DBSJ Japanese Journal, Vol.13-J, No.2
February 2015

データ分割と協調的マージに基づく GPU 上の効率的ソートアルゴリズム

An Efficient Sorting Algorithm on GPUs via Data Partitioning and Cooperative Merge

小澤 佑介♡ 天笠 俊之♢ 北川 博之♢

Yusuke KOZAWA Toshiyuki AMAGASA Hiroyuki KITAGAWA

ソートはデータベースをはじめ，コンピュータサイエンスにおける基本的な処理の一つであり，その高速化は非常に重要である．そのため，近年急速に性能が向上している **GPU (Graphics Processing Unit)** を用いたソートの研究が数多く存在する．**GPU** を用いたソートアルゴリズムの代表の一つにマージソートベースのものがある．このアルゴリズムには，ソートするデータが巨大な場合に性能が劣化する，という問題がある．本稿では，この欠点を解消した新たなアルゴリズムを提案する．本アルゴリズムは，データを複数のバケットに分割してから，これらのバケットを **GPU** 上で協調的にソートすることで，高性能を達成する．

Sorting is a fundamental operation in computer science, especially database systems, and its acceleration has significant importance. Thus there have been a large number of sort algorithms using GPUs (Graphics Processing Units), which rapidly increase their performance. One of the fastest algorithms on GPUs is an algorithm based on merge sort. This algorithm, however, has the problem that it slows down when sorting large-scale data. This paper presents a novel GPU-based algorithm suited for sorting large-scale data. The algorithm achieves high performance by first partitioning the data into multiple buckets and sorting the buckets cooperatively.

1. はじめに

ソートはコンピュータサイエンスにおける基礎的な処理の一つであり，様々なアルゴリズムやアプリケーションに活用されている [6]．データベースにおいても，ソートは基本的な演算であり，その高速化は非常に重要である [4]．そのため，並列化を含めた多数の研究が存在する [3,4]．中でも近年では，性能向上が目覚ましい GPU (Graphics Processing Unit) を用いた高速化が注目されている [2,7,9,12]．

GPU は元来グラフィックス処理用プロセッサであったが，現在は多数のコアを搭載する並列処理用プロセッサとして注目を集めている．このように，GPU を一般的な処理の高速化に利用する手法を GPU コンピューティングという [11]．GPU は低消費電力かつ安価で，非常に高い理論ピーク性能を持つため，科学計算をはじめ様々な分野で利用が進んでいる．前述の通り，近年では GPU によるソートの研究も多数存在する．

♡ 学生会員　筑波大学大学院システム情報工学研究科
kyusuke@kde.cs.tsukuba.ac.jp
♢ 正会員　筑波大学システム情報系
{amagasa, kitagawa}@cs.tsukuba.ac.jp

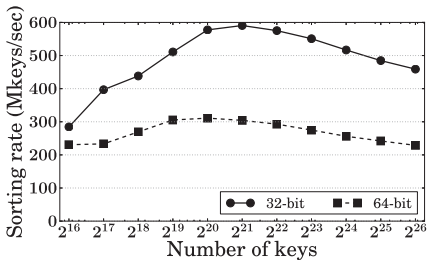


図 1: Baxter によるマージソート [2] の性能．横軸は入力の要素数，縦軸は性能 (一秒間にどれだけソートできたか) を示している．

現在までに提案されている GPU 上のソートアルゴリズムの中で代表的なものとして，基数ソートによるもの [9] と，Baxter のマージソート [2] の二つがある．基数ソートは 32-bit のキーをソートする場合には高速であるが，大きなキーをソートする場合には大幅に性能が劣化することが知られている．また，基数ソートはデータのビット表現を利用するアルゴリズムであるため，適用できるデータの種類が限られている．そこで本研究では，より汎用的なアルゴリズムである後者のマージソートによる実装に着目し，新たな GPU 上のソートアルゴリズムを提案する．

