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

SEIYA KOZAKAI[1,a)] NORIYUKI FUJIMOTO[2,b)] KOICHI WADA[3,c)]

Abstract: In this paper, we propose integer sorting algorithms based on histogram and prefix-sums and we show that their GPU-implementations are faster than the fastest sorting GPU-implementations in Thrust and/or CUB library for several input data. In particular, our algorithm is very useful in the cases that the maximum number of input data and/or the number of kinds of input data are smaller than the number of input data.

1. Introduction

Sorting is one of the fundamental and well-studied problems in various fields of computer science. The introduction of GPUs has attracted remarkable attention for new challenges in designing fast parallel sorting algorithms [1], [2].

This paper focuses on efficient GPU implementations of *integer sorting* on GPU and evaluates their performance comparing with the known fastest sorting GPU-implementations built-in thrust and/or cub libraries on GPU [3], [4]. Integer sorting is a sorting of n input data taken from integer values between 0 and $maxVal-1$, where $maxVal$ is known beforehand. Therefore, integer sorting can be implemented by using characteristics of the input like counting sort and radix sort[5], and these algorithms are suitable for parallel implementation. There are several research about GPU implementations of integer sorting such as [5], [6], [7], [8], [9], [10].

In this paper, we have developed faster integer sorting algorithms than the sorting one which is known to be the fastest implemented on GPGPU[3], [4]. First, since Histogram and Prefix-sums operations can be efficiently implemented on GPU[11], we implemented an integer sorting algorithm (called H-P algorithm) only with these operations[12] on GPU, and compared it with the built-in fastest algorithms in Thrust and/or CUB libraries[3], [4]. We call these sorting algorithms Thrust-sort and CUB-sort, respectively. This integer sorting algorithm was proposed as a very fast time ($O(\log^* n)$) and cost-optimal one on a sum-CRCW PRAM [12] and consists of repeating Histogram and Prefix-sums twice. We show that this H-P algorithm can be changed into one with one Histogram and one Prefix-sums in the case that all input data are distinct. We call the modified algorithm 1-H-P algorithm.

We have mainly shown the followings; Let δ be defined as the ratio of the number of data n to the maximum value of input data $maxVal$ (that is, $\delta = \frac{n}{maxVal}$)

(1) In the case that input data are distinct, 1-H-P algorithm is at most 2.0 times faster than Thrust-sort and CUB-sort[*1] for the case that n is between one hundred thousand (100k) and one million (1M) and $\delta = 1$. We have obtained the similar results for $\delta = 0.5$ and $\delta = 0.25$.

(2) In general case, H-P algorithm is $2.77 \sim 2.35$ times faster than Thrust-sort and CUB-sort for the case that n is between one million (1M) and ten million (10M) and $\delta = 50$.

Secondly, we have proposed a more efficient integer sorting algorithm in the case that the number of kinds of input data (denoted as len) is smaller than $maxVal$. This algorithm is called 0-Compressed H-P algorithm because this is a variant of H-P algorithm but we use an array of size len instead of that of size $maxVal$ to store the result of Histogram. We have mainly shown the followings. Let σ be defined as the maximum value of input data $maxVal$ to len (that is, $\sigma = \frac{maxVal}{len}$)

(3) In the case that n is between one million (1M) and ten million (10M) and $\delta = 50$ and $\sigma = 1$, 0-Compressed H-P algorithm is almost the same performance as H-P algorithm.

(4) We consider the two cases of input data. One is that input data are arbitrary taken from the interval $[maxVal - len..maxVal-1]$ (called the interval data), and the other is that input data are arbitrary taken from $[0..maxVal-1]$ such that the number of kinds of input data is len (called the arbitrary data).

(4-1) Considering the interval data, in the case that n is between one million (1M) and ten million (10M), 0-Compressed H-P algorithm is $2.69 \sim 1.99$ times faster than Thrust-sort and CUB-sort and faster than H-P algorithm for $\delta = 50$ and

1 Graduate school, Hosei University Koganei, Tokyo 184–8584, Japan
2 Graduate school, Osaka Prefecture University Sakai, Osaka 599–8531, Japan
3 Hosei University Koganei, Tokyo 184–8584, Japan
a) seiya.kozakai.2q@stu.hosei.ac.jp
b) fujimoto@cs.osakafu-u.ac.jp
c) wada@hosei.ac.jp

*1 It means the faster algorithm of the two.

$\sigma = 100$.

(4-2) Considering the arbitrary data, in the case that n is between one million (1M) and ten million (10M), 0-Compressed H-P algorithm is $2.72 \sim 2.14$ times faster than Thrust-sort and CUB-sort and faster than H-P algorithm for $\delta = 50$ and $\sigma = 100$.

The paper is organized as follows. In Section 2, we present proposed algorithms, H-P algorithm, 1-H-P algorithm, and 0-Compressed H-P algorithm. Section 3 shows an implemetation of our algorithms for GPGPU. Section 4, we report exprerimental results performed on GPGPU. We conclude in the last section.

2. Proposed Algorithms

2.1 H-P algorithm

We use an integer sorting algorithm based on histograms and prefix-sums and call it *H-P algorithm*. This algorithm was proposed as an $O(\log^* n)$ time algorithm on a sum-CRCW PRAM[12]. Histogram and Prefix-sums are shown in Algorithm 1 and H-P algorithm is shown in Algorithm 2. Since this algorithm is an integer sorting algorithm, the maximum value among input data is predetermined and input data are taken from values between 0 and $maxVal - 1$. An example of the execution of Algorithm 2 is depicted in Fig. 1. In Fig. 1, array x is input, A stores the Histogram of x, A_p is Prefix-sums of A, B stores the Histogram of A_p, and the Prefix-sums of B is output y, which is correctly sorted. Note that since the maximum value in A_p is n, the size of B becomes $n+1$. However, the output is sufficient to compute Prefix-sums of $B[0], B[1], \ldots, B[n-1]$ and $B[n]$ is not used in the algorithm.

