template Parameters

- **HistogramConfig** – - configuration of histogram kernel. Must be *kernel_config*.
- **MaxGridSize** – - maximum number of blocks to launch.
- **SharedImplMaxBins** – - maximum total number of bins for all active channels for the shared memory histogram implementation (samples -> shared memory bins -> global memory bins), when exceeded the global memory implementation is used (samples -> global memory bins).
- **SharedImplHistograms** – - number of histograms in the shared memory to reduce bank conflicts for atomic operations with narrow sample distributions. Sweetspot for 9xx and 10xx is 3.

Subclassed by rocprim::detail::default_histogram_config< arch, value_type, channels, active_channels, enable >

histogram_even

template<class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t **rocprim::histogram_even**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int size, *Counter* *histogram, unsigned int levels, *Level* lower_level, *Level* upper_level, hipStream_t stream = 0, bool debug_synchronous = false)

Computes a histogram from a sequence of samples using equal-width bins.

- The number of histogram bins is (`levels` - 1).
- Bins are evenly-segmented and include the same width of sample values: (`upper_level` - `lower_level`) / (`levels` - 1).
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Example

In this example a device-level histogram of 5 bins is computed on an array of float samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int size;        // e.g., 8
float * samples;          // e.g., [-10.0, 0.3, 9.5, 8.1, 1.5, 1.9, 100.0, 5.1]
int * histogram;          // empty array of at least 5 elements
unsigned int levels;      // e.g., 6 (for 5 bins)
float lower_level;        // e.g., 0.0
float upper_level;        // e.g., 10.0

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::histogram_even(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, lower_level, upper_level
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histogram
rocprim::histogram_even(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, lower_level, upper_level
);
// histogram: [3, 0, 1, 0, 2]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **size** – **[in]** - number of elements in the samples range.
- **histogram** – **[out]** - pointer to the first element in the histogram range.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins.
- **lower_level** – **[in]** - lower sample value bound (inclusive) for the first histogram bin.
- **upper_level** – **[in]** - upper sample value bound (exclusive) for the last histogram bin.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::`**histogram_even**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int columns, unsigned int rows, size_t row_stride_bytes, *Counter* *histogram, unsigned int levels, *Level* lower_level, *Level* upper_level, hipStream_t stream = 0, bool debug_synchronous = false)

Computes a histogram from a two-dimensional region of samples using equal-width bins.

- The two-dimensional region of interest within `samples` can be specified using the `columns`, `rows` and `row_stride_bytes` parameters.
- The row stride must be a whole multiple of the sample data type size, i.e., `(row_stride_bytes % sizeof(std::iterator_traits<SampleIterator>::value_type)) == 0`.
- The number of histogram bins is (`levels` - 1).
- Bins are evenly-segmented and include the same width of sample values: (`upper_level` - `lower_level`) / (`levels` - 1).

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Example

In this example a device-level histogram of 5 bins is computed on an array of float samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int columns;     // e.g., 4
unsigned int rows;        // e.g., 2
size_t row_stride_bytes;  // e.g., 6 * sizeof(float)
float * samples;          // e.g., [-10.0, 0.3, 9.5, 8.1, -, -, 1.5, 1.9, 100.0,
↪ 5.1, -, -]
int * histogram;          // empty array of at least 5 elements
unsigned int levels;      // e.g., 6 (for 5 bins)
float lower_level;        // e.g., 0.0
float upper_level;        // e.g., 10.0

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::histogram_even(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, lower_level, upper_level
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histogram
rocprim::histogram_even(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, lower_level, upper_level
);
// histogram: [3, 0, 1, 0, 2]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.

- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **columns** – **[in]** - number of elements in each row of the region.
- **rows** – **[in]** - number of rows of the region.
- **row_stride_bytes** – **[in]** - number of bytes between starts of consecutive rows of the region.
- **histogram** – **[out]** - pointer to the first element in the histogram range.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins.
- **lower_level** – **[in]** - lower sample value bound (inclusive) for the first histogram bin.
- **upper_level** – **[in]** - upper sample value bound (exclusive) for the last histogram bin.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

multi_histogram_even

template<unsigned int **Channels**, unsigned int **ActiveChannels**, class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::`**multi_histogram_even**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int size, *Counter* *histogram[*ActiveChannels*], unsigned int levels[*ActiveChannels*], *Level* lower_level[*ActiveChannels*], *Level* upper_level[*ActiveChannels*], hipStream_t stream = 0, bool debug_synchronous = false)

Computes histograms from a sequence of multi-channel samples using equal-width bins.

- The input is a sequence of *pixel* structures, where each pixel comprises a record of `Channels` consecutive data samples (e.g., `Channels` = 4 for *RGBA* samples).
- The first `ActiveChannels` channels of total `Channels` channels will be used for computing histograms (e.g., `ActiveChannels` = 3 for computing histograms of only *RGB* from *RGBA* samples).
- For channel the number of histogram bins is (`levels`[i] - 1).
- For channel bins are evenly-segmented and include the same width of sample values: (`upper_level`[i] - `lower_level`[i]) / (`levels`[i] - 1).
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Notes

- Currently the `Channels` template parameter has no strict restriction on its value. However, internally a vector type of elements of type `SampleIterator` and length `Channels` is used to represent the input items, so the amount of local memory available will limit the range of possible values for this template parameter.

- `ActiveChannels` must be less or equal than `Channels`.

Example

In this example histograms for 3 channels (RGB) are computed on an array of 8-bit RGBA samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int size;        // e.g., 8
unsigned char * samples;  // e.g., [(3, 1, 5, 255), (3, 1, 5, 255), (4, 2, 6,␣
↪127), (3, 2, 6, 127),
                          //        (0, 0, 0, 100), (0, 1, 0, 100), (0, 0, 1,␣
↪255), (0, 1, 1, 255)]
int * histogram[3];       // 3 empty arrays of at least 256 elements each
unsigned int levels[3];   // e.g., [257, 257, 257] (for 256 bins)
int lower_level[3];       // e.g., [0, 0, 0]
int upper_level[3];       // e.g., [256, 256, 256]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::multi_histogram_even<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, lower_level, upper_level
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histograms
rocprim::multi_histogram_even<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, lower_level, upper_level
);
// histogram: [[4, 0, 0, 3, 1, 0, 0, ..., 0],
//             [2, 4, 2, 0, 0, 0, 0, ..., 0],
//             [2, 2, 0, 0, 0, 2, 2, ..., 0]]
```

Template Parameters

- **Channels** – - number of channels interleaved in the input samples.
- **ActiveChannels** – - number of channels being used for computing histograms.
- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **size** – **[in]** - number of pixels in the samples range.
- **histogram** – **[out]** - pointers to the first element in the histogram range, one for each active channel.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins in each active channel.
- **lower_level** – **[in]** - lower sample value bound (inclusive) for the first histogram bin in each active channel.
- **upper_level** – **[in]** - upper sample value bound (exclusive) for the last histogram bin in each active channel.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

template<unsigned int **Channels**, unsigned int **ActiveChannels**, class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::multi_histogram_even`(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int columns, unsigned int rows, size_t row_stride_bytes, *Counter* *histogram[*ActiveChannels*], unsigned int levels[*ActiveChannels*], *Level* lower_level[*ActiveChannels*], *Level* upper_level[*ActiveChannels*], hipStream_t stream = 0, bool debug_synchronous = false)

Computes histograms from a two-dimensional region of multi-channel samples using equal-width bins.

- The two-dimensional region of interest within `samples` can be specified using the `columns`, `rows` and `row_stride_bytes` parameters.
- The row stride must be a whole multiple of the sample data type size, i.e., `(row_stride_bytes % sizeof(std::iterator_traits<SampleIterator>::value_type)) == 0`.
- The input is a sequence of *pixel* structures, where each pixel comprises a record of `Channels` consecutive data samples (e.g., `Channels` = 4 for *RGBA* samples).
- The first `ActiveChannels` channels of total `Channels` channels will be used for computing histograms (e.g., `ActiveChannels` = 3 for computing histograms of only *RGB* from *RGBA* samples).
- For channel the number of histogram bins is (`levels`[i] - 1).
- For channel bins are evenly-segmented and include the same width of sample values: (`upper_level`[i] - `lower_level`[i]) / (`levels`[i] - 1).

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Notes

- Currently the `Channels` template parameter has no strict restriction on its value. However, internally a vector type of elements of type `SampleIterator` and length `Channels` is used to represent the input items, so the amount of local memory available will limit the range of possible values for this template parameter.
- `ActiveChannels` must be less or equal than `Channels`.

Example

In this example histograms for 3 channels (RGB) are computed on an array of 8-bit RGBA samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int columns;     // e.g., 4
unsigned int rows;        // e.g., 2
size_t row_stride_bytes;  // e.g., 5 * sizeof(unsigned char)
unsigned char * samples;  // e.g., [(3, 1, 5, 255), (3, 1, 5, 255), (4, 2, 6,␣
↪127), (3, 2, 6, 127), (-, -, -, -),
                          //        (0, 0, 0, 100), (0, 1, 0, 100), (0, 0, 1,␣
↪255), (0, 1, 1, 255), (-, -, -, -)]
int * histogram[3];       // 3 empty arrays of at least 256 elements each
unsigned int levels[3];   // e.g., [257, 257, 257] (for 256 bins)
int lower_level[3];       // e.g., [0, 0, 0]
int upper_level[3];       // e.g., [256, 256, 256]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::multi_histogram_even<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, lower_level, upper_level
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histograms
rocprim::multi_histogram_even<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, lower_level, upper_level
);
// histogram: [[4, 0, 0, 3, 1, 0, 0, ..., 0],
//             [2, 4, 2, 0, 0, 0, 0, ..., 0],
//             [2, 2, 0, 0, 0, 2, 2, ..., 0]]
```

Template Parameters

- **Channels** – - number of channels interleaved in the input samples.

- **ActiveChannels** – - number of channels being used for computing histograms.
- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **columns** – **[in]** - number of elements in each row of the region.
- **rows** – **[in]** - number of rows of the region.
- **row_stride_bytes** – **[in]** - number of bytes between starts of consecutive rows of the region.
- **histogram** – **[out]** - pointers to the first element in the histogram range, one for each active channel.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins in each active channel.
- **lower_level** – **[in]** - lower sample value bound (inclusive) for the first histogram bin in each active channel.
- **upper_level** – **[in]** - upper sample value bound (exclusive) for the last histogram bin in each active channel.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

histogram_range

template<class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::`**histogram_range**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int size, *Counter* *histogram, unsigned int levels, *Level* *level_values, hipStream_t stream = 0, bool debug_synchronous = false)

Computes a histogram from a sequence of samples using the specified bin boundary levels.

- The number of histogram bins is (`levels` - 1).
- The range for bin is [`level_values[j]`, `level_values[j+1]`).

- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Example

In this example a device-level histogram of 5 bins is computed on an array of float samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int size;        // e.g., 8
float * samples;          // e.g., [-10.0, 0.3, 9.5, 8.1, 1.5, 1.9, 100.0, 5.1]
int * histogram;          // empty array of at least 5 elements
unsigned int levels;      // e.g., 6 (for 5 bins)
float * level_values;     // e.g., [0.0, 1.0, 5.0, 10.0, 20.0, 50.0]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::histogram_range(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, level_values
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histogram
rocprim::histogram_range(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, level_values
);
// histogram: [1, 2, 3, 0, 0]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **size** – **[in]** - number of elements in the samples range.

- **histogram** – **[out]** - pointer to the first element in the histogram range.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins.
- **level_values** – **[in]** - pointer to the array of bin boundaries.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

template<class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::`**histogram_range**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int columns, unsigned int rows, size_t row_stride_bytes, *Counter* *histogram, unsigned int levels, *Level* *level_values, hipStream_t stream = 0, bool debug_synchronous = false)

Computes a histogram from a two-dimensional region of samples using the specified bin boundary levels.

- The two-dimensional region of interest within `samples` can be specified using the `columns`, `rows` and `row_stride_bytes` parameters.
- The row stride must be a whole multiple of the sample data type size, i.e., `(row_stride_bytes % sizeof(std::iterator_traits<SampleIterator>::value_type)) == 0`.
- The number of histogram bins is (`levels` - 1).
- The range for bin is [`level_values[j]`, `level_values[j+1]`).
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Example

In this example a device-level histogram of 5 bins is computed on an array of float samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int columns;     // e.g., 4
unsigned int rows;        // e.g., 2
size_t row_stride_bytes;  // e.g., 6 * sizeof(float)
float * samples;          // e.g., [-10.0, 0.3, 9.5, 8.1, 1.5, 1.9, 100.0, 5.1]
int * histogram;          // empty array of at least 5 elements
unsigned int levels;      // e.g., 6 (for 5 bins)
float level_values;       // e.g., [0.0, 1.0, 5.0, 10.0, 20.0, 50.0]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::histogram_range(
    temporary_storage_ptr, temporary_storage_size_bytes,
```

(continues on next page)

(continued from previous page)

```
    samples, columns, rows, row_stride_bytes,
    histogram, levels, level_values
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histogram
rocprim::histogram_range(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, level_values
);
// histogram: [1, 2, 3, 0, 0]
```

Template Parameters

- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **columns** – **[in]** - number of elements in each row of the region.
- **rows** – **[in]** - number of rows of the region.
- **row_stride_bytes** – **[in]** - number of bytes between starts of consecutive rows of the region.
- **histogram** – **[out]** - pointer to the first element in the histogram range.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins.
- **level_values** – **[in]** - pointer to the array of bin boundaries.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

multi_histogram_range

template<unsigned int **Channels**, unsigned int **ActiveChannels**, class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::`**multi_histogram_range**(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int size, *Counter* *histogram[*ActiveChannels*], unsigned int levels[*ActiveChannels*], *Level* *level_values[*ActiveChannels*], hipStream_t stream = 0, bool debug_synchronous = false)

Computes histograms from a sequence of multi-channel samples using the specified bin boundary levels.

- The input is a sequence of *pixel* structures, where each pixel comprises a record of `Channels` consecutive data samples (e.g., `Channels` = 4 for *RGBA* samples).
- The first `ActiveChannels` channels of total `Channels` channels will be used for computing histograms (e.g., `ActiveChannels` = 3 for computing histograms of only *RGB* from *RGBA* samples).
- For channel the number of histogram bins is (`levels`[i] - 1).
- For channel the range for bin is [`level_values[i][j]`, `level_values[i][j+1]`).
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Notes

- Currently the `Channels` template parameter has no strict restriction on its value. However, internally a vector type of elements of type `SampleIterator` and length `Channels` is used to represent the input items, so the amount of local memory available will limit the range of possible values for this template parameter.
- `ActiveChannels` must be less or equal than `Channels`.

Example

In this example histograms for 3 channels (RGB) are computed on an array of 8-bit RGBA samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int size;        // e.g., 8
unsigned char * samples;  // e.g., [(0, 0, 80, 255), (120, 0, 80, 255), (123, 0,
↪ 82, 127), (10, 1, 83, 127),
                          //         (51, 1, 8, 100), (52, 1, 8, 100), (53, 0,␣
↪81, 255), (54, 50, 81, 255)]
int * histogram[3];       // 3 empty arrays of at least 256 elements each
unsigned int levels[3];   // e.g., [4, 4, 3]
int * level_values[3];    // e.g., [[0, 50, 100, 200], [0, 20, 40, 60], [0, 10,␣
↪100]]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::multi_histogram_range<4, 3>(
```

(continues on next page)

(continued from previous page)

```
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, level_values
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histograms
rocprim::multi_histogram_range<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, size,
    histogram, levels, level_values
);
// histogram: [[2, 4, 2], [7, 0, 1], [2, 6]]
```

Template Parameters

- **Channels** – - number of channels interleaved in the input samples.
- **ActiveChannels** – - number of channels being used for computing histograms.
- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **size** – **[in]** - number of pixels in the samples range.
- **histogram** – **[out]** - pointers to the first element in the histogram range, one for each active channel.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins in each active channel.
- **level_values** – **[in]** - pointer to the array of bin boundaries for each active channel.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

template<unsigned int **Channels**, unsigned int **ActiveChannels**, class **Config** = *default_config*, class **SampleIterator**, class **Counter**, class **Level**>
inline hipError_t `rocprim::multi_histogram_range`(void *temporary_storage, size_t &storage_size, *SampleIterator* samples, unsigned int columns, unsigned int rows, size_t row_stride_bytes, *Counter* *histogram[*ActiveChannels*], unsigned int levels[*ActiveChannels*], *Level* *level_values[*ActiveChannels*], hipStream_t stream = 0, bool debug_synchronous = false)

Computes histograms from a two-dimensional region of multi-channel samples using the specified bin boundary levels.

- The two-dimensional region of interest within `samples` can be specified using the `columns`, `rows` and `row_stride_bytes` parameters.
- The row stride must be a whole multiple of the sample data type size, i.e., `(row_stride_bytes % sizeof(std::iterator_traits<SampleIterator>::value_type)) == 0`.
- The input is a sequence of *pixel* structures, where each pixel comprises a record of `Channels` consecutive data samples (e.g., `Channels` = 4 for *RGBA* samples).
- The first `ActiveChannels` channels of total `Channels` channels will be used for computing histograms (e.g., `ActiveChannels` = 3 for computing histograms of only *RGB* from *RGBA* samples).
- For channel the number of histogram bins is (`levels`[i] - 1).
- For channel the range for bin is [`level_values[i][j]`, `level_values[i][j+1]`).
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` in a null pointer.

Notes

- Currently the `Channels` template parameter has no strict restriction on its value. However, internally a vector type of elements of type `SampleIterator` and length `Channels` is used to represent the input items, so the amount of local memory available will limit the range of possible values for this template parameter.
- `ActiveChannels` must be less or equal than `Channels`.

Example

In this example histograms for 3 channels (RGB) are computed on an array of 8-bit RGBA samples.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
unsigned int columns;     // e.g., 4
unsigned int rows;        // e.g., 2
size_t row_stride_bytes;  // e.g., 5 * sizeof(unsigned char)
unsigned char * samples;  // e.g., [(0, 0, 80, 0), (120, 0, 80, 0), (123, 0, 82,
↪ 0), (10, 1, 83, 0), (-, -, -, -),
                          //        (51, 1, 8, 0), (52, 1, 8, 0), (53, 0, 81,␣
↪0), (54, 50, 81, 0), (-, -, -, -)]
int * histogram[3];       // 3 empty arrays
unsigned int levels[3];   // e.g., [4, 4, 3]
```

(continues on next page)

(continued from previous page)

```
int * level_values[3];    // e.g., [[0, 50, 100, 200], [0, 20, 40, 60], [0, 10,␣
↪100]]

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::multi_histogram_range<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, level_values
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// compute histograms
rocprim::multi_histogram_range<4, 3>(
    temporary_storage_ptr, temporary_storage_size_bytes,
    samples, columns, rows, row_stride_bytes,
    histogram, levels, level_values
);
// histogram: [[2, 4, 2], [7, 0, 1], [2, 6]]
```

Template Parameters

- **Channels** – - number of channels interleaved in the input samples.
- **ActiveChannels** – - number of channels being used for computing histograms.
- **Config** – - [optional] Configuration of the primitive, must be *default_config* or *histogram_config*.
- **SampleIterator** – - random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **Counter** – - integer type for histogram bin counters.
- **Level** – - type of histogram boundaries (levels)

Parameters

- **temporary_storage** – **[in]** - pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the reduction operation.
- **storage_size** – **[inout]** - reference to a size (in bytes) of `temporary_storage`.
- **samples** – **[in]** - iterator to the first element in the range of input samples.
- **columns** – **[in]** - number of elements in each row of the region.
- **rows** – **[in]** - number of rows of the region.
- **row_stride_bytes** – **[in]** - number of bytes between starts of consecutive rows of the region.
- **histogram** – **[out]** - pointers to the first element in the histogram range, one for each active channel.
- **levels** – **[in]** - number of boundaries (levels) for histogram bins in each active channel.

- **level_values** – **[in]** - pointer to the array of bin boundaries for each active channel.
- **stream** – **[in]** - [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful histogram operation; otherwise a HIP runtime error of type `hipError_t`.

2.3.18 DeviceCopy

Configuring the kernel

template<unsigned int **NonBlevBlockSize** = 256, unsigned int **NonBlevBuffersPerThreaed** = 2, unsigned int **TlevBytesPerThread** = 8, unsigned int **BlevBlockSize** = 128, unsigned int **BlevBytesPerThread** = 32, unsigned int **WlevSizeThreshold** = 128, unsigned int **BlevSizeThreshold** = 1024>
struct **batch_copy_config** : public rocprim::*batch_memcpy_config*<256, 2, 8, 128, 32, 128, 1024>

Template Parameters

- **NonBlevBlockSize** – - number of threads per block for thread- and warp-level copy.
- **NonBlevBuffersPerThreaed** – - number of buffers processed per thread.
- **TlevBytesPerThread** – - number of bytes per thread for thread-level copy.
- **BlevBlockSize** – - number of thread per block for block-level copy.
- **BlevBytesPerThread** – - number of bytes per thread for block-level copy.
- **WlevSizeThreshold** – - minimum size to use warp-level copy instead of thread-level.
- **BlevSizeThreshold** – - minimum size to use block-level copy instead of warp-level.

batch_copy

template<class **Config_** = *default_config*, class **InputBufferItType**, class **OutputBufferItType**, class **BufferSizeItType**>
static inline hipError_t `rocprim::`**batch_copy**(void *temporary_storage, size_t &storage_size, *InputBufferItType* sources, *OutputBufferItType* destinations, *BufferSizeItType* sizes, uint32_t num_copies, hipStream_t stream = hipStreamDefault, bool debug_synchronous = false)

Copy `sizes[i]` elements from `sources[i]` to `destinations[i]` for all `i` in the range [0, `num_copies`].

Performs multiple device to device copies as a single batched operation. Roughly equivalent to

```
for (auto i = 0; i < num_copies; ++i) {
    auto* src = sources[i];
    auto* dst = destinations[i];
    auto size = sizes[i];
    for (auto j = 0; j < size; ++j)
    {
        dst[j] = src[j];
    }
}
```

except executed on the device in parallel. Note that sources and destinations do not have to be part of the same array. I.e. you can copy from both array A and B to array C and D with a single call to this function. Source ranges are allowed to overlap, however, destinations overlapping with either other destinations or with sources is not allowed, and will result in undefined behaviour.

Example

In this example multiple sections of data are copied from `a` to `b` .

```
#include <rocprim/rocprim.hpp

// Device allocated data:
int* a;             // e.g, [9, 1, 2, 3, 4, 5, 6, 7, 8]
int* b;             // e.g, [0, 0, 0, 0, 0, 0, 0, 0, 0]

// Batch memcpy parameters:
int   num_copies;   // Number of buffers to copy.
                    // e.g, 4.
int** sources;      // Pointer to source pointers.
                    // e.g, [&a[0], &a[4] &a[7]]
int** destinations; // Pointer to destination pointers.
                    // e.g, [&b[5], &b[2] &b[0]]
int*  sizes;        // Size of buffers to copy.
                    // e.g., [3, 2, 2]

// Calculate the required temporary storage.
size_t temporary_storage_size_bytes;
void* temporary_storage_ptr = nullptr;
rocprim::batch_copy(
    temporary_storage_ptr,
    temporary_storage_size_bytes,
    sources,
    destinations,
    sizes,
    num_buffers);

// Allocate temporary storage.
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// Copy buffers.
rocprim::batch_copy(
    temporary_storage_ptr,
    temporary_storage_size_bytes,
    sources,
    destinations,
    sizes,
    num_copies);

// b is now: [7, 8, 4, 5, 0, 9, 1, 2, 0]
//   3rd copy ^--^  ^--^     ^--^--^ 1st copy
//                2nd copy
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *batch_copy_config*.
- **InputBufferItType** – type of iterator to source pointers.
- **OutputBufferItType** – type of iterator to desetination pointers.
- **BufferSizeItType** – type of iterator to sizes.

Parameters

- **temporary_storage** – **[in]** pointer to device-accessible temporary storage. When a null pointer is passed, the required allocation size in bytes is written to `storage_size` and the function returns without performing the copy.
- **storage_size** – **[inout]** reference to the size in bytes of `temporary_storage`.
- **sources** – **[in]** iterator of source pointers.
- **destinations** – **[in]** iterator of destination pointers.
- **sizes** – **[in]** iterator of range sizes to copy.
- **num_copies** – **[in]** number of ranges to copy
- **stream** – **[in]** [optional] HIP stream object to enqueue the copy on. Default is `hipStreamDefault`.
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

2.3.19 Memcpy

Configuring the kernel

template<unsigned int **NonBlevBlockSize** = 256, unsigned int **NonBlevBuffersPerThreaed** = 2, unsigned int **TlevBytesPerThread** = 8, unsigned int **BlevBlockSize** = 128, unsigned int **BlevBytesPerThread** = 32, unsigned int **WlevSizeThreshold** = 128, unsigned int **BlevSizeThreshold** = 1024>
struct **batch_memcpy_config**

Template Parameters

- **NonBlevBlockSize** – - number of threads per block for thread- and warp-level copy.
- **NonBlevBuffersPerThreaed** – - number of buffers processed per thread.
- **TlevBytesPerThread** – - number of bytes per thread for thread-level copy.
- **BlevBlockSize** – - number of thread per block for block-level copy.
- **BlevBytesPerThread** – - number of bytes per thread for block-level copy.
- **WlevSizeThreshold** – - minimum size to use warp-level copy instead of thread-level.
- **BlevSizeThreshold** – - minimum size to use block-level copy instead of warp-level.

batch_memcpy

template<class **Config_** = *default_config*, class **InputBufferItType**, class **OutputBufferItType**, class **BufferSizeItType**>
static inline hipError_t `rocprim::batch_memcpy`(void *temporary_storage, size_t &storage_size, *InputBufferItType* sources, *OutputBufferItType* destinations, *BufferSizeItType* sizes, uint32_t num_copies, hipStream_t stream = hipStreamDefault, bool debug_synchronous = false)

Copy `sizes[i]` bytes from `sources[i]` to `destinations[i]` for all `i` in the range [0, `num_copies`].

Performs multiple device to device memory copies as a single batched operation. Roughly equivalent to

```
for (auto i = 0; i < num_copies; ++i) {
    char* src = sources[i];
    char* dst = destinations[i];
    auto size = sizes[i];
    hipMemcpyAsync(dst, src, size, hipMemcpyDeviceToDevice, stream);
}
```

except executed on the device in parallel. Note that sources and destinations do not have to be part of the same array. I.e. you can copy from both array A and B to array C and D with a single call to this function. Source ranges are allowed to overlap, however, destinations overlapping with either other destinations or with sources is not allowed, and will result in undefined behaviour.