図 1 は Baxter のマージソートの性能調査の結果を示している．ここでは，一様分布に基づき 32-bit と 64-bit の整数の配列をそれぞれ生成し，各配列のソートを行った．この図から，要素数が 2^{20} 付近で性能がピークを迎え，入力を大きくするにつれて，性能が低下していくことが分かる．これは，メモリバンド幅がボトルネックとなるためと考えられる．マージソートは計算量が $O(n \log n)$ であり，これにしたがってメモリアクセス回数も増加する．そのため，小さいデータではメモリアクセス回数も少なく，また GPU の並列度を十分に活かせていないため，2^{20} あたりまでは性能が向上していく．一方，データが大きくなってくると，メモリバンド幅がボトルネックとなり，性能が劣化してしまう．

本稿では，以上の欠点を解消した新たな GPU 上のソートアルゴリズムを提案する．本アルゴリズムは，はじめにデータを複数のバケットに分割することで，後のマージの回数を削減する．データの分割はマージと比較して，メモリアクセスが少なく実現できるため，性能向上が期待できる．また，複数のバケットを作ったうえでソートを行う際に考慮すべき点として，負荷分散がある．本研究ではこの点を考慮して，すべてのバケット内を GPU 上で協調的にマージしていく方法を提案する．さらに，評価実験によって，提案アルゴリズムが Baxter のマージソートより最大で 39%高速となることを示す．

以降の本稿の構成は以下の通りである．2. 節で予備知識について説明する．具体的には，GPU コンピューティングと GPU 上のソートアルゴリズムについて述べる．3. 節で提案アルゴリズムについて説明し，4. 節でその評価を行う．最後に 5. 節でまとめと今後の課題について述べる．

2. 予備知識

2.1 CUDA

GPU コンピューティングとは，元来グラフィックス処理用プロセッサであった GPU を，汎用的なプロセッサとして利用し計算の高速化を図る技術を指す．GPU を用いたアプリケーションを作成する際には，NVIDIA 社によって提供されている **CUDA** (Compute Unified Device Architecture) [1] が主に用いられる．CUDA が対象とする GPU は以下のような構成に成っている．GPU は複数の SM (Streaming Multiprocessor) を含んでおり，各 SM は 数十

[1] https://developer.nvidia.com/cuda-toolkit

から数百の SP (Streaming Processor) を搭載している．CUDA では，一つのカーネル (GPU 上で実行される関数) に対して，数万から数十万の軽量のスレッドを並行に走らせることで，高い並列度を実現している．

CUDA のスレッドは以下のような階層構造を成している．まず，一つのカーネルの実行にともなって，一つのグリッドが作成される．グリッドは複数のブロックから構成されており，各ブロックは数百のスレッドを持つ．ブロックは一つの SM に割り当てられ，その SM 上で処理が実行される．ブロック内のスレッドは，SM が持つ高速にアクセス可能な共有メモリを利用することで，同じブロック内のスレッドとデータのやり取りができる．一方，SM は複数のブロックを並行に走らせることが可能である．このとき，同じ SM に同時に割り当てられているブロックは，SM が持つ資源を共有しながら実行することになる．基本的にはブロック間の同期はできないため，ブロック間ではデータの依存関係が無いように問題空間を分割する必要がある．ブロック間同期が必要な場合には，複数のカーネルを用いることで対処する．

GPU 向けの最適化で重要な点の一つに，種々のメモリの活用がある．CUDA で主に利用されるメモリには三種類のものがある．グローバルメモリは GPU 上で最大のメモリで，数 GB の容量を持つ．GPU で処理するデータは基本的にはこのメモリ上に置かれる．データの再利用や計算の途中経過を保持するには，高速にアクセス可能なオンチップの共有メモリやレジスタが利用可能である．

グローバルメモリの利用において留意すべき点として，コアレスドアクセスがある．通常，グローバルメモリへのアクセスは複数のトランザクションにわけられ処理される．しかし，一つの warp (32 スレッドの集まり) 内のスレッド全てが，アラインされた 128 バイト領域にアクセスしている場合，これは一つのトランザクションで実行される．これをコアレスドアクセスと呼ぶ．コアレスドアクセスが実現されているか否かで大きく性能が変化するため，アルゴリズム設計時に十分に考慮する必要がある．

