**Table of Contents**

[1. About this document 1](#_Toc441603812)

[2. HykSort 1](#_Toc441603813)

[2.1. For Normal primitive types 2](#_Toc441603814)

[2.2. For Key-value data type 4](#_Toc441603815)

[3. HyperQuickSort 6](#_Toc441603816)

[3.1. For Non-key-value data type 6](#_Toc441603817)

[3.2. For Key-value data type 8](#_Toc441603818)

[4. SampleSort 10](#_Toc441603819)

[4.1. For Non-key-value data type 10](#_Toc441603820)

[4.2. For Key-value data type 11](#_Toc441603821)

[5. About ParUtils::ArrangeValues() function 14](#_Toc441603822)

# 1. About this document

This document describes main process flows of the gpusort library. The gpusort library includes three sorting algorithm: HykSort, HyperQuickSort and SampleSort. Each algorithm supports sorting with three types of data:

* Normal keys data (primitive types).
* User-defined data.
* Key-value data.

The following sections will describe more clearly how each algorithm does it’s functions.

# 2. HykSort

This section list all tools and libraries which are required to compile and install the gpusort library from source successfully.

## 2.1. For Normal primitive types

This graph below describes a process flow of the HykSort algorithm for primitive data types:

**HykSortHelper::Sort** (keys\_ptr)

Input data (keys\_ptr)

Sorted keys\_ptr

**SampleSort::Sort**(keys\_ptr)

npes and kway are both a power of 2

Yes

No

npes is the number of processes

Details of **HykSortHelper::Sort()** function:

Input data (keys\_ptr)

Is using gpu?

Yes

No

Split items into chunks rely on available memory of the current GPU device

Using **thrust::sort** () to sort each chunk in turn

Using **CudaUtils::ParallelMergeArrays** () to merge all chunks with **OmpUtils::Merge**()

**CudaUtils::Sort**()

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks with **std::merge**()

**OmpUtils::MergeSort** ()

Select a GPU device rely on rank of the current process

**SelectSplitters**()

Sorted keys\_ptr

**TransferData** ()

**Merge** ()

Split MPI communicator

npes > 1

Yes

No

The number of processes in the current communicator

## 2.2. For Key-value data type

This graph below describes a process flow of the HykSort algorithm for Key-value data type:

**HykSortHelper::SortByKey**(keys\_ptr, global\_idx\_ptr)

Input data (keys\_ptr, values\_ptrs)

**ParUtils::ArrangeValues**(global\_idx\_ptr, values\_ptr)

values1\_ptr  
values2\_ptr

…

valuesN\_ptr

Sorted keys\_ptr, values\_ptrs

Sorted global\_idx\_ptr

**SampleSort::SortByKey**(keys\_ptr, values\_ptrs)

npes and kway are both a power of 2

Yes

No

npes is the number of processes

Details of **HykSortHelper::SortByKey()** function:

Input data (keys\_ptr, values\_ptr)

Sorted keys\_ptr, values\_ptr

Is using gpu?

Yes

No

Split items into chunks rely on available memory of the current GPU device

Using **thrust::sort\_by\_key** () to sort each chunk in turn

Using **CudaUtils::ParallelMergeArrays** () to merge all chunks of items with **OmpUtils::Merge**()

Create a vector containing all paired elements of keys\_ptr and values\_ptr:

**std::vector**<**std::pair**<K, V>> items

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks

**OmpUtils::MergeSort** ()

Select a GPU device rely on rank of the current process

Output each chunk to a vector containing all paired elements of sorted keys and values:

**std::vector**<**std::pair**<K, V>> items

**SelectSplitters**()

**TransferData** ()

**Merge** ()

Split MPI communicator

npes > 1

Yes

No

The number of processes in the current communicator

**CudaUtils::SortByKey**()

# 3. HyperQuickSort

## 3.1. For Non-key-value data type

This graph below describes a process flow of the HyperQuickSort algorithm for Non-key-value data types:

**HyperQuickSortHelper::Sort** (keys\_ptr)

Input data (keys\_ptr)

Sorted keys\_ptr

**SampleSort::Sort** (keys\_ptr)

npes is a power of 2

Yes

No

The number of processes

Details of **HyperQuickSortHelper::Sort()** function:

Input data (keys\_ptr)

Sorted keys\_ptr

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks with **std::merge**()

