ヒストグラムと接頭辞和に基づく整数ソーティングのためのGPUによる効率的な実装

小堺, 海叶 / KOZAKAI, Seiya

(出版者 / Publisher)
法政大学大学院理工学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 理工学研究科編
(巻 / Volume)
63
(開始ページ / Start Page)
1
(終了ページ / End Page)
8
(発行年 / Year)
2022-03-24
(URL)
https://doi.org/10.15002/00025376

ヒストグラムと接頭辞和に基づく整数ソーティングのための GPU による効率的な実装

EFFICIENT GPU IMPLEMENTATION FOR INTEGER SORTING
BASED ON HISTOGRAM AND PREFIX SUMS

小堺 海叶
Seiya KOZAKAI
指導教員　和田幸一

法政大学大学院理工学研究科応用情報工学専攻修士課程

In this study, we devised and implemented an algorithm based on histograms and prefix sums for integer sorting on GPUs. We conducted comparison experiments between the devised sorting algorithm and an algorithm known to be the fastest sorting on GPUs. We show that the speedup is particularly efficient when the maximum value is small compared to the number of data in the input data or when the number of data types is small.
Key Words *: GPGPU, Integer sorting, CRCW PRAM, Histogram, Prefix-Sums*

1　はじめに

ソーティングはコンピューターサイエンスの様々な分野で研究されている基本的な問題の一つである．GPGPU の発展に伴い，高速な並列ソーティングアルゴリズムの設計に新たな挑戦がなされ，注目されている分野である [1, 5].

1.1　研究の目的

本研究の目的は，GPU による基本的なアルゴリズムの高速化である．これまでの研究では，GPU 上での整数ソーティングの効率的な実装に注目し，GPU 上で実装されている CUB ライブラリ [10] に組み込まれた既知の最速のソーティングアルゴリズムと比較して，その性能を評価した (以降このソーティングアルゴリズムを CUB ソートと表記する)．整数ソーティングとは，$minVal$ から $maxVal-1$ までの整数値によって生成された n 個の入力データをソートするものである．ここでいう $minVal$ と $maxVal$ とは予め分かっている入力データの下限値と上限値をそれぞれ表している．そのため，整数ソーティングは計数ソートや基数ソートのように入力データの特徴を利用して実装することができ [2]，これらのアルゴリズムは並列での実装に向いている．整数ソーティングの GPU 実装については，[1, 4, 5, 9, 13, 14] などの研究論文がある．

1.2　従来の研究

H-P ソートは計数ソートを接頭辞和で並列化したものと見なすことができる．したがって，H-P ソートや本研究の拡張アルゴリズムは計数ソートに基づいている．GPU 上の計数ソートの並列化についてはいくつかの既存研究があるため紹介する．

Sum と Ma[13] は CUDA による実装で，接頭辞和を一度だけ呼び出す素朴な実装を発表している．彼らはその実装を 2GHz AMD Athlon 64 X2 3800+ CPU と NVIDIA GeForce 9600 GSO GPU を使用して評価し，500 万個の整数に対して CPU の実装よりも 8.08 倍高速に動作させることに成功している．

Kolonias ら [9] は CUDA による実装で，最後のステップで同期を必要としない実装を発表している．彼らはその実装を 2 つのプラットフォーム使用して，223 個の整数に対して評価を行った．一つは 3.0GHz Intel Core 2 Duo E8400 CPU と NVIDIA GeForce GTX 260 GPU を搭載したものであり，もう 1 つは 2.8GHz Intel Core i7 930 CPU と NVIDIA GeForce GTX 480 GPU を搭載したものである．後者は CUDA SDK が提供する基数ソートの実装と比較して，3 倍以上の性能向上を示している．

Svenningsson ら [14] は，Haskell ベースの高位関数型言語である Obsidian で書かれた，GPU のカーネルの実装を発表している．彼らの実装は NVIDIA GeForce GTX 670 によって，thrust::sort() と最大 3200 万整数まで比較実験を行っている．成績係数はおおよそ 0.5 から 0.6 の範囲であるが，通常，彼らの実装は thrust::sort() よりも少し速く動作している．

Faujdar と Ghrera[4] は，1000 から 1000 万の整数に対して，逐次計数ソートと並列計数ソートの性能の評価を行っているが，詳細な実装は示されていない．2.93GHz Intel Core i3-530 と NVIDIA GeForce GTX 460 での彼らの CUDA における実装は，2.60GHz Intel Core i5-3230M での逐次実装より最大で 66 倍速く動作した．

1.3 研究の成果

本研究の成果として，まず sum-CRCW PRAM(詳細は 3 章に記載) 上での動作を前提に考案されていた整数ソーティングアルゴリズム [3] を GPU 上で実装した (このソーティングを以降 H-P ソートと表記).

また，H-P ソートのアルゴリズムを基にした二つのソーティングアルゴリズムを新しく考案し，同様に GPU 上で実装した．一つは重複のない入力データ用に H-P ソートに対して改良を施した 1-H-P ソートである．1-H-P ソートは重複のないデータへの使用を前提とすることで，H-P ソートの一部の工程を省略することができる．もう一つは，入力データのヒストグラムを取った際に，$minVAl$ から $maxVal-1$ までの整数値で存在しない区間を圧縮することでソーティングの負荷の軽減を図った 0-圧縮 H-P ソートである．

そして考案・実装した三つのソーティングアルゴリズムを評価するために CUB ソートとの比較実験を行った．その際のパラメータである δ と σ について予め定義しておく．以降，入力データのデータ数 n とデータの範囲 ($range$) の比率を δ とする ($range = maxVal - minVal$，$\delta = \frac{n}{range}$)．また，入力データの種類数を len とし，len に対する $range$ の比率を σ とする ($\sigma = \frac{range}{len}$).