Lemma 1. [12] *Let n and maxVal* -1 *be the number of input data and the maximum value of input data, respectively. If* $0 \leq x[i] < maxVal(0 \leq i \leq n)$*, then Algorithm 2 sorts* $x[0], x[1], \ldots, x[n-1]$ *correctly in* $O(max(n, maxVal))$ *time.*

Algorithm 2 can be implemented on a PRAM (sum-CRCW PRAM) in $O(\log^* m)$ time by using $O(m/\log^* m)$ processors, where $m = max(n, maxVal)$. On the same PRAM Histogram can be computed in constant time by using m processors*2 and Prefix-sums can be computed in constant time by using $O(m \log m)$ processors[13]. Thus, H-P algorithm can be computed in constant time by using $O(m \log m)$ processors on sum-CRCW PRAM. When considering the implementation on GPU, if Histogram and Prefix-sums can be implemented on GPU efficiently, H-P algorithm can be also implemented on GPU efficiently.

H-P algorithm can be simplified in the case that $n \geq maxVal$ and the input data are different. That is, in that case it is sufficient to perform Histogram and Prefix-sums once. If the input data are different, after the first Prefix-sums for the Histogram of input, if $A_p[0] \neq 0$ then 0 is the smallest value in the input. Otherwise, 0 is not included in the input data and the smallest value is the smallest i such that $A_p[i] \neq A_p[i-1]$. In general, the difference between $A_p[i]$ and $A_p[i-1]$ $(1 \leq i \leq maxVal - 1)$ is at most one and $A_p[i] - A_p[i-1] = 1$ if and only if i is the $A_p[i-1]$-th smallest value. Therefore, Algorithm 3 can perform sorting correctly.

*2 On sum-CRCW PRAM Histogram is trivially computed in constant time with $O(m)$ processors.

Lemma 2. *Let n and maxVal* -1 *be the number of input data and the maximum value of input data, respectively. If* $0 \leq x[i] < maxVal(0 \leq i \leq n)$ *and* $x[i] \neq x[j](0 \leq i < j \leq n)$*, then Algorithm 3 sorts* $x[0], x[1], \ldots, x[n-1]$ *correctly in* $O(maxVal)$ *time.*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
x	0	9	8	15	9	8	0	9	5	15	9	0	15	5	8	15	8	9	8	9	
A	3	0	0	0	0	2	0	0	5	6	0	0	0	0	0	4					
A_p	3	3	3	3	3	5	5	5	10	16	16	16	16	16	16	20					
B	0	0	0	5	0	3	0	0	0	0	1	0	0	0	0	0	6	0	0	0	1
y	0	0	0	5	5	8	8	8	8	8	9	9	9	9	9	9	15	15	15	15	

Fig. 1 An execution example of H-P algorithm ($n = 20$ and $maxVal = 16$).

Algorithm 1 Histogram and Prefix-Sums

```
subroutine Histogram(int data[], hist[], num, bins)
  for (int i = 0; i < bins; i++)
      hist[i] = 0;
  for ( int i = 1; i < n; i++)
      hist[data[i]]++;

subroutine Prefix-Sums(int data[], data_p[], num)
  int sum = 0;
  for ( int i = 0; i < num; i++) {
      sum += data[i];
      data_p[i] = sum;
  }
```

Algorithm 2 H-P algorithm

```
Assumptions:
    maxVal − 1: maximum value among input data.

input: x[0], ..., x[n−1], maxVal;
output: y[0], (≤)y[1], (≤) ..., (≤)y[n−2], (≤)y[n−1];

variables
int A[maxVal], A_p[maxVal], B[n+1];

Algorithm
  Histogram(x, A, n, maxVal);
  Prefix-Sums(A, A_p, maxVal);
  Histogram(A_p, B, maxVal, n+1);
  Prefix-Sums(B, y, n);
```

2.2 0-compressed H-P algorithms

H-P algorithm is an integer sorting algorithm and if $maxVal$ is smaller than n, it is computed in $O(n)$ time. However, otherwise, it is computed in $O(maxVal)$ time. Therefore, we propose a variant of the H-P algorithm which is an efficient sorting algorithm in the case that the number of kinds of input data is small

Algorithm 3 1-H-P algorithm

Assumptions:

$maxVal-1$: maximum value among input data.
input data are different.

input: $x[0],\ldots,x[n-1](x[i]\neq x[j](i\neq j)$, $maxVal$;
output: $y[0],(\leq)y[1],(\leq)\ldots,(\leq)y[n-2],(\leq)y[n-1]$;

variables
int $A[maxVal], A_p[maxVal]$;

Algorithm

```
Histogram(x, A, n, maxVal);
Prefix-Sums(A, A_p, maxVal);
if A_p[0] ≠ 0 then y[0] = 0;//The input has 0.
for(int i = 1; i < maxVal; i++)
    if A_p[i] ≠ A_p[i-1] then y[A_p[i-1]] = i
```

even if $maxVal$ is larger than n. The idea is that when computing Prefix-sums of the histogram A of input x, it is not computed directly from A but instead creating a new array C whose size is the number of kinds of input (denoted as len) and which consists of non-zero elements of A, it is computed from C and some additional information. If C can be computed efficiently, Prefix-sums of the histogram A of input x can be computed in $O(len)$ time not in $O(maxVal)$ time.

The abstract level of the algorithm is shown in **Algorithm** 4, where len is the number of kinds of input data, $C[len+1]$ has non-zero elements in A (Histogram of input data) and letting $i_1, i_2, \ldots, l_{len}$ be these indices of non-zero elements, $C[j] = A[i_j](1\leq j\leq len)$ and $iC[j] = i_j(1\leq j\leq len)$. $iC[j]$ indicates the index in $A[maxVal]$ for $C[j]$ and is used to compute A_p (Prefix-sums of A) with $C[j]$.

Let $A_p[maxVal]$ be Prefix-sums of $A[maxVal]$ (Histogram of $x[n]$) and $B[n+1]$ be its Histogram. And let $C_p[len+1]$ be Prefix-sums of $C[len+1]$. $A01[maxVal]$ is defined as

$$A01[j](1\leq j\leq len) = \begin{cases} 0 & (\textbf{if } A[j]=0) \\ 1 & (\textbf{if } A[j]\neq 0) \end{cases},$$

and its Prefix-sums is denoted as $A01_p[maxVal]$.

The following lemmas are used to implement Algorithm 4.

