INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

ICPP/2021/CHICAGO/USA

acm In-Cooperation

sighpc

AUGUST 9-12, 2021

Efficient GPU-Implementation for Integer Sorting Based on Histogram and Prefix-Sums

Seiya Kozakai† Noriyuki Fujimoto†† Koichi Wada†

†Hosei University (Tokyo, Japan)

††Osaka Prefecture University (Osaka, Japan)

Introduction

- Background to this experiment.

CUB sort

Sorting provided by the CUB library included in the CUDA toolkit.

Highly optimized for GPUs so fast.

I want to devise an algorithm that is faster than these sorts.

Motivation for research

- CUB sorting is a general-purpose sorting algorithm.

- An algorithm that specializes in integer sorting may be able to devise a faster algorithm.

Integer sorting

Sort input data restricted to non-negative integers greater than or equal to minVal(=0) and less than maxVal.

Research content

- Implementation of integer sorting algorithm based on Histogram (H) and Prefix-sums(P) on GPU.

H-P sort

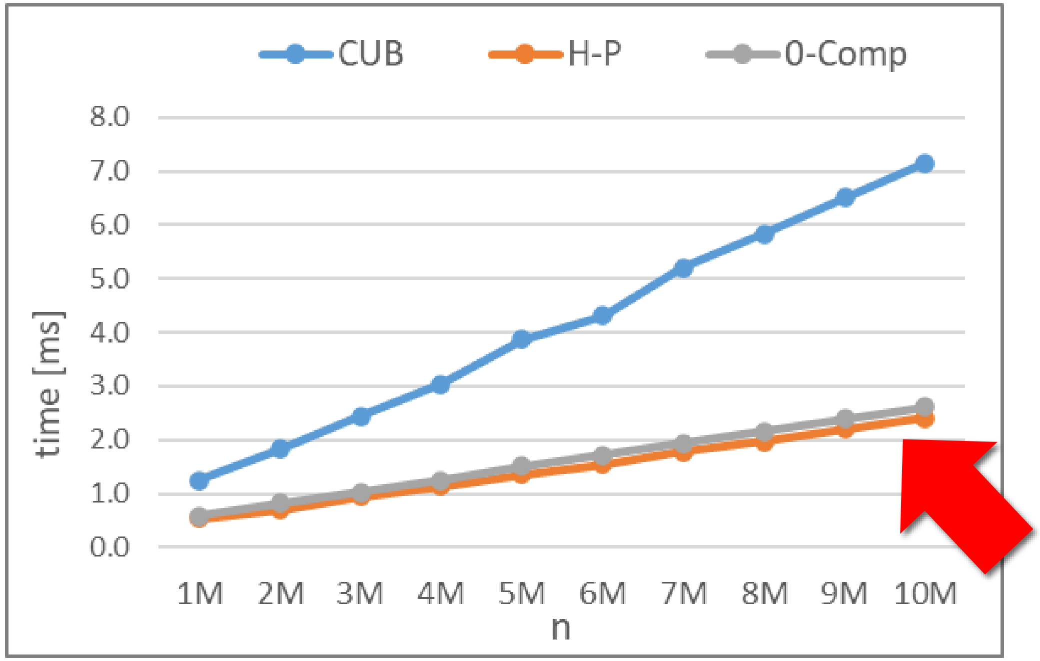
Integer sorting algorithm based on Histogram and Prefix sums, which was devised for operation on PRAM.

※ Eisenstat, S. C.: $O(\log n)$ algorithms on a Sum-CRCW PRAM, *Computing*,Vol. 79, pp. 93–97 (2007).

0-Compressed H-P sort

Newly devised algorithm for speeding up by compressing the Histogram.

How efficient is H-P sort?