2 GPGPU について

2.1 GPU

GPU(Graphics Processing Unit) は複数のプロセッサを並列に機能させることで，大量のデータを高速に処理することができる，画像処理に特化したハードウェアである．高精細なグラフィックを描画するには単純な処理を膨大に繰り返し行わなければならず，複雑かつ逐次的に命令を処理する CPU にとって負担が大きい．そのため多量のコアを並列に動作させることで，単純かつ膨大な計算処理を高速に行える GPU が描画処理に使われてきた [17].

2.2 GPGPU

多量のコアを並列に動作させることで，単純かつ膨大な計算処理を高速に行えるという GPU の特性を，描画処理だけでなく計算に応用しようとして“GPGPU”（General Purpose computing on GPU）が考え出された．GPU はもともと画像処理機能に特化したハードウェアであるため，逐次的かつ複雑な命令の処理を得意とする CPU とは異なり，並列的かつ単純な命令の処理を得意としている．そのため GPGPU では定型的な処理を繰り返すといった単純なプログラムを，GPU に処理させることで処理速度の高速化を行うことができる．

2.3 CUDA

CUDA とは“Compute Unified Device Architecture”の略であり，NVIDIA 製 GPU に対する統合開発環境である．本来描画処理に使われる GPU を CPU 上でのプログラミングが可能なように設計されており，GPU で処理する部分以外は C 言語のように記述することができる [17]．CUDA のプログラム構成は，CPU を動作させる「ホスト・コード」と GPU を動作させる「デバイス・コード」からなる．GPU を使って計算処理を行うには，まず計算で扱うデータをデバイス側に渡す必要がある．デバイス側はホスト側から渡されたデータを使って処理を行い，出力されたデータをホスト側に送り返すという手順を踏まなければならない.

3 CRCW PRAM モデルについて

CPU での実装を想定した従来の逐次アルゴリズムは，1 つのプロセッサがメモリの読み込みと書き込みを行いつつ計算する RAM(Random Access Machine) モデルと呼ばれる計算モデルを仮定して評価を行っている．その RAM を複数個持つことで並列計算を行えるようにしたモデルが PRAM(Parallel Ramdon Access Machine) であるため，PRAM は並列コンピューティングに適用可能なアルゴリズムを設計するための抽象なモデルとなっている．PRAM は同期式のモデルであり，全プロセッサはクロックを共有して動作する．各クロックにおいて，メモリアクセスないし演算を 1 回実行できる．つまり PRAM のプロセッサ数を x としたとき，与えられた問題大きさが x 以下であるのならば，その PRAM は 1 ステップ，単位時間で処理を行うことができる．その PRAM にはいくつかの種類がある．そのうち，複数のプロセッサが同一番地に対して，自由に読み書きできる CRCW PRAM モデル上で動作する整数ソートアルゴリズムを用いる [16].

本研究では CRCW PRAM の中でも Sum-CRCW-PRAM モデルを使用している．このモデルは，1 つのセルに対して複数のプロセッサから同時に書き込まれたときに，書き込みを行ったプロセッサの数を書き込ませるモデルである [15].

4 アルゴリズムついて

4.1 ヒストグラムと接頭辞和

本研究において GPU 上で実装した三つのアルゴリズムの根幹をなすヒストグラムと接頭辞和について記す．まずヒストグラムとは，与えられた入力配列に対して，各要素数を計数した度数分布表を出力する問題のことを指す．本研究においては，bin_{min} から $bin_{max}(0 \leq bin_{min} < bin_{max})$ までの n 個の整数からなる入力データに対して，各要素数を計数した度数分布表を出力する．そして接頭辞和とは，与えられた入力配列に対して，出力配列の要素の k 番目 $(0 \leq k \leq n)$ に入力配列の 0 番目から k 番目までの合計値をそれぞれ出力するという問題のことを指す．この二つの問題を解くためのアルゴリズムをアルゴリズム 1 に示す.

4.2 H-P ソートのアルゴリズム

ヒストグラムと接頭辞和に基づく整数ソーティングアルゴリズムを H-P ソートと呼ぶ．このアルゴリズムは入力データが 0 から $maxVal-1$ までの整数によって構成されたデータに対して，sum-CRCW PRAM 上で動作することを前提に，$O(\log^* n)$ 時間のアルゴリズムとして提案された [3]．本研究ではその区間を一般化し，入力データを $minVal$ から $maxVal-1$ までの整数で構成されるものとした.