Lemma 3. *For $j(1\leq j\leq len)$, $C[j] = A[A01_p[i]]$(**if** $(j = A01_p[i])$ **and** $(A[A01_p[i]]>0)$), and $iC[j] = i$(**if** $(j = A01_p[i])$.*

In the following we assume $C[0]=0$ and $iC[0]=0$.

Lemma 4. *For $i(0\leq i\leq n-1)$,*

$$B[i] = \begin{cases} iC[j+1]-iC[j] & (\textbf{\textit{if}}\ i = C_P[j]) \\ 0 & (\textbf{\textit{otherwise}}). \end{cases}$$

We can implement Algorithm 4 as Algorithm 5 using Lemmas 3 and 4. An example of the execution of Algorithm 5 is shown in Fig. 2. Fig. 2 shows Algorithm 5 works correctly. In fact, we have the following lemma.

Lemma 5. *Let n and $maxVal-1$ be the number of input data and the maximum value of input data, respectively. If $0\leq x[i] < maxVal(0\leq i\leq n)$, then Algorithm 5 sorts $x[0],x[1],\ldots,x[n-1]$ correctly in $O(max(n,maxVal))$ time.*

The time complexity of Algorithm 5 is the same as that of Algorithm 2. However, comparing these two algorithms closely, the same parts are the first Histogram and Prefix-sums with size $maxVal$[*3], and the last Prefix-sums with size n. Then we should compare the second Histogram with size $maxVal$ to obtain B in Algorithm 2 with computing C and iC with size $maxVal$ and B with size len in Algorithm 5. The big difference is that Algorithm 5 does not need computing Histogram. Histogram operation is time-consuming on GPGPU [11], we have possibility that Algorithm 5 can be faster than Algorithm 2 when implementing them on GPGPU. In fact, we will show that Algorithm 5 is faster than Algorithm 2 in Section 4.

Algorithm 4 0-Compressed H-P algorithm (abstract)

Assumptions:

n: number of input.
$maxVal-1$: maximum value among input data.

Input: $x[0],\ldots,x[n-1]$, $maxVal$;
Output: $y[0],(\leq)y[1],(\leq)\ldots,(\leq)y[n-2],(\leq)y[n-1]$;

Variables
int $A[maxVal], B[n+1], C[len+1], iC[len+1]$;
where len is the number of kinds of input data.

Algorithm

Histogram($x, A, n, maxVal$);
Let $i_1, i_2, \ldots, i_{len}$ be increasing indices of A such that $A[i] > 0$, where len is the number of kinds of input data.
Let $C[len+1]$ and $iC[len+1]$ be defined as follows:

$$C[j] = \begin{cases} unused & (\textbf{if } j=0)) \\ A[i_j] & (\textbf{if } 1\leq j\leq len) \end{cases}$$

$$iC[j] = \begin{cases} unused & (\textbf{if } j=0)) \\ i_j & (\textbf{if } 1\leq j\leq len) \end{cases}$$

Compute Histogram B of Prefix-Sum A_p of A by using C and iC;
Prefix-Sums(B, y, n);

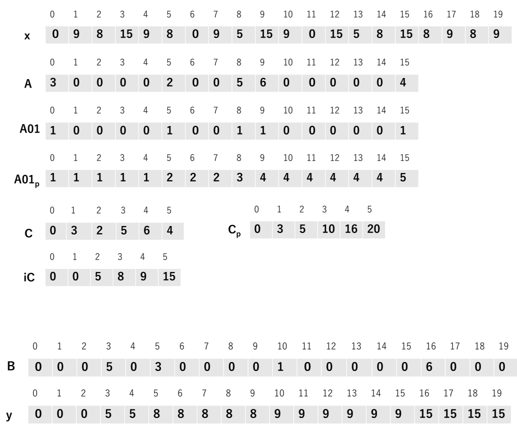

Fig. 2 An execution example of 0-compressed H-P algorithm ($n=20$, $maxVal=16$ and $len=5$).

*3 Although in Algorithm 2, computing $A01$ has a little bit extra time, but these two are considered to be almost the same.

Algorithm 5 0-Compressed H-P algorithm (Implementation)

Assumptions:

n:number of input.

$maxVal-1$: maximum value among input data.

Input:$x[0],\dots,x[n-1]$, $maxVal$;
Output:$y[0],(\le)y[1],(\le)\dots,(\le)y[n-2],(\le)y[n-1]$;

Variables
int $A[maxVal],A_p[maxVal],B[n+1]$,
$A01[maxVal],A01_p[maxVal]$,
$C[len+1],C_p[len+1],iC[len+1]$;

Algorithm
1: Histogram($x,A,n,maxVal$);
2: **for**(**int** $i=0;i<maxVal;i++$)
3: $A01[i]=(A[i]>0)?1:0$;
4: Prefix-Sums($A01,A01_p,maxVal$);
5: **int** $len=A01_p[maxVal-1]$;
6: **int** $C[len+1],C_p]len+1],iC[len+1]$;
7: //where len is the number of kinds of input data.
8: C[0]=0;
9: iC[0]=0;
10: **for**(**int** $i=0;i<maxVal;i++$)
11: **if** $(A[i]>0)$ {
12: $C[A01_p[i]]=A[i]$;
13: $iC[A01_p[i]]=i$;
14: }
15: Prefix-Sums($C,C_p,len+1$);
16: **for**(**int** $i=0;i<len;i++$) $B[i]=0$
17: **for**(**int** $i=0;i<len;i++$) $B[C_p[i]]=iC[i+1]-iC[i]$;
18: Prefix-Sums(B,y,n);

3. Implementation on GPU

We implement all of H-P algorithm (Algorithm 2), 1-H-P algorithm (Algorithm 3), and 0-Compressed H-P algorithm (Algorithm 5) in CUDA C/C++ language [14] partially with CUB library.

Every algorithm uses subroutine Histogram and Prefix-Sums. Subroutine Prefix-Sums is implemented just by calling CUB library function cub::DeviceScan::InclusiveSum(). The CUB implementation is the fastest Prefix-Sums implementation as far as we know. Subroutine Histogram is implemented using CUDA atomic function atomicAdd(), as shown in Fig.3. In CUDA kernel incCnt, we divide "atomicAdd(&cnt[a[i]], 1);" into three steps shown from lines 6 to 8 in Fig.3. This aims at separating coalescing access to array a[] and non-coalescing access to array cnt[]. That is, our three step implementation intends not to overlap execution of the coalescing access with execution of the non-coalescing access. In our preliminary experiments, our three step implementation was faster than the naive single step implementation.