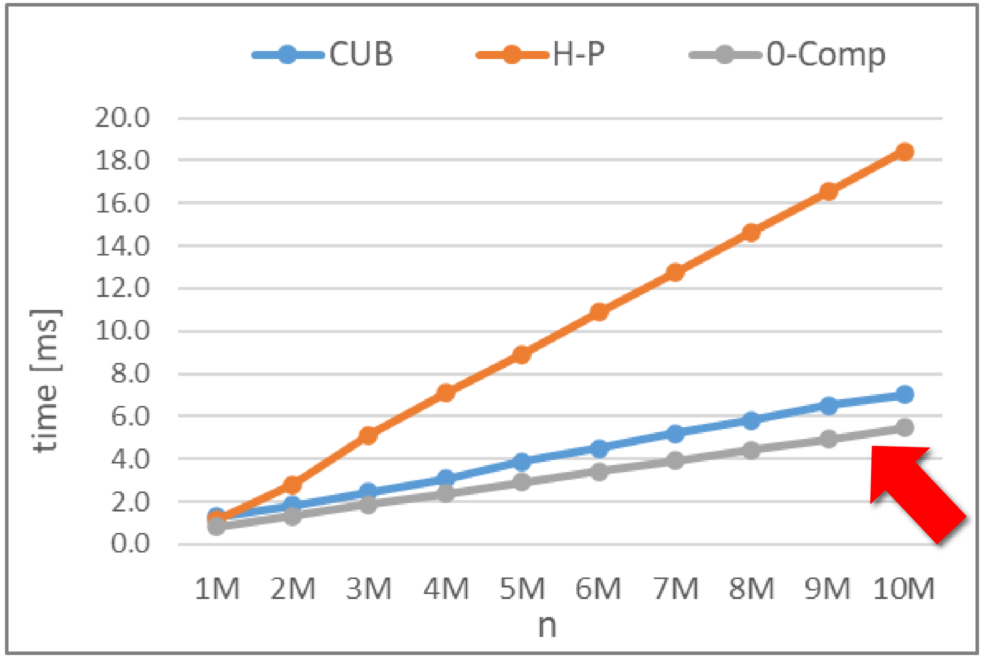


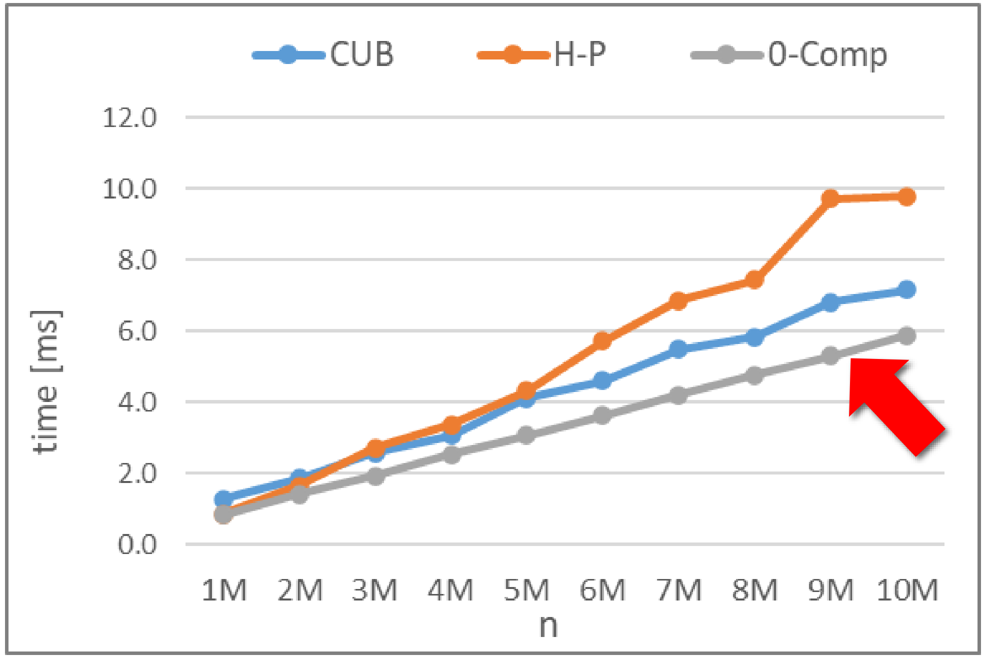
- $n = 1\text{M}\sim 10\text{M},\ minVal = 0,\ maxVal = n/50$
- For small $maxVal$, H-P sort is faster than CUB sort.