H-P ソートのアルゴリズムをアルゴリズム 2 に示す．H-P ソートは整数ソーティングアルゴリズムであるため，入力データの最小値 ($minVal$ とする) と最大値 ($maxVal-1$) は予め決められており，$minVal$ から $maxVal-1$ までの整数値から取得される (この入力データの要素の範囲の大きさを

アルゴリズム 1: ヒストグラム (Histogram) と接頭辞和 (Prefix-Sums)

subroutine Histogram(**int** $data[], hist[], n, bin_{min}, bin_{max}$)
1: **for** (**int** $i = 0; i < bin_{max} - bin_{min}; i++$)
2: $hist[i] = 0;$
3: **for** (**int** $i = 0; i < n; i++$)
4: $hist[data[i] - bin_{min}]++;$

subroutine Prefix-Sums(**int** $data[], data_p[], from, to, init$)
5: **int** $sum = init;$
6: **for** (**int** $i = 0; i < to - from; i++$) {
7: $sum+ = data[i];$
8: $data_p[i] = sum;$
9: }

$range(= maxVal - minVal)$ とする).

アルゴリズム 2: H-P ソート

Assumptions:
$maxVal - 1$: maximum value among input data.
$minVal$: minimum value among input data.

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

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

Algorithm
1: Histogram$(x, A, n, minVal, maxVal)$;
2: Prefix-Sums$(A, A_p, minVal, maxVal, 0)$;
3: Histogram$(A_p, B, range, 0, n)$;
4: Prefix-Sums$(B, y, 0, n, minVal)$;

定理 1. $n, maxVal-1, minVal$ をそれぞれ入力データのデータ数, 最大値, 最小値であるとし, $range = maxVal - minVal$ であるとする．もし $minVal \leq x[i] < maxVal (0 \leq i \leq n)$ であるのならば，アルゴリズム *2* は $x[0], x[1], \ldots, x[n-1]$ を $O(max(n, range))$ 時間で正しくソーティングすることができる.

アルゴリズム 2 は sum-CRCW PRAM 上で, $O(m/\log^* m)$ プロセッサ ($m = max(n, range)$) を用いれば，$O(\log^* m)$ 時間で実装することができる．ヒストグラムは m 個のプロセッサを用いて定数時間で計算でき[*1]，接頭辞和は $O(m \log^* m)$ 個のプロセッサを用いて定数時間で計算できる [6]．したがって，H-P ソートは $O(m \log m)$ 個のプロセッサを使用して定数時間で計算することができる.

[*1] sum-CRCW PRAM 上では，ヒストグラムは $O(m)$ プロセッサで定数時間で計算できる.

4.3 1-H-P ソートのアルゴリズム

1-H-P ソートは H-P ソートに改良を加えて新しく考案したアルゴリズムである．これは H-P ソートのアルゴリズム上，入力データに重複がない場合に計算過程を簡略化できる点に着目した改良である．この場合，ヒストグラムと接頭辞和の適用は一回ずつで良い．入力データに重複がない場合，入力配列のヒストグラムに対する一回目の接頭辞和の適用後，$A_p[minVal] \neq 0$ であるならば，$minVal$ は入力データの最小値である．そうでなければ $minVal$ は入力データには含まれず，$A_p[i] \neq A_p[i-1]$ となる最小の i が入力データの最小値となる．一般的に，$A_p[i]$ と $A_p[i-1]$ $(1 \leq i \leq maxVal - 1)$ の差は最大でも 1 であり，i が $A_p[i-1]$ 番目の最小の値の場合にのみ，$A_p[i] - A_p[i-1] = 1$ が成立する．したがって，アルゴリズム 3 は正しくソーティングすることができる.

定理 2. $n, maxVal-1, minVal$ をそれぞれ入力データのデータ数，入力データの最大値，最小値とする．もし $minVal \leq x[i] < maxVal (0 \leq i \leq n)$ であり，$x[i] \neq x[j] (0 \leq i < j \leq n)$ であるのならば，アルゴリズム *3* は $x[0], x[1], \ldots, x[n-1]$ を $O(range)$ 時間で正しくソーティングすることができる.

アルゴリズム 3: 1-H-P ソート

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

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

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

Algorithm
1: Histogram$(x, A, n, maxVal, minVal)$;
2: Prefix-Sums$(A, A_p, minVal, maxVal, 0)$;
3: **if**$A_p[0] \neq 0$ **then** $y[0] = minVal$;
// The input has $minVal$.
4: **for**(**int** $i = 1; i < rangekon; i++$)
5: **if** $A_p[i] \neq A_p[i-1]$
then $y[A_p[i-1]] = minVal + i$

4.4 0-圧縮 H-P ソートのアルゴリズム

H-P アルゴリズムは整数ソーティングアルゴリズムであり，$range$ が n より小さい場合，$O(n)$ 時間で計算される．そうでない場合は入力データの種類数が少なくても $O(range)$ 時間で計算される．そこで本研究では，入力データの種類数が少ない場合 (len とする)，$range$ が n より大きい場合でも効率的なソーティングアルゴリズムである，H-P ソートの変形を考案した．それは入力配列 x のヒストグラムを取った配列 A の接頭辞和を計算するとき，A から直接計算するのではなく，大