As shown in Algorithm 2, 3 and 5, every algorithm uses auxiliary arrays except input and output arrays. These arrays are dynamically allocated using CUDA library function cudaMalloc() because their sizes are dynamically determined according to input. If we call cudaMalloc() array by array, it takes long time

```
1  __global__ void incCnt(int n, int *a, int *cnt)
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i >= n) return;
5
6      int pos = a[i];
7      __syncthreads();
8      atomicAdd(&cnt[pos], 1);
9  }
10
11 inline void Histogram(int *data, int *hist, int num, int bins)
12 {
13     cudaMemset(hist, 0, sizeof(int) * bins);
14     incCnt<<< (num + 255) / 256, 256 >>>(num, data, hist);
15 }
```

Fig. 3 Our Implementation of Subroutine Histogram

```
__global__ void OnePrefixSums(int maxVal, int* a, int* in, int* out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= maxVal) return;

    if (i == 0) out[a[i]] = i;
    else if(a[i - 1] != a[i]) out[a[i]-1] = i;
}
```

Fig. 4 Our Implementation of lines 3 to 5 in Algorithm 3

because cudaMalloc() is time-consuming. Therefore, in our implementation of each algorithm, we call cudaMalloc() and cudaFree() only once to allocate and free a memory block of size required for all auxiliary arrays in each algorithm. Each auxiliary array is manually allocated to a part of the memory block. Auxiliary arrays C, C_p, and iC in Algorithm 5 have size $len+1$, which depends on the content of input data. The size cannot be determined until line 5 in Algorithm 5. If we allocate memory for the three arrays after the size is determined, we must call cudaMalloc() twice, which makes the resultant implementation very slow. However, we have $len \le n$ where input array size n is independent of the content of input data. Therefore, we allocate size $n+1$ instead of $len+1$ for each array to realize a single call of cudaMalloc() and cudaFree(). Note that at the start of each algorithm we can determine the sizes of auxiliary arrays A, A_p, $A01$, and $A01_p$ in Algorithm 2, 3 and 5 because $maxVal$ is given as a part of input of each algorithm.

As for Algorithm 3, lines 3 to 5 are implemented as a single CUDA kernel as shown in Fig.4. Lines 6 to 7 are equivalent to "out[a[in[i]]-1] = in[i];". However, in our preliminary experiments, this single line implementation was slower.

As for Algorithm 5, lines 2 to 3, lines 10 to 14, and lines 16 to 17 are respectively implemented as a single CUDA kernel as shown in Fig.5, 6, and 7. Due to lines 6 to 9 in Fig.7, in our implementation of Algorithm 5, line 9 in Algorithm 5 can be ignored. In contrast, line 8 in Algorithm 5 is implemented just by calling CUDA library function cudaMemset().

4. Experimental Results

This section describes the experimental environment, the experimental content, and the experimental results.

4.1 Experimental Environment

The experiments were performed in the environment shown in

```
__global__ void binarize(int n, int* cnt, int* cnt01)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;

    cnt01[i] = (cnt[i] > 0) ? 1 : 0;
}
```

Fig. 5 Our Implementation of lines 2 to 3 in Algorithm 5

```
__global__
void compressA01p(int maxVal, int* A, int* A01p, int* C, int* iC)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= maxVal) return;

    if (A[i]) {
        int x = A01p[i];
        int y = A[i];
        __syncthreads();
        C[x] = y;
        iC[x] = i;
    }
}
```

Fig. 6 Our Implementation of lines 10 to 14 in Algorithm 5

```
__global__ void expandToB(int len, int* Cp, int* iC, int* B)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= len) return;

    if (i == 0) {
        B[Cp[0]] = iC[1];
        return;
    }

    int x = Cp[i];
    __syncthreads();
    B[x] = iC[i + 1] - iC[i];
}
```

Fig. 7 Our Implementation of lines 16 to 17 in Algorithm 5

Table 1. The CPU was an Intel Xeon CPU E5-2620 v3 and the GPU was an NVIDIA Tesla K40c. We used CUDA toolkit version 10.0.130.

Table 1 Experimental Environment

	CPU	GPU
Cores	6	2880
Memory Size	768GB DDR4	12GB GDDR5
Memory Bandwidth	56 GB/s	288 GB/s

n	Thrust	CUB	H-P	0-Comp	1-H-P
100k	0.336	0.578	0.332	0.392	0.306
200k	0.702	0.664	0.357	0.459	0.331
300k	0.836	0.732	0.410	0.533	0.371
400k	0.939	0.811	0.523	0.676	0.480
500k	1.096	0.915	0.681	0.831	0.606
600k	1.241	1.003	0.815	0.993	0.729
700k	1.292	1.091	0.942	1.153	0.860
800k	1.386	1.153	1.091	1.374	0.998
900k	1.486	1.220	1.228	1.513	1.127
1M	1.676	1.277	1.379	1.662	1.267

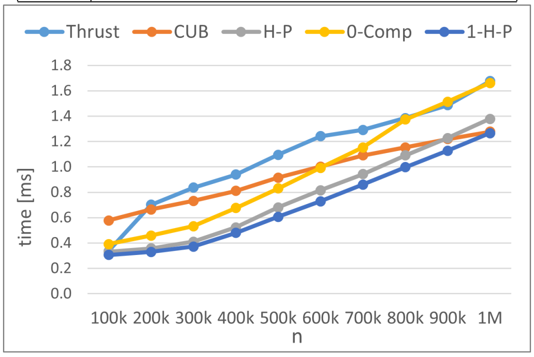

Fig. 8 Computing time for distinct data ($\delta = 1$)

n	Thrust	CUB	H-P	0-Comp	1-H-P
100k	0.340	0.588	0.344	0.417	0.322
200k	0.705	0.690	0.435	0.549	0.407
300k	0.827	0.730	0.579	0.718	0.549
400k	0.946	0.815	0.736	0.921	0.680
500k	1.088	0.905	0.900	1.107	0.850
600k	1.189	1.006	1.051	1.299	0.990
700k	1.290	1.075	1.211	1.494	1.151
800k	1.383	1.163	1.393	1.685	1.295
900k	1.485	1.219	1.552	1.883	1.449
1M	1.666	1.279	1.708	2.102	1.605