Example

In this example multiple sections of data are copied from `a` to `b` .

```
#include <rocprim/rocprim.hpp

// Device allocated data:
int* a;             // e.g, [9, 1, 2, 3, 4, 5, 6, 7, 8]
int* b;             // e.g, [0, 0, 0, 0, 0, 0, 0, 0, 0]

// Batch memcpy parameters:
int   num_copies;   // Number of buffers to copy.
                    // e.g, 4.
int** sources;      // Pointer to source pointers.
                    // e.g, [&a[0], &a[4] &a[7]]
int** destinations; // Pointer to destination pointers.
                    // e.g, [&b[5], &b[2] &b[0]]
int*  sizes;        // Size of buffers to copy.
                    // e.g., [3 * sizeof(int), 2 * sizeof(int), 2 *␣
↪sizeof(int)]

// Calculate the required temporary storage.
size_t temporary_storage_size_bytes;
void* temporary_storage_ptr = nullptr;
rocprim::batch_memcpy(
    temporary_storage_ptr,
    temporary_storage_size_bytes,
    sources,
    destinations,
    sizes,
    num_buffers);

// Allocate temporary storage.
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);
```

(continues on next page)

(continued from previous page)

```
// Copy buffers.
rocprim::batch_memcpy(
    temporary_storage_ptr,
    temporary_storage_size_bytes,
    sources,
    destinations,
    sizes,
    num_copies);

// b is now: [7, 8, 4, 5, 0, 9, 1, 2, 0]
//   3rd copy ^--^  ^--^     ^--^--^ 1st copy
//               2nd copy
```

Template Parameters

- **Config** – [optional] Configuration of the primitive, must be *default_config* or *batch_memcpy_config*.
- **InputBufferItType** – type of iterator to source pointers.
- **OutputBufferItType** – type of iterator to desetination pointers.
- **BufferSizeItType** – type of iterator to sizes.

Parameters

- **temporary_storage** – **[in]** pointer to device-accessible temporary storage. When a null pointer is passed, the required allocation size in bytes is written to `storage_size` and the function returns without performing the copy.
- **storage_size** – **[inout]** reference to the size in bytes of `temporary_storage`.
- **sources** – **[in]** iterator of source pointers.
- **destinations** – **[in]** iterator of destination pointers.
- **sizes** – **[in]** iterator of range sizes to copy.
- **num_copies** – **[in]** number of ranges to copy
- **stream** – **[in]** [optional] HIP stream object to enqueue the copy on. Default is `hipStreamDefault`.
- **debug_synchronous** – **[in]** - [optional] If true, synchronization after every kernel launch is forced in order to check for errors. The default value is `false`.

2.3.20 Find first of

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**>

struct **find_first_of_config** : public rocprim::detail::find_first_of_config_params

Configuration of device-level find_first_of.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread.

Subclassed by rocprim::detail::default_find_first_of_config< arch, value_type, enable >

find_first_of

template<class **Config** = *default_config*, class **InputIterator1**, class **InputIterator2**, class **OutputIterator**, class **BinaryFunction** = ::rocprim::equal_to<typename std::iterator_traits<*InputIterator1*>::value_type>>
inline hipError_t `rocprim::`**`find_first_of`**(void *temporary_storage, size_t &storage_size, *InputIterator1* input, *InputIterator2* keys, *OutputIterator* output, size_t size, size_t keys_size, *BinaryFunction* compare_function = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Searches the range [input, input + size) for any of the elements in the range [keys, keys + keys_size).

Overview

- The contents of the inputs are not altered by the function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is
- Accepts custom compare_function.

Example

In this example a device-level find_first_of is performed where inputs and keys are represented by an array of unsigned integers.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;                // e.g., 8
size_t keys_size;           // e.g., 2
unsigned int* input;        // e.g., [ 6, 3, 5, 4, 1, 8, 2, 7 ]
unsigned int* keys;         // e.g., [ 10, 5 ]
unsigned int* keys_output;  // 1 element

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::find_first_of(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, keys, output, size, keys_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform find_first_of
rocprim::find_first_of(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, keys, output, size, keys_size
);
// output: [ 2 ]
```

Template Parameters

- **Config** – [optional] configuration of the primitive. It has to be *find_first_of_config*.

- **InputIterator1** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **InputIterator2** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – [inferred] random-access iterator type of the output range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **CompareFunction** – [inferred] Type of binary function that accepts two arguments of the type `InputIterator1` and returns a value convertible to bool. Default type is `rocprim::equal_to<>`.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the search.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** iterator to the range of elements to examine.
- **keys** – **[in]** iterator to the range of elements to search for.
- **output** – **[out]** iterator to the output range. `output` should be able to be written for 1 element. `*output` constains the position of the first element in the range [input, input + size) that is equal to an element from the range [keys, keys + keys_size).
- **size** – **[in]** number of elements to examine.
- **keys_size** – **[in]** number of elements to search for.
- **compare_function** – **[in]** binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful search; otherwise a HIP runtime error of type `hipError_t`.

2.3.21 Find end

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, unsigned int **MaxSharedKeyBytes**>

struct **search_config** : public rocprim::detail::search_config_params

Configuration of device-level find_end.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread.
- **MaxSharedKeyBytes** – maximum number of bytes for which a shared key is used.

find_end

template<class **Config** = *default_config*, class **InputIterator1**, class **InputIterator2**, class **OutputIterator**, class **BinaryFunction** = rocprim::equal_to<typename std::iterator_traits<*InputIterator1*>::value_type>>
inline hipError_t **rocprim::find_end**(void *temporary_storage, size_t &storage_size, *InputIterator1* input, *InputIterator2* keys, *OutputIterator* output, size_t size, size_t keys_size, *BinaryFunction* compare_function = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Searches for the last occurrence of the sequence.

Searches the input for the last occurence of a sequence, according to a particular comparison function. If found, the index of the first item of the found sequence in the input is returned. Otherwise, returns the size of the input.

Overview

- The contents of the inputs are not altered by the function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Accepts custom compare_functions for find_end across the device.
- Streams in graph capture mode are supported

Example

In this example a device-level find_end is performed where input values are represented by an array of unsigned integers and the key is also an array of unsigned integers.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;           // e.g., 10
size_t key_size;       // e.g., 3
unsigned int * input;  // e.g., [ 6, 3, 5, 4, 1, 8, 2, 5, 4, 1 ]
unsigned int * key;    // e.g., [ 5, 4, 1 ]
unsigned int * output; // e.g., empty array of size 1

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::find_end(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, key, output, size, key_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform find_end
rocprim::find_end(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, key, output, size, key_size
);
// output:   [ 7 ]
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *search_config*.
- **InputIterator1** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **InputIterator2** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] Type of binary function that accepts two arguments of the type `InputIterator1` and returns a value convertible to bool. Default type is `rocprim::less<>`.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the find_end.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** iterator to the input range.
- **keys** – **[in]** iterator to the key range.
- **output** – **[out]** iterator to the output range. The output is one element.
- **size** – **[in]** number of elements in the input range.
- **keys_size** – **[in]** number of elements in the key range.
- **compare_function** – **[in]** binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The comparator must meet the C++ named requirement BinaryPredicate. The default value is `BinaryFunction()`.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful search; otherwise a HIP runtime error of type `hipError_t`.

2.3.22 Search

Configuring the kernel

template<unsigned int **BlockSize**, unsigned int **ItemsPerThread**, unsigned int **MaxSharedKeyBytes**>

struct **search_config** : public rocprim::detail::search_config_params

Configuration of device-level find_end.

Template Parameters

- **BlockSize** – number of threads in a block.
- **ItemsPerThread** – number of items processed by each thread.

- **MaxSharedKeyBytes** – maximum number of bytes for which a shared key is used.

search

template<class **Config** = *default_config*, class **InputIterator1**, class **InputIterator2**, class **OutputIterator**, class **BinaryFunction** = rocprim::equal_to<typename std::iterator_traits<*InputIterator1*>::value_type>>
inline hipError_t **rocprim::search**(void *temporary_storage, size_t &storage_size, *InputIterator1* input, *InputIterator2* keys, *OutputIterator* output, size_t size, size_t keys_size, *BinaryFunction* compare_function = *BinaryFunction*(), hipStream_t stream = 0, bool debug_synchronous = false)

Searches for the first occurrence of the sequence.

Searches the input for the first occurence of a sequence, according to a particular comparison function. If found, the index of the first item of the found sequence in the input is returned. Otherwise, returns the size of the input.

Overview

- The contents of the inputs are not altered by the function.
- Returns the required size of `temporary_storage` in `storage_size` if `temporary_storage` is a null pointer.
- Accepts custom compare_functions for search across the device.
- Streams in graph capture mode are supported

Example

In this example a device-level search is performed where input values are represented by an array of unsigned integers and the key is also an array of unsigned integers.

```
#include <rocprim/rocprim.hpp>

// Prepare input and output (declare pointers, allocate device memory etc.)
size_t size;           // e.g., 10
size_t key_size;       // e.g., 3
unsigned int * input;  // e.g., [ 6, 3, 5, 4, 1, 8, 2, 5, 4, 1 ]
unsigned int * key;    // e.g., [ 5, 4, 1 ]
unsigned int * output; // e.g., empty array of size 1

size_t temporary_storage_size_bytes;
void * temporary_storage_ptr = nullptr;
// Get required size of the temporary storage
rocprim::search(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, key, output, size, key_size
);

// allocate temporary storage
hipMalloc(&temporary_storage_ptr, temporary_storage_size_bytes);

// perform search
rocprim::search(
    temporary_storage_ptr, temporary_storage_size_bytes,
    input, key, output, size, key_size
```

(continues on next page)

(continued from previous page)

```
);
// output:   [ 2 ]
```

Template Parameters

- **Config** – [optional] configuration of the primitive, must be *default_config* or *search_config*.
- **InputIterator1** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **InputIterator2** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ InputIterator concept. It can be a simple pointer type.
- **OutputIterator** – [inferred] random-access iterator type of the input range. Must meet the requirements of a C++ OutputIterator concept. It can be a simple pointer type.
- **BinaryFunction** – [inferred] Type of binary function that accepts two arguments of the type `InputIterator1` and returns a value convertible to bool. Default type is `rocprim::less<>`.

Parameters

- **temporary_storage** – **[in]** pointer to a device-accessible temporary storage. When a null pointer is passed, the required allocation size (in bytes) is written to `storage_size` and function returns without performing the search.
- **storage_size** – **[inout]** reference to a size (in bytes) of `temporary_storage`.
- **input** – **[in]** iterator to the input range.
- **keys** – **[in]** iterator to the key range.
- **output** – **[out]** iterator to the output range. The output is one element.
- **size** – **[in]** number of elements in the input range.
- **keys_size** – **[in]** number of elements in the key range.
- **compare_function** – **[in]** binary operation function object that will be used for comparison. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The comparator must meet the C++ named requirement BinaryPredicate. The default value is `BinaryFunction()`.
- **stream** – **[in]** [optional] HIP stream object. Default is `0` (default stream).
- **debug_synchronous** – **[in]** [optional] If true, synchronization after every kernel launch is forced in order to check for errors. Default value is `false`.

Returns

`hipSuccess` (`0`) after successful search; otherwise a HIP runtime error of type `hipError_t`.

2.4 Block-Wide Operations

- *Operation classes*
  - *Load*
  - *Store*
  - *Adjacent difference*

  - *Discontinuity*
  - *Scan*
  - *Reduce*
  - *Shuffle*
  - *Exchange*
  - *Sort*
  - *Histogram*
- *Data movement functions*

2.4.1 Operation classes

- *Load*
- *Store*
- *Adjacent difference*
- *Discontinuity*
- *Scan*
- *Reduce*
- *Shuffle*
- *Exchange*
- *Sort*
- *Histogram*

Load

Class

template<class **T**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread**, *block_load_method* **Method** = *block_load_method*::*block_load_direct*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_load**

The `block_load` class is a block level parallel primitive which provides methods for loading data from continuous memory into a blocked arrangement of items across the thread block.

Overview

- The `block_load` class has a number of different methods to load data:
  - `block_load_method::block_load_direct`
  - `block_load_method::block_load_striped`
  - `block_load_method::block_load_vectorize`
  - `block_load_method::block_load_transpose`
  - `block_load_method::block_load_warp_transpose`

Example:

In the examples load operation is performed on block of 128 threads, using type `int` and 8 items per thread.

```
__global__ void example_kernel(int * input, ...)
{
    const int offset = blockIdx.x * 128 * 8;
    int items[8];
    rocprim::block_load<int, 128, 8, load_method> blockload;
    blockload.load(input + offset, items);
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - the number of items to be processed by each thread.
- **Method** – - the method to load data.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union with other storage types to increase shared memory reusability.

Public Functions

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*])

Loads data from continuous memory into an arrangement of items across the thread block.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters
InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Loads data from continuous memory into an arrangement of items across the thread block, which is guarded by range `valid`.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters

InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.
- **valid** – **[in]** - maximum range of valid numbers to load.

template<class **InputIterator**, class **Default**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds)

Loads data from continuous memory into an arrangement of items across the thread block, which is guarded by range with a fall-back value for out-of-bound elements.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters

- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer.
- **Default** – - [inferred] The data type of the default value.

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.
- **valid** – **[in]** - maximum range of valid numbers to load.
- **out_of_bounds** – **[in]** - default value assigned to out-of-bound items.

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*], *storage_type* &storage)

Loads data from continuous memory into an arrangement of items across the thread block, using temporary storage.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int items[8];
    using block_load_int = rocprim::block_load<int, 128, 8>;
    block_load_int bload;
    __shared__ typename block_load_int::storage_type storage;
    bload.load(..., items, storage);
    ...
}
```

Template Parameters

InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - temporary storage for inputs.

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *storage_type* &storage)

Loads data from continuous memory into an arrangement of items across the thread block, which is guarded by range `valid`, using temporary storage.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int items[8];
    using block_load_int = rocprim::block_load<int, 128, 8>;
    block_load_int bload;
    tile_static typename block_load_int::storage_type storage;
    bload.load(..., items, valid, storage);
    ...
}
```

Template Parameters

InputIterator – - [inferred] an iterator type for input (can be a simple pointer

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.

- **valid** – **[in]** - maximum range of valid numbers to load.
- **storage** – **[in]** - temporary storage for inputs.

template<class **InputIterator**, class **Default**>
__device__ inline void **load**(*InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds, *storage_type* &storage)

Loads data from continuous memory into an arrangement of items across the thread block, which is guarded by range with a fall-back value for out-of-bound elements, using temporary storage.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int items[8];
    using block_load_int = rocprim::block_load<int, 128, 8>;
    block_load_int bload;
    __shared__ typename block_load_int::storage_type storage;
    bload.load(..., items, valid, out_of_bounds, storage);
    ...
}
```

Template Parameters

- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer.
- **Default** – - [inferred] The data type of the default value.

Parameters

- **block_input** – **[in]** - the input iterator from the thread block to load from.
- **items** – **[out]** - array that data is loaded to.
- **valid** – **[in]** - maximum range of valid numbers to load.
- **out_of_bounds** – **[in]** - default value assigned to out-of-bound items.
- **storage** – **[in]** - temporary storage for inputs.

Algorithms

enum class `rocprim::block_load_method`

`block_load_method` enumerates the methods available to load data from continuous memory into a blocked arrangement of items across the thread block

Values:

enumerator **block_load_direct**

Data from continuous memory is loaded into a blocked arrangement of items.

Performance Notes:

- Performance decreases with increasing number of items per thread (stride between reads), because of reduced memory coalescing.

enumerator **block_load_striped**

A striped arrangement of data is read directly from memory.

enumerator **block_load_vectorize**

Data from continuous memory is loaded into a blocked arrangement of items using vectorization as an optimization.

Performance Notes:

- Performance remains high due to increased memory coalescing, provided that vectorization requirements are fulfilled. Otherwise, performance will default to `block_load_direct`.

Requirements:

- The input offset (`block_input`) must be quad-item aligned.
- The following conditions will prevent vectorization and switch to default `block_load_direct`:
  - `ItemsPerThread` is odd.
  - The datatype `T` is not a primitive or a HIP vector type (e.g. int2, int4, etc.

enumerator **block_load_transpose**

A striped arrangement of data from continuous memory is locally transposed into a blocked arrangement of items.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to `block_load_direct` and `block_load_vectorize` due to reordering on local memory.

enumerator **block_load_warp_transpose**

A warp-striped arrangement of data from continuous memory is locally transposed into a blocked arrangement of items.

Requirements:

- The number of threads in the block must be a multiple of the size of hardware warp.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to `block_load_direct` and `block_load_vectorize` due to reordering on local memory.

enumerator **default_method**

Defaults to `block_load_direct`.

Store

Class

template<class **T**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread**, *block_store_method* **Method** = *block_store_method*::*block_store_direct*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_store**

The *block_store* class is a block level parallel primitive which provides methods for storing an arrangement of items into a blocked/striped arrangement on continous memory.

Overview

- The *block_store* class has a number of different methods to store data:
  - *block_store_method::block_store_direct*
  - *block_store_method::block_store_striped*
  - *block_store_method::block_store_vectorize*
  - *block_store_method::block_store_transpose*
  - *block_store_method::block_store_warp_transpose*

Example:

In the examples store operation is performed on block of 128 threads, using type `int` and 8 items per thread.

```
__global__ void kernel(int * output)
{
    const int offset = blockIdx.x * 128 * 8;
    int items[8];
    rocprim::block_store<int, 128, 8, store_method> blockstore;
    blockstore.store(output + offset, items);
    ...
}
```

Template Parameters

- **T** – - the output/output type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - the number of items to be processed by each thread.
- **Method** – - the method to store data.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to

an externally allocated memory, or be a part of a union with other storage types to increase shared memory reusability.

Public Functions

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* block_output, *T* (&items)[*ItemsPerThread*])

Stores an arrangement of items from across the thread block into an arrangement on continuous memory.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters
OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **block_output** – **[out]** - the output iterator from the thread block to store to.
- **items** – **[in]** - array that data is read from.

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Stores an arrangement of items from across the thread block into an arrangement on continuous memory, which is guarded by range `valid`.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters
OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **block_output** – **[out]** - the output iterator from the thread block to store to.
- **items** – **[in]** - array that data is read from.
- **valid** – **[in]** - maximum range of valid numbers to read.

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], *storage_type* &storage)

Stores an arrangement of items from across the thread block into an arrangement on continuous memory, using temporary storage.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void kernel(...)
{
    int items[8];
    using block_store_int = rocprim::block_store<int, 128, 8>;
    block_store_int bstore;
    __shared__ typename block_store_int::storage_type storage;
    bstore.store(..., items, storage);
    ...
}
```

Template Parameters

OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **block_output** – **[out]** - the output iterator from the thread block to store to.
- **items** – **[in]** - array that data is read from.
- **storage** – **[in]** - temporary storage for outputs.

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], unsigned int valid, *storage_type* &storage)

Stores an arrangement of items from across the thread block into an arrangement on continuous memory, which is guarded by range `valid`, using temporary storage.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void kernel(...)
{
    int items[8];
    using block_store_int = rocprim::block_store<int, 128, 8>;
    block_store_int bstore;
    __shared__ typename block_store_int::storage_type storage;
    bstore.store(..., items, valid, storage);
    ...
}
```

Template Parameters

OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **block_output** – **[out]** - the output iterator from the thread block to store to.
- **items** – **[in]** - array that data is read from.
- **valid** – **[in]** - maximum range of valid numbers to read.
- **storage** – **[in]** - temporary storage for outputs.

Algorithms

enum class rocprim::**block_store_method**

block_store_method enumerates the methods available to store a striped arrangement of items into a blocked/striped arrangement on continuous memory

Values:

enumerator **block_store_direct**

A blocked arrangement of items is stored into a blocked arrangement on continuous memory.

Performance Notes:

- Performance decreases with increasing number of items per thread (stride between reads), because of reduced memory coalescing.

enumerator **block_store_striped**

A striped arrangement of items is stored into a blocked arrangement on continuous memory.

enumerator **block_store_vectorize**

A blocked arrangement of items is stored into a blocked arrangement on continuous memory using vectorization as an optimization.

Performance Notes:

- Performance remains high due to increased memory coalescing, provided that vectorization requirements are fulfilled. Otherwise, performance will default to block_store_direct.

Requirements:

- The output offset (block_output) must be quad-item aligned.
- The following conditions will prevent vectorization and switch to default block_store_direct:
  - ItemsPerThread is odd.
  - The datatype T is not a primitive or a HIP vector type (e.g. int2, int4, etc.

enumerator **block_store_transpose**

A blocked arrangement of items is locally transposed and stored as a striped arrangement of data on continuous memory.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to block_store_direct and block_store_vectorize due to reordering on local memory.

enumerator **block_store_warp_transpose**

A blocked arrangement of items is locally transposed and stored as a warp-striped arrangement of data on continuous memory.

Requirements:

- The number of threads in the block must be a multiple of the size of hardware warp.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to `block_store_direct` and `block_store_vectorize` due to reordering on local memory.

enumerator **default_method**

Defaults to `block_store_direct`.

Adjacent difference

template<class **T**, unsigned int **BlockSizeX**, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>

class **block_adjacent_difference**

The *block_adjacent_difference* class is a block level parallel primitive which provides methods for applying binary functions for pairs of consecutive items partition across a thread block.

Overview

- There are two types of flags:
  - Head flags.
  - Tail flags.
- The above flags are used to differentiate items from their predecessors or successors.
- E.g. Head flags are convenient for differentiating disjoint data segments as part of a segmented reduction/scan.

Examples

In the examples discontinuity operation is performed on block of 128 threads, using type `int`.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
    block_adjacent_difference_int b_discontinuity;
```

(continues on next page)

(continued from previous page)

```
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **T** – - the input type.
- **BlockSize** – - the number of threads in a block.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `head_flags` that indicate discontinuities between items partitioned across the thread block, where the first item has no reference and is always flagged.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_left()* or *block_discontinuity::flag_heads()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
```

(continues on next page)

(continued from previous page)

```
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_left()* or *block_discontinuity::flag_heads()* instead. This overload does not take a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `head_flags` that indicate discontinuities between items partitioned across the thread block, where the first item of the first thread is compared against a `tile_predecessor_item`.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_left()* or *block_discontinuity::flag_heads()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, tile_item, input, flag_op_type(),
                               storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_left()* or *block_discontinuity::flag_heads()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item has no reference and is always flagged.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_right()* or *block_discontinuity::flag_tails()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_tails(tail_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_right()* or *block_discontinuity::flag_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item of the last thread is compared against a `tile_successor_item`.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_right()* or *block_discontinuity::flag_tails()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_tails(tail_flags, tile_item, input, flag_op_type(),
                               storage);
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *subtract_right()* or *block_discontinuity::flag_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
```

(continues on next page)

(continued from previous page)

```
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tail_flags, input,
                                         flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>

__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item of the last thread is compared against a `tile_successor_item`.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tail_flags, tile_item,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.

- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the first item of the first thread is compared against a `tile_predecessor_item`.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;
```

(continues on next page)

(continued from previous page)

```
    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tile_item, tail_flags,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the first and last items of the first and last thread is compared against a `tile_predecessor_item` and a `tile_successor_item`.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_adjacent_difference_int = rocprim::block_adjacent_difference
↪<int, 128>;
    // allocate storage in shared memory
    __shared__ block_adjacent_difference_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_predecessor_item = 0;
    int tile_successor_item = 0;
    if (threadIdx.x == 0)
    {
        tile_predecessor_item = ...
        tile_successor_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_adjacent_difference_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tile_predecessor_item,
                                         tail_flags, tile_successor_item,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.

- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Deprecated:

The flags API of *block_adjacent_difference* is deprecated, use *block_discontinuity::flag_heads_and_tails()* instead.

This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_left**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the left item.

The first item in the first thread is copied from the input then for the rest the following code applies.

```
// For each i in [1, block_size * ItemsPerThread) across threads in a block
output[i] = op(input[i], input[i-1]);
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread

- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **storage** – reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_left**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, const *T* tile_predecessor, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the left item, with an explicit item before the tile.

```
// For the first item on the first thread use the tile predecessor
output[0] = op(input[0], tile_predecessor)
// For other items, i in [1, block_size * ItemsPerThread) across threads in a␣
↪block
output[i] = op(input[i], input[i-1]);
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread
- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **tile_predecessor** – **[in]** - the item before the tile, will be used as the input of the first application of `op`
- **storage** – - reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_left_partial**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, const unsigned int valid_items, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the left item, in a partial tile.

```
output[0] = input[0]
// For each item i in [1, valid_items) across threads in a block
output[i] = op(input[i], input[i-1]);
// Just copy "invalid" items in [valid_items, block_size * ItemsPerThread)
output[i] = input[i]
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread
- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **valid_items** – **[in]** - number of items in the block which are considered "valid" and will be used. Must be less or equal to `BlockSize` * `ItemsPerThread`
- **storage** – - reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_left_partial**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, const *T* tile_predecessor, const unsigned int valid_items, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the left item, in a partial tile with a predecessor.

This combines *subtract_left_partial()* with a tile predecessor.

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread
- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **tile_predecessor** – **[in]** - the item before the tile, will be used as the input of the first application of `op`
- **valid_items** – **[in]** - number of items in the block which are considered "valid" and will be used. Must be less or equal to `BlockSize * ItemsPerThread`
- **storage** – - reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_right**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the right item.

The last item in the last thread is copied from the input then for the rest the following code applies.

```
// For each i in [0, block_size * ItemsPerThread - 1) across threads in a block
output[i] = op(input[i], input[i+1]);
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread
- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **storage** – - reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_right**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, const *T* tile_successor, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the right item, with an explicit item after the tile.

```
// For each items i in [0, block_size * ItemsPerThread - 1) across threads in a␣
↪block
output[i] = op(input[i], input[i+1]);
// For the last item on the last thread use the tile successor
output[block_size * ItemsPerThread - 1] =
     op(input[block_size * ItemsPerThread - 1], tile_successor)
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread
- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **tile_successor** – **[in]** - the item after the tile, will be used as the input of the last application of `op`
- **storage** – - reference to a temporary storage object of type *storage_type*

template<typename **Output**, unsigned int **ItemsPerThread**, typename **BinaryFunction**>
__device__ inline void **subtract_right_partial**(const *T* (&input)[*ItemsPerThread*], *Output* (&output)[*ItemsPerThread*], const *BinaryFunction* op, const unsigned int valid_items, *storage_type* &storage)

Apply a function to each consecutive pair of elements partitioned across threads in the block and write the output to the position of the right item, in a partial tile.

```
// For each item i in [0, valid_items) across threads in a block
output[i] = op(input[i], input[i + 1]);
// Just copy "invalid" items in [valid_items, block_size * ItemsPerThread)
output[i] = input[i]
```

Storage reuse

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()* .

Template Parameters

- **Output** – - [inferred] the type of output, must be assignable from the result of `op`
- **ItemsPerThread** – - [inferred] the number of items processed by each thread

- **BinaryFunction** – - [inferred] the type of the function to apply

Parameters

- **input** – **[in]** - array that data is loaded from partitioned across the threads in the block
- **output** – **[out]** - array where the result of function application will be written to
- **op** – **[in]** - binary function applied to the items. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b)` The signature does not need to have `const &` but the function object must not modify the objects passed to it.
- **valid_items** – **[in]** - number of items in the block which are considered "valid" and will be used. Must be less or equal to `BlockSize` * `ItemsPerThread`
- **storage** – - reference to a temporary storage object of type *storage_type*

Discontinuity

template<class **T**, unsigned int **BlockSizeX**, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>

class **block_discontinuity**

The *block_discontinuity* class is a block level parallel primitive which provides methods for flagging items that are discontinued within an ordered set of items across threads in a block.

Overview

- There are two types of flags:
  - Head flags.
  - Tail flags.
- The above flags are used to differentiate items from their predecessors or successors.
- E.g. Head flags are convenient for differentiating disjoint data segments as part of a segmented reduction/scan.