きさが len で，A の非零要素からなる新しい配列 C を作成し，C といくつかの付加情報に基づいて計算することである．C が効率的に計算できれば，x のヒストグラム A の接頭辞和は，$O(range)$ 時間ではなく $O(len)$ 時間で計算されることになる．このアルゴリズムの概要をアルゴリズム 4 に示す．len は入力データの種類数であり，$C[len+1]$ は A(入力データのヒストグラム) 中の非零要素を持ち，$i_1, i_2, \ldots, i_{len+1}$ は入力データのヒストグラム中の非零要素のインデックスを示しているため，$C[j] = A[i_j](1 \leq j \leq len)$ かつ $iC[j] = i_j(1 \leq j \leq len)$ となる．$iC[j]$ は $C[j]$ の $A[minVal..maxVal-1]$ におけるインデックスを示し，Ap(A の接頭辞和) を $C[j]$ で計算する際に使用される．

$A_p[minVal], \ldots, A_p[maxVal-1]$ を $A[minVal], \ldots, A[maxVal-1]$ ($x[n]$ のヒストグラム) の接頭辞和，$B[n+1]$ をそのヒストグラムとする．また，$C_p[len+1]$ を $C[len+1]$ の接頭辞和とする．このとき $A01[minVal..maxVal-1]$ は次のように定義される．

$A01[j](minVal \leq j \leq maxVal-1) = \begin{cases} 0 & (\textbf{if } A[j]=0) \\ 1 & (\textbf{if } A[j] \neq 0) \end{cases}$.

この $A01[minVal..maxVal-1]$ の接頭辞和を $A01_p[minVal..maxVal-1]$ と定義する．アルゴリズム 4 の実装には以下の補題を用いるが，これは定義から容易に示すことができる．

補題 1. $j(1 \leq j \leq len)$ であるとき，
$C[j] = A[A01_p[i]](\textbf{\textit{if}}\ (j = A01_p[i]))$ であり $(A[A01_p[i]] > 0))$ かつ $iC[j] = i(\textbf{\textit{if}}\ (j = A01_p[i]))$ である．

以下では，$C[0] = 0$，$iC[0] = 0$ と仮定する．

補題 2. $i(0 \leq i \leq n-1)$ であるとき，

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

である．

アルゴリズム 4 は補題 1 と補題 2 を用いてアルゴリズム 5 として実装することができる．この証明は 2 つの補題から明らかである．アルゴリズム 5 が正しく動作していることを示すのに，次の定理がある．

定理 3. $n, maxVal-1, minVal$ をそれぞれ入力データのデータ数, 最大値, 最小値とする．$minVal \leq x[i] < maxVal(0 \leq i \leq n)$ ならば，アルゴリズム *5* は $x[0], x[1], \ldots, x[n-1]$ を $O(max(n, range))$ 時間で正しくソーティングすることができる．

アルゴリズム 5 の計算量は，アルゴリズム 2 の計算量と同じである．しかし，この二つのアルゴリズムをよく比較すると，同一の工程は最初のヒストグラムと大きさ $range$[*2]の接頭辞和に，最後の大きさ n の接頭辞和である．そしてアルゴリズム 2 の B を得るための大きさ $range$ の二回目のヒストグラムと，アルゴリズム 5 の大きさ $len+1$ の C と iC に，大きさ n の B の計算を比較する必要がある．大きな違いは，アルゴリズム 5 では二回目のヒストグラムの計算が不要なことである．GPGPU ではヒストグラムの計算に時間がかかるため [8]，GPGPU で実装した場合，アルゴリズム 5 はアルゴリズム 2 よりも速くなる可能性がある．次章以降では実際に GPGPU で実装した際の実験について説明していく．

[*2] アルゴリズム 5 では，$A01$ の計算に少し余分な時間があるが，これら二つはほぼ同じであると見なされる．

アルゴリズム 4: 0-Compressed H-P algorithm (abstract)

Assumptions:
$maxVal-1$: maximum value among input data.
$minVal$: minimum value among input data.

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

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

Algorithm
1: Histogram$(x, A, n, minVal, maxVal)$;
2: 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.
3: Let $C[len+1]$ and $iC[len+1]$ be defined as follows:
4: $C[j] = \begin{cases} unused & (\textbf{if } j = 0)) \\ A[i_j] & (\textbf{if } 1 \leq j \leq len) \end{cases}$
5: $iC[j] = \begin{cases} unused & (\textbf{if } j = 0)) \\ i_j & (\textbf{if } 1 \leq j \leq len) \end{cases}$
6: Compute Histogram B of Prefix-Sum A_p of A by using C and iC;
7: Prefix-Sums$(B, y, 0, n, 0)$;

---

5 GPU での実装について

本章では，4 章で紹介したアルゴリズムの GPGPU での実装方法や実装に伴う工夫点などについて説明していく．

本研究では H-P ソート (アルゴリズム 2)，1-H-P ソート (アルゴリズム 3)，0-圧縮 H-P ソート (アルゴリズム 5) のすべてが CUDA C/C++ 言語 [11] と CUB ライブラリ [10] の一部によって実装されている．また，すべてのアルゴリズムが上述したヒストグラムと接頭辞和のサブルーチンを使用している．

サブルーチンヒストグラム (Histogram) は，図 1 のように CUDA のアトミック関数である atomicAdd() を用いて実装されている．CUDA のカーネル関数として定義した incCnt 関数では，”atomicAdd(&cnt[a[i]-minVal], 1);”を図中の 6〜8 行目に示す 3 ステップに分割して実装している．これは配列 $a[]$ へのコアレスアクセスと，配列 $cnt[]$ への非コアレスアクセスを分離することが目的である．つまりこの 3 ステップの実装は，コアレスアクセスの実行と非コアレスアクセスの

アルゴリズム 5: 0-Compressed H-P algorithm (Implementation)

Assumptions:
$maxVal-1$: maximum value among input data.
$minVal$: minimum value among input data.