**OmpUtils::MergeSort** ()

**SelectSplitters**()

**TransferData** ()

**Merge** ()

Split MPI communicator

npes > 1

Yes

No

The number of processes in the current communicator

## 3.2. For Key-value data type

This graph below describes a process flow of the HyperQuickSort algorithm for Key-value data types:

**HyperQuickSortHelper::SortByKey**(keys\_ptr, global\_idx\_ptr)

Input data (keys\_ptr, values\_ptrs)

**ParUtils::ArrangeValues**(global\_idx\_ptr, values\_ptr)

values1\_ptr  
values2\_ptr

…

valuesN\_ptr

Sorted keys\_ptr, values\_ptrs

Sorted global\_idx\_ptr

**SampleSort::SortByKey**(keys\_ptr, values\_ptrs)

npes is a power of 2

Yes

No

The number of processes

Details of **HyperQuickSortHelper::SortByKey()** function:

Input data (keys\_ptr, values\_ptr)

Create a vector containing all paired elements of keys\_ptr and values\_ptr:

**std::vector**<**std::pair**<K, V>> items

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks with **std::merge**()

**OmpUtils::MergeSort** ()

Sorted keys\_ptr, values\_ptr

**SelectSplitters**()

**TransferData** ()

**Merge** ()

Split MPI communicator

npes > 1

Yes

No

The number of processes in the current communicator

# 4. SampleSort

## 4.1. For Non-key-value data type

This graph below describes a process flow of the SampleSort algorithm for Non-key-value data types:

**SampleSortHelper::Sort** (keys\_ptr)

Input data (keys\_ptr)

Sorted keys\_ptr

Details of **SampleSortHelper::Sort()** function:

Input data (keys\_ptr)

Sorted keys\_ptr

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks with **std::merge**()

**OmpUtils::MergeSort** ()

**SelectSplitters**()

**TransferData** ()

**Merge** data

npes > 1

Yes

No

The number of processes in the current communicator

## 4.2. For Key-value data type

This graph below describes a process flow of the SampleSort algorithm for Key-value data type:

**SampleSortHelper::SortByKey**(keys\_ptr, global\_idx\_ptr)

Input data (keys\_ptr, values\_ptrs)

**ParUtils::ArrangeValues**(global\_idx\_ptr, values\_ptr)

values1\_ptr  
values2\_ptr

…

valuesN\_ptr

Sorted keys\_ptr, values\_ptrs

Sorted global\_idx\_ptr

Details of **SampleSortHelper::SortByKey()** function:

Input data (keys\_ptr, values\_ptr)

Sorted keys\_ptr, values\_ptr

Create a vector containing all paired elements of keys\_ptr and values\_ptr:

**std::vector**<**std::pair**<K, V>> items

Split items into p chunks

(p is the number of threads)

Using **std::sort** () to sort all chunks

Using **OmpUtils::Merge** ()

to merge all chunks with **std::merge**()

**OmpUtils::MergeSort** ()

**SelectSplitters**()

**TransferData** ()

**Merge** data

npes > 1

Yes

No

The number of processes in the current communicator

# 5. About ParUtils::ArrangeValues() function

This functions sorts a value vector base on sorted global indexes. Main steps of this function are listed below:

* **Input**:
* *global\_idx\_ptr*: A vector containing sorted global indexes by input keys.
* *values\_ptr*: A value vector to sort base on the global indexes.
* **Step 1**: Split global\_idx\_ptr into N chunks. Let kMaxSend = size of each chunk. Let *loop = 1*.

This will guarantee that size of transferring data among processes is not too large on each loop.

* **Step 2**: Base on global indexes on this current chunk, find all senders who have values data to send to current process. With each sender found, create a corresponding vector of pair objects in form: *<remote\_idx, local\_idx>*. With *remote\_idx* is index of to-be-send data element in the sender’s original value vector, *local\_idx* is index in the local value vector (current process) to store the data element in.
* **Step 3**: Synchronize send and receive information on both senders and receivers processes. That information includes: how many data elements to transfer between a pair of two processes, each process sends its’ created vectors on Step 2 to corresponding processes (which are called senders on Step 2).
* **Step 4**: Each process which has received data information on Step 3 will send its’ relevant data elements to corresponding processes.
* **Step 5**: Each process which has received data on Step 4 will store the data to destination value vector.
* **Step 6**: *loop = loop + 1.* If *loop ≤ N* then go to Step 2.
* **Output**: Sorted value vector.