Examples

In the examples discontinuity operation is performed on block of 128 threads, using type `int`.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **T** – - the input type.
- **BlockSize** – - the number of threads in a block.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `head_flags` that indicate discontinuities between items partitioned across the thread block, where the first item has no reference and is always flagged.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not take a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `head_flags` that indicate discontinuities between items partitioned across the thread block, where the first item of the first thread is compared against a `tile_predecessor_item`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads(head_flags, tile_item, input, flag_op_type(),
                               storage);
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item has no reference and is always flagged.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
```

(continues on next page)

(continued from previous page)

```
    int input[8];
    ...
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_tails(tail_flags, input, flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags `tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item of the last thread is compared against a `tile_successor_item`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
```

(continues on next page)

(continued from previous page)

```
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_tails(tail_flags, tile_item, input, flag_op_type(),
                               storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_tails**(*Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    ...
    int head_flags[8];
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tail_flags, input,
                                         flag_op_type(), storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the last item of the last thread is compared against a `tile_successor_item`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tail_flags, tile_item,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.

- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the first item of the first thread is compared against a `tile_predecessor_item`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_item = 0;
    if (threadIdx.x == 0)
    {
        tile_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
```

(continues on next page)

(continued from previous page)

```
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tile_item, tail_flags,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>
__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op, *storage_type* &storage)

Tags both `head_flags` and`tail_flags` that indicate discontinuities between items partitioned across the thread block, where the first and last items of the first and last thread is compared against a `tile_predecessor_item` and a `tile_successor_item`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize discontinuity for int and a block of 128 threads
    using block_discontinuity_int = rocprim::block_discontinuity<int, 128>;
    // allocate storage in shared memory
    __shared__ block_discontinuity_int::storage_type storage;

    // segment of consecutive items to be used
    int input[8];
    int tile_predecessor_item = 0;
    int tile_successor_item = 0;
    if (threadIdx.x == 0)
    {
        tile_predecessor_item = ...
        tile_successor_item = ...
    }
    ...
    int head_flags[8];
    int tail_flags[8];
    block_discontinuity_int b_discontinuity;
    using flag_op_type = typename rocprim::greater<int>;
    b_discontinuity.flag_heads_and_tails(head_flags, tile_predecessor_item,
                                         tail_flags, tile_successor_item,
                                         input, flag_op_type(),
                                         storage);
    ...
}
```

Template Parameters

- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread.
- **Flag** – - [inferred] the flag type.
- **FlagOp** – - [inferred] type of binary function used for flagging.

Parameters

- **head_flags** – **[out]** - array that contains the head flags.
- **tile_predecessor_item** – **[in]** - first tile item from thread to be compared against.
- **tail_flags** – **[out]** - array that contains the tail flags.
- **tile_successor_item** – **[in]** - last tile item from thread to be compared against.
- **input** – **[in]** - array that data is loaded from.
- **flag_op** – **[in]** - binary operation function object that will be used for flagging. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);` or `bool (const T& a, const T& b, unsigned int b_index);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**, class **Flag**, class **FlagOp**>

__device__ inline void **flag_heads_and_tails**(*Flag* (&head_flags)[*ItemsPerThread*], *T* tile_predecessor_item, *Flag* (&tail_flags)[*ItemsPerThread*], *T* tile_successor_item, const *T* (&input)[*ItemsPerThread*], *FlagOp* flag_op)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This overload does not accept a reference to temporary storage, instead it is declared as part of the function itself. Note that this does NOT decrease the shared memory requirements of a kernel using this function.

Scan

Class

template<class **T**, unsigned int **BlockSizeX**, *block_scan_algorithm* **Algorithm** = *block_scan_algorithm*::*default_algorithm*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_scan**

The *block_scan* class is a block level parallel primitive which provides methods for performing inclusive and exclusive scan operations of items partitioned across threads in a block.

Overview

- Supports non-commutative scan operators. However, a scan operator should be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Computation can more efficient when:
  - `ItemsPerThread` is greater than one,
  - `T` is an arithmetic type,
  - scan operation is simple addition operator, and
  - the number of threads in the block is a multiple of the hardware warp size (see rocprim::arch::wavefront::min_size()).
- *block_scan* has two alternative implementations: *block_scan_algorithm::using_warp_scan* and *block_scan_algorithm::reduce_then_scan*.

Examples

In the examples scan operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize warp_scan for int and logical warp of 192 threads
    using block_scan_int = rocprim::block_scan<int, 192>;
    // allocate storage in shared memory
    __shared__ block_scan_int::storage_type storage;

    int value = ...;
    // execute inclusive scan
    block_scan_int().inclusive_scan(
        value, // input
        value, // output
        storage
    );
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **BlockSizeX** – - the number of threads in a block's x dimension.
- **Algorithm** – - selected scan algorithm, *block_scan_algorithm::default_algorithm* by default.
- **BlockSizeY** – - the number of threads in a block's y dimension, defaults to 1.
- **BlockSizeZ** – - the number of threads in a block's z dimension, defaults to 1.

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive min scan operations performed on a block of 256 threads, each provides one `float` value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for float and block of 256 threads
    using block_scan_f = rocprim::block_scan<float, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_float::storage_type storage;

    float input = ...;
    float output;
    // execute inclusive min scan
    block_scan_float().inclusive_scan(
```

(continues on next page)

(continued from previous page)

```
        input,
        output,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, -2, 3, -4, ..., 255, -256}`, then `output` values in will be `{1, -2, -2, -4, ..., -254, -256}`.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* input, *T* &output, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* input, *T* &output, *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan and reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive min scan operations performed on a block of 256 threads, each provides one `float` value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for float and block of 256 threads
    using block_scan_f = rocprim::block_scan<float, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_float::storage_type storage;

    float input = ...;
    float output;
    float reduction;
    // execute inclusive min scan
    block_scan_float().inclusive_scan(
        input,
        output,
        reduction,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, -2, 3, -4, ..., 255, -256}`, then `output` values in will be `{1, -2, -2, -4, ..., -254, -256}`, and the `reduction` will be `-256`.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* input, *T* &output, *T* &reduction, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan and reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **PrefixCallback**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *PrefixCallback* &prefix_callback_op, *BinaryFunction* scan_op)

Performs inclusive scan across threads in a block, and uses `prefix_callback_op` to generate prefix value for the whole block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive prefix sum operations performed on a block of 256 threads, each thread provides one `int` value.

```
struct my_block_prefix
{
    int prefix;

    __device__ my_block_prefix(int prefix) : prefix(prefix) {}

    __device__ int operator()(int block_reduction)
    {
        int old_prefix = prefix;
        prefix = prefix + block_reduction;
        return old_prefix;
    }
};

__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for int and block of 256 threads
    using block_scan_f = rocprim::block_scan<int, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_int::storage_type storage;
```

(continues on next page)

(continued from previous page)

```
    // init prefix functor
    my_block_prefix prefix_callback(10);

    int input;
    int output;
    // execute inclusive prefix sum
    block_scan_int().inclusive_scan(
        input,
        output,
        storage,
        prefix_callback,
        rocprim::plus<int>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, 1, 1, ..., 1}`, then `output` values in will be `{11, 12, 13, ..., 266}`, and the `prefix` will be `266`.

Template Parameters

- **PrefixCallback** – - type of the unary function object used for generating block-wide prefix value for the scan operation.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **prefix_callback_op** – **[inout]** - function object for generating block prefix value. The signature of the `prefix_callback_op` should be equivalent to the following: `T f(const T &block_reduction);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The object will be called by the first warp of the block with block reduction of `input` values as input argument. The result of the first thread will be used as the block-wide prefix.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive maximum scan operations performed on a block of 128 threads, each provides two `long` value.

```
__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for long and block of 128 threads
    using block_scan_f = rocprim::block_scan<long, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_long::storage_type storage;

    long input[2] = ...;
    long output[2];
    // execute inclusive min scan
    block_scan_long().inclusive_scan(
        input,
        output,
        storage,
        rocprim::maximum<long>()
    );
    ...
}
```

If the `input` values across threads in a block are `{-1, 2, -3, 4, ..., -255, 256}`, then `output` values in will be `{-1, 2, 2, 4, ..., 254, 256}`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan and reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive maximum scan operations performed on a block of 128 threads, each provides two `long` value.

```
__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for long and block of 128 threads
    using block_scan_f = rocprim::block_scan<long, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_long::storage_type storage;

    long input[2] = ...;
    long output[2];
    long reduction;
    // execute inclusive min scan
    block_scan_long().inclusive_scan(
        input,
        output,
        reduction,
        storage,
        rocprim::maximum<long>()
    );
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

If the `input` values across threads in a block are `{-1, 2, -3, 4, ..., -255, 256}`, then `output` values in will be `{-1, 2, 2, 4, ..., 254, 256}` and the `reduction` will be `256`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **inclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* &reduction, *BinaryFunction* scan_op = *BinaryFunction*())

Performs inclusive scan and reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **PrefixCallback**, class **BinaryFunction**>
__device__ inline void **inclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *storage_type* &storage, *PrefixCallback* &prefix_callback_op, *BinaryFunction* scan_op)

Performs inclusive scan across threads in a block, and uses `prefix_callback_op` to generate prefix value for the whole block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive prefix sum operations performed on a block of 128 threads, each thread provides two `int` value.

```
struct my_block_prefix
{
    int prefix;

    __device__ my_block_prefix(int prefix) : prefix(prefix) {}

    __device__ int operator()(int block_reduction)
    {
        int old_prefix = prefix;
        prefix = prefix + block_reduction;
        return old_prefix;
    }
};

__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for int and block of 128 threads
    using block_scan_f = rocprim::block_scan<int, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_int::storage_type storage;

    // init prefix functor
    my_block_prefix prefix_callback(10);

    int input[2] = ...;
    int output[2];
    // execute inclusive prefix sum
    block_scan_int().inclusive_scan(
        input,
        output,
        storage,
        prefix_callback,
        rocprim::plus<int>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, 1, 1, ..., 1}`, then `output` values in will

be {11, 12, 13, ..., 266}, and the prefix will be 266.

Template Parameters

- **ItemsPerThread** – - number of items in the input array.
- **PrefixCallback** – - type of the unary function object used for generating block-wide prefix value for the scan operation.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with input.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **prefix_callback_op** – **[inout]** - function object for generating block prefix value. The signature of the prefix_callback_op should be equivalent to the following: T f(const T &block_reduction);. The signature does not need to have const &, but function object must not modify the objects passed to it. The object will be called by the first warp of the block with block reduction of input values as input argument. The result of the first thread will be used as the block-wide prefix.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: T f(const T &a, const T &b); . The signature does not need to have const &, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* input, *T* &output, *T* init, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan across threads in a block.

Storage reusage

Synchronization barrier should be placed before storage is reused or repurposed: __syncthreads() or *rocprim::syncthreads()*.

Examples

The examples present exclusive min scan operations performed on a block of 256 threads, each provides one float value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for float and block of 256 threads
    using block_scan_f = rocprim::block_scan<float, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_float::storage_type storage;

    float init = ...;
    float input = ...;
    float output;
    // execute exclusive min scan
    block_scan_float().exclusive_scan(
```

(continues on next page)

(continued from previous page)

```
        input,
        output,
        init,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, -2, 3, -4, ..., 255, -256}` and `init` is `0`, then `output` values in will be `{0, 0, -2, -2, -4, ..., -254, -254}`.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* input, *T* &output, *T* init, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`

. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* input, *T* &output, *T* init, *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan and reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive min scan operations performed on a block of 256 threads, each provides one `float` value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for float and block of 256 threads
    using block_scan_f = rocprim::block_scan<float, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_float::storage_type storage;

    float init = 0;
    float input = ...;
    float output;
    float reduction;
    // execute exclusive min scan
    block_scan_float().exclusive_scan(
        input,
        output,
        init,
        reduction,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, -2, 3, -4, ..., 255, -256}` and `init` is `0`, then `output` values in will be `{0, 0, -2, -2, -4, ..., -254, -254}` and the `reduction` will be `-256`.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.

- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* input, *T* &output, *T* init, *T* &reduction, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan and reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **PrefixCallback**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *PrefixCallback* &prefix_callback_op, *BinaryFunction* scan_op)

Performs exclusive scan across threads in a block, and uses `prefix_callback_op` to generate prefix value for the whole block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive prefix sum operations performed on a block of 256 threads, each thread provides one `int` value.

```
struct my_block_prefix
{
    int prefix;

    __device__ my_block_prefix(int prefix) : prefix(prefix) {}

    __device__ int operator()(int block_reduction)
    {
        int old_prefix = prefix;
        prefix = prefix + block_reduction;
        return old_prefix;
    }
};

__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_scan for int and block of 256 threads
    using block_scan_f = rocprim::block_scan<int, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_int::storage_type storage;

    // init prefix functor
    my_block_prefix prefix_callback(10);

    int input;
    int output;
    // execute exclusive prefix sum
    block_scan_int().exclusive_scan(
        input,
        output,
        storage,
        prefix_callback,
        rocprim::plus<int>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, 1, 1, ..., 1}`, then `output` values in will be `{10, 11, 12, 13, ..., 265}`, and the `prefix` will be `266`.

Template Parameters

- **PrefixCallback** – - type of the unary function object used for generating block-wide prefix value for the scan operation.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

- **prefix_callback_op** – **[inout]** - function object for generating block prefix value. The signature of the `prefix_callback_op` should be equivalent to the following: `T f(const T &block_reduction);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The object will be called by the first warp of the block with block reduction of `input` values as input argument. The result of the first thread will be used as the block-wide prefix.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* init, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive maximum scan operations performed on a block of 128 threads, each provides two `long` value.

```
__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for long and block of 128 threads
    using block_scan_f = rocprim::block_scan<long, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_long::storage_type storage;

    long init = ...;
    long input[2] = ...;
    long output[2];
    // execute exclusive min scan
    block_scan_long().exclusive_scan(
        input,
        output,
        init,
        storage,
        rocprim::maximum<long>()
    );
    ...
}
```

If the `input` values across threads in a block are `{-1, 2, -3, 4, ..., -255, 256}` and `init` is 0, then `output` values in will be `{0, 0, 2, 2, 4, ..., 254, 254}`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.

- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* init, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* init, *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan and reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive maximum scan operations performed on a block of 128 threads, each provides two `long` value.

```
__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for long and block of 128 threads
    using block_scan_f = rocprim::block_scan<long, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_long::storage_type storage;

    long init = ...;
    long input[2] = ...;
    long output[2];
    long reduction;
    // execute exclusive min scan
    block_scan_long().exclusive_scan(
        input,
        output,
        init,
        reduction,
        storage,
        rocprim::maximum<long>()
    );
    ...
}
```

If the `input` values across threads in a block are `{-1, 2, -3, 4, ..., -255, 256}` and `init` is 0, then `output` values in will be `{0, 0, 2, 2, 4, ..., 254, 254}` and the `reduction` will be `256`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **exclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *T* init, *T* &reduction, *BinaryFunction* scan_op = *BinaryFunction*())

Performs exclusive scan and reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a block.
- **reduction** – **[out]** - result of reducing of all `input` values in a block.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **PrefixCallback**, class **BinaryFunction**>
__device__ inline void **exclusive_scan**(*T* (&input)[*ItemsPerThread*], *T* (&output)[*ItemsPerThread*], *storage_type* &storage, *PrefixCallback* &prefix_callback_op, *BinaryFunction* scan_op)

Performs exclusive scan across threads in a block, and uses `prefix_callback_op` to generate prefix value for the whole block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive prefix sum operations performed on a block of 128 threads, each thread provides two `int` value.

```
struct my_block_prefix
{
    int prefix;

    __device__ my_block_prefix(int prefix) : prefix(prefix) {}
```

(continues on next page)

(continued from previous page)

```
    __device__ int operator()(int block_reduction)
    {
        int old_prefix = prefix;
        prefix = prefix + block_reduction;
        return old_prefix;
    }
};

__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_scan for int and block of 128 threads
    using block_scan_f = rocprim::block_scan<int, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_scan_int::storage_type storage;

    // init prefix functor
    my_block_prefix prefix_callback(10);

    int input[2] = ...;
    int output[2];
    // execute exclusive prefix sum
    block_scan_int().exclusive_scan(
        input,
        output,
        storage,
        prefix_callback,
        rocprim::plus<int>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, 1, 1, ..., 1}`, then `output` values in will be `{10, 11, 12, 13, ..., 265}`, and the `prefix` will be `266`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **PrefixCallback** – - type of the unary function object used for generating block-wide prefix value for the scan operation.
- **BinaryFunction** – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **prefix_callback_op** – **[inout]** - function object for generating block prefix value. The signature of the `prefix_callback_op` should be equivalent to the following: `T f(const T &block_reduction);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it. The object will be called by the first warp

of the block with block reduction of `input` values as input argument. The result of the first thread will be used as the block-wide prefix.

- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

Algorithms

enum class `rocprim::block_scan_algorithm`

Available algorithms for *block_scan* primitive.

Values:

enumerator **using_warp_scan**

A *warp_scan* based algorithm.

enumerator **reduce_then_scan**

An algorithm which limits calculations to a single hardware warp.

enumerator **default_algorithm**

Default *block_scan* algorithm.

Reduce

Class

template<class **T**, unsigned int **BlockSizeX**, *block_reduce_algorithm* **Algorithm** = *block_reduce_algorithm*::*default_algorithm*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_reduce**

The *block_reduce* class is a block level parallel primitive which provides methods for performing reductions operations on items partitioned across threads in a block.

Overview

- Supports non-commutative reduce operators. However, a reduce operator should be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Computation can more efficient when:
  - `ItemsPerThread` is greater than one,
  - `T` is an arithmetic type,
  - reduce operation is simple addition operator, and
  - the number of threads in the block is a multiple of the hardware warp size (see rocprim::arch::wavefront::min_size()).
- *block_reduce* has three alternative implementations: *block_reduce_algorithm::using_warp_reduce*, *block_reduce_algorithm::raking_reduce* and *block_reduce_algorithm::raking_reduce_commutative_*
- If the block sizes less than 64 only one warp reduction is used. The block reduction algorithm stores the result only in the first thread(lane_id = 0 warp_id = 0), when the block size is larger then the warp size.

Examples

In the examples reduce operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize warp_reduce for int and logical warp of 192 threads
    using block_reduce_int = rocprim::block_reduce<int, 192>;
    // allocate storage in shared memory
    __shared__ block_reduce_int::storage_type storage;

    int value = ...;
    // execute reduce
    block_reduce_int().reduce(
        value, // input
        value, // output
        storage
    );
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **BlockSize** – - the number of threads in a block.
- **Algorithm** – - selected reduce algorithm, *block_reduce_algorithm::default_algorithm* by default.

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* input, *T* &output, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *`rocprim::syncthreads()`*.

Examples

The examples present min reduce operations performed on a block of 256 threads, each provides one `float` value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_reduce for float and block of 256 threads
    using block_reduce_f = rocprim::block_reduce<float, 256>;
    // allocate storage in shared memory for the block
    __shared__ block_reduce_float::storage_type storage;

    float input = ...;
    float output;
    // execute min reduce
    block_reduce_float().reduce(
        input,
        output,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

If the `input` values across threads in a block are `{1, -2, 3, -4, ..., 255, -256}`, then `output` value will be `{-256}`.

Template Parameters

BinaryFunction – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* input, *T* &output, *BinaryFunction* reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

BinaryFunction – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* (&input)[*ItemsPerThread*], *T* &output, *storage_type* &storage,
BinaryFunction reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present maximum reduce operations performed on a block of 128 threads, each provides two `long` value.

```
__global__ void example_kernel(...) // blockDim.x = 128
{
    // specialize block_reduce for long and block of 128 threads
    using block_reduce_f = rocprim::block_reduce<long, 128>;
    // allocate storage in shared memory for the block
    __shared__ block_reduce_long::storage_type storage;

    long input[2] = ...;
    long output[2];
    // execute max reduce
    block_reduce_long().reduce(
        input,
        output,
        storage,
        rocprim::maximum<long>()
    );
    ...
}
```

If the `input` values across threads in a block are `{-1, 2, -3, 4, ..., -255, 256}`, then `output` value will be `{256}`.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.

- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* (&input)[*ItemsPerThread*], *T* &output, *BinaryFunction* reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* input, *T* &output, unsigned int valid_items, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present min reduce operations performed on a block of 256 threads, each provides one `float` value.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize block_reduce for float and block of 256 threads
    using block_reduce_f = rocprim::block_reduce<float, 256>;
```

(continues on next page)

(continued from previous page)

```
    // allocate storage in shared memory for the block
    __shared__ block_reduce_float::storage_type storage;

    float input = ...;
    unsigned int valid_items = 250;
    float output;
    // execute min reduce
    block_reduce_float().reduce(
        input,
        output,
        valid_items,
        storage,
        rocprim::minimum<float>()
    );
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **valid_items** – **[in]** - number of items that will be reduced in the block.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>>
__device__ inline void **reduce**(*T* input, *T* &output, unsigned int valid_items, *BinaryFunction* reduce_op = *BinaryFunction*())

Performs reduction across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

- **ItemsPerThread** – - number of items in the `input` array.
- **BinaryFunction** – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - reference to an array containing thread input values.
- **output** – **[out]** - reference to a thread output array. May be aliased with `input`.
- **valid_items** – **[in]** - number of items that will be reduced in the block.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

Algorithms

enum class `rocprim::block_reduce_algorithm`

Available algorithms for *block_reduce* primitive.

Values:

enumerator **using_warp_reduce**

A *warp_reduce* based algorithm.

enumerator **raking_reduce**

An algorithm which limits calculations to a single hardware warp.

enumerator **raking_reduce_commutative_only**

raking reduce that supports only commutative operators

enumerator **default_algorithm**

Default *block_reduce* algorithm.

Shuffle

template<class **T**, unsigned int **BlockSizeX**, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>

class **block_shuffle**

The *block_shuffle* class is a block level parallel primitive which provides methods for shuffling data partitioned across a block.

Overview

It is commonplace for blocks of threads to rearrange data items between threads. The BlockShuffle abstraction allows threads to efficiently shift items either (a) up to their successor or (b) down to their predecessor.

- Computation can more efficient when:
  - `ItemsPerThread` is greater than one,
  - `T` is an arithmetic type,
  - the number of threads in the block is a multiple of the hardware warp size (see *rocprim::warp_size()*).

Examples

In the examples shuffle operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block__shuffle_int for int and logical warp of 192 threads
    using block__shuffle_int = rocprim::block_shuffle<int, 192>;
    // allocate storage in shared memory
    __shared__ block_shuffle::storage_type storage;

    int value = ...;
    // execute block shuffle
    block__shuffle_int().inclusive_up(
        value, // input
        value, // output
        storage
    );
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **BlockSizeX** – - the number of threads in a block's x dimension, it has no defaults value.
- **BlockSizeY** – - the number of threads in a block's y dimension, defaults to 1.
- **BlockSizeZ** – - the number of threads in a block's z dimension, defaults to 1.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implementation the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

__device__ inline void **offset**(*T* input, *T* &output, int distance = 1)

Shuffles data across threads in a block, offseted by the distance value.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance), where distance may be a negative value.

Any shuffle operation with invalid input or output threadIds are not carried out, i.e. threadId < 0 || threadId >= BlockSize.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block__shuffle_int for int and logical warp of 192 threads
```

(continues on next page)

(continued from previous page)

```
    using block_shuffle_int = rocprim::block_shuffle<int, 192>;

    int value = ...;
    // execute block shuffle
    block_shuffle_int().offset(
        value, // input
        value  // output
    );
    ...
}
```

Parameters

- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId.

__device__ inline void **offset**(const size_t &flat_id, *T* input, *T* &output, int distance)

Shuffles data across threads in a block, offseted by the distance value.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance), where distance may be a negative value.

Any shuffle operation with invalid input or output threadIds are not carried out, i.e. threadId < 0 || threadId >= BlockSize.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId.

__device__ inline void **offset**(const size_t &flat_id, *T* input, *T* &output, int distance, *storage_type* &storage)

Shuffles data across threads in a block, offseted by the distance value, using temporary storage.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance), where distance may be a negative value.

Any shuffle operation with invalid input or output threadIds are not carried out, i.e. threadId < 0 || threadId >= BlockSize.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId.

- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

__device__ inline void **rotate**(*T* input, *T* &output, int distance = 1)

Shuffles data across threads in a block, offseted by the distance value.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance) % BlockSize, where distance may be a negative value.

Data is rotated around the block, using (input_threadId + distance) modulous BlockSize to ensure valid threadIds.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_shuffle_int for int and logical warp of 192 threads
    using block_shuffle_int = rocprim::block_shuffle<int, 192>;

    int value = ...;
    // execute block shuffle
    block_shuffle_int().rotate(
        value, // input
        value  // output
    );
    ...
}
```

Parameters

- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId. Distance magnitude should be <= BlockSize.

__device__ inline void **rotate**(const size_t &flat_id, *T* input, *T* &output, int distance)

Shuffles data across threads in a block, offseted by the distance value.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance) % BlockSize, where distance may be a negative value.

Data is rotated around the block, using (input_threadId + distance) modulous BlockSize to ensure valid threadIds.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId.

__device__ inline void **rotate**(const size_t &flat_id, *T* input, *T* &output, int distance, *storage_type* &storage)

Shuffles data across threads in a block, offseted by the distance value, using temporary storage.

A thread with threadIdx i receives data from a thread with threadIdx (i - distance) % BlockSize, where distance may be a negative value.

Data is rotated around the block, using (input_threadId + distance) modulous BlockSize to ensure valid threadIds.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - input data to be shuffled to another thread.
- **output** – **[out]** - reference to a output value, that receives data from another thread
- **distance** – **[in]** - The input threadId + distance = output threadId.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**>
__device__ inline void **up**(*T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*])

The thread block rotates a blocked arrange of input items, shifting it up by one item.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block__shuffle_int for int and logical warp of 192 threads
    using block__shuffle_int = rocprim::block_shuffle<int, 192>;

    int value = ...;
    // execute block shuffle
    block__shuffle_int().up(
        value, // input
        value  // output
    );
    ...
}
```

Parameters

- **input** – **[in]** - The calling thread's input items
- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_0$.

template<unsigned int **ItemsPerThread**>
__device__ inline void **up**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*])

The thread block rotates a blocked arrange of input items, shifting it up by one item.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items

- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`). The item `prev`[0] is not updated for *thread*$_0$.

template<unsigned int **ItemsPerThread**>
__device__ inline void **up**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*], *storage_type* &storage)

The thread block rotates a blocked arrange of input items, shifting it up by one item, using temporary storage.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items
- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`).
- **storage** – **[in]** - reference to a temporary storage object of type storage_type. The item `prev`[0] is not updated for *thread*$_0$.

template<unsigned int **ItemsPerThread**>
__device__ inline void **up**(*T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*], *T* &block_suffix)

The thread block rotates a blocked arrange of input items, shifting it up by one item.

Parameters

- **input** – **[in]** - The calling thread's input items
- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`). The item `prev`[0] is not updated for *thread*$_0$.
- **block_suffix** – **[out]** - The item `input`[ItemsPerThread-1] from *thread*, provided to all threads

template<unsigned int **ItemsPerThread**>
__device__ inline void **up**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*], *T* &block_suffix)

The thread block rotates a blocked arrange of input items, shifting it up by one item.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items
- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`). The item `prev`[0] is not updated for *thread*$_0$.
- **block_suffix** – **[out]** - The item `input`[ItemsPerThread-1] from *thread*, provided to all threads

template<int **ItemsPerThread**>
__device__ inline void **up**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&prev)[*ItemsPerThread*], *T* &block_suffix, *storage_type* &storage)

The thread block rotates a blocked arrange of input items, shifting it up by one item, using temporary storage.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items

- **prev** – **[out]** - The corresponding predecessor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_0$.
- **block_suffix** – **[out]** - The item `input`[ItemsPerThread-1] from *thread*, provided to all threads
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(*T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*])

The thread block rotates a blocked arrange of input items, shifting it down by one item.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block__shuffle_int for int and logical warp of 192 threads
    using block__shuffle_int = rocprim::block_shuffle<int, 192>;

    int value = ...;
    // execute block shuffle
    block__shuffle_int().down(
        value, // input
        value  // output
    );
    ...
}
```

Parameters

- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{\text{BlockSize - 1}}$.