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

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

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

実行が重ならないようにする意図がある．そのため，3 ステップの実装では__syncthreads() 関数をコアレスアクセスと非コアレスアクセスの実行の間 (2 ステップ目) に使用することで，同じスレッドブロックが同時に動作しないようにしている．

サブルーチン接頭辞和 (Prefix-Sums) は，カーネル関数を呼び出してパラメータ $init$ を $data[from]$ に追加した後，CUB ライブラリの cub::DeviceScan::InclusiveSum() 関数 (($data+from$) から始まる配列で，$data$ が C 言語の整数へのポインタとして実装されている場合) を呼ぶことで実装されている．このカーネルは $data[from]$ を破棄してしまうが，本研究のアルゴリズムでは $data[]$ は常に作業配列を指すので問題はない．使用している CUB ライブラリでの実装は，現在 GPGPU での最速の接頭辞和の実装である．

アルゴリズム 2，アルゴリズム 3，アルゴリズム 5 に示すように，全てのアルゴリズムで入出力配列以外の補助配列が使用されている．これらの配列は入力に応じて動的にサイズが決定されるため，CUDA ライブラリの cudaMalloc() 関数を用いて動的に確保される．しかし cudaMalloc() 関数自体が実行に時間がかかるため，配列単位で呼び出してしまうと全体の実行時間に大幅な遅延が生じてしまう．そこで各アルゴリズムの実装では，cudaMalloc() 関数と cudaFree() 関数を一回だけ呼び出すことで，各アルゴリズムのすべての補助配列に必要なサイズのメモリブロックを確保・解放している．一括で確保した後に，各補助配列はメモリブロックの一部に手動で割り当てられている．アルゴリズム 5 の補助配列 C，C_p，iC は，入力データの内容に依存した大きさ $len+1$ を持っている．アルゴリズム 5 の 5 行目が実行されるまで，この大きさは決定していない．大きさが決定してから三つの配列のメモリを確保すると，cudaMalloc() 関数を二回呼び出す必要があり，結果的に実装が非常に遅くなってしまう．しかし，入力配列の大きさ n が入力データの内容に依存しない $len\leq n$ があるため，三つの配列のそれぞれについて，$len+1$ の代わりに大きさ $n+1$ を割り当てることで，cudaMalloc() 関数と cudaFree() 関数の呼び出しを一回のみにすることができる．アルゴリズム 2，アルゴリズム 3，アルゴリズム 5 では，$maxVal$ と $minVal$ が各アルゴリズムの入力として与えられるため，各アルゴリズムの開始時に補助配列 A，A_p，$A01$，$A01_p$ の大きさを決定することができる点に注意してほしい．

アルゴリズム 3 については，図 2 に示すように，3〜5 行目を一つの CUDA カーネルとして実装している．このカーネルを"OneHPTail <<< $(maxVal-minVal+255)/256,256$ >>> $(minVal,maxVal,A_p,y)$;" と表記している．そしてアルゴリズム 3 の 4〜5 行目は，"**for**(**int** $i=0;i<n;i++$) $y[A_p[x[i]]-1]=x[i]$;"と等価である．しかしながら予備実験では，等価なコードに基づく実装の方が遅いことがわかった．これは配列内での参照を重ねるほどメモリへのアクセス数が増えることにより，より重くなるためだと思われる．

アルゴリズム 5 では，図中の 2〜3 行目，8〜12 行目，17 行目がそれぞれ単一の CUDA カーネルとして実装されている (図： 3，4，5)．図中のカーネルをそれぞれ"binarize <<< (range + 255) / 256, 256 >>> (range, A + minVal, A01 + minVal);"，"compressA01p <<< (range + 255) / 256, 256 >>> (minVal, maxVal, A, A01p, C, iC);"，"expandToB <<< (len + 255) / 256, 256 >>> (len, Cp, iC, B);"と表記している．アルゴリズム 5 の 15 行目と 16 行目は，それぞれ CUDA ライブラリの cudaMemset() 関数を呼び出すことで実装されている．

6 実験について

6.1 実験環境

実験は表 1 に示す環境で行った．CPU は Intel Xeon CPU E5-2650 v4 を，GPU は NVIDIA Tesla P100 をそれぞれ使用した．そして統合開発環境として CUDA toolkit version 11.1 を使用した．

```
1  __global__ void incCnt(int n, int *a, int *cnt,
       int minVal)
2  {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    if (i >= n) return;
5
6    int pos = a[i] - minVal;
7    __syncthreads();
8    atomicAdd(&cnt[pos], 1);
9  }
10
11 inline void
12 Histogram(int *data, int *hist, int num, int
       bins_min, int bins_max)
13 {
14   cudaMemset(hist + bins_min, 0, sizeof(int) * (
         bins_max - bins_min));
15   incCnt<<< (num + 255) / 256, 256 >>>(num, data
         , hist);
16 }
```