H-P sort is up to 2.97 times faster than the CUB sort.

How efficient is 0-Compressed H-P sort?

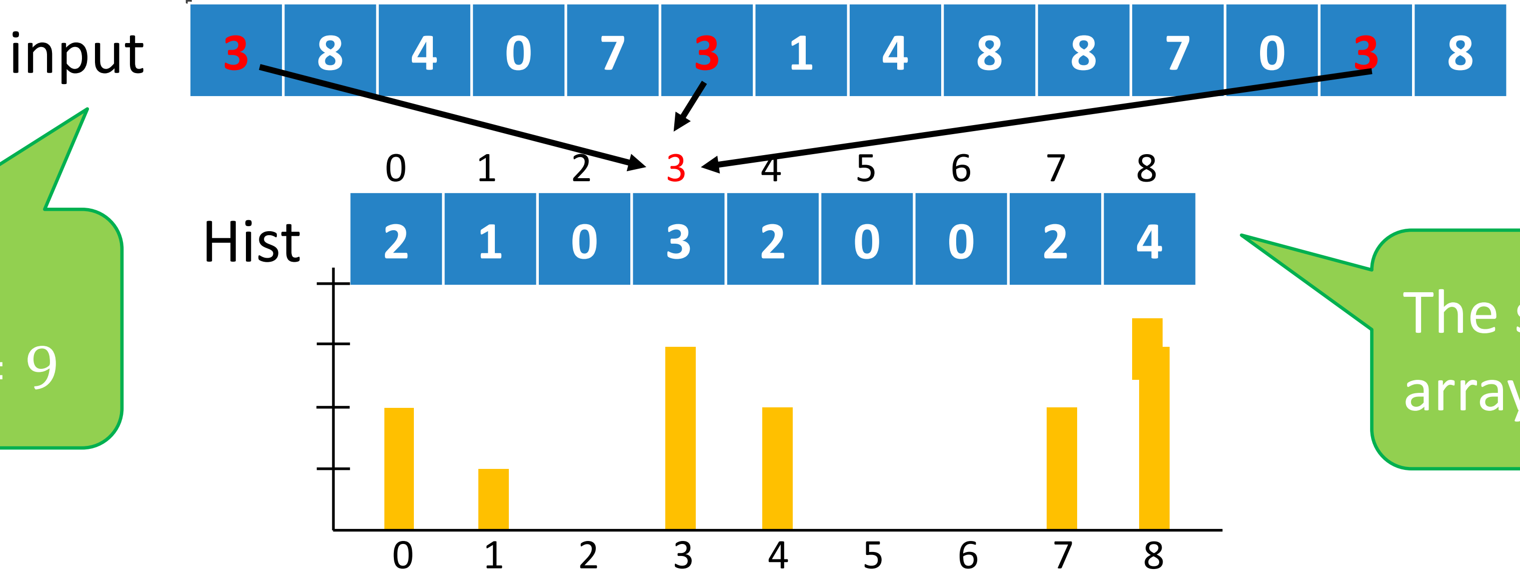
- $n = 1\text{M} \sim 10\text{M},\ minVal = 0,\ maxVal = n$(For large $maxVal$)
- The number of kinds of input data is 100 or 1000.





Histogram

- Frequency distribution table that counts the number of elements of the input value.
- Computational complexity of histogram generation: $O\ (n)$ n is the number of input data.