2.2 GPU 上のソートアルゴリズム

既存の GPU 上のソートアルゴリズムは大きく分けて，基数ソートによるもの [9] と，比較ソートベースのもの [2,7,12] の二種類がある．GPU 上の基数ソートに関する研究はいくつか存在するが，Merrill ら [9] によるものが最速と考えられる．この研究をもとに，更に最適化を施した基数ソートの実装が，ライブラリの一部として公開されている [8]．基数ソートは，32-bit のキーを対象としたソートの場合，他のアルゴリズムよりも高速である．一方，基数ソートはキーのビット表現を利用しているため，ソート可能なデータが限定されるという欠点がある．また，計算量がソートに用いるキーのビット長に依存しているため，64-bit 整数キーなど，キー長が長い場合には著しく性能が落ちるという問題もある．

マージソートなどの比較ソートベースの研究も数多く存在する．ここで比較ソートとは，基数ソートとは異なり，キー同士の比較のみによってソートするアルゴリズムを指す．GPU 上の比較ソートの代表的なものとしては，マージソート [2]，バイトニックソート [1,12]，サンプルソート [7] などがある．これらのアルゴリズムは，キー同士の比較関数さえ用意できれば，ソート対象のデータには制限がない．ただし，理論上，比較ソートは基数ソートよりも計算量自体は大きくなってしまう [6]．一方，計算量がビット長に依存しないため，64-bit のキーのソートの場合では基数ソートより高速となる．

本研究では，高速な GPU 上の比較ソートアルゴリズムを提案する．以降では，本研究がベースとしている，サンプルソート [3,7] とマージソートの一実装 [2] について説明する．

2.2.1 サンプルソート

Leischner ら [7] はサンプルソート [3] の GPU 上の実装を提案した．サンプルソートは，簡単に言うと，クイックソートを拡張したアルゴリズムである．クイックソートが入力を再帰的に二分割していくのに対し，サンプルソートでは k 個に分割していく．

Algorithm 1: サンプルソート

```
   入力: ソート対象の配列 A
1  if |A| > M then
2  |  S ← 入力 A から αk 個の要素をランダムに選択
3  |  S をソート
4  |  (s_0, s_1, ..., s_k) ← (−∞, S_α, S_2α, ..., S_(k−1)α, ∞)  ▷ s_i は S の iα
   |  番目の要素．s_0 と s_k は最小と最大の値を表す番兵．
5  |  for x ∈ A do
6  |  |  s_(i−1) ≤ x < s_i となる i を探索
7  |  |  バケット B_i に x を格納
8  |  end
9  |  各バケット B_i に対して再帰的にサンプルソートを適用
10 |  A ← B_1, ..., B_k を結合したもの
11 else
12 |  他のアルゴリズムで A をソート
13 end
14 return A
```

サンプルソートの擬似コードは Algorithm 1 のように書ける．入力データがあるしきい値 M より大きければ，以下のように k 個のバケットに分割し再帰的にサンプルソートを適用する．

1. 入力 A から $k-1$ 個の分割点 s_i を抽出 (2–4 行)
2. 入力の各要素を k 個のバケット B_i に分割 (5–8 行)
3. 各バケット B_i に対しサンプルソートを適用 (9–10 行)

入力が十分に小さくなったら他のアルゴリズムによってソートを完了する．

Leischner らは，以上のアルゴリズムをもとに GPU 上でのサンプルソートを提案した．彼らの実装では，すべてのバケットがしきい値以下の大きさになるまで，並列に分割していく．その後，一つのバケットを一つのブロックでソートしていく．この実装の問題点の一つに，バケットの大きさの違いによる負荷の偏りがある．さらに，バケットが大き過ぎたり小さ過ぎる場合，一つのブロックでソートを行うと，十分に GPU の性能を引き出せないと考えられる．

本研究で提案するアルゴリズムでは，サンプルソートのデータ分割方法によって，一度だけ分割を行う．そして，すべてのバケット内を一つのカーネルで徐々にマージしていく．分割をどのように GPU 上で実現するかについては 3.1 節で説明する．基本的には Leischner らの研究に基づいているが，いくつかの改良点が存在する．

2.2.2 マージソート

上記のように，GPU 上の比較ソートの研究は多数存在するが，現在最速のものと考えられるのは，Baxter によるマージソート [2] である．このアルゴリズムは，**Merge Path** [5,10] を利用した高速なマージに基づいている．Merge Path によって，二つのソートされた配列を並列にマージする際の，各プロセッサへのデータの分割が，高速かつ負荷の偏りが無いように実現できる．データを分割した後は，各プロセッサはそれぞれ独立に担当範囲のデータをマージし，結果を出力する．