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*])

The thread block rotates a blocked arrange of input items, shifting it down by one item.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{\text{BlockSize - 1}}$.

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*], *storage_type* &storage)

The thread block rotates a blocked arrange of input items, shifting it down by one item, using temporary storage.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*

- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{BlockSize - 1}$.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(*T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*], *T* &block_prefix)

The thread block rotates a blocked arrange of input items, shifting it down by one item.

Parameters

- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{BlockSize - 1}$.
- **block_prefix** – **[out]** - The item `input`[0] from *thread*, provided to all threads

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*], *T* &block_prefix)

The thread block rotates a blocked arrange of input items, shifting it down by one item.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{BlockSize - 1}$.
- **block_prefix** – **[out]** - The item `input`[0] from *thread*, provided to all threads

template<unsigned int **ItemsPerThread**>
__device__ inline void **down**(const size_t &flat_id, *T* (&input)[*ItemsPerThread*], *T* (&next)[*ItemsPerThread*], *T* &block_prefix, *storage_type* &storage)

The thread block rotates a blocked arrange of input items, shifting it down by one item, using temporary storage.

Parameters

- **flat_id** – **[in]** - flat thread ID obtained from *rocprim::flat_block_thread_id*
- **input** – **[in]** - The calling thread's input items
- **next** – **[out]** - The corresponding successor items (may be aliased to `input`). The item `prev`[0] is not updated for $thread_{BlockSize - 1}$.
- **block_prefix** – **[out]** - The item `input`[0] from *thread*, provided to all threads
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

Exchange

template<class **T**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread**, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1, block_padding_hint **PaddingHint** = block_padding_hint::avoid_conflicts>
class **block_exchange**

The *block_exchange* class is a block level parallel primitive which provides methods for rearranging items partitioned across threads in a block.

Overview

- The *block_exchange* class supports the following rearrangement methods:
  - Transposing a blocked arrangement to a striped arrangement.
  - Transposing a striped arrangement to a blocked arrangement.
  - Transposing a blocked arrangement to a warp-striped arrangement.
  - Transposing a warp-striped arrangement to a blocked arrangement.
  - Scattering items to a blocked arrangement.
  - Scattering items to a striped arrangement.
- Data is automatically be padded to ensure zero bank conflicts.

Examples

In the examples exchange operation is performed on block of 128 threads, using type `int` with 8 items per thread.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items per␣
↪thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.blocked_to_striped(items, items, storage);
    ...
}
```

Template Parameters

- **T** – - the input type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - the number of items contributed by each thread.
- **PaddingHint** – - a hint that decides when to use padding. May not always be applicable.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implementation the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **U**>
__device__ inline void **blocked_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a blocked arrangement of items to a striped arrangement across the thread block.

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

template<class **U**>
__device__ inline void **blocked_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a blocked arrangement of items to a striped arrangement across the thread block, using temporary storage.

Storage reusage
Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.blocked_to_striped(items, items, storage);
    ...
}
```

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**>

__device__ inline void **striped_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a striped arrangement of items to a blocked arrangement across the thread block.

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

template<class **U**>
__device__ inline void **striped_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a striped arrangement of items to a blocked arrangement across the thread block, using temporary storage.

Storage reusage
Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.striped_to_blocked(items, items, storage);
    ...
}
```

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**>
__device__ inline void **blocked_to_warp_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a blocked arrangement of items to a warp-striped arrangement across the thread block.

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

template<class **U**>
__device__ inline void **blocked_to_warp_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a blocked arrangement of items to a warp-striped arrangement across the thread block, using temporary storage.

Storage reusage
Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.blocked_to_warp_striped(items, items, storage);
    ...
}
```

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**>
__device__ inline void **warp_striped_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a warp-striped arrangement of items to a blocked arrangement across the thread block.

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.

- **output** – **[out]** - array that data is loaded to.

template<class **U**>
__device__ inline void **warp_striped_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a warp-striped arrangement of items to a blocked arrangement across the thread block, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.warp_striped_to_blocked(items, items, storage);
    ...
}
```

Template Parameters

U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*])

Scatters items to a blocked arrangement based on their ranks across the thread block.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

- **ranks** – **[out]** - array that has rank of data.

template<class **U**, class **Offset**>
__device__ inline void **gather_from_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*])

Gathers items from a striped arrangement based on their ranks across the thread block.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Scatters items to a blocked arrangement based on their ranks across the thread block, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    int ranks[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.scatter_to_blocked(items, items, ranks, storage);
    ...
}
```

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**, class **Offset**>
__device__ inline void **gather_from_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Gathers items from a striped arrangement based on their ranks across the thread block, using temporary storage.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*])

Scatters items to a striped arrangement based on their ranks across the thread block.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Scatters items to a striped arrangement based on their ranks across the thread block, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    int ranks[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.scatter_to_striped(items, items, ranks, storage);
    ...
}
```

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **WarpSize** = arch::wavefront::min_size(), class **U**, class **Offset**>
__device__ inline void **scatter_to_warp_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Scatters items to a *warp* striped arrangement based on their ranks across the thread block, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
```

(continues on next page)

(continued from previous page)

```
    int ranks[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.scatter_to_warp_striped(items, items, ranks, storage);
    ...
}
```

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[out]** - array that has rank of data.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_striped_guarded**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*])

Scatters items to a striped arrangement based on their ranks across the thread block, guarded by rank.

Overview

- Items with rank -1 are not scattered.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[in]** - array that has rank of data.

template<class **U**, class **Offset**>
__device__ inline void **scatter_to_striped_guarded**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Scatters items to a striped arrangement based on their ranks across the thread block, guarded by rank, using temporary storage.

Overview

- Items with rank -1 are not scattered.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    int ranks[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.scatter_to_striped_guarded(items, items, ranks, storage);
    ...
}
```

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[in]** - array that has rank of data.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**, class **Offset**, class **ValidFlag**>
__device__ inline void **scatter_to_striped_flagged**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], const *ValidFlag* (&is_valid)[*ItemsPerThread*])

Scatters items to a striped arrangement based on their ranks across the thread block, with a flag to denote validity.

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.
- **ValidFlag** – - [inferred] the validity flag type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[in]** - array that has rank of data.

- **is_valid** – **[in]** - array that has flags to denote validity.

template<class **U**, class **Offset**, class **ValidFlag**>
__device__ inline void **scatter_to_striped_flagged**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *Offset* (&ranks)[*ItemsPerThread*], const *ValidFlag* (&is_valid)[*ItemsPerThread*], *storage_type* &storage)

Scatters items to a striped arrangement based on their ranks across the thread block, with a flag to denote validity, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    // specialize block_exchange for int, block of 128 threads and 8 items␣
↪per thread
    using block_exchange_int = rocprim::block_exchange<int, 128, 8>;
    // allocate storage in shared memory
    __shared__ block_exchange_int::storage_type storage;

    int items[8];
    int ranks[8];
    int flags[8];
    ...
    block_exchange_int b_exchange;
    b_exchange.scatter_to_striped_flagged(items, items, ranks, flags,␣
↪storage);
    ...
}
```

Template Parameters

- **U** – - [inferred] the output type.
- **Offset** – - [inferred] the rank type.
- **ValidFlag** – - [inferred] the validity flag type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[in]** - array that has rank of data.
- **is_valid** – **[in]** - array that has flags to denote validity.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

Sort

Generic Block Sort

template<class **Key**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread** = 1, class **Value** = empty_type, *block_sort_algorithm* **Algorithm** = *block_sort_algorithm*::*default_algorithm*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_sort**

The *block_sort* class is a block level parallel primitive which provides methods sorting items (keys or key-value pairs) partitioned across threads in a block using comparison-based sort algorithm.

Overview

- Accepts custom compare_functions for sorting across a block.
- Performance notes:
  - It is generally better if `BlockSize` and `ItemsPerThread` are powers of two.
  - The overloaded functions with `size` are generally slower.

Examples

In the examples sort is performed on a block of 256 threads, each thread provides 8 `int` values, results are returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_sort for int, block of 256 threads, and 8 items per␣
↪thread
    // key-only sort
    using block_sort_int = rocprim::block_sort<int, 256, 8>;
    // allocate storage in shared memory
    __shared__ block_sort_int::storage_type storage;

    int input[8] = ...;
    // execute block sort (ascending)
    block_sort_int().sort(
        input,
        storage
    );
    ...
}
```

Template Parameters

- **Key** – - the key type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - number of items processed by each thread. The total range will be BlockSize * ItemsPerThread long
- **Value** – - the value type. Default type empty_type indicates a keys-only sort.
- **Algorithm** – - selected sort algorithm. The available algorithms and default choice are documented in *block_sort_algorithm*.

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* &thread_key, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort for any data type.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*())

This overload allows an array of `ItemsPerThread` keys to be passed in so that each thread can process multiple items.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* &thread_key, *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort for any data type.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples sort is performed on a block of 256 threads, each thread provides one `int` value, results are returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_sort for int, block of 256 threads
    // key-only sort
    using block_sort_int = rocprim::block_sort<int, 256>;
    // allocate storage in shared memory
    __shared__ block_sort_int::storage_type storage;
```

(continues on next page)

(continued from previous page)

```
    int input = ...;
    // execute block sort (ascending)
    block_sort_int().sort(
        input,
        storage
    );
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*())

This overload allows arrays of `ItemsPerThread` keys to be passed in so that each thread can process multiple items.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* &thread_key, *Value* &thread_value, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort by key for any data type.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **thread_value** – **[inout]** - reference to a value provided by a thread.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*())

This overload allows an array of `ItemsPerThread` keys and values to be passed in so that each thread can process multiple items.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>

__device__ inline void **sort**(*Key* &thread_key, *Value* &thread_value, *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort by key for any data type.

In the examples sort is performed on a block of 256 threads, each thread provides 8 `int` keys and one `int` value, results are returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_sort for int, block of 256 threads, and 8 items per␣
↪thread
    using block_sort_int = rocprim::block_sort<int, 256, 8, int>;
    // allocate storage in shared memory
    __shared__ block_sort_int::storage_type storage;

    int key[8] = ...;
    int value = ...;
    // execute block sort (ascending)
    block_sort_int().sort(
        key,
        value,
        storage
    );
    ...
}
```

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **thread_value** – **[inout]** - reference to a value provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*())

This overload allows an array of `ItemsPerThread` keys and values to be passed in so that each thread can process multiple items.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>

__device__ inline void **sort**(*Key* &thread_key, *storage_type* &storage, const unsigned int size, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort for any data type. This function sorts up to `size` elements blocked across threads.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **size** – **[in]** - custom size of block to be sorted.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *storage_type* &storage, const unsigned int size, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort for any data type. This function sorts up to `size` elements blocked across threads.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – **[inout]** - reference to keys provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **size** – **[in]** - custom size of block to be sorted.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* &thread_key, *Value* &thread_value, *storage_type* &storage, const unsigned int size, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort by key for any data type. This function sorts up to `size` elements blocked across threads.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – **[inout]** - reference to a key provided by a thread.
- **thread_value** – **[inout]** - reference to a value provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **size** – **[in]** - custom size of block to be sorted.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>>
__device__ inline void **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *storage_type* &storage, const unsigned int size, *BinaryFunction* compare_function = *BinaryFunction*())

Block sort by key for any data type. This function sorts up to `size` elements blocked across threads.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – **[inout]** - reference to keys provided by a thread.
- **thread_values** – **[inout]** - reference to values provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **size** – **[in]** - custom size of block to be sorted.
- **compare_function** – **[in]** - comparison function object which returns true if the first argument is is ordered before the second. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

enum class `rocprim::block_sort_algorithm`

Available algorithms for *block_sort* primitive.

Values:

enumerator **bitonic_sort**

A bitonic sort based algorithm.

```
\par Stability
\p bitonic_sort is <b>not stable</b>: it doesn't necessarily preserve the␣
↪relative ordering
of equivalent keys.
That is, given two keys \p a and \p b and a binary boolean operation \p op such␣
↪that:
  * \p a precedes \p b in the input keys, and
  * op(a, b) and op(b, a) are both false,
then it is <b>not guaranteed</b> that \p a will precede \p b as well in the␣
↪output
(ordered) keys.
```

enumerator **merge_sort**

A merge sort based algorithm.

```
\par Stability
\p merge_sort <b>may</b> use \p stable_merge_sort as the underlying␣
↪implementation.
However, \p merge_sort is <b>not guaranteed to be stable</b>: it doesn't␣
↪necessarily
preserve the relative ordering of equivalent keys.
That is, given two keys \p a and \p b and a binary boolean operation \p op such␣
↪that:
```

(continues on next page)

(continued from previous page)

```
  * \p a precedes \p b in the input keys, and
  * op(a, b) and op(b, a) are both false,
then it is <b>not guaranteed</b> that \p a will precede \p b as well in the␣
↪output
(ordered) keys.
```

enumerator **stable_merge_sort**

A merged sort based algorithm which sorts stably.

```
\par Stability
\p stable_merge_sort is \b stable: it preserves the relative ordering of␣
↪equivalent keys.
That is, given two keys \p a and \p b and a binary boolean operation \p op such␣
↪that:
  * \p a precedes \p b in the input keys, and
  * op(a, b) and op(b, a) are both false,
then it is \b guaranteed that \p a will precede \p b as well in the output␣
↪(ordered) keys.
```

enumerator **default_algorithm**

Default *block_sort* algorithm.

Radix sort

template<class **Key**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread**, class **Value** = empty_type, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1, unsigned int **RadixBitsPerPass** = (*BlockSizeX* * *BlockSizeY* * *BlockSizeZ*) % arch::wavefront::min_size() == 0 ? 8 : 4, block_radix_rank_algorithm **RadixRankAlgorithm** = (*BlockSizeX* * *BlockSizeY* * *BlockSizeZ*) % arch::wavefront::min_size() == 0 ? block_radix_rank_algorithm::match : block_radix_rank_algorithm::basic_memoize, block_padding_hint **PaddingHint** = block_padding_hint::lds_occupancy_bound>

class **block_radix_sort**

The *block_radix_sort* class is a block level parallel primitive which provides methods for sorting of items (keys or key-value pairs) partitioned across threads in a block using radix sort algorithm.

Overview

- `Key` type must be an arithmetic type (that is, an integral type or a floating-point type).
- Performance depends on `BlockSize` and `ItemsPerThread`.
  - It is usually better for `BlockSize` to be a multiple of the size of the hardware warp.
  - It is usually increased when `ItemsPerThread` is greater than one. However, when there are too many items per thread, each thread may need so much registers and/or shared memory that occupancy will fall too low, decreasing the performance.
  - If `Key` is an integer type and the range of keys is known in advance, the performance can be improved by setting `begin_bit` and `end_bit`, for example if all keys are in range [100, 10000], `begin_bit = 0` and `end_bit = 14` will cover the whole range.

Stability

block_radix_sort is **stable:** it preserves the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- a precedes b in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **guaranteed** that a will precede b as well in the output (ordered) keys.

Examples

In the examples radix sort is performed on a block of 256 threads, each thread provides eight `int` value, results are returned using the same array as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for int, block of 256 threads,
    // and eight items per thread; key-only sort
    using block_rsort_int = rocprim::block_radix_sort<int, 256, 8>;
    // allocate storage in shared memory
    __shared__ block_rsort_int::storage_type storage;

    int input[8] = ...;
    // execute block radix sort (ascending)
    block_rsort_int().sort(
        input,
        storage
    );
    ...
}
```

Template Parameters

- **Key** – - the key type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - the number of items contributed by each thread.
- **Value** – - the value type. Default type empty_type indicates a keys-only sort.
- **RadixBitsPerPass** – - amount of bits to sort per pass. The Default is 4.
- **RadixRankAlgorithm** – the rank algorithm used.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over keys partitioned across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two `float` value, results are returned using the same array as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for float, block of 128 threads,
    // and two items per thread; key-only sort
    using block_rsort_float = rocprim::block_radix_sort<float, 128, 2>;
    // allocate storage in shared memory
    __shared__ block_rsort_float::storage_type storage;

    float input[2] = ...;
    // execute block radix sort (ascending)
    block_rsort_float().sort(
        input,
        storage
    );
    ...
}
```

If the `input` values across threads in a block are `{[256, 255], ..., [4, 3], [2, 1]}`, then then after sort they will be equal `{[1, 2], [3, 4] ..., [255, 256]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over keys partitioned across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over keys partitioned across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *`rocprim::syncthreads()`*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two `float` value, results are returned using the same array as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for float, block of 128 threads,
    // and two items per thread; key-only sort
    using block_rsort_float = rocprim::block_radix_sort<float, 128, 2>;
    // allocate storage in shared memory
    __shared__ block_rsort_float::storage_type storage;

    float input[2] = ...;
    // execute block radix sort (descending)
    block_rsort_float().sort_desc(
        input,
        storage
    );
    ...
}
```

If the `input` values across threads in a block are `{[1, 2], [3, 4] ..., [255, 256]}`, then after sort they will be equal `{[256, 255], ..., [4, 3], [2, 1]}`.

Template Parameters
Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over keys partitioned across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters
Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs partitioned across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two key-value `int-float` pairs, results are returned using the same arrays as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for int-float pairs, block of 128
    // threads, and two items per thread
    using block_rsort_ii = rocprim::block_radix_sort<int, 128, 2, int>;
    // allocate storage in shared memory
    __shared__ block_rsort_ii::storage_type storage;

    int keys[2] = ...;
    float values[2] = ...;
    // execute block radix sort-by-key (ascending)
    block_rsort_ii().sort(
        keys, values,
        storage
    );
    ...
}
```

If the `keys` across threads in a block are `{[256, 255], ..., [4, 3], [2, 1]}` and the `values` are `{[1, 1], [2, 2] ..., [128, 128]}`, then after sort the `keys` will be equal `{[1, 2], [3, 4] ..., [255, 256]}` and the `values` will be equal `{[128, 128], [127, 127] ..., [2, 2], [1, 1]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>

__device__ inline void **sort**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs partitioned across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `* sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs partitioned across threads in a block.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *`rocprim::syncthreads()`*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two key-value `int-float` pairs, results are returned using the same arrays as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for int-float pairs, block of 128
    // threads, and two items per thread
    using block_rsort_ii = rocprim::block_radix_sort<int, 128, 2, int>;
    // allocate storage in shared memory
```

(continues on next page)

(continued from previous page)

```
    __shared__ block_rsort_ii::storage_type storage;

    int keys[2] = ...;
    float values[2] = ...;
    // execute block radix sort-by-key (descending)
    block_rsort_ii().sort_desc(
        keys, values,
        storage
    );
    ...
}
```

If the `keys` across threads in a block are `{[1, 2], [3, 4] ..., [255, 256]}` and the `values` are `{[128, 128], [127, 127] ..., [2, 2], [1, 1]}`, then after sort the `keys` will be equal `{[256, 255], ..., [4, 3], [2, 1]}` and the `values` will be equal `{[1, 1], [2, 2] ..., [128, 128]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs partitioned across threads in a block.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_to_striped**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over keys partitioned across threads in a block, results are saved in a striped arrangement.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *`rocprim::syncthreads()`*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two `float` value, results are returned using the same array as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for float, block of 128 threads,
    // and two items per thread; key-only sort
    using block_rsort_float = rocprim::block_radix_sort<float, 128, 2>;
    // allocate storage in shared memory
    __shared__ block_rsort_float::storage_type storage;

    float keys[2] = ...;
    // execute block radix sort (ascending)
    block_rsort_float().sort_to_striped(
        keys,
        storage
    );
    ...
}
```

If the `input` values across threads in a block are `{[256, 255], ..., [4, 3], [2, 1]}`, then then after sort they will be equal `{[1, 129], [2, 130] ..., [128, 256]}`.

Template Parameters
Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_to_striped**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over keys partitioned across threads in a block, results are saved in a striped arrangement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters
Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_to_striped**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over keys partitioned across threads in a block, results are saved in a striped arrangement.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples radix sort is performed on a block of 128 threads, each thread provides two `float` value, results are returned using the same array as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for float, block of 128 threads,
    // and two items per thread; key-only sort
    using block_rsort_float = rocprim::block_radix_sort<float, 128, 2>;
    // allocate storage in shared memory
    __shared__ block_rsort_float::storage_type storage;

    float input[2] = ...;
    // execute block radix sort (descending)
    block_rsort_float().sort_desc_to_striped(
        input,
        storage
    );
    ...
}
```

If the `input` values across threads in a block are `{[1, 2], [3, 4] ..., [255, 256]}`, then after sort they will be equal `{[256, 128], ..., [130, 2], [129, 1]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_to_striped**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over keys partitioned across threads in a block, results are saved in a striped arrangement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs partitioned across threads in a block, results are saved in a striped arrangement.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *`rocprim::syncthreads()`*.

Examples

In the examples radix sort is performed on a block of 4 threads, each thread provides two key-value `int-float` pairs, results are returned using the same arrays as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for int-float pairs, block of 4
    // threads, and two items per thread
    using block_rsort_ii = rocprim::block_radix_sort<int, 4, 2, int>;
    // allocate storage in shared memory
    __shared__ block_rsort_ii::storage_type storage;

    int keys[2] = ...;
    float values[2] = ...;
    // execute block radix sort-by-key (ascending)
    block_rsort_ii().sort_to_striped(
        keys, values,
        storage
    );
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

If the `keys` across threads in a block are `{[8, 7], [6, 5], [4, 3], [2, 1]}` and the `values` are `{[-1, -2], [-3, -4], [-5, -6], [-7, -8]}`, then after sort the `keys` will be equal `{[1, 5], [2, 6], [3, 7], [4, 8]}` and the `values` will be equal `{[-8, -4], [-7, -3], [-6, -2], [-5, -1]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *`rocprim::tuple`* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs partitioned across threads in a block, results are saved in a striped arrangement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.

- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs partitioned across threads in a block, results are saved in a striped arrangement.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples radix sort is performed on a block of 4 threads, each thread provides two key-value `int-float` pairs, results are returned using the same arrays as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_radix_sort for int-float pairs, block of 4
    // threads, and two items per thread
    using block_rsort_ii = rocprim::block_radix_sort<int, 4, 2, int>;
    // allocate storage in shared memory
    __shared__ block_rsort_ii::storage_type storage;

    int keys[2] = ...;
    float values[2] = ...;
    // execute block radix sort-by-key (descending)
    block_rsort_ii().sort_desc_to_striped(
        keys, values,
        storage
    );
    ...
}
```

If the `keys` across threads in a block are `{[1, 2], [3, 4], [5, 6], [7, 8]}` and the `values` are `{[80, 70], [60, 50], [40, 30], [20, 10]}`, then after sort the `keys` will be equal `{[8, 4], [7, 3], [6, 2], [5, 1]}` and the `values` will be equal `{[10, 50], [20, 60], [30, 70], [40, 80]}`.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.

- **values** – **[inout]** - reference to an array of values provided by a thread.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

Pre

Method is enabled only if `Value` type is different than empty_type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs partitioned across threads in a block, results are saved in a striped arrangement.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters

Decomposer – The type of the decomposer argument. Defaults to the identity decomposer.

Parameters

- **keys** – **[inout]** - reference to an array of keys provided by a thread.
- **values** – **[inout]** - reference to an array of values provided by a thread.
- **begin_bit** – **[in]** - [optional] index of the first (least significant) bit used in key comparison. Must be in range `[0; 8 * sizeof(Key))`. Default value: `0`.
- **end_bit** – **[in]** - [optional] past-the-end index (most significant) bit used in key comparison. Must be in range `(begin_bit; 8 * sizeof(Key)]`. Default value: `8 * sizeof(Key)`.
- **decomposer** – **[in]** [optional] If `Key` is not an arithmetic type (integral, floating point), a custom decomposer functor should be passed that produces a *rocprim::tuple* of references to fundamental types from this custom type.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>

__device__ inline void **sort_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs in a *warp-striped order*

See also

block_radix_sort::sort_to_striped partitioned across threads in a block, results are saved in a striped arrangement.

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs ascending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_desc_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], typename std::enable_if<*WithValues*, *Value*>::type (&values)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_desc_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], *storage_type* &storage, unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_desc_to_striped

template<bool **WithValues** = with_values, class **Decomposer** = ::rocprim::identity_decomposer>
__device__ inline void **sort_desc_warp_striped_to_striped**(*Key* (&keys)[*ItemsPerThread*], unsigned int begin_bit = 0, unsigned int end_bit = 8 * sizeof(*Key*), *Decomposer* decomposer = {})

Performs descending radix sort over key-value pairs in a *warp-striped order* partitioned across threads in a block, results are saved in a striped arrangement.

See also

block_radix_sort::sort_desc_to_striped

Histogram

Class

template<class **T**, unsigned int **BlockSizeX**, unsigned int **ItemsPerThread**, unsigned int **Bins**, *block_histogram_algorithm* **Algorithm** = *block_histogram_algorithm*::*default_algorithm*, unsigned int **BlockSizeY** = 1, unsigned int **BlockSizeZ** = 1>
class **block_histogram**

The *block_histogram* class is a block level parallel primitive which provides methods for constructing block-wide histograms from items partitioned across threads in a block.

Overview

- *block_histogram* has two alternative implementations: *block_histogram_algorithm::using_atomic* and *block_histogram_algorithm::using_sort*.

Examples

In the examples histogram operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_histogram for int, logical block of 192 threads,
    // 2 items per thread and a bin size of 192.
    using block_histogram_int = rocprim::block_histogram<int, 192, 2, 192>;
    // allocate storage in shared memory
    __shared__ block_histogram_int::storage_type storage;
    __shared__ int hist[192];

    int value[2];
    ...
    // execute histogram
    block_histogram_int().histogram(
        value, // input
        hist, // output
        storage
    );
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **BlockSize** – - the number of threads in a block.
- **ItemsPerThread** – - the number of items to be processed by each thread.
- **Bins** – - the number of bins within the histogram.
- **Algorithm** – - selected histogram algorithm, *block_histogram_algorithm::default_algorithm* by default.

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **Counter**>
__device__ inline void **init_histogram**(*Counter* hist[*Bins*])

Initialize histogram counters to zero.

Template Parameters
Counter – - [inferred] counter type of histogram.

Parameters
hist – **[out]** - histogram bin count.

template<class **Counter**>
__device__ inline void **composite**(*T* (&input)[*ItemsPerThread*], *Counter* hist[*Bins*], *storage_type* &storage)

Update an existing block-wide histogram. Each thread composites an array of input elements.