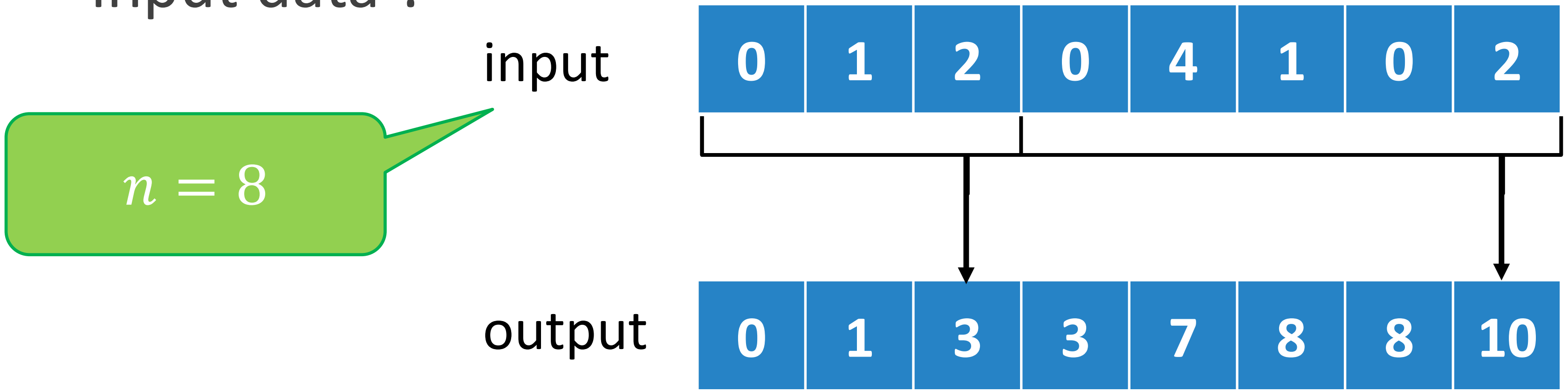


$n = 14$
$maxVal = 9$

The size of the output array will be $maxVal$

Prefix sums

- Output the sum of the 0'th to k'th elements of the input array to the k'th element of the output array.
- Computational complexity of prefix sum: $O\ (n)$ $\quad n$ is "number of input data".



$$\text{output}[\mathrm{k}] = input[0] + input[1] + \cdots + input[k]$$

HP sort algorithm

Parameters

n : Number of input data

$maxVal$: Input data interval $[0, maxVal - 1]$

Array used

Input array x (size : n), output array y (size : n)

Work array A (size : $maxVal$), $\mathrm{A_p}$ (size : $maxVal$), B (size : $n + 1$)

HP sort algorithm

Input array x $(\text{size}: n)$, output array y $(\text{size}: n)$

Work array A $(\text{size}: maxVal)$, $\mathrm{A_p}$ $(\text{size}: maxVal)$, B $(\text{size}: n+1)$

① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate $\mathrm{A_p}$ ($maxVal$)

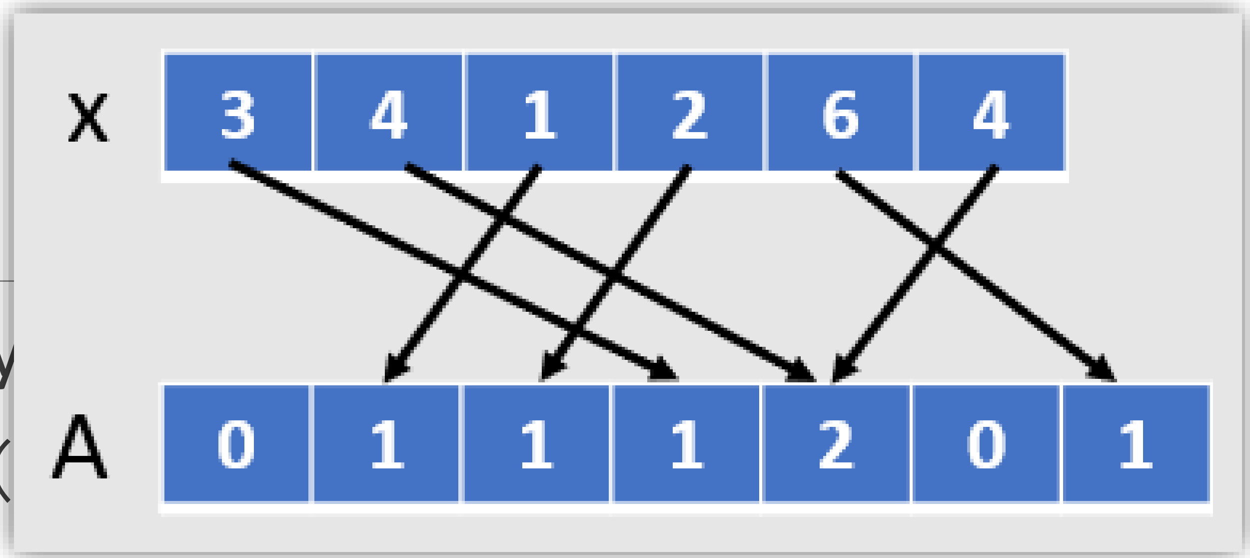
③ Generate B (n + 1), which is a Histogram of $\mathrm{A_p}$