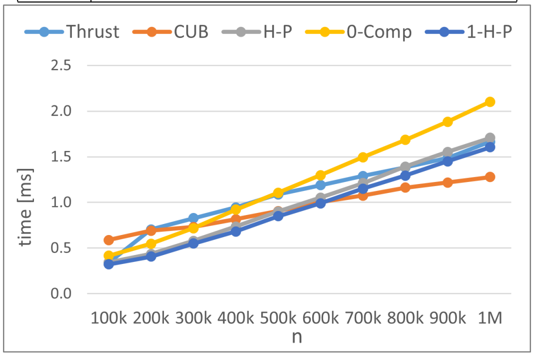

Fig. 9 Computing time for distinct data ($\delta = 0.5$)

4.2 Experimental Content

In the experiments, we compare the fastest sorting algorithms in the Thrust and CUB libraries (denoted as "Thrust sort" and "CUB sort", respectively) and the three algorithms introduced in Section 2. We denote the three algorithms to be compared implemented on the GPU as follows: Sorting using the H-P algorithm (Algorithm 2) is denoted as "H-P sort", using the 1-H-P algorithm (Algorithm 3) is denoted as "1-H-P sort", and using the 0-Compressed H-P algorithm (Algorithm 4-5) is denoted as "0-Comp sort".

For each algorithm, measurements were performed with n, *maxVal*, and *len* (which imply δ and σ) as parameters.

Data used in the measurements can be characterized by the parameters. We used four kinds of data set. In the following, we devote a sub-subsection to each kind of data set to describe the experiments. In the following tables, time unit is measured by millisecond (ms).

4.2.1 Distinct data

To evaluate 1-H-P sort, we compare the computing time of the five algorithms for distinct data. The results are shown in Figs. 8 to 10. We see 1-H-P sort is the fastest with $\delta = 1$ and n between 100k and 1M, with $\delta = 0.5$ and n between 100k and 600k, and with $\delta = 0.25$ and n between 100k and 300k.

4.2.2 Data with *maxVal* kinds of values

We compare the computing time the four algorithms except 1-H-P sort for non-distinct data. The results are shown in Figs. 11 to 13. We see that CUB sort is the fastest with $\sigma = 1$ and that H-P sort and 0-Comp sort are faster with $\sigma = 50$. Smaller and smaller *maxVal* is, faster and faster H-P sort is. The fastest one changes

n	Thrust	CUB	H-P	0-Comp	1-H-P
100k	0.340	0.610	0.419	0.522	0.405
200k	0.705	0.654	0.563	0.715	0.528
300k	0.827	0.740	0.744	0.939	0.711
400k	0.946	0.815	0.950	1.177	0.888
500k	1.088	0.923	1.133	1.424	1.078
600k	1.189	1.002	1.340	1.670	1.260
700k	1.290	1.082	1.518	1.913	1.449
800k	1.383	1.167	1.706	2.153	1.632
900k	1.485	1.252	1.908	2.429	1.817
1M	1.666	1.296	2.090	2.642	2.020

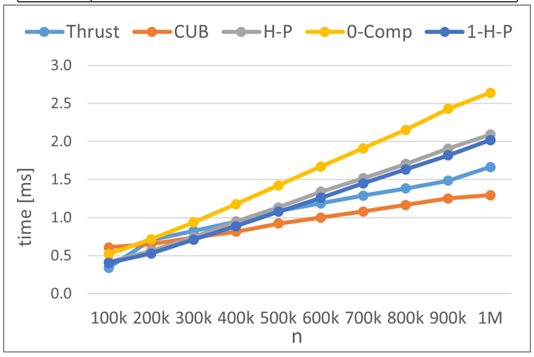

Fig. 10 Computing time for distinct data ($\delta = 0.25$)

n	Thrust	CUB	H-P	0-Comp
1M	1.300	1.301	1.377	1.628
2M	1.960	1.961	2.896	3.366
3M	2.648	2.633	4.456	5.097
4M	3.310	3.305	5.989	6.868
5M	4.218	4.227	7.558	8.744
6M	4.920	4.921	9.175	10.425
7M	5.626	5.628	10.693	12.188
8M	6.291	6.271	12.266	13.949
9M	7.064	7.026	13.866	15.726
10M	7.713	7.709	15.439	17.514

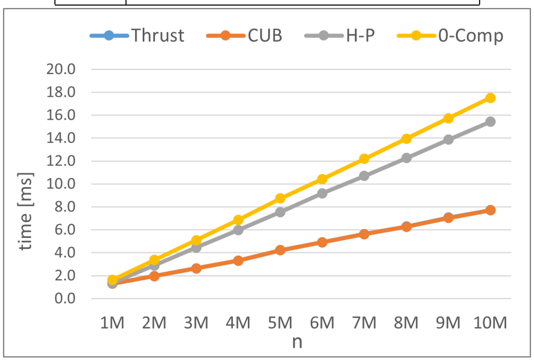

Fig. 11 Computing time in case that $\delta = 1$ and $\sigma = 1$

n	Thrust	CUB	H-P	0-Comp
1M	1.232	1.249	0.551	0.632
2M	1.968	1.953	0.800	0.945
3M	2.628	2.631	1.356	1.435
4M	3.279	3.268	2.289	2.509
5M	3.977	3.977	3.621	3.770
6M	4.703	4.644	4.993	5.133
7M	5.305	5.297	6.141	6.399
8M	5.928	5.930	7.325	7.773
9M	6.620	6.622	8.839	9.207
10M	7.281	7.267	10.283	10.673