Baxter のマージソートは以下の 2 フェイズから成る．

1. タイル内ソート：各ブロックがそれぞれ担当する範囲のデータをソートする
2. マージ：各ブロックがソートしたデータをもとに，協調的にマージをしていく

図 2 はマージソートの処理の流れを示した図である．まず，入力の配列は複数のタイルに分割され，1 ブロックが 1 タイルをソー

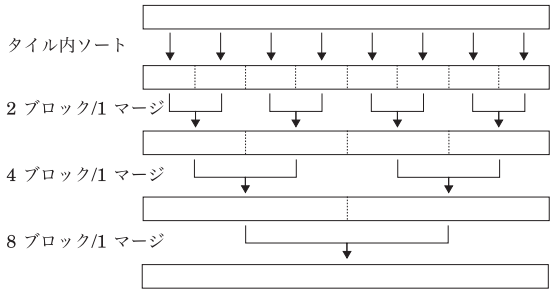


図 2: マージソートの処理の流れ.

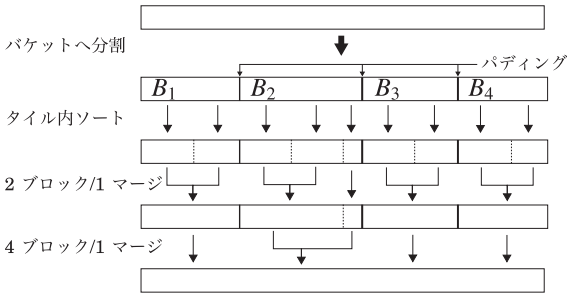


図 3: 提案アルゴリズムの処理の流れ.

トする．ここでタイルとは，一つのブロックによって独立に処理されるデータ領域を指す．タイルの大きさは，利用する GPU やデータの性質によって適切な値を設定する必要がある．

一段階のマージは以下の二つのステップで行っていく．

1. Merge Path によって各ブロックが担当する範囲を計算
2. ステップ 1 の計算結果をもとに，各ブロックが並列にマージをして出力

図 2 から分かるようにブロックの数は常に一定で，各ブロックが処理する要素数も一定である．これによって，負荷の偏りが発生しないようにしている．

本研究では，上記のマージアルゴリズムをもとに，複数バケット内のマージを一つのカーネルで協調的に行うアルゴリズムを提案する．多くのバケットに分割することで必要なマージの回数を減らすことができ，全体のメモリアクセス回数が少なく済む．そのため，メモリバンド幅がボトルネックとなるような場合に高速化が期待できる．

3. 提案アルゴリズム

本節では，サンプルソートのデータ分割方法と，Merge Path を利用した協調的マージに基づく，提案アルゴリズムについて説明する．提案アルゴリズムの処理の流れは図 3 のようになる．まず，入力の配列を受け取ったら，この配列を k 個のバケットへと分割する．その後，各バケット内をマージソートによってソートしていく．データを $k = 2^l$ 個のバケットに分割する場合，最大でマージの回数を l 回削減可能である．ただし，バケットサイズにはある程度ずれが生じるため，l 回未満の削減にとどまることもある．以降では，3.1 節でデータ分割について述べ，3.2 節で協調的マージの説明をする．

3.1 データ分割

ここでは，Leischner らの研究 [7] をもとに改良を加えた，GPU 上のデータ分割アルゴリズムについて説明していく．このアルゴリズムは以下の四つのステップからなる：1. 分割点の選択，2. バケットサイズの計算，3. Prefix sum の計算，4. データの再配置．以降では k 個のバケットへ分割することを考える．

3.1.1 分割点の選択

第一ステップでは，$k-1$ 個の分割点を入力データから選択する．まず，入力からランダムに αk 個の要素を取得する．ここで α は oversampling factor であり，大きいほど分割後のバケットサイズが均等になる [3]．実際の分割点 s_i には，取得したサンプルをソートしたときの，$i\alpha$ 番目の要素を用いる．