④ Apply Prefix sums to B to generate y (n)

HP sort algorithm

Input array x(size: n), output array

Work array A(size : $maxVal$), $\mathrm{A_p}$ (



① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate $\mathrm{A_p}$ ($maxVal$)

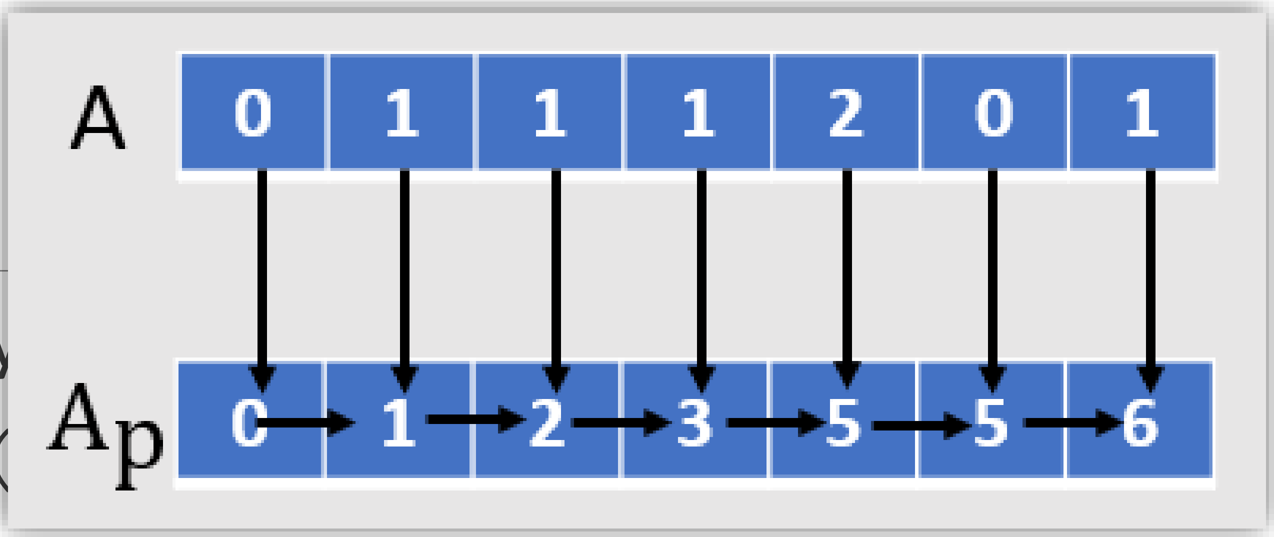
③ Generate B ($n + 1$), which is a Histogram of $\mathrm{A_p}$

④ Apply Prefix sums to B to generate y (n)

HP sort algorithm

Input array x(size : n), output array

Work array A(size : $maxVal$), $\mathrm{A_p}$ (



① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate $\mathrm{A_p}$ ($maxVal$)