図 1　サブルーチン：ヒストグラム (Histogram) の実装

```
1  __global__ void OneHPTail(int minVal, int maxVal
       , int* Ap, int* y)
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.
           x;
4      if (i > maxVal - minVal) return;
5
6      if (i == 0) {
7          if (Ap[0]) y[0] = minVal;
8      }
9      else if (Ap[i - 1] != Ap[i]) y[Ap[i - 1]] =
           minVal + i;
10 }
```

図 2　アルゴリズム 3 の 3〜5 行目の実装

```
1  __global__ void binarize(int n, int* cnt, int*
       cnt01)
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.
           x;
4      if (i >= n) return;
5
6      cnt01[i] = (cnt[i] > 0) ? 1 : 0;
7  }
```

図 3　アルゴリズム 5 の 2〜3 行目の実装

表 1　実験環境

	CPU	GPU
コア数	12x2	3584
メモリ容量	32GB DDR4	16GB GDDR5
メモリ帯域幅	76.8 GB/s	720 GB/s

6.2　実験方法

実験では，CUB ライブラリの中で最も高速なソートアルゴリズムである CUB ソートと，4 章で紹介した三つのアルゴリズムの計算時間を比較することで性能の評価を行った．

各アルゴリズムについて、n、$maxVal$、$minVal$(または $range = maxVal - minVal$ や $\delta = \frac{n}{range}$)，len(または $\sigma = \frac{range}{len}$) をパラメータとして入力データに対して計測を行なった．予備実験では $range$ が同じであれば，各アルゴリズムの計算時間は $maxVal$ と $minVal$ にほとんど依存しない．よって，以下では $minVal = 0$ とする．したがって入力データのパラメータは特に断らない限り δ と σ(1 章で定義), n である．提案したアルゴリズムと H-P ソートの計算時間はこれらのパラメータに依存する．一方，CUB ソートの演算時間は以下の実験に示すように，これらのパラメータに影響されず，ほぼ入力データのデータ数である n にのみ影響される．

```
1  __global__ void
2  compressA01p(int minVal, int maxVal, int* A, int
       * A01p,
3  int* C, int* iC)
4  {
5      int i = blockIdx.x * blockDim.x + threadIdx.
           x;
6      if (i >= maxVal - minVal) return;
7
8      if (A[i]) {
9          int x = A01p[i];
10         int y = A[i];
11         __syncthreads();
12         C[x] = y;
13         iC[x] = minVal + i;
14     }
15 }
```

図 4　アルゴリズム 5 の 8〜12 行目の実装

```
1  __global__ void expandToB(int len, int* Cp, int*
       iC, int* B)
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.
           x;
4      if (i >= len) return;
5
6      int x = Cp[i];
7      __syncthreads();
8      B[x] = iC[i + 1] - iC[i];
9  }
```

図 5　アルゴリズム 5 の 17 行目の実装

測定に使用するデータはパラメータで特徴付けることができる．使用したデータセットは 5 種類であり，以下ではそれぞれのデータセット毎に小節を設け，実験内容について説明する．以下の図において時間単位はいずれもミリ秒 (ms) であり，”CUB ソート”を”CUB”，”H-P ソート”を”H-P”，”0-圧縮 H-P ソート””0-圧縮”，”1-H-P ソート”を”1-H-P”と表記している．

6.3　実験結果

6.3.1　重複のないデータ

1-H-P ソートを評価するために，4 つのアルゴリズムの計算時間を異なるデータを用いて比較した．ここでは $\delta = 1, 0.5, 0.25$ の場合について検討した．ただし，len は n と等しいため，n と δ で σ が決定される．n が，20 万から 200 万までの $\delta = 1$ と，20 万から 130 万までの $\delta = 0.5$，20 万から 70 万までの $\delta = 0.25$ において，1-H-P が最速であることが分かった．つまり，20 万以上の小さな n では 1-H-P ソートが最速である．この際の実験結果を図 6 に示す．また δ が 1 に近しいほど，1-H-P が最も高速に動作する n の区間が増加する．これは δ が増加すると $maxVal$ が減少するためである．

6.3.2　重複のあるデータ

H-P ソートと 0-圧縮 H-P ソートを評価するために重複のあるデータに対して，$\delta \in \{1, 10, 50\}$ と $\sigma \in \{10, 50, 100\}$ の全ての組み合わせで，1-H-P ソート以外の 3 つのアルゴリズムの計算時間を比較した．δ が小さい場合，n が小さくない場合は CUB ソートが最速となる．しかし，δ が大きい場合，$maxVal$ が小さいため H-P ソートや 0-圧縮 H-P ソートは CUB ソートより高速になる．n が 100 万以上 2000 万以下の場合，$delta$

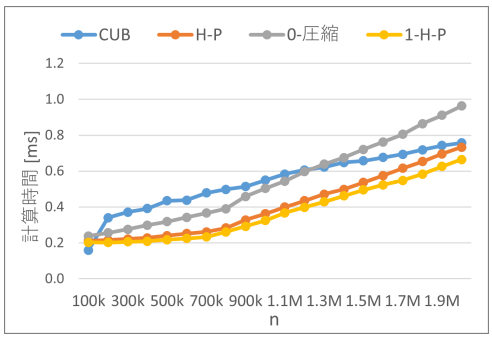

図 6　重複のないデータの計算時間 ($\delta = 1$)

