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 $\text{minVal}(=0)$ and less than $\text{maxVal}$. 
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.

**0-Compressed H-P sort**
Newly devised algorithm for speeding up by compressing the Histogram.
How efficient is H-P sort?

- \( n = 1M \sim 10M, \ minVal = 0, \ maxVal = n/50 \)

- 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?

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

![Graphs showing time [ms] vs. n for CUB, H-P, and 0-Comp](https://via.placeholder.com/150)
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: $O(n)$ $n$ is "number of input data“.

$$\text{output}[k] = \text{input}[0] + \text{input}[1] + \cdots + \text{input}[k]$$
HP sort algorithm

Parameters

\[ n: \text{Number of input data} \]
\[ \text{maxVal}: \text{Input data interval}[0, \text{maxVal} - 1] \]

Array used

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

Work array \( A(\text{size}:\text{maxVal}), A_p(\text{size}:\text{maxVal}), B(\text{size}:n + 1) \)
HP sort algorithm

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

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

① Generate $A$ ($\text{maxVal}$), which is a Histogram of $x$ ($n$)
② Apply Prefix sums to $A$ to generate $A_p$ ($\text{maxVal}$)
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), A_p (size: maxVal).

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: $maxVal$), $A_p$ (size: $maxVal$)

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

1. Generate $A$ ($\text{maxVal}$), which is a Histogram of $x$ ($n$)
2. Apply Prefix sums to $A$ to generate $A_p$ ($\text{maxVal}$)
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: $\text{maxVal}$), $A_p$ (size: $\text{maxVal}$), $B$ (size: $n+1$)

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: \( maxVal \)), \( A_p \) (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 \( A_p \) (\( maxVal \))
③ Generate \( B \) (\( n + 1 \)), which is a Histogram of \( A_p \)
④ Apply Prefix sums to \( B \) to generate \( y \) (\( n \))

※ Sort by doing Histogram and Prefix sums twice each

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 \( x(\text{size}: n) \), output array \( y(\text{size}: n) \)
Work array \( A(\text{size}: maxVal) \), \( B(\text{size}: n) \), \( C(\text{size}: len + 1) \)
0-Compressed H-P sort algorithm

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

Work array \( A \) (size: \( \text{max}Val \)), \( B \) (size: \( n \)), \( C \) (size: \( \text{len} + 1 \))

1. Generate \( A \) (\( \text{max}Val \)), which is a Histogram of \( x \) (\( n \))
2. Generate \( C \) (\( \text{len} + 1 \)) 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 $C$
4. Apply Prefix sums to $B$ to generate $y$ ($n$)

$O(len)$
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

Simple implementation

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

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

int pos = x[i];
__syncthreads();
atomicAdd(&A[pos], 1);
Devised on the implementation

- Consolidation of memory allocation for work arrays

**Naive implementation**

- 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

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel Xeon CPU E5-2620 v3</td>
<td>NVIDIA Tesla K40c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>6</td>
<td>2880</td>
</tr>
<tr>
<td>Memory size</td>
<td>768GB DDR4</td>
<td>12GB GDDR5</td>
</tr>
<tr>
<td>Memory band</td>
<td>56 GB/s</td>
<td>288 GB/s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Range</th>
<th>Maximum value ((maxVal)) – Minimum value ((minVal))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(len)</td>
<td>the number of kinds of input data</td>
</tr>
</tbody>
</table>
Summary of Experiment

1. Distinct Data
2. Non-Distinct Data
3. Data with a fixed \( \text{len} \)
4. Data with \( \text{range} \) Kinds of Values

H-P sort is up to 2.97 times faster
0-Compressed H-P sort is up to 2.73 times faster

\( 10^4 \leq n \leq 10^5 \)

\( \text{maxVal} \) is small

\( 10^6 \leq n \leq 10^7 \)

\( \text{maxVal} \) is large
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.