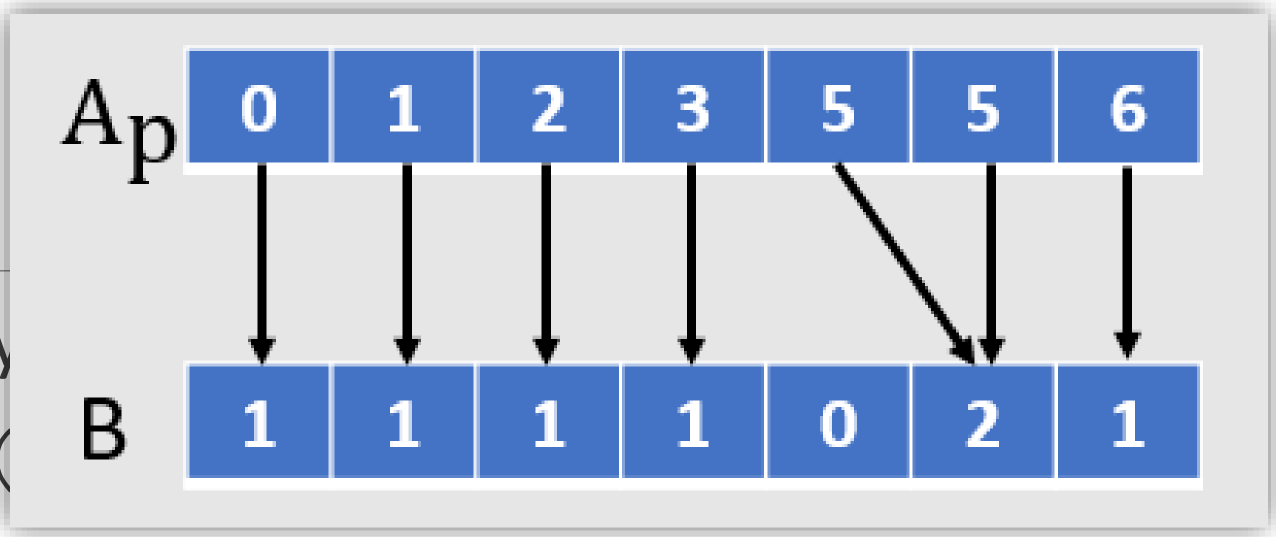
③ Generate B ($n + 1$), which is a Histogram of $\mathrm{A_p}$

④ Apply Prefix sums to B to generate y (n)

HP sort algorithm

Input array x(size : n), output array

Work array A(size : $maxVal$), $\mathrm{A_p}$ (



① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate $\mathrm{A_p}$ ($maxVal$)