このステップは一つのカーネルとして実現され，1 ブロックのみで処理を行う．入力から得られたサンプルは，共有メモリに置きバイトニックソート [1] によりソートする．Leischner らは oversampling factor α を固定の値としていたが，本研究では，バケットサイズの平滑化のため，共有メモリを可能な限り使用するように設定する．例えば，共有メモリが 48 KB，$k = 128$, 入力データが 32-bit の整数であるとき，$\alpha = 48 \cdot 1024 / (4 \cdot k) = 96$ となる．抽出した分割点は Leischner らが提案しているように，二分探索木として保持する [7]．これによって，後の，ある要素がどのバケットに属するかの計算が高速に行える．

3.1.2 バケットサイズの計算

第二ステップでは，入力配列を複数のタイルに分割して，入力データがどのバケットにいくつ含まれるかのヒストグラムを各タイルごとに計算する．この結果に対して，次ステップで prefix sum (後述) を計算することで，データ分割後のバケットの位置が決定できる．

一つのブロックが一つのタイルを担当し，各ブロックが並列に計算を行う．タイル内のデータに対するバケットサイズを計算するために，共有メモリ上に，現在のタイル内におけるバケットのデータ数を保持しておく．バケットサイズを計算する際には，1 スレッドが 1 要素を調べていく．前ステップで作成した分割点の二分探索木を用いることで，各要素がどのバケットに属するかを決定する．そして，この結果をもとに，共有メモリ上の対応するバケットサイズを 1 増加させる．この際，同じバケットのサイズを同時に変更する可能性があるため，atomic 関数を用いて加算を行う．

タイル内の計算が終了したら，そのタイルのヒストグラムをグローバルメモリに出力する．このとき，異なるタイルの同じバケット ID のバケットサイズが連続するようにメモリ上に配置する．こうすることで，次ステップの prefix sum の計算で，データ分割後における各タイルの各バケットが，どの位置からはじまるかが分かるようになる．最終的に，出力結果は図 4 左の行列のようになる．この行列はグローバルメモリ上に行優先順序で保持される．

3.1.3 Prefix Sum の計算

第三ステップでは，前ステップで計算したバケットサイズの配列に対して，prefix sum の計算を行う．ここで，prefix sum とは，ある配列 $[a_1, a_2, ..., a_n]$ に対する，新たな配列 $\left[0, a_1, a_1 + a_2, ..., \sum_{i=1}^{n-1} a_i\right]$ を指す．本研究では，Yan ら [13] のアルゴリズムを実装し利用した．

本研究では，このステップで新たに二つの最適化を導入する．一つめは，prefix sum の行列の転置である．これは，次ステップでコアレスドアクセスを最大限発生させるための最適化となる．次ステップでは，j 番目のタイルを処理するブロックが，prefix sum 結果の $s_{i,j}$ $(1 \le i \le k)$ を読む必要がある．すなわち，図 4 中央の行列における一列が必要となる．しかし，この行列は行優先で保持されているため，このままでは一つの warp 中のスレッドがばらばらのアドレスにアクセスしてしまい，コアレスドアクセスが全く発生しない．一方，行列の転置を行うことで，今度は図 4 右の行列における一行を読めばよくなる．この場合，一つの warp 中のスレッドが連続した領域を読むので，コアレスドアクセスの条件を満たす．