Storage reusage
Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples histogram operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_histogram for int, logical block of 192 threads,
```

(continues on next page)

(continued from previous page)

```
    // 2 items per thread and a bin size of 192.
    using block_histogram_int = rocprim::block_histogram<int, 192, 2, 192>;
    // allocate storage in shared memory
    __shared__ block_histogram_int::storage_type storage;
    __shared__ int hist[192];

    int value[2];
    ...
    // initialize histogram
    block_histogram_int().init_histogram(
        hist // output
    );

    rocprim::syncthreads();

    // update histogram
    block_histogram_int().composite(
        value, // input
        hist, // output
        storage
    );
    ...
}
```

Template Parameters
Counter – - [inferred] counter type of histogram.

Parameters

- **input** – **[in]** - reference to an array containing thread input values. The function expects each value to satisfy 0 <= input[i] < BINS.
- **hist** – **[out]** - histogram bin count.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **Counter**>
__device__ inline void **composite**(*T* (&input)[*ItemsPerThread*], *Counter* hist[*Bins*])

Update an existing block-wide histogram. Each thread composites an array of input elements.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters
Counter – - [inferred] counter type of histogram.

Parameters

- **input** – **[in]** - reference to an array containing thread input values. The function expects each value to satisfy 0 <= input[i] < BINS.
- **hist** – **[out]** - histogram bin count.

template<class **Counter**>
__device__ inline void **histogram**(*T* (&input)[*ItemsPerThread*], *Counter* hist[*Bins*], *storage_type* &storage)

Construct a new block-wide histogram. Each thread contributes an array of input elements.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples histogram operation is performed on block of 192 threads, each provides one `int` value, result is returned using the same variable as for input.

```
__global__ void example_kernel(...)
{
    // specialize block_histogram for int, logical block of 192 threads,
    // 2 items per thread and a bin size of 192.
    using block_histogram_int = rocprim::block_histogram<int, 192, 2, 192>;
    // allocate storage in shared memory
    __shared__ block_histogram_int::storage_type storage;
    __shared__ int hist[192];

    int value[2];
    ...
    // execute histogram
    block_histogram_int().histogram(
        value, // input
        hist, // output
        storage
    );
    ...
}
```

Template Parameters

Counter – - [inferred] counter type of histogram.

Parameters

- **input** – **[in]** - reference to an array containing thread input values. The function expects each value to satisfy 0 <= input[i] < BINS.
- **hist** – **[out]** - histogram bin count.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **Counter**>
__device__ inline void **histogram**(*T* (&input)[*ItemsPerThread*], *Counter* hist[*Bins*])

Construct a new block-wide histogram. Each thread contributes an array of input elements.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

- This overload does not accept storage argument. Required shared memory is allocated by the method itself.

Template Parameters
Counter – - [inferred] counter type of histogram.

Parameters

- **input** – **[in]** - reference to an array containing thread input values. The function expects each value to satisfy 0 <= input[i] < BINS.
- **hist** – **[out]** - histogram bin count.

Algorithms

enum class rocprim::**block_histogram_algorithm**

Available algorithms for *block_histogram* primitive.

Values:

enumerator **using_atomic**

Atomic addition is used to update bin count directly.

Performance Notes:

- Performance is dependent on hardware implementation of atomic addition.
- Performance may decrease for non-uniform random input distributions where many concurrent updates may be made to the same bin counter.

enumerator **using_sort**

A two-phase operation is used:-

- Data is sorted using radix-sort.
- ”Runs” of same-valued keys are detected using discontinuity; run-lengths are bin counts.

Performance Notes:

- Performance is consistent regardless of sample bin distribution.

enumerator **default_algorithm**

Default *block_histogram* algorithm.

2.4.2 Data movement functions

Direct Blocked

Load

template<class **InputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void rocprim::**block_load_direct_blocked**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*])

Loads data from continuous memory into a blocked arrangement of items across the thread block.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer

- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to

template<class **InputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **rocprim::block_load_direct_blocked**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Loads data from continuous memory into a blocked arrangement of items across the thread block, which is guarded by range `valid`.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to
- **valid** – - maximum range of valid numbers to load

template<class **InputIterator**, class **T**, unsigned int **ItemsPerThread**, class **Default**>
__device__ inline void **rocprim::block_load_direct_blocked**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds)

Loads data from continuous memory into a blocked arrangement of items across the thread block, which is guarded by range with a fall-back value for out-of-bound elements.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread
- **Default** – - [inferred] The data type of the default value

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to

- **valid** – - maximum range of valid numbers to load
- **out_of_bounds** – - default value assigned to out-of-bound items

Store

template<class **OutputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **rocprim::block_store_direct_blocked**(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*])

Stores a blocked arrangement of items from across the thread block into a blocked arrangement on continuous memory.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

Template Parameters

- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block

template<class **OutputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **rocprim::block_store_direct_blocked**(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Stores a blocked arrangement of items from across the thread block into a blocked arrangement on continuous memory, which is guarded by range `valid`.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

Template Parameters

- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block
- **valid** – - maximum range of valid numbers to store

Direct Blocked Vectorized

Load

template<class **T**, class **U**, unsigned int **ItemsPerThread**>
__device__ inline auto **rocprim::block_load_direct_blocked_vectorized**(unsigned int flat_id, *T* *block_input, *U* (&items)[*ItemsPerThread*]) -> typename std::enable_if<detail::is_vectorizable<*T*, *ItemsPerThread*>::value>::type

Loads data from continuous memory into a blocked arrangement of items across the thread block.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

The input offset (`block_input` + offset) must be quad-item aligned.

The following conditions will prevent vectorization and switch to default block_load_direct_blocked:

- `ItemsPerThread` is odd.
- The datatype `T` is not a primitive or a HIP vector type (e.g. int2, int4, etc.

The type `T` must be such that it can be implicitly converted to `U`.

Template Parameters

- **T** – - [inferred] the input data type
- **U** – - [inferred] the output data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to

Store

template<class **T**, class **U**, unsigned int **ItemsPerThread**>
__device__ inline auto **rocprim::block_store_direct_blocked_vectorized**(unsigned int flat_id, *T* *block_output, *U* (&items)[*ItemsPerThread*]) -> typename std::enable_if<detail::is_vectorizable<*T*, *ItemsPerThread*>::value>::type

Stores a blocked arrangement of items from across the thread block into a blocked arrangement on continuous memory.

The block arrangement is assumed to be (block-threads * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

The input offset (`block_output` + offset) must be quad-item aligned.

The following conditions will prevent vectorization and switch to default block_load_direct_blocked:

- `ItemsPerThread` is odd.
- The datatype `T` is not a primitive or a HIP vector type (e.g. int2, int4, etc.

The type `U` must be such that it can be implicitly converted to `T`.

Template Parameters

- **`T`** – - [inferred] the output data type
- **`U`** – - [inferred] the input data type
- **`ItemsPerThread`** – - [inferred] the number of items to be processed by each thread

Parameters

- **`flat_id`** – - a local flat 1D thread id in a block (tile) for the calling thread
- **`block_output`** – - the input iterator from the thread block to load from
- **`items`** – - array that data is loaded to

Direct Striped

Load

template<unsigned int **`BlockSize`**, class **`InputIterator`**, class **`T`**, unsigned int **`ItemsPerThread`**>
__device__ inline void `rocprim::`**`block_load_direct_striped`**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*])

Loads data from continuous memory into a striped arrangement of items across the thread block.

The striped arrangement is assumed to be (`BlockSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **`BlockSize`** – - the number of threads in a block
- **`InputIterator`** – - [inferred] an iterator type for input (can be a simple pointer
- **`T`** – - [inferred] the data type
- **`ItemsPerThread`** – - [inferred] the number of items to be processed by each thread

Parameters

- **`flat_id`** – - a local flat 1D thread id in a block (tile) for the calling thread
- **`block_input`** – - the input iterator from the thread block to load from
- **`items`** – - array that data is loaded to

template<unsigned int **`BlockSize`**, class **`InputIterator`**, class **`T`**, unsigned int **`ItemsPerThread`**>
__device__ inline void `rocprim::`**`block_load_direct_striped`**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Loads data from continuous memory into a striped arrangement of items across the thread block, which is guarded by range `valid`.

The striped arrangement is assumed to be (`BlockSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **BlockSize** – - the number of threads in a block
- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to
- **valid** – - maximum range of valid numbers to load

template<unsigned int **BlockSize**, class **InputIterator**, class **T**, unsigned int **ItemsPerThread**, class **Default**>
__device__ inline void `rocprim::block_load_direct_striped`(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds)

Loads data from continuous memory into a striped arrangement of items across the thread block, which is guarded by range with a fall-back value for out-of-bound elements.

The striped arrangement is assumed to be (`BlockSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

Template Parameters

- **BlockSize** – - the number of threads in a block
- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread
- **Default** – - [inferred] The data type of the default value

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to
- **valid** – - maximum range of valid numbers to load
- **out_of_bounds** – - default value assigned to out-of-bound items

Store

template<unsigned int **BlockSize**, class **OutputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void `rocprim::block_store_direct_striped`(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*])

Stores a striped arrangement of items from across the thread block into a blocked arrangement on continuous memory.

The striped arrangement is assumed to be (`BlockSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

Template Parameters

- **BlockSize** – - the number of threads in a block

- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block

template<unsigned int **BlockSize**, class **OutputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **rocprim::block_store_direct_striped**(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Stores a striped arrangement of items from across the thread block into a blocked arrangement on continuous memory, which is guarded by range `valid`.

The striped arrangement is assumed to be (`BlockSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

Template Parameters

- **BlockSize** – - the number of threads in a block
- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block
- **valid** – - maximum range of valid numbers to store

Direct Warp Striped

Load

template<unsigned int **WarpSize** = arch::wavefront::min_size(), class **InputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **block_load_direct_warp_striped**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*])

Loads data from continuous memory into a warp-striped arrangement of items across the thread block.

The warp-striped arrangement is assumed to be (`WarpSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

- The number of threads in the block must be a multiple of `WarpSize`.
- The default `WarpSize` is a hardware warpsize and is an optimal value.
- `WarpSize` must be a power of two and equal or less than the size of hardware warp.
- Using `WarpSize` smaller than hardware warpsize could result in lower performance.

Template Parameters

- **WarpSize** – - [optional] the number of threads in a warp
- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to

template<unsigned int **WarpSize** = arch::wavefront::min_size(), class **InputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **block_load_direct_warp_striped**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Loads data from continuous memory into a warp-striped arrangement of items across the thread block, which is guarded by range `valid`.

The warp-striped arrangement is assumed to be (`WarpSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

- The number of threads in the block must be a multiple of `WarpSize`.
- The default `WarpSize` is a hardware warpsize and is an optimal value.
- `WarpSize` must be a power of two and equal or less than the size of hardware warp.
- Using `WarpSize` smaller than hardware warpsize could result in lower performance.

Template Parameters

- **WarpSize** – - [optional] the number of threads in a warp
- **InputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_input** – - the input iterator from the thread block to load from
- **items** – - array that data is loaded to
- **valid** – - maximum range of valid numbers to load

template<unsigned int **WarpSize** = arch::wavefront::min_size(), class **InputIterator**, class **T**, unsigned int **ItemsPerThread**, class **Default**>
__device__ inline void **block_load_direct_warp_striped**(unsigned int flat_id, *InputIterator* block_input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds)

Loads data from continuous memory into a warp-striped arrangement of items across the thread block, which is guarded by range with a fall-back value for out-of-bound elements.

The warp-striped arrangement is assumed to be (`WarpSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to load a range of `ItemsPerThread` into `items`.

- The number of threads in the block must be a multiple of `WarpSize`.
- The default `WarpSize` is a hardware warpsize and is an optimal value.
- `WarpSize` must be a power of two and equal or less than the size of hardware warp.
- Using `WarpSize` smaller than hardware warpsize could result in lower performance.

Template Parameters

- **`WarpSize`** – - [optional] the number of threads in a warp
- **`InputIterator`** – - [inferred] an iterator type for input (can be a simple pointer
- **`T`** – - [inferred] the data type
- **`ItemsPerThread`** – - [inferred] the number of items to be processed by each thread
- **`Default`** – - [inferred] The data type of the default value

Parameters

- **`flat_id`** – - a local flat 1D thread id in a block (tile) for the calling thread
- **`block_input`** – - the input iterator from the thread block to load from
- **`items`** – - array that data is loaded to
- **`valid`** – - maximum range of valid numbers to load
- **`out_of_bounds`** – - default value assigned to out-of-bound items

Store

template<unsigned int **`WarpSize`** = arch::wavefront::min_size(), class **`OutputIterator`**, class **`T`**, unsigned int **`ItemsPerThread`**>
__device__ inline void **`block_store_direct_warp_striped`**(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*])

Stores a warp-striped arrangement of items from across the thread block into a blocked arrangement on continuous memory.

The warp-striped arrangement is assumed to be (`WarpSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

- The number of threads in the block must be a multiple of `WarpSize`.
- The default `WarpSize` is a hardware warpsize and is an optimal value.
- `WarpSize` must be a power of two and equal or less than the size of hardware warp.
- Using `WarpSize` smaller than hardware warpsize could result in lower performance.

Template Parameters

- **`WarpSize`** – - [optional] the number of threads in a warp

- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block

template<unsigned int **WarpSize** = arch::wavefront::min_size(), class **OutputIterator**, class **T**, unsigned int **ItemsPerThread**>
__device__ inline void **block_store_direct_warp_striped**(unsigned int flat_id, *OutputIterator* block_output, *T* (&items)[*ItemsPerThread*], unsigned int valid)

Stores a warp-striped arrangement of items from across the thread block into a blocked arrangement on continuous memory, which is guarded by range `valid`.

The warp-striped arrangement is assumed to be (`WarpSize` * `ItemsPerThread`) items across a thread block. Each thread uses a `flat_id` to store a range of `ItemsPerThread` `items` to the thread block.

- The number of threads in the block must be a multiple of `WarpSize`.
- The default `WarpSize` is a hardware warpsize and is an optimal value.
- `WarpSize` must be a power of two and equal or less than the size of hardware warp.
- Using `WarpSize` smaller than hardware warpsize could result in lower performance.

Template Parameters

- **WarpSize** – - [optional] the number of threads in a warp
- **OutputIterator** – - [inferred] an iterator type for input (can be a simple pointer
- **T** – - [inferred] the data type
- **ItemsPerThread** – - [inferred] the number of items to be processed by each thread

Parameters

- **flat_id** – - a local flat 1D thread id in a block (tile) for the calling thread
- **block_output** – - the input iterator from the thread block to store to
- **items** – - array that data is stored to thread block
- **valid** – - maximum range of valid numbers to store

2.5 Warp-Level Operations

- *Load*
- *Store*
- *Reduce*
- *Scan*
- *Sort*

- *Shuffle*
- *Exchange*

2.5.1 Load

Class

template<class **T**, unsigned int **ItemsPerThread**, unsigned int **WarpSize** = ::rocprim::arch::wavefront::min_size(), *warp_load_method* **Method** = *warp_load_method*::*warp_load_direct*>
class **warp_load**

The *warp_load* class is a warp level parallel primitive which provides methods for loading data from continuous memory into a blocked arrangement of items across a warp.

Overview

- The *warp_load* class has a number of different methods to load data:
  - *warp_load_direct*
  - *warp_load_striped*
  - *warp_load_vectorize*
  - *warp_load_transpose*

Example:

In the example a load operation is performed on a warp of 8 threads, using type `int` and 4 items per thread.

```
__global__ void example_kernel(int * input, ...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_per_
↪warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    const int offset = blockIdx.x * threads_per_block * items_per_thread
        + warp_id * threads_per_warp * items_per_thread;
    int items[items_per_thread];
    rocprim::warp_load<int, items_per_thread, threads_per_warp, load_method>␣
↪warp_load;
    warp_load.load(input + offset, items);
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **ItemsPerThread** – - the number of items to be processed by each thread.
- **WarpSize** – - the number of threads in the warp. It must be a divisor of the kernel block size.
- **Method** – - the method to load data.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union with other storage types to increase shared memory reusability.

Public Functions

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* input, *T* (&items)[*ItemsPerThread*], *storage_type*&)

Loads data from continuous memory into an arrangement of items across the warp.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters
InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **input** – **[in]** - the input iterator to load from.
- **items** – **[out]** - array that data is loaded to.
- **-** – **[in]** temporary storage for inputs.

template<class **InputIterator**>
__device__ inline void **load**(*InputIterator* input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *storage_type*&)

Loads data from continuous memory into an arrangement of items across the warp.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters
InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **input** – **[in]** - the input iterator to load from.
- **items** – **[out]** - array that data is loaded to.
- **valid** – **[in]** - maximum range of valid numbers to load.
- **-** – **[in]** temporary storage for inputs.

template<class **InputIterator**, class **Default**>

__device__ inline void **load**(*InputIterator* input, *T* (&items)[*ItemsPerThread*], unsigned int valid, *Default* out_of_bounds, *storage_type*&)

Loads data from continuous memory into an arrangement of items across the warp.

Overview

- The type `T` must be such that an object of type `InputIterator` can be dereferenced and then implicitly converted to `T`.

Template Parameters

InputIterator – - [inferred] an iterator type for input (can be a simple pointer.

Parameters

- **input** – **[in]** - the input iterator to load from.
- **items** – **[out]** - array that data is loaded to.
- **valid** – **[in]** - maximum range of valid numbers to load.
- **out_of_bounds** – **[in]** - default value assigned to out-of-bound items.
- **-** – **[in]** temporary storage for inputs.

Algorithms

enum class `rocprim::warp_load_method`

`warp_load_method` enumerates the methods available to load data from continuous memory into a blocked/striped arrangement of items across the warp

Values:

enumerator **warp_load_direct**

Data from continuous memory is loaded into a blocked arrangement of items.

Performance Notes:

- Performance decreases with increasing number of items per thread (stride between reads), because of reduced memory coalescing.

enumerator **warp_load_striped**

A striped arrangement of data is read directly from memory.

enumerator **warp_load_vectorize**

Data from continuous memory is loaded into a blocked arrangement of items using vectorization as an optimization.

Performance Notes:

- Performance remains high due to increased memory coalescing, provided that vectorization requirements are fulfilled. Otherwise, performance will default to `warp_load_direct`.

Requirements:

- The input offset (`block_input`) must be quad-item aligned.
- The following conditions will prevent vectorization and switch to default `warp_load_direct:`

- `ItemsPerThread` is odd.
- The datatype `T` is not a primitive or a HIP vector type (e.g. int2, int4, etc.

enumerator **warp_load_transpose**

A striped arrangement of data from continuous memory is locally transposed into a blocked arrangement of items.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to `warp_load_direct` and `warp_load_vectorize` due to reordering on local memory.

enumerator **default_method**

Defaults to `warp_load_direct`.

2.5.2 Store

Class

template<class **T**, unsigned int **ItemsPerThread**, unsigned int **WarpSize** = ::rocprim::arch::wavefront::min_size(), *warp_store_method* **Method** = *warp_store_method*::*warp_store_direct*>
class **warp_store**

The *warp_store* class is a warp level parallel primitive which provides methods for storing an arrangement of items into a blocked/striped arrangement on continous memory.

Overview

- The *warp_store* class has a number of different methods to store data:
  - *warp_store_direct*
  - *warp_store_striped*
  - *warp_store_vectorize*
  - *warp_store_transpose*

Example:

In the example a store operation is performed on a warp of 8 threads, using type `int` and 4 items per thread.

```
__global__ void example_kernel(int * output, ...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_per_
↪warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    const int offset = blockIdx.x * threads_per_block * items_per_thread
        + warp_id * threads_per_warp * items_per_thread;
    int items[items_per_thread];
    rocprim::warp_store<int, items_per_thread, threads_per_warp, load_method>␣
```

(continues on next page)

(continued from previous page)

```
↪warp_store;
    warp_store.store(output + offset, items);
    ...
}
```

Template Parameters

- **T** – - the output/output type.
- **ItemsPerThread** – - the number of items to be processed by each thread.
- **WarpSize** – - the number of threads in a warp. It must be a divisor of the kernel block size.
- **Method** – - the method to store data.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union with other storage types to increase shared memory reusability.

Public Functions

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* output, *T* (&items)[*ItemsPerThread*], *storage_type*&)

Stores an arrangement of items from across the warp into an arrangement on continuous memory.

Overview

- The type `T` must be such that an object of type `OutputIterator` can be dereferenced and then implicitly assigned from `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Template Parameters

OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **output** – **[out]** - the output iterator to store to.
- **items** – **[in]** - array that data is read from.

template<class **OutputIterator**>
__device__ inline void **store**(*OutputIterator* output, *T* (&items)[*ItemsPerThread*], unsigned int valid, *storage_type*&)

Stores an arrangement of items from across the warp into an arrangement on continuous memory, which is guarded by range `valid`, using temporary storage.

Overview

- The type `T` must be such that an object of type `OutputIterator` can be dereferenced and then implicitly assigned from `T`.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Template Parameters

OutputIterator – - [inferred] an iterator type for output (can be a simple pointer.

Parameters

- **output** – **[out]** - the output iterator to store to.
- **items** – **[in]** - array that data is read from.
- **valid** – **[in]** - maximum range of valid numbers to read.

Algorithms

enum class `rocprim::warp_store_method`

`warp_store_method` enumerates the methods available to store a blocked/striped arrangement of items into a blocked/striped arrangement in continuous memory

Values:

enumerator **warp_store_direct**

A blocked arrangement of items is stored into a blocked arrangement on continuous memory.

Performance Notes:

- Performance decreases with increasing number of items per thread (stride between reads), because of reduced memory coalescing.

enumerator **warp_store_striped**

A striped arrangement of items is stored into a blocked arrangement on continuous memory.

enumerator **warp_store_vectorize**

A blocked arrangement of items is stored into a blocked arrangement on continuous memory using vectorization as an optimization.

Performance Notes:

- Performance remains high due to increased memory coalescing, provided that vectorization requirements are fulfilled. Otherwise, performance will default to `warp_store_direct`.

Requirements:

- The output offset (`block_output`) must be quad-item aligned.
- The following conditions will prevent vectorization and switch to default `warp_store_direct`:
  - `ItemsPerThread` is odd.
  - The datatype `T` is not a primitive or a HIP vector type (e.g. int2, int4, etc.

enumerator **warp_store_transpose**

A blocked arrangement of items is locally transposed and stored as a striped arrangement of data on continuous memory.

Performance Notes:

- Performance remains high due to increased memory coalescing, regardless of the number of items per thread.
- Performance may be better compared to `warp_store_direct` and `warp_store_vectorize` due to reordering on local memory.

enumerator **default_method**

Defaults to `warp_store_direct`.

2.5.3 Reduce

template<class **T**, unsigned int **WarpSize** = arch::wavefront::min_size(), bool **UseAllReduce** = false>

class **warp_reduce**

The *warp_reduce* class is a warp level parallel primitive which provides methods for performing reduction operations on items partitioned across threads in a hardware warp.

Overview

- `WarpSize` must be equal to or less than the size of hardware warp (see rocprim::arch::wavefront::min_size()). If it is less, reduce is performed separately within groups determined by WarpSize.

  For example, if `WarpSize` is 4, hardware warp is 64, reduction will be performed in logical warps grouped like this: `{ {0, 1, 2, 3}, {4, 5, 6, 7 }, ..., {60, 61, 62, 63} }` (thread is represented here by its id within hardware warp).
- Logical warp is a group of `WarpSize` consecutive threads from the same hardware warp.
- Supports non-commutative reduce operators. However, a reduce operator should be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Number of threads executing *warp_reduce*'s function must be a multiple of `WarpSize`;
- All threads from a logical warp must be in the same hardware warp.

Examples

In the examples reduce operation is performed on groups of 16 threads, each provides one `int` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 64.

```
__global__ void example_kernel(...)
{
    // specialize warp_reduce for int and logical warp of 16 threads
    using warp_reduce_int = rocprim::warp_reduce<int, 16>;
    // allocate storage in shared memory
    __shared__ warp_reduce_int::storage_type temp[4];

    int logical_warp_id = threadIdx.x/16;
    int value = ...;
```

(continues on next page)

(continued from previous page)

```
    // execute reduce
    warp_reduce_int().reduce(
        value, // input
        value, // output
        temp[logical_warp_id]
    );
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **WarpSize** – - the size of logical warp size, which can be equal to or less than the size of hardware warp (see rocprim::arch::wavefront::min_size()). Reduce operations are performed separately within groups determined by WarpSize.
- **UseAllReduce** – - input parameter to determine whether to broadcast final reduction value to all threads (default is false).

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **reduce**(*T* input, *T* &output, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples reduce operation is performed on groups of 16 threads, each provides one `int` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 64.

```
__global__ void example_kernel(...)
{
    // specialize warp_reduce for int and logical warp of 16 threads
    using warp_reduce_int = rocprim::warp_reduce<int, 16>;
```

(continues on next page)

(continued from previous page)

```
    // allocate storage in shared memory
    __shared__ warp_reduce_int::storage_type temp[4];

    int logical_warp_id = threadIdx.x/16;
    int value = ...;
    // execute reduction
    warp_reduce_int().reduce(
        value, // input
        value, // output
        temp[logical_warp_id],
        rocprim::minimum<float>()
    );
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **reduce**(*T*, *T*&, *storage_type*&, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs reduction across threads in a logical warp. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **reduce**(*T* input, *T* &output, int valid_items, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

In the examples reduce operation is performed on groups of 16 threads, each provides one `int` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 64.

```
__global__ void example_kernel(...)
{
    // specialize warp_reduce for int and logical warp of 16 threads
    using warp_reduce_int = rocprim::warp_reduce<int, 16>;
    // allocate storage in shared memory
    __shared__ warp_reduce_int::storage_type temp[4];

    int logical_warp_id = threadIdx.x/16;
    int value = ...;
    int valid_items = 4;
    // execute reduction
    warp_reduce_int().reduce(
        value, // input
        value, // output
        valid_items,
        temp[logical_warp_id]
    );
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **valid_items** – **[in]** - number of items that will be reduced in the warp.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **reduce**(*T*, *T*&, int, *storage_type*&, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs reduction across threads in a logical warp. Invalid Warp Size.

template<class **Flag**, class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **head_segmented_reduce**(*T* input, *T* &output, *Flag* flag, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs head-segmented reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Template Parameters

- **Flag** – - type of head flags. Must be contextually convertible to `bool`.
- **BinaryFunction** – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **flag** – **[in]** - thread head flag, `true` flags mark beginnings of segments.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **Flag**, class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **head_segmented_reduce**(*T*, *T*&, *Flag*, *storage_type*&, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs head-segmented reduction across threads in a logical warp. Invalid Warp Size.

template<class **Flag**, class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **tail_segmented_reduce**(*T* input, *T* &output, *Flag* flag, *storage_type* &storage, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs tail-segmented reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Template Parameters

- **Flag** – - type of tail flags. Must be contextually convertible to `bool`.
- **BinaryFunction** – - type of binary function used for reduce. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **flag** – **[in]** - thread tail flag, `true` flags mark ends of segments.

- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **reduce_op** – **[in]** - binary operation function object that will be used for reduce. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **Flag**, class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **tail_segmented_reduce**(*T*, *T*&, *Flag*, *storage_type*&, *BinaryFunction* reduce_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs tail-segmented reduction across threads in a logical warp. Invalid Warp Size.

2.5.4 Scan

template<class **T**, unsigned int **WarpSize** = arch::wavefront::min_size()>

class **warp_scan**

The *warp_scan* class is a warp level parallel primitive which provides methods for performing inclusive and exclusive scan operations of items partitioned across threads in a hardware warp.

Overview

- `WarpSize` must be equal to or less than the size of hardware warp (see rocprim::arch::wavefront::min_size()). If it is less, scan is performed separately within groups determined by WarpSize.

  For example, if `WarpSize` is 4, hardware warp is 64, scan will be performed in logical warps grouped like this: `{ {0, 1, 2, 3}, {4, 5, 6, 7 }, ..., {60, 61, 62, 63} }` (thread is represented here by its id within hardware warp).
- Logical warp is a group of `WarpSize` consecutive threads from the same hardware warp.
- Supports non-commutative scan operators. However, a scan operator should be associative. When used with non-associative functions the results may be non-deterministic and/or vary in precision.
- Number of threads executing *warp_scan*'s function must be a multiple of `WarpSize`;
- All threads from a logical warp must be in the same hardware warp.

Examples

In the examples scan operation is performed on groups of 16 threads, each provides one `int` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 64.