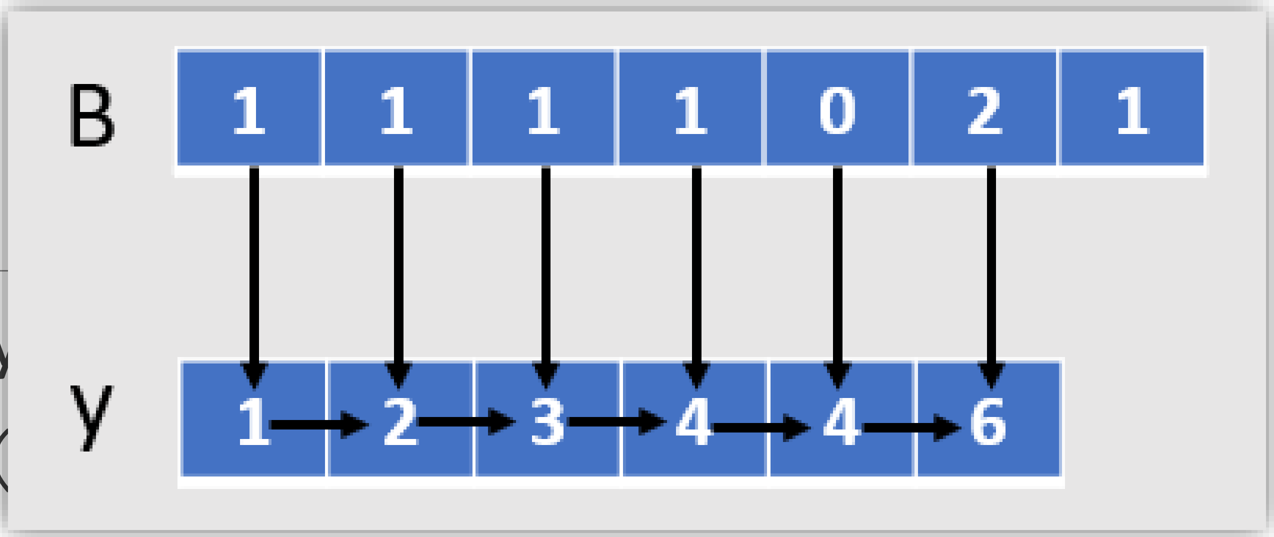
③ Generate B ($n + 1$), which is a Histogram of $\mathrm{A_p}$

④ Apply Prefix sums to B to generate y (n)

HP sort algorithm

Input array x(size : n), output array

Work array A(size : $maxVal$), A_p (



① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate A_p ($maxVal$)

③ Generate B (n + 1), which is a Histogram of A_p

④ Apply Prefix sums to B to generate y (n)

HP sort algorithm

Input array x (size : n), output array y (size : n)

Work array A (size : $maxVal$), $\mathrm{A_p}$ (size : $maxVal$), B (size : $n+1$)

① Generate A ($maxVal$), which is a Histogram of x (n)

② Apply Prefix sums to A to generate $\mathrm{A_p}$ ($maxVal$)

③ Generate B ($n + 1$), which is a Histogram of A

④ Apply Prefix sums to B to generate y (n)

※ Sort by doing Histogram and Prefix sums twice each

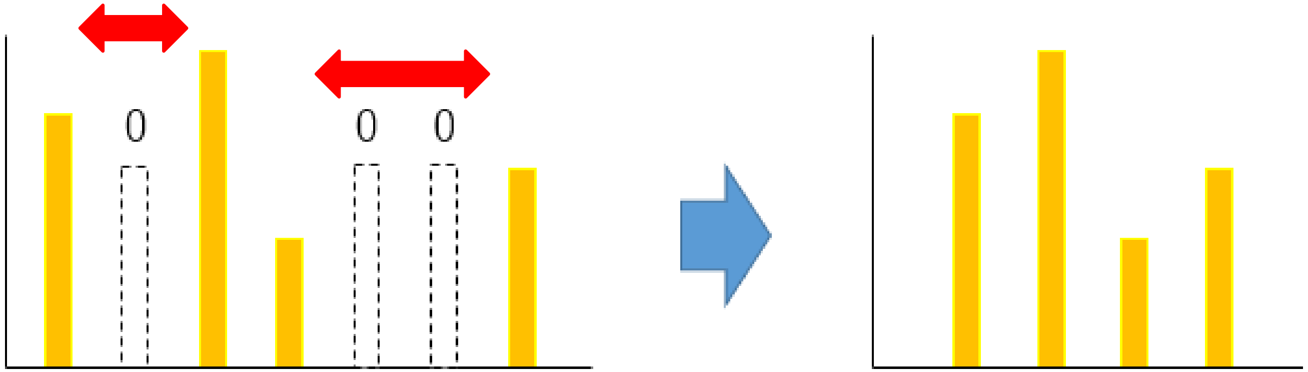
Distinct Data

1-H-P sort

Sort by doing Histogram and Prefix sums once each

0-Compressed H-P sort

- This algorithm aims to reduce the load of Prefix sums on the Histogram by compressing the "number: 0" part when generating the Histogram of H-P sort.