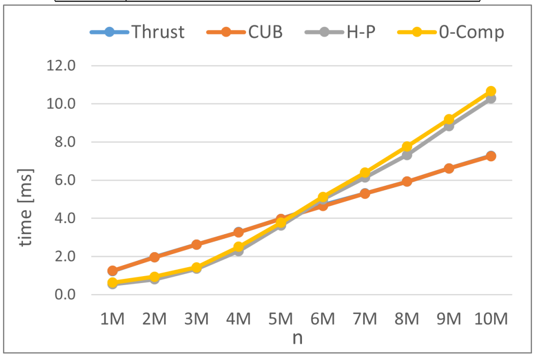

Fig. 12 Computing time in case that $\delta = 10$ and $\sigma = 1$

n	Thrust	CUB	H-P	0-Comp
1M	1.237	1.240	0.526	0.591
2M	1.848	1.844	0.750	0.844
3M	2.470	2.467	0.993	1.085
4M	3.092	3.088	1.214	1.332
5M	3.741	3.744	1.432	1.618
6M	4.397	4.455	1.687	1.877
7M	5.286	5.274	1.902	2.113
8M	5.912	5.911	2.139	2.369
9M	6.614	6.597	2.392	2.623
10M	7.255	7.234	2.664	2.915

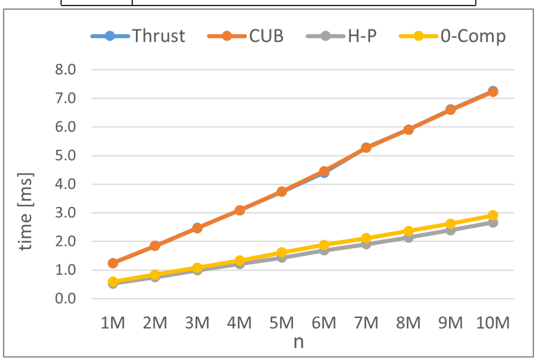

Fig. 13 Computing time in case that $\delta = 50$ and $\sigma = 1$

when σ is 10.

4.2.3 Data with larger σ

In this sub-subsection, we fix *maxVal* and decrease the number of kinds of data. The results are shown in Figs. 14 to 17. We see that 0-Comp sort is the slowest for arbitrary data. However, we see also that 0-Comp sort is the fastest for interval data and larger σ.

4.2.4 Data with $\sigma = 100$

The results are shown in Figs. 18 to 23. We see that 0-Comp sort and H-P sort are the fastest for arbitrary data. We see also that 0-Comp sort is the fastest for interval data although H-P sort catches up with 0-Comp sort when *maxVal* becomes small.

5. Conclusion and Future Work

We have presented efficient integer sorting algorithms based on Histogram and Prefix-sums and have shown that their implementations on GPGPU are faster than the sorting algorithms Thrust-sort and CUB-sort which are known to be the fastest implementation on GPGPU.

Stable sorting algorithms maintain in the output the relative order of input appearance in the case of equally valued data. This property is important and interesting. In fact,in Radix sort each digit sort must be stable in order for radix sort to work correctly. Unfortunately, proposed algorithms in this paper are not stable. Making these algorithms stable while preserving their efficiency is one of the interesting future work.

Acknowledgments

This work was supported in part by JST SICORP Grant Number JPMJSC1806, and by JSPS KAKENHI Grant Num-

n	Thrust	CUB	H-P	0-Comp
1M	1.303	1.293	1.986	0.801
2M	1.950	1.945	3.678	1.324
3M	2.641	2.620	5.690	2.019
4M	3.277	3.283	8.040	3.272
5M	3.968	3.979	10.706	4.596
6M	4.646	4.645	13.474	6.121
7M	5.310	5.308	16.139	7.565
8M	5.923	5.980	18.879	9.180
9M	6.942	6.938	21.747	10.700
10M	7.269	7.309	24.454	12.256

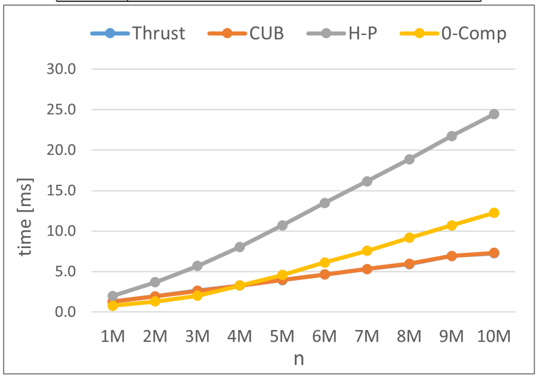

Fig. 14 Computing time for interval data ($\delta = 1$, $\sigma = 10$)

n	Thrust	CUB	H-P	0-Comp
1M	1.338	1.241	2.113	0.776
2M	1.857	1.845	3.887	1.193
3M	2.484	2.468	5.683	1.588
4M	3.100	3.072	7.468	2.023
5M	3.967	3.941	9.261	2.471
6M	4.493	4.352	11.060	2.880
7M	5.294	5.255	12.851	3.310
8M	5.965	5.863	14.652	3.716
9M	6.617	6.569	16.455	4.132
10M	7.276	7.196	18.295	4.572

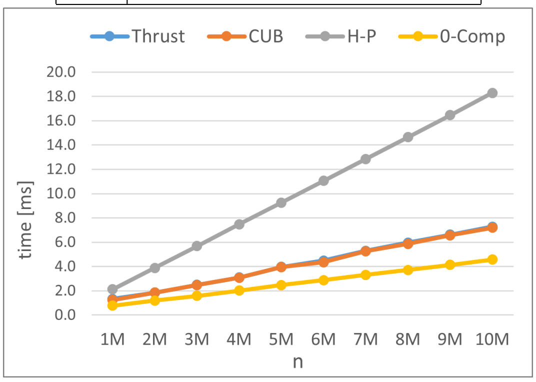

Fig. 15 Computing time for interval data ($\delta = 1$, $\sigma = 100$)

ber 17K00171 20K11685 20K11842.

n	Thrust	CUB	H-P	0-Comp
1M	1.741	1.303	1.426	1.551
2M	2.805	1.961	2.970	3.199
3M	3.953	2.638	4.564	4.843
4M	5.569	3.296	6.135	6.561
5M	6.698	4.207	7.746	8.263
6M	7.926	4.913	9.363	9.967
7M	9.369	5.620	10.972	11.624
8M	11.213	6.258	12.570	13.302
9M	12.445	6.987	14.224	15.055
10M	13.833	7.668	15.820	16.698