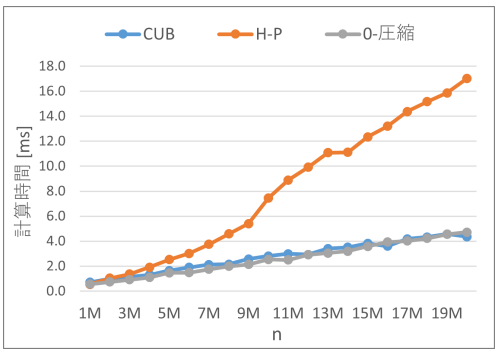

図 8　小さい len で固定したデータの計算時間 ($\delta = 1$, $len = 100$)

が小さい場合と大きい場合の境界は $maxVal = 10$ となる．また，最速であるパターンは σ にも依存しており，H-P ソートと 0-圧縮 H-P ソートが高速に動作する最小の n は σ が大きくなるにつれて増加していく．この際の実験結果を図 7 に示す．また，H-P ソートは多くの場合，0-圧縮 H-P ソートより高速であるが，その差はほとんど全ての場合において非常に小さいものである．

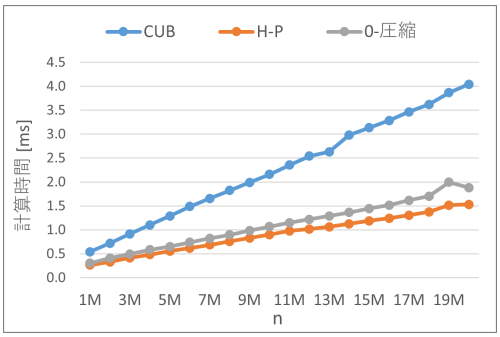

図 7　重複のあるデータの計算時間 ($\delta = 50$, $\sigma = 50$)

6.3.3　len を固定したデータ

len が小さい場合，0-圧縮 H-P ソートはより高速であると予想される．それを検証するため重複のあるデータで n が 20 万以上 2000 万以下における，1-H-P ソート以外の三つのアルゴリズムの計算時間を，小さい値の len で固定して比較する．len を固定すると n に応じて σ が変動する点に注意してほしい．δ が大きい場合，H-P ソートや 0-圧縮 H-P ソートは CUB ソートよりも高速である．これは”重複のあるデータ”で示したように，一般的な len の場合と同じ傾向である．また δ が小さくても，例えば $\delta = 1$ のときに len が小さい場合には 0-圧縮 H-P ソートが最も速いことが分かる．この際の実験結果を図 8 に示す．

6.3.4　$range$ に依存するデータ

$len = range$ の場合に，重複のあるデータに対する 1-H-P ソートを除いた三つのアルゴリズムの計算時間を比較する．このデータは $range$ に依存しているため，$\sigma = 1$ である．$\sigma = 1$ とし，n が 100 万以上 2000 万以下，$\delta = 1, 10, 50$ の場合について検証する．$\delta = 1$ のとき，$n = 100$ 万以外では CUB ソートが最も速い．これは δ が非常に小さいことによって $range$ が非常に大きくなっていることが原因である．H-P ソートと 0-圧縮 H-P ソートは $\delta = 10, 50$ の場合は $range$ が小さいため，CUB ソートよりも高速であることが分かる．また，この場合 len が非常に大きいため，H-P ソートは 0-圧縮 H-P ソートより高速となる．しかしながら，多くの場合において H-P ソートと 0-圧縮 H-P ソートの差は，H-P ソートと CUB ソートの差に比べれば無視できるほど小さい．

ここまでの実験結果からも分かる通り，H-P ソートと 0-圧縮 H-P ソートは基本的に δ が大きくなるにつれて高速になる．

6.3.5　$maxVal$ を固定したデータ

定理 1 と定理 3 より，三つのアルゴリズムの計算量は $maxVal$ に依存することがわかる．そこで重複のあるデータの n が 20 万以上 2000 万以下における，1-H-P ソートを除く三つのアルゴリズムの計算時間を，任意の値に固定された $maxVal \in \{1000, 10000, 100000\}$ と $\sigma \in \{1, 10, 50, 100\}$ の組み合わせを変えて比較した．なお，$maxVal$ を固定すると δ が n に応じて変動していく点に注意すること．このデータセットにおいて H-P ソートと 0-圧縮 H-P ソートは $maxVal \in \{1000, 10000, 100000\}$ が大きいほど速くなっていることがわかった．図 9 では $maxVal = 100$ と非常に小さいにも関わらず，最も早いのは CUB ソートであることがわかった．また $maxVal = 1000$，$\sigma = 100$ の際には，徐々に CUB ソートの方が高速になっていくのことがわかった．

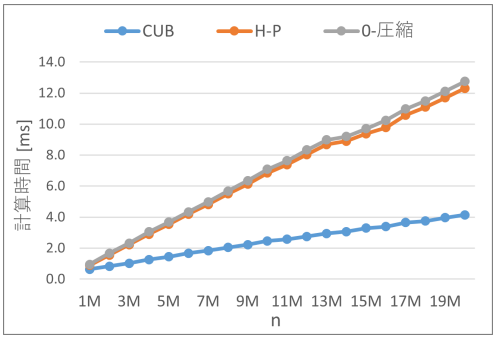

図 9　小さい $maxVal$ で固定したデータの計算時間 ($range = 100$, $len = 100$)