```
__global__ void example_kernel(...)
{
    // specialize warp_scan for int and logical warp of 16 threads
    using warp_scan_int = rocprim::warp_scan<int, 16>;
    // allocate storage in shared memory
    __shared__ warp_scan_int::storage_type temp[4];

    int logical_warp_id = threadIdx.x/16;
    int value = ...;
```

(continues on next page)

(continued from previous page)

```
    // execute inclusive scan
    warp_scan_int().inclusive_scan(
        value, // input
        value, // output
        temp[logical_warp_id]
    );
    ...
}
```

Template Parameters

- **T** – - the input/output type.
- **WarpSize** – - the size of logical warp size, which can be equal to or less than the size of hardware warp (see rocprim::arch::wavefront::min_size()). Scan operations are performed separately within groups determined by WarpSize.

Public Types

using **storage_type** = typename base_type::storage_type

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **inclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs inclusive scan across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive min scan operations performed on groups of 32 threads, each provides one `float` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for float and logical warp of 32 threads
    using warp_scan_f = rocprim::warp_scan<float, 32>;
    // allocate storage in shared memory
```

(continues on next page)

(continued from previous page)

```
    __shared__ warp_scan_float::storage_type temp[8]; // 256/32 = 8

    int logical_warp_id = threadIdx.x/32;
    float value = ...;
    // execute inclusive min scan
    warp_scan_float().inclusive_scan(
        value, // input
        value, // output
        temp[logical_warp_id],
        rocprim::minimum<float>()
    );
    ...
}
```

If the input values across threads in a block/tile are `{1, -2, 3, -4, ..., 255, -256}`, then output values in the first logical warp will be `{1, -2, -2, -4, ..., -32}`, in the second: `{33, -34, -34, -36, ..., -64}` etc.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **inclusive_scan**(*T*, *T*&, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs inclusive scan across threads in a logical warp. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **inclusive_scan**(*T* input, *T* &output, *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs inclusive scan and reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive prefix sum operations performed on groups of 64 threads, each thread provides one `int` value. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for int and logical warp of 64 threads
    using warp_scan_int = rocprim::warp_scan<int, 64>;
    // allocate storage in shared memory
    __shared__ warp_scan_int::storage_type temp[4]; // 256/64 = 4

    int logical_warp_id = threadIdx.x/64;
    int input = ...;
    int output, reduction;
    // inclusive prefix sum
    warp_scan_int().inclusive_scan(
        input,
        output,
        reduction,
        temp[logical_warp_id]
    );
    ...
}
```

If the `input` values across threads in a block/tile are `{1, 1, 1, 1, ..., 1, 1}`, then `output` values in the every logical warp will be `{1, 2, 3, 4, ..., 64}`. The `reduction` will be equal `64`.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **reduction** – **[out]** - result of reducing of all `input` values in logical warp.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **inclusive_scan**(*T*, *T*&, *T*&, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs inclusive scan and reduction across threads in a logical warp. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T* input, *T* &output, *T* init, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs exclusive scan across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive min scan operations performed on groups of 32 threads, each provides one `float` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for float and logical warp of 32 threads
    using warp_scan_f = rocprim::warp_scan<float, 32>;
    // allocate storage in shared memory
    __shared__ warp_scan_float::storage_type temp[8]; // 256/32 = 8

    int logical_warp_id = threadIdx.x/32;
    float value = ...;
    // execute exclusive min scan
    warp_scan_float().exclusive_scan(
        value, // input
        value, // output
        100.0f, // init
        temp[logical_warp_id],
        rocprim::minimum<float>()
    );
    ...
}
```

If the initial value is `100` and input values across threads in a block/tile are `{1, -2, 3, -4, ..., 255, -256}`, then output values in the first logical warp will be `{100, 1, -2, -2, -4, ..., -30}`, in the second: `{100, 33, -34, -34, -36, ..., -62}` etc.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a logical warp.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T*, *T*&, *T*, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs exclusive scan across threads in a logical warp. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T* input, *T* &output, *T* init, *T* &reduction, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs exclusive scan and reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present exclusive prefix sum operations performed on groups of 64 threads, each thread provides one `int` value. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for int and logical warp of 64 threads
    using warp_scan_int = rocprim::warp_scan<int, 64>;
    // allocate storage in shared memory
    __shared__ warp_scan_int::storage_type temp[4]; // 256/64 = 4

    int logical_warp_id = threadIdx.x/64;
    int input = ...;
    int output, reduction;
    // exclusive prefix sum
    warp_scan_int().exclusive_scan(
        input,
        output,
        10, // init
        reduction,
        temp[logical_warp_id]
    );
    ...
}
```

If the initial value is `10` and `input` values across threads in a block/tile are `{1, 1, ..., 1, 1}`, then `output` values in every logical warp will be `{10, 11, 12, 13, ..., 73}`. The `reduction` will be 64.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **output** – **[out]** - reference to a thread output value. May be aliased with `input`.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a logical warp.
- **reduction** – **[out]** - result of reducing of all `input` values in logical warp. `init` value is not included in the reduction.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T*, *T*&, *T*, *T*&, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs exclusive scan and reduction across threads in a logical warp. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> void

Performs exclusive scan without an initial value across threads in a logical warp.

Template Parameters
BinaryFunction – binary function used for scan

Parameters

- **input** – Thread input value
- **output** – **[out]** Reference to thread output value. Each threads value for the scan will be written to it. May be aliased with `input`. The value written is unspecified for the first thread of each logical warp.
- **storage** – **[in]** Reference to a temporary storage object of type storage_type.
- **scan_op** – The function object used to combine elements used for the scan

template<class **BinaryFunction** = ::rocprim::plus<>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **exclusive_scan**(*T* input, *T* &output, *storage_type* &storage, *T* &reduction, *BinaryFunction* scan_op = *BinaryFunction*()) -> void

Performs exclusive scan and reduction without an initial value across threads in a logical warp.

Template Parameters
BinaryFunction – binary function used for scan

Parameters

- **input** – Thread input value
- **output** – **[out]** Reference to thread output value. Each threads value for the scan will be written to it. May be aliased with `input`. The value written is unspecified for the first thread of each logical warp.
- **reduction** – **[out]** Result of reducing of all `input` values in the logical warp.
- **storage** – **[in]** Reference to a temporary storage object of type storage_type.
- **scan_op** – The function object used to combine elements used for the scan

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **scan**(*T* input, *T* &inclusive_output, *T* &exclusive_output, *T* init, *storage_type* &storage, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Performs inclusive and exclusive scan operations across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present min inclusive and exclusive scan operations performed on groups of 32 threads, each provides one `float` value, result is returned using the same variable as for input. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for float and logical warp of 32 threads
    using warp_scan_f = rocprim::warp_scan<float, 32>;
    // allocate storage in shared memory
    __shared__ warp_scan_float::storage_type temp[8]; // 256/32 = 8

    int logical_warp_id = threadIdx.x/32;
    float input = ...;
    float ex_output, in_output;
    // execute exclusive min scan
    warp_scan_float().scan(
        input,
        in_output,
        ex_output,
        100.0f, // init
        temp[logical_warp_id],
        rocprim::minimum<float>()
    );
    ...
}
```

If the initial value is `100` and input values across threads in a block/tile are `{1, -2, 3, -4, ..., 255, -256}`, then `in_output` values in the first logical warp will be `{1, -2, -2, -4, ..., -32}`, in the second: `{33, -34, -34, -36, ..., -64}` and so forth, `ex_output` values in the first logical warp will be `{100, 1, -2, -2, -4, ..., -30}`, in the second: `{100, 33, -34, -34, -36, ..., -62}` etc.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **inclusive_output** – **[out]** - reference to a thread inclusive-scan output value.
- **exclusive_output** – **[out]** - reference to a thread exclusive-scan output value.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a logical warp.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **scan**(*T*, *T*&, *T*&, *T*, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) ->
typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()),
void>::type

Performs inclusive and exclusive scan operations across threads Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **scan**(*T* input, *T* &inclusive_output, *T* &exclusive_output, *T* init, *T* &reduction,
storage_type &storage, *BinaryFunction* scan_op = *BinaryFunction*()) ->
typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()),
void>::type

Performs inclusive and exclusive scan operations, and reduction across threads in a logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Examples

The examples present inclusive and exclusive prefix sum operations performed on groups of 64 threads, each thread provides one `int` value. Hardware warp size is 64. Block (tile) size is 256.

```
__global__ void example_kernel(...) // blockDim.x = 256
{
    // specialize warp_scan for int and logical warp of 64 threads
    using warp_scan_int = rocprim::warp_scan<int, 64>;
    // allocate storage in shared memory
    __shared__ warp_scan_int::storage_type temp[4]; // 256/64 = 4

    int logical_warp_id = threadIdx.x/64;
    int input = ...;
    int in_output, ex_output, reduction;
    // inclusive and exclusive prefix sum
    warp_scan_int().scan(
        input,
        in_output,
        ex_output,
        init,
        reduction,
        temp[logical_warp_id]
    );
    ...
}
```

If the initial value is `10` and `input` values across threads in a block/tile are `{1, 1, ..., 1, 1}`, then `in_output` values in every logical warp will be `{1, 2, 3, 4, ..., 63, 64}`, and `ex_output` values in every logical warp will be `{10, 11, 12, 13, ..., 73}`. The `reduction` will be 64.

Template Parameters

BinaryFunction – - type of binary function used for scan. Default type is rocprim::plus<T>.

Parameters

- **input** – **[in]** - thread input value.
- **inclusive_output** – **[out]** - reference to a thread inclusive-scan output value.

- **exclusive_output** – **[out]** - reference to a thread exclusive-scan output value.
- **init** – **[in]** - initial value used to start the exclusive scan. Should be the same for all threads in a logical warp.
- **reduction** – **[out]** - result of reducing of all `input` values in logical warp. `init` value is not included in the reduction.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.
- **scan_op** – **[in]** - binary operation function object that will be used for scan. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);` . The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::plus<*T*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **scan**(*T*, *T*&, *T*&, *T*, *T*&, *storage_type*&, *BinaryFunction* scan_op = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Performs inclusive and exclusive scan operations across threads Invalid Warp Size.

template<unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **broadcast**(*T* input, const unsigned int src_lane, *storage_type* &storage) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), *T*>::type

Broadcasts value from one thread to all threads in logical warp.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Parameters

- **input** – **[in]** - value to broadcast.
- **src_lane** – **[in]** - id of the thread whose value should be broadcasted
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **broadcast**(*T*, const unsigned int, *storage_type*&) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), *T*>::type

Broadcasts value from one thread to all threads in logical warp. Invalid Warp Size.

2.5.5 Sort

template<class **Key**, unsigned int **WarpSize** = arch::wavefront::min_size(), class **Value** = empty_type>

class **warp_sort** : private rocprim::detail::warp_sort_shuffle<*Key*, arch::wavefront::min_size(), empty_type>

The *warp_sort* class provides warp-wide methods for computing a parallel sort of items across thread warps. This class currently implements parallel bitonic sort, and only accepts warp sizes that are powers of two.

Overview

- `WarpSize` must be power of two.
- `WarpSize` must be equal to or less than the size of hardware warp (see rocprim::arch::wavefront::min_size()). If it is less, sort is performed separately within groups determined by WarpSize. For example, if `WarpSize` is 4, hardware warp is 64, sort will be performed in logical warps grouped like this: `{ {0, 1, 2, 3}, {4, 5, 6, 7 }, ..., {60, 61, 62, 63} }` (thread is represented here by its id within hardware warp).
- Accepts custom compare_functions for sorting across a warp.
- Number of threads executing *warp_sort*'s function must be a multiple of `WarpSize`.

Stability

warp_sort is **not stable**: it doesn't necessarily preserve the relative ordering of equivalent keys. That is, given two keys `a` and `b` and a binary boolean operation `op` such that:

- `a` precedes `b` in the input keys, and
- op(a, b) and op(b, a) are both false, then it is **not guaranteed** that `a` will precede `b` as well in the output (ordered) keys.

Example:

Every thread within the warp uses the *warp_sort* class by first specializing the *warp_sort* type, and instantiating an object that will be used to invoke a member function.

```
__global__ void example_kernel(...)
{
    const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    int value = input[i];
    rocprim::warp_sort<int, 64> wsort;
    wsort.sort(value);
    input[i] = value;
}
```

Below is a snippet demonstrating how to pass a custom compare function:

```
__device__ bool customCompare(const int& a, const int& b)
{
    return a < b;
}
...
__global__ void example_kernel(...)
{
    const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    int value = input[i];
    rocprim::warp_sort<int, 64> wsort;
    wsort.sort(value, customCompare);
    input[i] = value;
}
```

Template Parameters

- **Key** – Data type for parameter Key
- **WarpSize** – [optional] The number of threads in a warp

- **Value** – [optional] Data type for parameter Value. By default, it's empty_type

Public Types

typedef base_type::storage_type **storage_type**

Struct used to allocate a temporary memory that is required for thread communication during operations provided by related parallel primitive.

Depending on the implemention the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union with other storage types to increase shared memory reusability.

Public Functions

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* &thread_key, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort for any data type.

Template Parameters
BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – - input/output to pass to other threads
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort for any data type. Invalid Warp Size.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort for any data type.

Template Parameters
BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – - input/output keys to pass to other threads
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>

__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort for any data type. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* &thread_key, *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort for any data type using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int value = ...;
    using warp_sort_int = rp::warp_sort<int, 64>;
    warp_sort_int wsort;
    __shared__ typename warp_sort_int::storage_type storage;
    wsort.sort(value, storage);
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – - input/output to pass to other threads
- **storage** – - temporary storage for inputs
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key*&, *storage_type*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort for any data type using temporary storage. Invalid Warp Size.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort for any data type using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int value = ...;
    using warp_sort_int = rp::warp_sort<int, 64>;
    warp_sort_int wsort;
    __shared__ typename warp_sort_int::storage_type storage;
    wsort.sort(value, storage);
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – - input/output keys to pass to other threads
- **storage** – - temporary storage for inputs
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *storage_type*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort for any data type using temporary storage. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* &thread_key, *Value* &thread_value, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort by key for any data type.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – - input/output key to pass to other threads
- **thread_value** – - input/output value to pass to other threads
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>

__device__ inline auto **sort**(*Key*&, *Value*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort by key for any data type. Invalid Warp Size.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort by key for any data type.

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – - input/output keys to pass to other threads
- **thread_values** – - input/outputs values to pass to other threads
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort by key for any data type. Invalid Warp Size.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* &thread_key, *Value* &thread_value, *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort by key for any data type using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int value = ...;
    using warp_sort_int = rp::warp_sort<int, 64>;
    warp_sort_int wsort;
    __shared__ typename warp_sort_int::storage_type storage;
    wsort.sort(key, value, storage);
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_key** – - input/output key to pass to other threads
- **thread_value** – - input/output value to pass to other threads
- **storage** – - temporary storage for inputs
- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key*&, *Value*&, *storage_type*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort by key for any data type using temporary storage. Invalid Warp Size.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *storage_type* &storage, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* <= arch::wavefront::min_size()), void>::type

Warp sort by key for any data type using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    int value = ...;
    using warp_sort_int = rp::warp_sort<int, 64>;
    warp_sort_int wsort;
    __shared__ typename warp_sort_int::storage_type storage;
    wsort.sort(key, value, storage);
    ...
}
```

Template Parameters

BinaryFunction – - type of binary function used for sort. Default type is rocprim::less<T>.

Parameters

- **thread_keys** – - input/output keys to pass to other threads
- **thread_values** – - input/output values to pass to other threads
- **storage** – - temporary storage for inputs

- **compare_function** – - binary operation function object that will be used for sort. The signature of the function should be equivalent to the following: `bool f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<unsigned int **ItemsPerThread**, class **BinaryFunction** = ::rocprim::less<*Key*>, unsigned int **FunctionWarpSize** = *WarpSize*>
__device__ inline auto **sort**(*Key* (&thread_keys)[*ItemsPerThread*], *Value* (&thread_values)[*ItemsPerThread*], *storage_type*&, *BinaryFunction* compare_function = *BinaryFunction*()) -> typename std::enable_if<(*FunctionWarpSize* > arch::wavefront::min_size()), void>::type

Warp sort by key for any data type using temporary storage. Invalid Warp Size.

2.5.6 Shuffle

template<class **T**>
__device__ inline *T* **rocprim::warp_shuffle**(const *T* &input, const int src_lane, const int width = arch::wavefront::min_size())

Shuffle for any data type.

Each thread in warp obtains `input` from `src_lane`-th thread in warp. If `width` is less than arch::wavefront::min_size() then each subsection of the warp behaves as a separate entity with a starting logical lane id of 0. If `src_lane` is not in [0; `width`) range, the returned value is equal to `input` passed by the `src_lane modulo width` thread.

Note: The optional `width` parameter must be a power of 2; results are undefined if it is not a power of 2, or it is greater than arch::wavefront::min_size().

Parameters

- **input** – - input to pass to other threads
- **src_lane** – - warp if of a thread whose `input` should be returned
- **width** – - logical warp width

template<class **T**>
__device__ inline *T* **rocprim::warp_shuffle_down**(const *T* &input, const unsigned int delta, const int width = arch::wavefront::min_size())

Shuffle down for any data type.

`i`-th thread in warp obtains `input` from `i+delta`-th thread in warp. If is not in [0; `width`) range, thread's own `input` is returned.

Note: The optional `width` parameter must be a power of 2; results are undefined if it is not a power of 2, or it is greater than arch::wavefront::min_size().

Parameters

- **input** – - input to pass to other threads
- **delta** – - offset for calculating source lane id
- **width** – - logical warp width

template<class **T**>
__device__ inline *T* **rocprim::warp_shuffle_xor**(const *T* &input, const int lane_mask, const int width = arch::wavefront::min_size())

Shuffle XOR for any data type.

`i`-th thread in warp obtains `input` from `i^lane_mask`-th thread in warp.

Note: The optional `width` parameter must be a power of 2; results are undefined if it is not a power of 2, or it is greater than arch::wavefront::min_size().

Parameters

- **input** – - input to pass to other threads
- **lane_mask** – - mask used for calculating source lane id
- **width** – - logical warp width

2.5.7 Exchange

template<class **T**, unsigned int **ItemsPerThread**, unsigned int **WarpSize** = ::rocprim::arch::wavefront::min_size()>

class **warp_exchange**

The *warp_exchange* class is a warp level parallel primitive which provides methods for rearranging items partitioned across threads in a warp.

Overview

- The *warp_exchange* class supports the following rearrangement methods:
  - Transposing a blocked arrangement to a striped arrangement.
  - Transposing a striped arrangement to a blocked arrangement.

Examples

In the example an exchange operation is performed on a warp of 8 threads, using type `int` with 4 items per thread.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_per_
↪warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, items_per_thread,␣
↪threads_per_warp>;
    // allocate storage in shared memory
    __shared__ warp_exchange_int::storage_type storage[warps_per_block];

    int items[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.blocked_to_striped(items, items, storage[warp_id]);
    ...
}
```

Template Parameters

- **T** – - the input type.
- **ItemsPerThread** – - the number of items contributed by each thread.
- **WarpSize** – - the number of threads in a warp.

Public Types

using **storage_type** = storage_type_

Struct used to allocate a temporary memory that is required for thread communication during operations provided by the related parallel primitive.

Depending on the implementation the operations exposed by parallel primitive may require a temporary storage for thread communication. The storage should be allocated using keywords . It can be aliased to an externally allocated memory, or be a part of a union type with other storage types to increase shared memory reusability.

Public Functions

template<class **U**>
__device__ inline void **blocked_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a blocked arrangement of items to a striped arrangement across the warp, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_
↪per_warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, items_per_thread,␣
↪threads_per_warp>;
    // allocate storage in shared memory
    __shared__ warp_exchange_int::storage_type storage[warps_per_block];

    int items[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.blocked_to_striped(items, items, storage[warp_id]);
    ...
}
```

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**>
__device__ inline void **blocked_to_striped_shuffle**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a blocked arrangement of items to a striped arrangement across the warp, using warp shuffle operations. Uses an optimized implementation for when WarpSize is equal to ItemsPerThread. Caution: this API is experimental. Performance might not be consistent. ItemsPerThread must be a divisor of WarpSize.

Example.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_
↪per_warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, items_per_thread,␣
↪threads_per_warp>;

    int items[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.blocked_to_striped_shuffle(items, items);
    ...
}
```

Template Parameters
U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

template<class **U**>
__device__ inline void **striped_to_blocked**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], *storage_type* &storage)

Transposes a striped arrangement of items to a blocked arrangement across the warp, using temporary storage.

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_
↪per_warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, threads_per_warp,␣
↪items_per_thread>;
    // allocate storage in shared memory
    __shared__ warp_exchange_int::storage_type storage[warps_per_block];

    int items[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.striped_to_blocked(items, items, storage[warp_id]);
    ...
}
```

Template Parameters

U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

template<class **U**>
__device__ inline void **striped_to_blocked_shuffle**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*])

Transposes a striped arrangement of items to a blocked arrangement across the warp, using warp shuffle operations. Uses an optimized implementation for when WarpSize is equal to ItemsPerThread. Caution: this API is experimental. Performance might not be consistent. ItemsPerThread must be a divisor of WarpSize.

Example.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_
```

(continues on next page)

(continued from previous page)

```
↪per_warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, items_per_thread,␣
↪threads_per_warp>;

    int items[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.striped_to_blocked_shuffle(items, items);
    ...
}
```

Template Parameters

U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.

template<class **U**, class **OffsetT**>
__device__ inline void **scatter_to_striped**(const *T* (&input)[*ItemsPerThread*], *U* (&output)[*ItemsPerThread*], const *OffsetT* (&ranks)[*ItemsPerThread*], *storage_type* &storage)

Orders `input` values according to ranks using temporary storage, then writes the values to `output` in a striped manner. No values in `ranks` should exists that exceed `WarpSize*ItemsPerThread-1` .

Storage reusage

Synchronization barrier should be placed before `storage` is reused or repurposed: `__syncthreads()` or *rocprim::syncthreads()*.

Example.

```
__global__ void example_kernel(...)
{
    constexpr unsigned int threads_per_block = 128;
    constexpr unsigned int threads_per_warp  =   8;
    constexpr unsigned int items_per_thread  =   4;
    constexpr unsigned int warps_per_block   = threads_per_block / threads_
↪per_warp;
    const unsigned int warp_id = hipThreadIdx_x / threads_per_warp;
    // specialize warp_exchange for int, warp of 8 threads and 4 items per␣
↪thread
    using warp_exchange_int = rocprim::warp_exchange<int, items_per_thread,␣
↪threads_per_warp>;
    // allocate storage in shared memory
    __shared__ warp_exchange_int::storage_type storage[warps_per_block];

    int items[items_per_thread];
```

(continues on next page)

(continued from previous page)

```
    // data-type of `ranks` should be able to contain warp_size*items_per_
↪thread unique elements
    // unsigned short is sufficient for up to 1024*64 elements
    unsigned short ranks[items_per_thread];
    ...
    warp_exchange_int w_exchange;
    w_exchange.scatter_to_striped(items, items, ranks, storage[warp_id]);
    ...
}
```

Template Parameters

U – - [inferred] the output type.

Parameters

- **input** – **[in]** - array that data is loaded from.
- **output** – **[out]** - array that data is loaded to.
- **ranks** – **[in]** - array containing the positions.
- **storage** – **[in]** - reference to a temporary storage object of type storage_type.

2.6 Thread-Level Operations

- *Radix Key Encoder/Decoder*
- *Operators*
- *Load*
- *Reduce*
- *Scan*
- *Search*
- *Store*

2.6.1 Radix Key Encoder/Decoder

template<class **Key**, bool **Descending** = false, bool **is_fundamental** = ::rocprim::detail::radix_key_fundamental<*Key*>::value>
class **radix_key_codec** : protected rocprim::detail::radix_key_codec_base<*Key*>

Key encoder, decoder and bit-extractor for radix-based sorts.

Template Parameters

- **Key** – Type of the key used.
- **Descending** – Whether the sort is increasing or decreasing.

Public Types

using **bit_key_type** = typename base_type::bit_key_type

Type of the encoded key.

Public Static Functions

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline *bit_key_type* **encode**(*Key* key, *Decomposer* decomposer = {})

Encodes a key of type `Key` into `bit_key_type`.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters

- **key** – **[in]** Key to encode.
- **decomposer** – **[in]** [optional] Decomposer functor.

Returns
A `bit_key_type` encoded key.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline void **encode_inplace**(*Key* &key, *Decomposer* decomposer = {})

Encodes in-place a key of type `Key`.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters

- **key** – **[inout]** Key to encode.
- **decomposer** – **[in]** [optional] Decomposer functor.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline *Key* **decode**(*bit_key_type* bit_key, *Decomposer* decomposer = {})

Decodes an encoded key of type `bit_key_type` back into `Key`.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters

- **bit_key** – **[in]** Key to decode.
- **decomposer** – **[in]** [optional] Decomposer functor.

Returns
A `Key` decoded key.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline void **decode_inplace**(*Key* &key, *Decomposer* decomposer = {})

Decodes in-place an encoded key of type `Key`.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters

- **key** – **[inout]** Key to decode.
- **decomposer** – **[in]** [optional] Decomposer functor.

__host__ __device__ static inline unsigned int **extract_digit**(*bit_key_type* bit_key, unsigned int start, unsigned int radix_bits)

Extracts the specified bits from a given encoded key.

Parameters

- **bit_key** – **[in]** Encoded key.
- **start** – **[in]** Start bit of the sequence of bits to extract.
- **radix_bits** – **[in]** How many bits to extract.

Returns
Requested bits from the key.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline unsigned int **extract_digit**(*Key* key, unsigned int start, unsigned int radix_bits, *Decomposer* decomposer = {})

Extracts the specified bits from a given in-place encoded key.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters

- **key** – **[in]** Key.
- **start** – **[in]** Start bit of the sequence of bits to extract.
- **radix_bits** – **[in]** How many bits to extract.
- **decomposer** – **[in]** [optional] Decomposer functor.

Returns
Requested bits from the key.

template<class **Decomposer** = ::rocprim::identity_decomposer>
__host__ __device__ static inline *Key* **get_out_of_bounds_key**(*Decomposer* decomposer = {})

Gives the default value for out-of-bound keys of type `Key`.