0-Compressed H-P sort algorithm

Parameters

n : Number of input data

$maxVal$: Input data interval$[0, maxVal - 1]$

len : Number of kinds of input data

Array used

input array x(size : n), output array y (size : n)

Work array A (size: $maxVal$), B (size: n), C (size: $len + 1$)

0-Compressed H-P sort algorithm

input array x(size : n), output array y (size : n)

Work array A (size: $maxVal$), B (size: n), C (size: len + 1)

① Generate A ($maxVal$), which is a Histogram of x (n)

② Generate C (len + 1) by compressing the 0 part of the Histogram of A

③ Generate B (n) using C

④ Apply Prefix sums to B to generate y (n)

0-Compressed H-P sort algorithm

input array x(size : n), output array y (size : n)

Work array A (size: $maxVal$), B (size: n), C (size: $len + 1$)

① Generate A ($maxVal$), which is a Histogram of x (n)

② Generate C ($len + 1$) by compressing the 0 part of the Histogram of A

③ Generate B (n) using $\boxed{\text{C}}$

④ Apply Prefix sums to B to generate y (n)

$\boldsymbol{O(len)}$

GPU implementation of Histogram and Prefix sums

Histogram

- Realized using the "atomicAdd()" function of the GPU library.

Prefix sums

- Realized using the "InclusiveSum()" function of the GPU's CUB library.

Devised on the implementation ①

- Separation of coalescing access and random access

Random access to array A

```
atomicAdd(&A[x[i]], 1)
```

Simple implementation

Coalescing access to input array x

```
int pos = x[i];
__syncthreads();
atomicAdd(&A[pos], 1);
```

Slightly faster by separating the execution of coalescing access and random access.

Devised on the implementation ②

- Consolidation of memory allocation for work arrays

<u>Naive implementation</u>

- First, allocate memory for each array ("cudaMalloc")
- Finally, the memory allocated for each array is released ("cudaFree").

It takes longer than the calculated part of the algorithm.

Devised on the implementation ②

- Consolidation of memory allocation for work arrays
  - First, allocate the memory for all arrays at once ("cudaMalloc" is executed only once).

Memory for the total size of the work arrays

  - Partition memory to each array according to its size (with pointer).

Array A	Array A_p	Array B

  - Finally, release the batched memory ("cudaFree" is executed also only once).

Experimental environment

CPU	Intel Xeon CPU E5-2620 v3
GPU	NVIDIA　Tesla K40c

	CPU	GPU
Core	6	2880
Memory size	768GB DDR4	12GB GDDR5
Memory band	56 GB/s	288 GB/s

- NVIDIA Integrated development environment
  「CUDA」 ver.10.0.130

Details of Experiment

① Distinct Data

② Non-Distinct Data

③ Data with a fixed len

④ Data with $range$ Kinds of Values

$range$	$Maximum\ value(maxVal) - Minimum\ value(minVal)$
len	the number of kinds of input data

Summary of Experiment

① Distinct Data

② Non-Distinct Data

③ Data with a fixed len

④ Data with $range$ Kinds of Values

$10k \leq m \leq 100k$

$1M \leq m \leq 10M$

$maxVal$ is Large

$maxVal$ = Small

H-P sort is up to 2.97 times faster

0-Compressed H-P sort is up to 2.73 times faster

Conclusion

- The proposed algorithm works well only with certain types of data, but the applicability of our algorithm is quite large. It is also applicable when "$maxVal - minVal$" is smaller than n and/or the number of kinds of input data is smaller than " $maxVal - minVal$ ".

ex)

- Sorting exam scores of many examinees.
- Sorting ages of many people.

Future work

- Stable sorting algorithms maintain in the output the relative order of input appearance in the case of equally valued data. If the algorithm is stable, it can be used as a subroutine to sort each digit of a radix sort.

- Proposed algorithms in this paper are not stable.

- Making our algorithms stable while preserving their efficiency is future work.

Thank you for your attention.