7　結論

ヒストグラムと接頭辞和に基づく効率的な整数ソーティングアルゴリズムを提案し，それらの GPGPU 上での実装が，ソーティングアルゴリズムとして知られている CUB ソートよりも高速であることを示した．ただし，どのようなデータに対しても高速化が実現できたわけではない．

H-P ソートが最も効果的なのは，図 7 のように，n に対して $range$ が小さい場合である．ただし，GPU 上で実装する際は少数のメモリにアクセスが集中すると，atomicAdd() 関数を使用しているサブルーチンであるヒストグラムが遅くなってしまうため，$range$ に対して len が小さいほど効率が良い．

1-H-P ソートが最も効果的なのは，n が 10 万から 200 万，$\delta = 1$ のときであった．そして δ が 1 に近しいほど，1-H-P が最も高速に動作する n の区間が増加することがわかった．

0-圧縮 H-P ソートはほとんどの場合において H-P ソートと同等の性能を示したが，H-P ソートが苦手とする $maxVal$ の大きいデータでも，実験 6.3.3 の $len = 100$ と 1000 のように，len が小さければ 0-圧縮 H-P ソートは性能を保つことができた．これは 0-圧縮 H-P ソートがそのアルゴリズム上，$maxVal$ だけでなく len の大きさに依存しているためだと考えられる．

8 今後の課題

今後の重要な課題として，ソーティングアルゴリズムを安定化することが挙げられる．残念ながら本研究で提案したアルゴリズムは安定ではないため，これらのアルゴリズムの効率を保ちながら安定化させることは，非常に魅力的な今後の研究課題の一つである．

安定なソーティングアルゴリズムとは，入力データに同じ値がある場合，それら入力の相対的な順序維持して出力するものである．安定なソーティングアルゴリズムにすることができれば，ソーティングを正しく動作させるためには，各桁のソートが安定でなければならない基数ソートなどへの応用が可能となる．

参考文献

[1] D. I. Arkhipov, D. Wu, K. Li, and A. C. Regan. 2017. Sorting with GPUs: A survey. *arXiv:1709.02520v1* (2017), 1-17.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2010. *Introduction to Algorithms Third Edition*. PHI Learning.

[3] S. C. Eisenstat. 2007. $O(\log^* n)$ algorithms on a Sum-CRCW PRAM. *Computing* 79(2007), 93 – 97.

[4] N. Faujdar and S. Ghrera. 2016. Performance evaluation of parallel count sort using GPU computing with CUDA. *Indian Journal of Science and Technoogy* 9, 15(2016), 1 – 12.

[5] N. Faujdar and S. Saraswat. 2017. A roadmap of parallel sorting algorithms using GPU computing. In *Proceedings of International Conference on Computing, Communication and Automation, ICCCA2017*. 736 – 741.

[6] F. Frei and K.Wada. 2019. Efficient circuit simulation in MapReduce. In *Proceedings of ISAAC 2019, LIPIcs, Vol;. 149*. 55:1 – 55:22.

[7] Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel Prefix Sum (Scan) with CUDA. In GPU Gems 3, Hubert Nguyen (Ed.). Addison-Wesley Professional, Chapter 39, 851 – 876.

[8] S. Hellfritzsch. 2018. Efficient Histogram Computation on GPGPUs. In *Master' s Thesis, University of Copenhagen*. 1 – 98.

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

[10] NVIDIA Corp. 2021. *CUB*. Retrieved 2021-4-23 from https://docs.nvidia.com/cuda/cub/index.html

[11] NVIDIA Corp. 2021. *CUDA C++ Programming Guide*. Retrieved 2021-4-23 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] NVIDIA Corp. 2021. *Thrust*. Retrieved 2021-4-23 from https://docs.nvidia.com/cuda/index.html

[13] W. Sum and Z. Ma. 2009. Count sort for GPU computing. In *Proceedings of 2009 15th ICPDS*. 919 – 924.

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

[15] F. Frei, K. Wada. 2019. *Efficient Circuit Simulation in MapReduce*. arXiv:1907.01624v1 [cs.CC] 2.

[16] 久野 靖. 2012. 電子通信情報学会「知識の森」, 6 群 3 編 5 章. http://www.ieice-hbkb.org/

[17] J. Snaders, E. Kandrot. 2011. 株式会社クイープ, インプレスジャパン. *CUDA BY EXAMPLE An Introduction to General-Purpose GPU Programming*.

謝辞

本研究を行うにあたりご指導を頂いた和田幸一教授，ならびに大阪府立大学の藤本典幸教授に深く感謝いたします．また様々な面において貴重なご意見，ご指導頂いた本研究室の皆様に感謝いたします．

本論文に関する発表

1) 小堺 海叶, 藤本 典幸, 和田 幸一, ヒストグラムと接頭辞和計算に基づく整数ソーティングのための GPU による効率的な実装, 第 16 回情報科学ワークショップ, 2020
2) Seiya Kozakai, Noriyuki Fujimoto, Koichi Wada, Efficient GPU-Implementation for Integer Sorting Based on Histogram and Prefix-Sums, International Conference on Parallel Processing 2021, 2021
3) Seiya Kozakai, Noriyuki Fujimoto, Koichi Wada, Integer sortings algorithms suitable for GPU-implementation, 第 17 回情報科学ワークショップ, 2021