Template Parameters
Decomposer – Being `Key` a fundamental type, `Decomposer` should be `identity_decomposer`. This is also the type by default.

Parameters
decomposer – **[in]** [optional] Decomposer functor.

Returns
Out-of-bound keys' value.

2.6.2 Operators

Equality

struct **equality**

Functor that tests for equality.

Public Functions

template<class **T**>
__host__ __device__ inline constexpr bool **operator()**(const *T* &a, const *T* &b) const

Invocation operator.

Inequality

struct **inequality**

Functor that tests for inequality.

Public Functions

template<class **T**>
__host__ __device__ inline constexpr bool **operator()**(const *T* &a, const *T* &b) const

Invocation operator.

template<class **EqualityOp**>

struct **inequality_wrapper**

Functor that tests for inequality using a user-supplied equality comparator.

Public Functions

__host__ __device__ inline **inequality_wrapper**(*EqualityOp* op)

Constructs the wrapper using the provided equality comparator.

Parameters

op – a binary equality comparator. The signature of the function should be equivalent to the following: `T f(const T &a, const T &b);`. The signature does not need to have `const &`, but function object must not modify the objects passed to it.

template<class **T**>
__host__ __device__ inline bool **operator()**(const *T* &a, const *T* &b)

Invocation operator.

Public Members

EqualityOp **op**

A binary equality comparator (see constructor for details).

Sum

struct **sum**

Functor that returns the sum of its arguments.

Public Functions

template<class **T**>
__host__ __device__ inline constexpr *T* **operator()**(const *T* &a, const *T* &b) const

Invocation operator.

Max/Min

struct **max**

Functor that returns the maximum of its arguments.

Public Functions

template<class **T**>
__host__ __device__ inline constexpr *T* **operator()**(const *T* &a, const *T* &b) const

Invocation operator.

struct **min**

Functor that returns the minimum of its arguments.

Public Functions

template<class **T**>
__host__ __device__ inline constexpr *T* **operator()**(const *T* &a, const *T* &b) const

Invocation operator.

ArgMax/ArgMin

struct **arg_max**

Functor that returns the "arg max" of the given key-value pairs.

The "arg max" of a key-value pair is defined as:

- b: if b's value is larger, or if the values are equal but b's key is smaller.
- a: otherwise

Public Functions

template<class **Key**, class **Value**>
__host__ __device__ inline constexpr *key_value_pair*<*Key*, *Value*> **operator()**(const *key_value_pair*<*Key*, *Value*> &a, const *key_value_pair*<*Key*, *Value*> &b) const

Invocation operator.

struct **arg_min**

Functor that returns the "arg min" of the given key-value pairs.

The "arg min" of a key-value pair is defined as:

- b: if b's value is smaller, or if the values are equal but b's key is smaller.
- a: otherwise

Public Functions

template<class **Key**, class **Value**>

__host__ __device__ inline constexpr *key_value_pair*<*Key*, *Value*> **operator()**(const *key_value_pair*<*Key*, *Value*> &a, const *key_value_pair*<*Key*, *Value*> &b) const

Invocation operator (see struct description for details)

2.6.3 Load

group **Thread Load Functions**

Enums

enum **cache_load_modifier**

These enum values are used to specify caching behaviour on load.

Values:

enumerator **load_default**

Default (no modifier)

enumerator **load_ca**

Cache at all levels.

enumerator **load_cg**

Cache at global level.

enumerator **load_cs**

Cache streaming (likely to be accessed once)

enumerator **load_cv**

Cache as volatile (including cached system lines)

enumerator **load_ldg**

Cache as texture.

enumerator **load_volatile**

Volatile (any memory space)

Functions

template<*cache_load_modifier* **MODIFIER** = load_default, typename **InputIteratorT**>
__device__ inline std::iterator_traits<*InputIteratorT*>::value_type **thread_load**(*InputIteratorT* itr)

Store data using the default load instruction. No support for cache modified stores yet.

Template Parameters

- **MODIFIER** – Value in enum for determine which type of cache store modifier to be used
- **InputIteratorT** – Type of Output Iterator

Parameters
itr – [in] Iterator to location where data is to be stored

Returns
Data that is loaded from memory

template<*cache_load_modifier* **MODIFIER** = load_default, typename **T**>
__device__ inline *T* **thread_load**(*T* *ptr)

Load data using the default load instruction. No support for cache modified loads yet.

Template Parameters

- **MODIFIER** – Value in enum for determine which type of cache store modifier to be used
- **T** – Type of Data to be loaded

Parameters
ptr – [in] - Pointer to data to be loaded

Returns
Data that is loaded from memory

2.6.4 Reduce

group **Thread Reduce Functions**

Functions

template<int **LENGTH**, typename **T**, typename **ReductionOp**, bool **NoPrefix** = false>
__device__ inline auto **thread_reduce**(*T* *input, *ReductionOp* reduction_op, *T* prefix = *T*(0)) ->
std::enable_if_t<!rocprim::detail::is_tuple<*T*>::value, *T*>

Carry out a reduction on an array of elements in one thread.

Template Parameters

- **LENGTH** – Length of the array to be reduced
- **T** – the input/output type
- **ReductionOp** – Binary Operation that used to carry out the reduction
- **NoPrefix** – Boolean, determining whether to have a initialization value for the reduction accumulator

Parameters

- **input** – [in] Pointer to the first element of the array to be reduced
- **reduction_op** – [in] Instance of the reduction operator functor
- **prefix** – [in] Value to be used as prefix, if NoPrefix is false

Returns
Value obtained from reduction of input array

template<int **LENGTH**, typename **T**, typename **ReductionOp**>
__device__ inline *T* **thread_reduce**(*T* (&input)[*LENGTH*], *ReductionOp* reduction_op, *T* prefix)

Carry out a reduction on an array of elements in one thread.

Template Parameters

- **LENGTH** – Length of the array to be reduced

- **T** – the input/output type
- **ReductionOp** – Binary Operation that used to carry out the reduction

Parameters

- **input** – [in] Pointer to the first element of the array to be reduced
- **reduction_op** – [in] Instance of the reduction operator functor
- **prefix** – [in] Value to be used as prefix

Returns
Value obtained from reduction of input array

template<int **LENGTH**, typename **T**, typename **ReductionOp**>
__device__ inline *T* **thread_reduce**(*T* (&input)[*LENGTH*], *ReductionOp* reduction_op)

Carry out a reduction on an array of elements in one thread.

Template Parameters

- **LENGTH** – Length of the array to be reduced
- **T** – the input/output type
- **ReductionOp** – Binary Operation that used to carry out the reduction

Parameters

- **input** – [in] Pointer to the first element of the array to be reduced
- **reduction_op** – [in] Instance of the reduction operator functor

Returns
Value obtained from reduction of input array

2.6.5 Scan

Exclusive Scan

group **Thread Exclusive Scan Functions**

Functions

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_exclusive**(*T* inclusive, *T* exclusive, *T* *input, *T* *output, *ScanOp* scan_op, Int2Type<*LENGTH*>)

Perform a sequential exclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – Length of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **inclusive** – [in] Initial value for inclusive aggregate
- **exclusive** – [in] Initial value for exclusive aggregate

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_exclusive**(*T* *input, *T* *output, *ScanOp* scan_op, *T* prefix, bool apply_prefix = true)

Perform a sequential exclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – Length of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator
- **prefix** – [in] Prefix to seed scan with
- **apply_prefix** – [in] Whether or not the calling thread should apply its prefix. (Handy for preventing thread-0 from applying a prefix.)

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_exclusive**(*T* (&input)[*LENGTH*], *T* (&output)[*LENGTH*], *ScanOp* scan_op, *T* prefix, bool apply_prefix = true)

Perform a sequential exclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – Length of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator
- **prefix** – [in] Prefix to seed scan with

- **apply_prefix** – [in] Whether or not the calling thread should apply its prefix. (Handy for preventing thread-0 from applying a prefix.)

Returns
Aggregate of the scan

Inclusive Scan

group **Thread Inclusive Scan Functions**

Functions

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_inclusive**(*T* inclusive, *T* *input, *T* *output, *ScanOp* scan_op, Int2Type<*LENGTH*>)

Perform a sequential exclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – Length of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **inclusive** – [in] Initial value for inclusive aggregate
- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_inclusive**(*T* *input, *T* *output, *ScanOp* scan_op)

Perform a sequential inclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – LengthT of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_inclusive**(*T* (&input)[*LENGTH*], *T* (&output)[*LENGTH*], *ScanOp* scan_op)

Perform a sequential inclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – LengthT of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_inclusive**(*T* *input, *T* *output, *ScanOp* scan_op, *T* prefix, bool apply_prefix = true)

Perform a sequential inclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – LengthT of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator
- **prefix** – [in] Prefix to seed scan with
- **apply_prefix** – [in] Whether or not the calling thread should apply its prefix. (Handy for preventing thread-0 from applying a prefix.)

Returns
Aggregate of the scan

template<int **LENGTH**, typename **T**, typename **ScanOp**>
__device__ inline *T* **thread_scan_inclusive**(*T* (&input)[*LENGTH*], *T* (&output)[*LENGTH*], *ScanOp* scan_op, *T* prefix, bool apply_prefix = true)

Perform a sequential inclusive prefix scan over `LENGTH` elements of the `input` array. The aggregate is returned.

Template Parameters

- **LENGTH** – LengthT of `input` and `output` arrays
- **T** – **[inferred]** The data type to be scanned.
- **ScanOp** – **[inferred]** Binary scan operator type having member `T operator()(const T &a, const T &b)`

Parameters

- **input** – [in] Input array
- **output** – [out] Output array (may be aliased to `input`)
- **scan_op** – [in] Binary scan operator
- **prefix** – [in] Prefix to seed scan with
- **apply_prefix** – [in] Whether or not the calling thread should apply its prefix. (Handy for preventing thread-0 from applying a prefix.)

Returns
Aggregate of the scan

2.6.6 Search

group **Thread Search Functions**

Functions

template<class **AIteratorT**, class **BIteratorT**, class **OffsetT**, class **CoordinateT**, class **BinaryFunction** = rocprim::less<typename std::iterator_traits<*AIteratorT*>::value_type>>
__host__ __device__ inline void **merge_path_search**(*OffsetT* diagonal, *AIteratorT* a, *BIteratorT* b, *OffsetT* a_len, *OffsetT* b_len, *CoordinateT* &path_coordinate, *BinaryFunction* compare_function = *BinaryFunction*())

Computes the begin offsets into A and B for the specific diagonal.

template<typename **InputIteratorT**, typename **OffsetT**, typename **T**>
__device__ inline *OffsetT* **lower_bound**(*InputIteratorT* input, *OffsetT* num_items, *T* val)

Returns the offset of the first value within `input` which does not compare less than `val`.

Template Parameters

- **InputIteratorT** – **[inferred]** Type of iterator for the input data to be searched
- **OffsetT** – **[inferred]** The data type of num_items
- **T** – **[inferred]** The data type of the input sequence elements

Parameters

- **input** – [in] Input sequence
- **num_items** – [in] Input sequence length
- **val** – [in] Search Key

Returns
Offset at which val was found

template<typename **InputIteratorT**, typename **OffsetT**, typename **T**>

__device__ inline *OffsetT* **upper_bound**(*InputIteratorT* input, *OffsetT* num_items, *T* val)

Returns the offset of the first value within `input` which compares greater than `val`.

Template Parameters

- **InputIteratorT** – **[inferred]** Type of iterator for the input data to be searched
- **OffsetT** – **[inferred]** The data type of num_items
- **T** – **[inferred]** The data type of the input sequence elements

Parameters

- **input** – [in] Input sequence
- **num_items** – [in] Input sequence length
- **val** – [in] Search Key

Returns

Offset at which val was found

template<int **MaxNumItems**, typename **InputIteratorT**, typename **OffsetT**, typename **T**>
__device__ inline *OffsetT* **static_upper_bound**(*InputIteratorT* input, *OffsetT* num_items, *T* val)

Returns the offset of the first value within `input` which compares greater than `val` computed as a statically unrolled loop.

Template Parameters

- **MaxNumItems** – The maximum number of items.
- **InputIteratorT** – **[inferred]** Type of iterator for the input data to be searched
- **OffsetT** – **[inferred]** The data type of num_items
- **T** – **[inferred]** The data type of the input sequence elements

Parameters

- **input** – [in] Input sequence
- **num_items** – [in] Input sequence length
- **val** – [in] Search Key

Returns

Offset at which val was found

2.6.7 Store

group **Thread Store Functions**

Enums

enum **cache_store_modifier**

These enum values are used to specify caching behaviour on store.

Values:

enumerator **store_default**

Default (no modifier)

enumerator **store_wb**

Cache write-back all coherent levels.

enumerator **store_cg**

Cache at global level.

enumerator **store_cs**

Cache streaming (likely to be accessed once)

enumerator **store_wt**

Cache write-through (to system memory)

enumerator **store_volatile**

Volatile shared (any memory space)

Functions

template<*cache_store_modifier* **MODIFIER** = store_default, typename **OutputIteratorT**, typename **T**>
__device__ inline void **thread_store**(*OutputIteratorT* itr, *T* val)

Store data using the default load instruction. No support for cache modified stores yet.

Template Parameters

- **MODIFIER** – Value in enum for determine which type of cache store modifier to be used
- **OutputIteratorT** – Type of Output Iterator
- **T** – Type of Data to be stored

Parameters

- **itr** – [in] Iterator to location where data is to be stored
- **val** – [in] Data to be stored

template<*cache_store_modifier* **MODIFIER** = store_default, typename **T**>
__device__ inline void **thread_store**(*T* *ptr, *T* val)

Store data using the default load instruction. No support for cache modified stores yet.

Template Parameters

- **MODIFIER** – Value in enum for determine which type of cache store modifier to be used
- **T** – Type of Data to be stored

Parameters

- **ptr** – [in] Pointer to location where data is to be stored
- **val** – [in] Data to be stored

2.7 Iterators

2.7.1 Constant

template<class **ValueType**, class **Difference** = std::ptrdiff_t>

class **constant_iterator**

A random-access input (read-only) iterator which generates a sequence of homogeneous values.

Overview

- A *constant_iterator* represents a pointer into a range of same values.
- Using it for simulating a range filled with a sequence of same values saves memory capacity and bandwidth.

Template Parameters

- **ValueType** – - type of value that can be obtained by dereferencing the iterator.
- **Difference** – - a type used for identify distance between iterators

Public Types

using **value_type** = typename std::remove_const<*ValueType*>::type

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = *value_type*

A reference type of the type iterated over (`value_type`). It's same as `value_type` since *constant_iterator* is a read-only iterator and does not have underlying buffer.

using **pointer** = const *value_type**

A pointer type of the type iterated over (`value_type`). It's `const` since *constant_iterator* is a read-only iterator.

using **difference_type** = *Difference*

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline explicit **constant_iterator**(const *value_type* value, const size_t index = 0)

Creates *constant_iterator* and sets its initial value to `value`.

Parameters

- **value** – initial value
- **index** – optional index for *constant_iterator*

__host__ __device__ inline *value_type* **operator[]**(*difference_type*) const

Constant_iterator is not writable, so we don't return reference, just something convertible to reference. That matches requirement of RandomAccessIterator concept

Note

For example, `constant_iterator(20)` generates the infinite sequence:

```
20
20
20
...
```

2.7.2 Counting

template<class **Incrementable**, class **Difference** = std::ptrdiff_t>

class **counting_iterator**

A random-access input (read-only) iterator over a sequence of consecutive integer values.

Overview

- A *counting_iterator* represents a pointer into a range of sequentially increasing values.
- Using it for simulating a range filled with a sequence of consecutive values saves memory capacity and bandwidth.

Template Parameters

- **Incrementable** – - type of value that can be obtained by dereferencing the iterator.
- **Difference** – - a type used for identify distance between iterators

Public Types

using **value_type** = typename std::remove_const<*Incrementable*>::type

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = *value_type*

A reference type of the type iterated over (`value_type`). It's same as `value_type` since *constant_iterator* is a read-only iterator and does not have underlying buffer.

using **pointer** = const *value_type**

A pointer type of the type iterated over (`value_type`). It's `const` since *counting_iterator* is a read-only iterator.

using **difference_type** = *Difference*

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline **~counting_iterator**() = default

Creates *counting_iterator* with its initial value initialized to its default value (usually 0).

__host__ __device__ inline explicit **counting_iterator**(const *value_type* value)

Creates *counting_iterator* and sets its initial value to `value_`.

Parameters

value – initial value

Note

For example, `counting_iterator(20)` generates the infinite sequence:

```
20
21
22
23
...
```

2.7.3 Transform

template<class **InputIterator**, class **UnaryFunction**, class **ValueType** = typename ::rocprim::invoke_result<*UnaryFunction*, typename std::iterator_traits<*InputIterator*>::value_type>::type>
class **transform_iterator**

A random-access input (read-only) iterator adaptor for transforming dereferenced values.

Overview

- A *transform_iterator* uses functor of type UnaryFunction to transform value obtained by dereferencing underlying iterator.
- Using it for simulating a range filled with results of applying functor of type `UnaryFunction` to another range saves memory capacity and/or bandwidth.

Template Parameters

- **InputIterator** – - type of the underlying random-access input iterator. Must be a random-access iterator.
- **UnaryFunction** – - type of the transform functor.
- **ValueType** – - type of value that can be obtained by dereferencing the iterator. By default it is the return type of `UnaryFunction`.

Public Types

using **value_type** = *ValueType*

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = const *value_type*&

A reference type of the type iterated over (`value_type`). It's `const` since *transform_iterator* is a read-only iterator.

using **pointer** = const *value_type**

A pointer type of the type iterated over (`value_type`). It's `const` since *transform_iterator* is a read-only iterator.

using **difference_type** = typename std::iterator_traits<*InputIterator*>::difference_type

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

using **unary_function** = *UnaryFunction*

The type of unary function used to transform input range.

Public Functions

__host__ __device__ inline **transform_iterator**(*InputIterator* iterator, *UnaryFunction* transform)

Creates a new *transform_iterator*.

Parameters

- **iterator** – input iterator to iterate over and transform.
- **transform** – unary function used to transform values obtained from range pointed by `iterator`.

Note

`transform_iterator(sequence, transform)` should generate the sequence:

```
transform(sequence(0))
transform(sequence(1))
...
```

Predicate

template<class **DataIterator**, class **PredicateDataIterator**, class **UnaryPredicate**>

class **predicate_iterator**

A random-access iterator which can discard values assigned to it upon dereference based on a predicate.

Overview

- *predicate_iterator* can be used to ignore certain input or output of algorithms.
- When writing to *predicate_iterator*, it will only write to the underlying iterator if the predicate holds. Otherwise it will discard the value.
- When reading from *predicate_iterator*, it will only read from the underlying iterator if the predicate holds. Otherwise it will return the default constructed value.

Template Parameters

- **DataIterator** – Type of the data iterator that will be forwarded upon dereference.

- **PredicateDataIterator** – Type of the test iterator used to test the predicate function.
- **UnaryPredicate** – Type of the predicate function that tests the test.

Public Types

using **value_type** = typename std::iterator_traits<*DataIterator*>::value_type

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = typename std::iterator_traits<*DataIterator*>::reference

A reference type of the type iterated over (`value_type`).

using **pointer** = typename std::iterator_traits<*DataIterator*>::pointer

A pointer type of the type iterated over (`value_type`).

using **difference_type** = typename std::iterator_traits<*DataIterator*>::difference_type

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline **predicate_iterator**(*DataIterator* data_iterator, *PredicateDataIterator* predicate_iterator, *UnaryPredicate* predicate)

Creates a new *predicate_iterator*.

Parameters

- **data_iterator** – The data iterator that will be forwarded whenever the predicate is true.
- **predicate_iterator** – The test iterator that is used to test the predicate on.
- **predicate** – Unary function used to select values obtained. from range pointed by `iterator`.

struct **proxy**

Assignable proxy for values in `DataIterator`.

Public Types

using **capture_t** = decltype(*std::declval<*DataIterator*>())

The return type on the dereference operator. This may be different than `reference`.

Public Functions

__host__ __device__ inline **proxy**(*capture_t* val, const bool keep)

Constructs a `proxy` object with the given reference and keep flag.

Parameters

- **val** – The value or reference to be captured.
- **keep** – Boolean flag that indicates whether to keep the reference.

__host__ __device__ inline *proxy* &**operator=**(const *value_type* &value)

Assigns a value to the held reference if the keep flag is `true`.

Parameters

value – The value to assign to the captured value.

Returns

A reference to the (possibly) modified `proxy` object.

__host__ __device__ inline **operator** ***value_type***() const

Converts the `proxy` to the underlying value type.

Returns

The referenced value or the default-constructed value.

Note

`predicate_iterator(sequence, test, predicate)` generates the sequence:

```
predicate(test[0]) ? sequence[0] : default
predicate(test[1]) ? sequence[1] : default
predicate(test[2]) ? sequence[2] : default
...
```

2.7.4 Pairing Values with Indices

template<class **InputIterator**, class **Difference** = std::ptrdiff_t, class **InputValueType** = typename std::iterator_traits<*InputIterator*>::value_type>
class **arg_index_iterator**

A random-access input (read-only) iterator adaptor for pairing dereferenced values with their indices.

Overview

- Dereferencing *arg_index_iterator* return a value of `key_value_pair<Difference, InputValueType>` type, which includes value from the underlying range and its index in that range.
- `std::iterator_traits<InputIterator>::value_type` should be convertible to `InputValueType`.

Template Parameters

- **InputIterator** – - type of the underlying random-access input iterator. Must be a random-access iterator.
- **Difference** – - type used for identify distance between iterators and as the index type in the output pair type (see `value_type`).
- **InputValueType** – - value type used in the output pair type (see `value_type`).

Public Types

using **value_type** = ::rocprim::*key_value_pair*<*Difference*, *InputValueType*>

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = const *value_type*&

A reference type of the type iterated over (`value_type`). It's `const` since *arg_index_iterator* is a read-only iterator.

using **pointer** = const *value_type**

A pointer type of the type iterated over (`value_type`). It's `const` since *arg_index_iterator* is a read-only iterator.

using **difference_type** = *Difference*

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline **arg_index_iterator**(*InputIterator* iterator, *difference_type* offset = 0)

Creates a new *arg_index_iterator*.

Parameters

- **iterator** – input iterator pointing to the input range.
- **offset** – index of the `iterator` in the input range.

Note

`arg_index_iterator(sequence)` generates the sequence of tuples:

```
(0, sequence[0])
(1, sequence[1])
...
```

2.7.5 Zip

template<class **IteratorTuple**>

class **zip_iterator**

TBD.

Overview

- TBD

Template Parameters

IteratorTuple – -

Public Types

using **reference** = typename detail::tuple_of_references<*IteratorTuple*>::type

A reference type of the type iterated over.

The type of the tuple made of the reference types of the iterator types in the IteratorTuple argument.

using **value_type** = typename detail::tuple_of_values<*IteratorTuple*>::type

The type of the value that can be obtained by dereferencing the iterator.

using **pointer** = *value_type**

A pointer type of the type iterated over (`value_type`).

using **difference_type** = typename std::iterator_traits<typename ::rocprim::*tuple_element*<0, *IteratorTuple*>::type>::difference_type

A type used for identify distance between iterators.

The difference_type member of *zip_iterator* is the difference_type of the first of the iterator types in the IteratorTuple argument.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline **zip_iterator**(*IteratorTuple* iterator_tuple)

Creates a new *zip_iterator*.

Parameters
iterator_tuple – tuple of iterators

Note

`zip_iterator(sequence_X, sequence_Y)` generates the sequence of tuples:

```
(sequence_X[0], sequence_Y[0])
(sequence_X[1], sequence_Y[1])
...
```

2.7.6 Discard

class **discard_iterator**

A random-access iterator which discards values assigned to it upon dereference.

Overview

- *discard_iterator* does not have any underlying array (memory) and does not save values written to it upon dereference.
- *discard_iterator* can be used to safely ignore certain output of algorithms, which saves memory capacity and/or bandwidth.

Public Types

using **value_type** = discard_value

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = discard_value

A reference type of the type iterated over (`value_type`).

using **pointer** = discard_value*

A pointer type of the type iterated over (`value_type`).

using **difference_type** = std::ptrdiff_t

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

__host__ __device__ inline **discard_iterator**(size_t index = 0)

Creates a new *discard_iterator*.

Parameters

index – - optional index of discard iterator (default = 0).

2.7.7 Texture Cache

template<class **T**, class **Difference** = std::ptrdiff_t>

class **texture_cache_iterator**

A random-access input (read-only) iterator adaptor for dereferencing array values through texture cache. This iterator is not functional on gfx94x architectures.

Overview

- A *texture_cache_iterator* wraps a device pointer of type T, where values are obtained by dereferencing through texture cache.
- Can be exchanged and manipulated within and between host and device functions.
- Can only be constructed within host functions, and can only be dereferenced within device functions.
- Accepts any data type from memory, and loads through texture cache.
- This iterator is not functional on gfx94x architectures, as native texture fetch functions are not supported in gfx94x.

Template Parameters

- **T** – - type of value that can be obtained by dereferencing the iterator.
- **Difference** – - a type used for identify distance between iterators.

Public Types

using **value_type** = typename std::remove_const<*T*>::type

The type of the value that can be obtained by dereferencing the iterator.

using **reference** = const *value_type*&

A reference type of the type iterated over (`value_type`).

using **pointer** = const *value_type**

A pointer type of the type iterated over (`value_type`).

using **difference_type** = *Difference*

A type used for identify distance between iterators.

using **iterator_category** = std::random_access_iterator_tag

The category of the iterator.

Public Functions

template<class **Qualified**>
inline hipError_t **bind_texture**(*Qualified* *ptr, size_t bytes = size_t(-1), size_t texture_offset = 0)

Creates a `hipTextureObject_t` and binds this iterator to it.

Template Parameters
Texture – data pointer type

Parameters

- **ptr** – - pointer to the texture data on the device
- **bytes** – - size of the texture data (in bytes)
- **texture_offset** – - an offset from ptr to load texture data from (Defaults to 0)

inline hipError_t **unbind_texture**()

Destroys the texture object that this iterator points at.

2.8 Intrinsics

2.8.1 Bitwise

__device__ inline int **rocprim::get_bit**(int x, int i)

Returns a single bit at 'i' from 'x'.

__device__ inline unsigned int **rocprim::bit_count**(unsigned int x)

Bit count.

Returns the number of bit of **x** set.

__device__ inline unsigned int **rocprim::bit_count**(unsigned long long x)

Bit count.

Returns the number of bit of **x** set.

__host__ __device__ inline unsigned int **rocprim::ctz**(unsigned int x)

Count trailing zeroes.

Count the number of consecutive 0-bits, starting from the least significant bit.

__host__ __device__ inline unsigned int **rocprim::ctz**(unsigned long long x)

Count trailing zeroes.

Count the number of consecutive 0-bits, starting from the least significant bit.

2.8.2 Warp size

__host__ __device__ inline constexpr unsigned int **rocprim::warp_size**()

Returns a number of threads in a hardware warp.

It is constant for a device.

> **Warning**
>
> This function will be removed in a future release.

__host__ inline hipError_t **rocprim::host_warp_size**(const int device_id, unsigned int &warp_size)

Returns a number of threads in a hardware warp for the actual device. At host side this constant is available at runtime only.