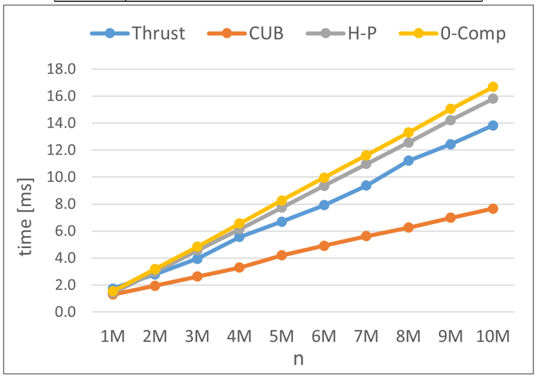

Fig. 16 Computing time for arbitrary data ($\delta = 1$, $\sigma = 10$)

n	Thrust	CUB	H-P	0-Comp
1M	1.732	1.337	0.929	1.024
2M	2.772	1.949	2.356	2.514
3M	3.904	2.630	3.941	4.056
4M	5.483	3.277	5.497	5.709
5M	6.601	4.158	7.113	7.333
6M	7.826	4.879	8.768	8.993
7M	9.241	5.581	10.350	10.660
8M	11.068	6.235	11.955	12.296
9M	12.269	6.952	13.614	13.994
10M	13.645	7.637	15.273	15.624

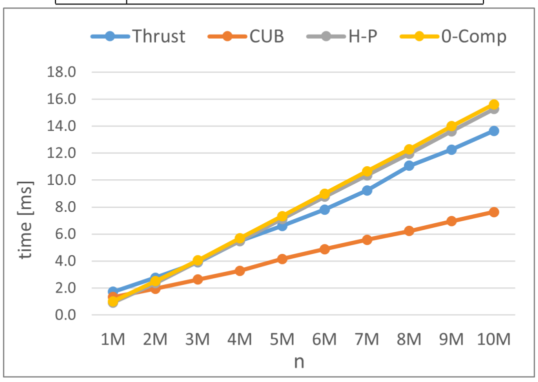

Fig. 17 Computing time for arbitrary data ($\delta = 1$, $\sigma = 100$)

References

[1] Arkhipov, D. I., Wu, D., Li, K. and Regan, A. C.: Sorting with GPUs: A survey, *arXiv:1709.02520v1* (2017).

[2] Faujdar, N. and Ghrera, S.: Performance evaluation of parallel count sort using GPU computing with CUDA, *Indian Journal of Science and Technoogy*, Vol. 9, No. 15, pp. 1–12 (2016).

[3] NVIDIA Corp.: Thrust, , available from ⟨https://docs.nvidia.com/cuda/index.html⟩ (accessed 2020-11-18).

[4] NVIDIA Corp.: CUB, , available from ⟨https://nvlabs.github.io/cub/⟩ (accessed 2020-11-18).

[5] Kolonias, V., Voyiatzis, A. G., Goulas, G. and Housos, E.: Design and implementation of an efficient integer count sort in CUDA GPUs, *Concurrency and Computation: Practice and Experience*, Vol. 23, pp. 2365–2381 (2011).

[6] Svenningsson, J., Svensson, B. J. and Sheeran, M.: Counting and occurrence sort for GPUs using an embedded language, *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, FHPC'13*, pp. 37–46 (2013).

n	Thrust	CUB	H-P	0-Comp
1M	1.221	1.173	0.634	0.553
2M	1.743	1.736	1.014	0.828
3M	2.513	2.446	1.383	1.027
4M	3.073	3.063	1.728	1.282
5M	3.712	3.715	2.107	1.539
6M	4.349	4.346	2.449	1.781
7M	4.971	4.965	2.813	2.015
8M	5.549	5.556	3.200	2.289
9M	6.231	6.207	3.635	2.490
10M	6.814	6.820	3.916	2.731

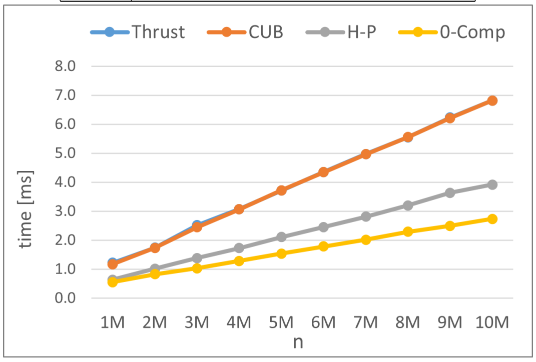

Fig. 18 Computing time for interval data ($\delta = 10$, $\sigma = 100$)

n	Thrust	CUB	H-P	0-Comp
1M	1.185	1.180	0.581	0.686
2M	1.740	1.734	0.753	0.821
3M	2.327	2.303	0.945	0.981
4M	2.889	2.877	1.075	1.121
5M	3.488	3.485	1.646	1.724
6M	4.092	4.082	1.479	1.832
7M	4.938	4.931	1.899	2.004
8M	5.205	5.210	2.041	2.117
9M	5.843	5.840	2.194	2.255
10M	6.417	6.417	2.344	2.448

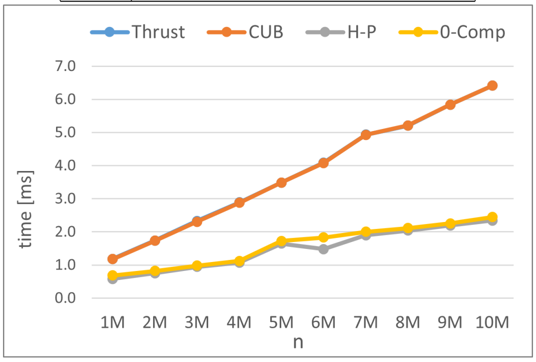

Fig. 20 Computing time for interval data ($\delta = 100$, $\sigma = 100$)

n	Thrust	CUB	H-P	0-Comp
1M	1.164	1.182	0.547	0.583
2M	1.740	1.737	0.716	0.748
3M	2.317	2.310	0.954	0.944
4M	2.881	2.879	1.226	1.219
5M	3.484	3.548	1.442	1.388
6M	4.319	4.319	1.792	1.618
7M	4.917	4.929	1.941	1.839
8M	5.519	5.514	2.086	2.050
9M	5.837	5.841	2.511	2.287
10M	6.424	6.421	2.573	2.491