It is constant for a device.

Parameters

- **device_id** – - the device that should be queried.
- **warp_size** – - out parameter for the warp size.

Returns

hipError_t any error that might occur.

__host__ inline hipError_t **rocprim::host_warp_size**(const hipStream_t stream, unsigned int &warp_size)

Returns the number of threads in a hardware warp for the device associated with the stream. At host side this constant is available at runtime only.

It is constant for a device.

Parameters

- **stream** – - the stream, whose device should be queried.
- **warp_size** – - out parameter for the warp size.

Returns

hipError_t any error that might occur.

__device__ inline constexpr unsigned int **rocprim::device_warp_size**()

Returns a number of threads in a hardware warp for the actual target. At device side this constant is available at compile time.

It is constant for a device.

> **Warning**
>
> This function will be removed in a future release.

2.8.3 Lane and Warp ID

__device__ inline unsigned int **warp_id**()

Returns warp id in a block (tile).

2.8.4 Flat ID

__device__ inline unsigned int **flat_block_thread_id**()

Returns flat (linear, 1D) thread identifier in a multidimensional block (tile).

__device__ inline unsigned int **flat_block_id**()

Returns flat (linear, 1D) block identifier in a multidimensional grid.

2.8.5 Flat Size

__device__ inline unsigned int `rocprim::`**flat_block_size**()

Returns flat size of a multidimensional block (tile).

__device__ inline unsigned int `rocprim::`**flat_tile_size**()

Returns flat size of a multidimensional tile (block).

2.8.6 Synchronization

__device__ inline void `rocprim::`**syncthreads**()

Synchronize all threads in a block (tile)

__device__ inline void `rocprim::`**wave_barrier**()

Synchronize all threads in the wavefront.

Wait for all threads in the wavefront before continuing execution. Memory ordering is guaranteed by this function between threads in the same wavefront. Threads can communicate by storing to global / shared memory, executing *wave_barrier()* then reading values stored by other threads in the same wavefront.

wave_barrier() should be executed by all threads in the wavefront in convergence, this means that if the function is executed in a conditional statement all threads in the wave must enter the conditional statement.

> **Note**
>
> On SIMT architectures all lanes come to a convergence point simultaneously, thus no special instruction is needed in the ISA.

2.8.7 Active threads

__device__ inline lane_mask_type `rocprim::`**ballot**(int predicate)

Evaluate predicate for all active work-items in the warp and return an integer whose `i`-th bit is set if and only if `predicate` is `true` for the `i`-th thread of the warp and the `i`-th thread is active.

Parameters

predicate – - input to be evaluated for all active lanes

__device__ inline bool rocprim::**group_elect**(lane_mask_type mask)

Elect a single lane for each group in mask.

Parameters

mask – **[in]** bit mask of the lanes in the same group as the calling lane. The i-th bit should be set if lane i is in the same group as the calling lane.

Returns

true for one unspecified lane in the mask, false for everyone else. Returns false for all lanes not in any group, that is lanes passing 0 as mask.

Pre

The relation specified by mask must be symmetric and transitive, in other words: the groups should be consistent between threads.

__device__ inline unsigned int rocprim::**masked_bit_count**(lane_mask_type x, unsigned int add = 0)

Masked bit count.

For each thread, this function returns the number of active threads which have i-th bit of x set and come before the current thread.

template<unsigned int **LabelBits**>
__device__ inline lane_mask_type rocprim::**match_any**(unsigned int label, bool valid = true)

Group active lanes having the same bits of label.

Threads that have the same least significant LabelBits bits are grouped into the same group. Every lane in the warp receives a mask of all active lanes participating in its group.

Template Parameters

LabelBits – number of bits to compare between labels

Parameters

- **label** – **[in]** the label for the calling lane
- **valid** – **[in]** lanes passing false will be ignored for comparisons, such lanes will not be part of any group, and will always return an empty mask (0)

Returns

A bit mask of lanes sharing the same bits for label. The bit at index lane i's result includes bit j in the lane mask if lane j is part of the same group as lane i, i.e. lane i and j called with the same value for label.

__device__ inline lane_mask_type rocprim::**match_any**(unsigned int label, unsigned int label_bits, bool valid = true)

Group active lanes having the same bits of label.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Threads that have the same least significant label_bits bits are grouped into the same group. Every lane in the warp receives a mask of all active lanes participating in its group.

This overload does not accept a template parameter for label bits. It is passed as a function parameter instead.

Parameters

- **label** – **[in]** the label for the calling lane
- **label_bits** – **[in]** number of bits to compare between labels
- **valid** – **[in]** lanes passing false will be ignored for comparisons, such lanes will not be part of any group, and will always return an empty mask (0)

Returns

A bit mask of lanes sharing the same bits for `label`. The bit at index lane `i`'s result includes bit `j` in the lane mask if lane `j` is part of the same group as lane `i`, i.e. lane `i` and `j` called with the same value for label.

2.9 Implementing traits for custom types in rocPRIM

2.9.1 Overview

This interface is designed to enable users to provide additional type trait information to rocPRIM, facilitating better compatibility with custom types.

Accurately describing custom types is important for performance optimization and computational correctness.

Custom types that implement arithmetic operators can behave like built-in arithmetic types but might still be interpreted by rocPRIM algorithms as generic `struct` or `class` types.

The rocPRIM type traits interface lets users add custom trait information for their types, improving compatibility between these types and rocPRIM algorithms.

This interface is similar to operator overloading.

Traits should be implemented as required by specific algorithms. Some traits can't be defined if they can be inferred from others.

2.9.2 Interface

template<class **T**>

struct **define**

#include <type_traits_interface.hpp>

Overview

This template struct provides an interface for downstream libraries to implement type traits for their custom types. Users can utilize this template struct to define traits for these types. Users should only implement traits as required by specific algorithms, and some traits cannot be defined if they can be inferred from others. This API is not static because of ODR.

Example

The example below demonstrates how to implement traits for a custom floating-point type.

```
// Your type definition
struct custom_float_type
{};
// Implement the traits
template<>
struct rocprim::traits::define<custom_float_type>
{
    using is_arithmetic = rocprim::traits::is_arithmetic::values<true>;
    using number_format = rocprim::traits::number_format::values<traits::number_
↪format::kind::floating_point_type>;
    using float_bit_mask = rocprim::traits::float_bit_mask::values<uint32_t, 10,
↪ 10, 10>;
};
```

The example below demonstrates how to implement traits for a custom integral type.

```
// Your type definition
struct custom_int_type
{};
// Implement the traits
template<>
struct rocprim::traits::define<custom_int_type>
{
    using is_arithmetic = rocprim::traits::is_arithmetic::values<true>;
    using number_format = rocprim::traits::number_format::values<traits::number_
↪format::kind::integral_type>;
    using integral_sign = rocprim::traits::integral_sign::values
↪<traits::integral_sign::kind::signed_type>;
};
```

Template Parameters

T – The type for which you want to define traits.

template<class **T**>

struct **get**

#include <type_traits_interface.hpp> predef

Overview

This template struct is designed to allow rocPRIM algorithms to retrieve trait information from C++ build-in arithmetic types, rocPRIM types, and custom types. This API is not static because of ODR.

- All member functions are `compiled only when invoked`.
- Different algorithms require different traits.

Example

The following code demonstrates how to retrieve the traits of type `T`.

```
// Get the trait in a template parameter
template<class T, std::enable_if<rocprim::traits::get<T>().is_integral()>
↪::type* = nullptr>
void get_traits_in_template_parameter(){}
// Get the trait in a function body
template<class T>
void get_traits_in_function_body(){
    constexpr auto input_traits = rocprim::traits::get<InputType>();
    // Then you can use the member functinos
    constexpr bool is_arithmetic = input_traits.is_arithmetic();
}
```

Template Parameters

T – The type from which you want to retrieve the traits.

Public Functions

inline constexpr bool **is_arithmetic**() const

Get the value of trait *is_arithmetic*.

Returns

`true` if `std::is_arithmetic_v<T>` is `true`, or if type `T` is a rocPRIM arithmetic type, or if the *is_arithmetic* trait has been defined as `true`; otherwise, returns `false`.

inline constexpr bool **is_fundamental**() const

Get trait *is_fundamental*.

Returns

`true` if `T` is a fundamental type (that is, rocPRIM arithmetic type, void, or nullptr_t); otherwise, returns `false`.

inline constexpr bool **is_compound**() const

If `T` is fundamental type, then returns `false`.

Returns

`false` if `T` is a fundamental type (that is, rocPRIM arithmetic type, void, or nullptr_t); otherwise, returns `true`.

inline constexpr bool **is_floating_point**() const

To check if `T` is floating-point type.

Warning

You cannot call this function when *is_arithmetic()* returns `false`; doing so will result in a compile-time error.

inline constexpr bool **is_integral**() const

To check if `T` is integral type.

Warning

You cannot call this function when *is_arithmetic()* returns `false`; doing so will result in a compile-time error.

inline constexpr bool **is_signed**() const

To check if `T` is signed integral type.

Warning

You cannot call this function when *is_integral()* returns `false`; doing so will result in a compile-time error.

inline constexpr bool **is_unsigned**() const

To check if `T` is unsigned integral type.

> **Warning**
>
> You cannot call this function when *is_integral()* returns `false`; doing so will result in a compile-time error.

inline constexpr bool **is_scalar**() const

Get trait *is_scalar*.

Returns

`true` if `std::is_scalar_v<T>` is `true`, or if type `T` is a rocPRIM arithmetic type, or if the *is_scalar* trait has been defined as `true`; otherwise, returns `false`.

inline constexpr auto **float_bit_mask**() const

Get trait *float_bit_mask*.

> **Warning**
>
> You cannot call this function when *is_floating_point()* returns `false`; doing so will result in a compile-time error.

Returns

A constexpr instance of the specialization of *rocprim::traits::float_bit_mask::values* as provided in the traits definition of type T. If the *float_bit_mask* `trait` is not defined, it returns the rocprim::detail::float_bit_mask values, provided a specialization of `rocprim::detail::float_bit_mask<T>` exists.

2.9.3 Available traits

struct **is_arithmetic**

#include <type_traits_interface.hpp>

Definability

- **Undefinable**: For types with `predefined traits`.
- **Optional**: For other types.

How to define

```
using is_arithmetic = rocprim::traits::is_arithmetic::values<true>;
```

How to use

```
rocprim::traits::get<InputType>().is_arithmetic();
```

template<bool **Val**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr auto **value** = *Val*

This indicates if the `InputType` is arithmetic.

struct **is_scalar**

#include <type_traits_interface.hpp> Arithmetic types, pointers, member pointers, and null pointers are considered scalar types.

Definability

- **Undefinable**: For types with `predefined traits`.
- **Optional**: For other types. If both *is_arithmetic* and *is_scalar* are defined, their values must be consistent; otherwise, a compile-time error will occur.

How to define

```
using is_scalar = rocprim::traits::is_scalar::values<true>;
```

How to use

```
rocprim::traits::get<InputType>().is_scalar();
```

template<bool **Val**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr auto **value** = *Val*

This indicates if the `InputType` is scalar.

struct **number_format**

#include <type_traits_interface.hpp>

Definability

- **Undefinable**: For types with `predefined traits` and non-arithmetic types.
- **Required**: If you define *is_arithmetic* as `true`, you must also define this trait; otherwise, a compile-time error will occur.

How to define

```
using number_format = rocprim::traits::number_format::values<number_
↪format::kind::integral_type>;
```

How to use

```
rocprim::traits::get<InputType>().is_integral();
rocprim::traits::get<InputType>().is_floating_point();
```

Public Types

enum class **kind**

The kind enum that indecates the values avaliable for this trait.

Values:

enumerator **unknown_type**

enumerator **floating_point_type**

enumerator **integral_type**

template<*kind* **Val**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr auto **value** = *Val*

This indicates if the `InputType` is floating_point_type or integral_type or unknown_type.

struct **integral_sign**

#include <type_traits_interface.hpp>

Definability

- **Undefinable**: For types with `predefined traits`, non-arithmetic types and floating-point types.
- **Required**: If you define *`number_format`* as `number_format::kind::floating_point_type`, you must also define this trait; otherwise, a compile-time error will occur.

How to define

```
using integral_sign = rocprim::traits::integral_sign::values<traits::integral_
↪sign::kind::signed_type>;
```

How to use

```
rocprim::traits::get<InputType>().is_signed();
rocprim::traits::get<InputType>().is_unsigned();
```

Public Types

enum class **kind**

The kind enum that indecates the values avaliable for this trait.

Values:

enumerator **unknown_type**

enumerator **signed_type**

enumerator **unsigned_type**

template<*kind* **Val**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr auto **value** = *Val*

This indicates if the `InputType` is signed_type or unsigned_type or unknown_type.

struct **float_bit_mask**

#include <type_traits_interface.hpp>

Definability

- **Undefinable**: For types with `predefined traits`, non-arithmetic types and integral types.
- **Required**: If you define *number_format* as `number_format::kind::unknown_type`, you must also define this trait; otherwise, a compile-time error will occur.

How to define

```
using float_bit_mask = rocprim::traits::float_bit_mask::values<int,1,1,1>;
```

How to use

```
rocprim::traits::get<InputType>().float_bit_mask();
```

> **Warning**
>
> For some types, if this trait is not implemented in their traits definition, it will link to `rocprim::detail::float_bit_mask` to maintain compatibility with downstream libraries. However, this linkage will be removed in the next major release. Please ensure that these types are updated to the latest interface.

template<class **BitType**, *BitType* **SignBit**, *BitType* **Exponent**, *BitType* **Mantissa**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr *BitType* **sign_bit** = *SignBit*

Trait sign_bit for the `InputType`.

static constexpr *BitType* **exponent** = *Exponent*

Trait exponent for the `InputType`.

static constexpr *BitType* **mantissa** = *Mantissa*

Trait mantissa for the `InputType`.

struct **is_fundamental**

#include <type_traits_interface.hpp> The trait *is_fundamental* is undefinable, as it is the union of `std::is_fundamental` and *rocprim::traits::is_arithmetic*.

Definability

- **Undefinable**: If you attempt to define this trait in any form, a compile-time error will occur.

How to use

```
rocprim::traits::get<InputType>().is_fundamental();
rocprim::traits::get<InputType>().is_compound();
```

template<bool **Val**>

struct **values**

#include <type_traits_interface.hpp> Value of this trait.

Public Static Attributes

static constexpr auto **value** = *Val*

This indicates if the `InputType` is fundamental.

2.9.4 Type traits wrappers

template<class **T**>
struct **is_floating_point** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_floating_point()>

#include <type_traits_interface.hpp> An extension of `std::is_floating_point` that supports additional arithmetic types, including `rocprim::half`, `rocprim::bfloat16`, and any types with trait *rocprim::traits::number_format::values*<number_format::kind::floating_point_type> implemented.

template<class **T**>
struct **is_integral** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_integral()>

#include <type_traits_interface.hpp> An extension of `std::is_integral` that supports additional arithmetic types, including `rocprim::int128_t`, `rocprim::uint128_t`, and any types with trait *rocprim::traits::number_format::values*<number_format::kind::integral_type> implemented.

template<class **T**>

struct **is_arithmetic** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_arithmetic()>

#include <type_traits_interface.hpp> An extension of `std::is_arithmetic` that supports additional arithmetic types, including any types with trait *rocprim::traits::is_arithmetic::values*<`true`> implemented.

template<class **T**>
struct **is_fundamental** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_fundamental()>

#include <type_traits_interface.hpp> An extension of `std::is_fundamental` that supports additional arithmetic types, including any types with trait *rocprim::traits::is_arithmetic::values*<`true`> implemented.

template<class **T**>
struct **is_unsigned** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_unsigned()>

#include <type_traits_interface.hpp> An extension of `std::is_unsigned` that supports additional arithmetic types, including `rocprim::uint128_t`, and any types with trait *rocprim::traits::integral_sign::values*<`integral_sign::kind::unsigned_type`> implemented.

template<class **T**>
struct **is_signed** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_signed()>

#include <type_traits_interface.hpp> An extension of `std::is_signed` that supports additional arithmetic types, including `rocprim::int128_t`, and any types with trait *rocprim::traits::integral_sign::values*<`integral_sign::kind::signed_type`> implemented.

template<class **T**>
struct **is_scalar** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_scalar()>

#include <type_traits_interface.hpp> An extension of `std::is_scalar` that supports additional arithmetic types, including any types with trait *rocprim::traits::is_scalar::values*<`true`> implemented.

template<class **T**>
struct **is_compound** : public std::integral_constant<bool, ::rocprim::traits::*get*<*T*>().is_compound()>

#include <type_traits_interface.hpp> An extension of `std::is_scalar` that supports additional non-arithmetic types.

2.9.5 Types with predefined traits

template<>
struct **define**<float>

Public Types

using **float_bit_mask** = traits::*float_bit_mask*::*values*<uint32_t, 0x80000000, 0x7F800000, 0x007FFFFF>

template<>
struct **define**<double>

Public Types

using **float_bit_mask** = traits::*float_bit_mask*::*values*<uint64_t, 0x8000000000000000, 0x7FF0000000000000, 0x000FFFFFFFFFFFFF>

template<>
struct **define**<rocprim::bfloat16>

Public Types

using **is_arithmetic** = traits::*is_arithmetic*::*values*<true>

using **number_format** = traits::*number_format*::*values*<traits::*number_format*::*kind*::*floating_point_type*>

using **float_bit_mask** = traits::*float_bit_mask*::*values*<uint16_t, 0x8000, 0x7F80, 0x007F>

template<>
struct **define**<rocprim::half>

Public Types

using **is_arithmetic** = traits::*is_arithmetic*::*values*<true>

using **number_format** = traits::*number_format*::*values*<traits::*number_format*::*kind*::*floating_point_type*>

using **float_bit_mask** = traits::*float_bit_mask*::*values*<uint16_t, 0x8000, 0x7F80, 0x007F>

template<>
struct **define**<rocprim::int128_t> : public std::conditional_t<std::is_arithmetic<rocprim::int128_t>::value, traits::*define*<void>, detail::define_int128_t>

template<>
struct **define**<rocprim::uint128_t> : public std::conditional_t<std::is_arithmetic<rocprim::uint128_t>::value, traits::*define*<void>, detail::define_uint128_t>

2.10 Utility types

2.10.1 Double buffer

template<class **T**>

class **double_buffer**

This class provides an convenient way to do double buffering.

It tracks two versions of the buffer ("current" and "alternate"), which can be accessed with functions of the same name. You can also swap the buffers.

Public Functions

__host__ __device__ inline **double_buffer**()

Constructs an empty double buffer object, initializing the buffer pointers to nullptr.

__host__ __device__ inline **double_buffer**(*T* *current, *T* *alternate)

Contructs a double buffer object using the supplied buffer pointers.

Parameters

- **current** – Pointer to the buffer to designate as "current" (in use).
- **alternate** – Pointer to the buffer to designate as "alternate" (not in use)

__host__ __device__ inline *T* ***current**() const

Returns a pointer to the current buffer.

__host__ __device__ inline *T* ***alternate**() const

Returns a pointer to the alternate buffer.

__host__ __device__ inline void **swap**()

Swaps the current and alternate buffers.

2.10.2 Future value

template<typename **T**, typename **Iter** = *T**>

class **future_value**

Allows passing values that are not yet known at launch time as parameters to device algorithms.

Note

It is the users responsibility to ensure that value is available when the algorithm executes. This can be guaranteed with stream dependencies or explicit external synchronization.

```
int* intermediate_result = nullptr;
hipMalloc(reinterpret_cast<void**>(&intermediate_result), sizeof(intermediate_
↪result));
hipLaunchKernelGGL(compute_intermediate, blocks, threads, 0, stream, arg1, arg2,␣
↪intermediate_result);
const auto initial_value = rocprim::future_value<int>{intermediate_result};
rocprim::exclusive_scan(temporary_storage,
                        storage_size,
                        input,
                        output,
                        initial_value,
                        size);
hipFree(intermediate_result)
```

Template Parameters

- **T** –
- **Iter** –

Public Types

using **value_type** = *T*

The type of the value this future will store.

using **iterator_type** = *Iter*

An iterator type that can point at the `value_type`.

Public Functions

__host__ __device__ inline explicit **future_value**(const *Iter* iter)

Constructs a future value.

Parameters

iter – An iterator that will point to the value when it becomes available.

__host__ __device__ inline **operator** *T*()

Returns the value by dereferencing the iterator that the constructor was passed.

> **Note**
>
> The value must be available at the point this is called.

__host__ __device__ inline **operator** *T*() const

Returns the value by dereferencing the iterator that the constructor was passed.

> **Note**
>
> The value must be available at the point this is called.

2.10.3 Key-value pair

template<class **Key_**, class **Value_**>

struct **key_value_pair**

Convenience struct that allows you to store key-value pairs as a composite entity.

Public Types

using **key_type** = *Key_*

the key's type

using **value_type** = *Value_*

the value's type

Public Functions

__host__ __device__ inline **key_value_pair**(const *key_type* key, const *value_type* value)

Constructs a key-value pair using the supplied values.

__host__ __device__ inline bool **operator!=**(const *key_value_pair* &kvb)

Returns true if the given key-value pair is not equal to this one. Otherwise returns false.

Public Members

key_type **key**

the stored key

value_type **value**

the stored value

2.10.4 Tuple

template<class ...**Types**>

class **tuple**

Fixed-size collection of heterogeneous values.

> **See also**
>
> std::tuple

Template Parameters

Types... – the types (zero or more) of the elements that the tuple stores.

Pre

- For all types in `Types...` following operations should not throw exceptions: construction, copy and move assignment, and swapping.

Public Functions

__host__ __device__ inline constexpr **tuple**() noexcept

Default constructor. Performs value-initialization of all elements.

This overload only participates in overload resolution if:

- `std::is_default_constructible<Ti>::value` is `true` for all `i`.

__host__ __device__ inline **tuple**(const *tuple*&) = default

Implicitly-defined copy constructor.

__host__ __device__ inline **tuple**(*tuple*&&) = default

Implicitly-defined move constructor.

__host__ __device__ inline explicit **tuple**(const *Types*&... values)

Direct constructor. Initializes each element of the tuple with the corresponding input value.

This overload only participates in overload resolution if:

- `std::is_copy_constructible<Ti>::value` is `true` for all `i`.

template<class ...**UTypes**>
__host__ __device__ inline explicit **tuple**(*UTypes*&&... values) noexcept

Converting constructor. Initializes each element of the tuple with the corresponding value in `rocprim::detail::custom_forward<UTypes>(values)`.

This overload only participates in overload resolution if:

- `sizeof...(Types) == sizeof...(UTypes)`,
- `sizeof...(Types) >= 1`, and
- `std::is_constructible<Ti, Ui&&>::value` is `true` for all `i`.

template<class ...**UTypes**>
__host__ __device__ inline **tuple**(const *tuple*<*UTypes*...> &other) noexcept

Converting copy constructor. Initializes each element of the tuple with the corresponding value from `other`.

This overload only participates in overload resolution if:

- `sizeof...(Types) == sizeof...(UTypes)`,
- `sizeof...(Types) >= 1`, and
- `std::is_constructible<Ti, Ui&&>::value` is `true` for all `i`.

template<class ...**UTypes**>
__host__ __device__ inline **tuple**(*tuple*<*UTypes*...> &&other) noexcept

Converting move constructor. Initializes each element of the tuple with the corresponding value from `other`.

This overload only participates in overload resolution if:

- `sizeof...(Types) == sizeof...(UTypes)`,
- `sizeof...(Types) >= 1`, and
- `std::is_constructible<Ti, Ui&&>::value` is `true` for all `i`.

__host__ __device__ inline **~tuple**() noexcept = default

Implicitly-defined destructor.

tuple &**operator=**(const *tuple* &other) noexcept

Copy assignment operator.

Parameters
other – tuple to replace the contents of this tuple

tuple &**operator=**(*tuple* &&other) noexcept

Move assignment operator.

Parameters
other – tuple to replace the contents of this tuple

template<class ...**UTypes**>
tuple &**operator=**(const *tuple*<*UTypes*...> &other) noexcept

For all `i`, assigns `rocprim::get<i>(other)` to `rocprim::get<i>(*this)`.

Parameters
other – tuple to replace the contents of this tuple

template<class ...**UTypes**>
tuple &**operator=**(*tuple*<*UTypes*...> &&other) noexcept

For all `i`, assigns `rocprim::detail::custom_forward<Ui>(get<i>(other))` to `rocprim::get<i>(*this)`.

Parameters
other – tuple to replace the contents of this tuple

__host__ __device__ inline void **swap**(*tuple* &other) noexcept

Swaps the content of the tuple (`*this`) with the content `other`.

Parameters
other – tuple of values to swap

template<class **T**>

class **tuple_size**

Provides access to the number of elements in a tuple as a compile-time constant expression.

`tuple_size<T>` is undefined for types `T` that are not tuples.

template<size_t **I**, class **T**>

struct **tuple_element**

Provides compile-time indexed access to the types of the elements of the tuple.

`tuple_element<I, T>` is undefined for types `T` that are not tuples.

2.10.5 Uninitialized Array

template<typename **T**, unsigned int **Count**, size_t **Alignment** = alignof(*T*)>

class **uninitialized_array**

Provides indexed & typed access to uninitialized memory. To be used with `ROCPRIM_SHARED_MEMORY`

> **Note**
>
> This class should be used to ensure that writes to the uninitialized memory block occur only via placement new.

> **Note**
>
> This class is non-copyable.

> **Note**
>
> The recommended pattern for usage is to first fill the array via calls to `emplace`, then read-only via *`get_unsafe_array()`*. Writing to the array reference returned by *`get_unsafe_array()`* should be avoided, if possible.

> **Note**
>
> The value of `Alignment` MUST be a valid alignment for `T`.

Template Parameters

- **T** – The item type which is provided via the accessors.
- **Count** – The number of T items to store.
- **Alignment** – The alignment of the backing storage, in bytes.

Public Types

using **c_array_t** = *T*[*Count*]

Type of the represented C array.

Public Functions

__host__ __device__ **uninitialized_array**(*uninitialized_array*&&) = default

Default move constructor.

__host__ __device__ *uninitialized_array* &**operator=**(*uninitialized_array*&&) = default

Default move assignment.

template<typename ...**Args**>
__host__ __device__ inline *T* &**emplace**(const unsigned int index, *Args*&&... args)

Constructs a value in-place at the specified array index.

> **Note**
>
> This function calls the constructor of `T` with the specified arguments. If an instance of `T&` is passed, the copy constructor is called, whereas in the case of a `T&&`, the move constructor is called.

> **Note**
>
> If an object is created at the same index more than once, the old object's destructor is **not** called. If `T` is not trivially destructible, the behaviour is undefined.

Template Parameters
Args – The types of the argument values.

Parameters

- **index** – The index in the array where the new object is constructed.
- **args** – The arguments to call the constructor with.

Returns
A reference to the newly constructed element.

__host__ __device__ inline *c_array_t* &**get_unsafe_array**()

Returns a reference to the underlying memory as a typed array.

> **Note**
>
> Manipulating items in the returned array reference at indices that were not previously filled by calls to `emplace` MUST be avoided.

2.11 Acknowledgements

The following contributors helped to make this documentation better:

- v01dXYZ has proposed a new structure for the documentation.

CHAPTER

THREE

LICENSE

INDEX

S

T

U

W