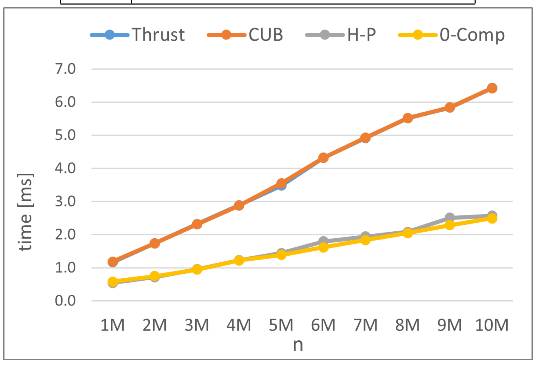

Fig. 19 Computing time for interval data ($\delta = 50$, $\sigma = 100$)

n	Thrust	CUB	H-P	0-Comp
1M	1.681	1.239	0.519	0.590
2M	2.783	1.925	0.766	0.831
3M	3.915	2.581	0.987	1.075
4M	5.553	3.219	1.210	1.320
5M	6.621	3.912	1.451	1.631
6M	7.833	4.629	1.809	1.977
7M	9.243	5.214	2.639	2.544
8M	11.089	5.831	3.235	3.577
9M	12.311	6.522	3.915	4.201
10M	13.695	7.153	4.754	5.105

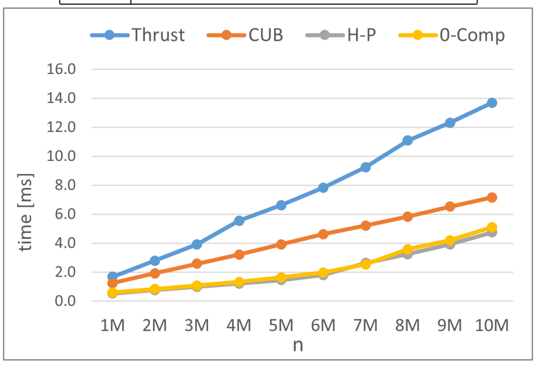

Fig. 21 Computing time for arbitrary data ($\delta = 10$, $\sigma = 100$)

[7] Faujdar, N. and Saraswat, S.: A roadmap of parallel sorting algorithms using GPU computing, *Proceedings of International Conference on Computing, Communication and Automation, ICCCA2017*, pp. 736–741 (2017).
[8] Usmani, A. R.: A novel time and space complexity efficient variant of counting-sort algorithm, *Proceedings of 2019 IEEE International Conference on Innovative Computing, ICIC* (2019).
[9] Yokoyama, E., Yasuoka, K., Okabe, Y. and Kanazawa, M.: Implementation of a fast integer sorting algorithm for distributed-memory parallel vector supercomputers, *IPSJ SIG Technical Reports on High Performance Computing, 42(3)*, pp. 45–53 (2001).
[10] Sum, W. and Ma, Z.: Count sort for GPU computing, *Proceedings of 2009 15th ICPDS*, pp. 919–924 (2009).
[11] Hellfritzsch, S.: Efficient Histogram Computation on GPGPUs, *Master's Thesis, University of Copenhagen*, pp. 1–98 (2018).
[12] Eisenstat, S. C.: $O(\log^* n)$ algorithms on a Sum-CRCW PRAM, *Computing*, Vol. 79, pp. 93–97 (2007).
[13] Frei, F. and Wada, K.: Efficient circuit simulation in MapReduce, *Proceedings of ISAAC 2019, LIPIcs, Vol;. 149*, pp. 55:1–55:22 (2019).
[14] NVIDIA Corp.: CUDA C++ Programming Guide, , available from ⟨https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html⟩ (accessed 2020-11-18).

n	Thrust	CUB	H-P	0-Comp
1M	1.646	1.219	0.509	0.569
2M	2.684	1.821	0.702	0.809
3M	3.781	2.433	0.942	1.011
4M	5.308	3.049	1.134	1.230
5M	6.413	3.683	1.341	1.552
6M	7.590	4.310	1.551	1.712
7M	8.946	5.204	1.758	1.954
8M	10.689	5.837	1.972	2.158
9M	11.886	6.507	2.204	2.392
10M	13.247	7.145	2.411	2.685

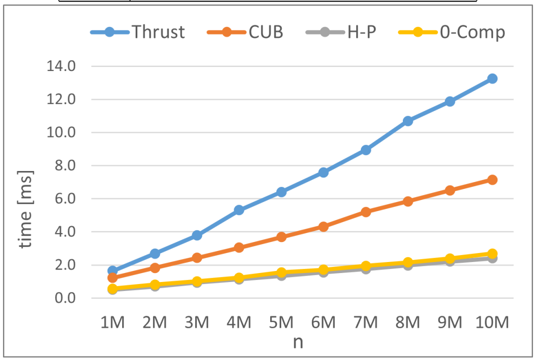

Fig. 22 Computing time for arbitrary data ($\delta = 50$, $\sigma = 100$)

n	Thrust	CUB	H-P	0-Comp
1M	1.618	1.215	0.510	0.565
2M	2.661	1.825	0.700	0.845
3M	3.744	2.429	0.937	1.017
4M	5.244	3.036	1.133	1.243
5M	6.330	3.669	1.343	1.498
6M	7.483	4.290	1.553	1.700
7M	8.835	4.911	1.737	1.947
8M	10.561	5.500	1.955	2.151
9M	11.734	6.142	2.181	2.388
10M	13.062	6.723	2.390	2.584

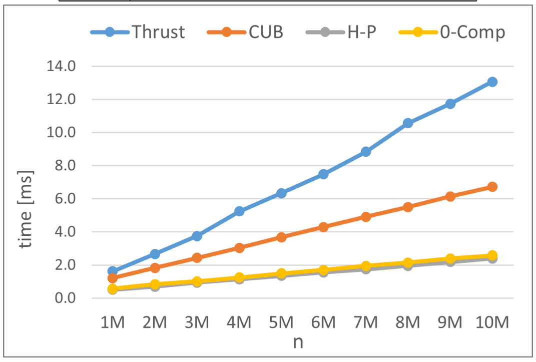

Fig. 23 Computing time for arbitrary data ($\delta = 100$, $\sigma = 100$)