二つめの最適化は，バケットのアラインメントである．この最適化もコアレスドアクセスに関するもので，次ステップとマージ

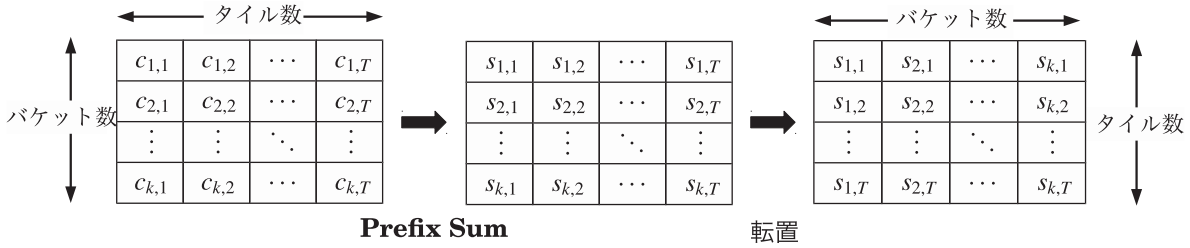


図 4: Prefix sum の計算前後の処理の流れ．ここで $c_{i,j}$ はタイル j 内のバケット i に含まれるキーの数を表す．

に影響する．コアレスドアクセスは，2.1 節で述べたように，一つの warp 内のスレッドがアラインされた **128** バイト領域にアクセスしている場合に発生する．しかし，各バケットの開始アドレスはデータやパラメータによって決まり，通常 128 バイト境界にアラインされていない．すなわち，あるバケットに対する一つの warp の読み書きは，少なくとも二つ以上のトランザクションとして処理される．これを解消するためには，各バケットの開始アドレスを調べ，アラインメントが達成されるように調整をする必要がある．これは，prefix sum 結果の $s_{i,1}$ $(1 \leq i \leq k)$ を見ることで容易に実現可能である．

3.1.4 データの再配置

ここまでに計算した結果をもとにデータの再配置を行う．第二ステップの場合と同様に，入力配列を複数のタイルに分割して，各タイルを並列に処理する．ブロックはタイル内のデータに対して，どのバケットに属するかの判定をもう一度行う．これは，再計算した方が，計算結果を保持してそれを読むよりも，高速に処理できるためである [7]．判定を行った後，対応するバケットサイズを atomic 関数により増加させる．この atomic 関数の返り値は，増加させる前の値であり，この値と前ステップで計算した prefix sum の値を調べることで，ある要素をどの位置に出力すべきかが決定できる．前ステップでアラインメントのために調整を行ったため，データの再配置後は，図 3 に示されているように，バケット間に隙間がある状態となる．

ここで，グローバルメモリへの出力を単純に行うと，一つの warp 中のスレッドがばらばらな位置に出力してしまう．すなわち，2.1 節で述べたコアレスドアクセスの条件を満たさず，性能が劣化してしまう．そこで，一度共有メモリ上でデータを並び替えてから，グローバルメモリに出力する．これによって，可能な限りコアレスドアクセスが発生するようにできる．Leischner ら [7] は単純にグローバルメモリに出力したほうが高速であると述べていたが，本研究の予備実験では，一度共有メモリ上で並び替えたほうが高速であったため，こちらを採用した．

3.2 協調的マージ

前節のアルゴリズムによってデータを k 個のバケットに分割した後，バケット内をソートしていく．本研究ではこれを 2.2.2 節で説明したマージソートのアルゴリズムを基に行う．

マージをはじめる前に，あるブロックがどのバケットを担当するかを決定する必要がある．これは，バケットごとにサイズが異なり，静的には割り当てが決定できないためである．まず，データ分割中に計算した結果をもとに，バケットサイズを保持する配列を作る．この配列から，各バケットが何個のタイルに分割されるかを計算し，その結果の prefix sum を計算する．これによって，各ブロックに関する以下の二つの情報を取得する．

- どのバケットを処理するか
- 同じバケットを処理するブロックのうち何番目のブロックか

ブロックの割り当てが終了したら，マージソートの場合と同様に，タイル内ソートを行う．ここでの変更としては，バケットの境界は跨がないようにソートをするという点がある．通常，バケットはタイルによって綺麗に分割はできないので，図 3 に見られるように，マージソートの場合と比較して，必要なブロック数は多少増加する．その後，バケット内のタイルを対象に，隣接するタイルのマージを進めていく．このとき，図 3 からわかるように，対応するタイルが存在しない場合には，何もせずそのまま出力する．マージを繰り返していき，全てのバケット内がソートされたら終了となる．図 2 と図 3 を比較すると，マージの回数が削減できていることがわかる．

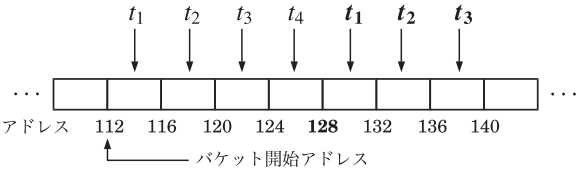


図 5: バケット位置を元に戻す際の最適化．この図は 32-bit 要素の配列に，複数のスレッドが並列にアクセスする様子を表している．ここで，t_i はあるブロック中の i 番目のスレッドを意味する．

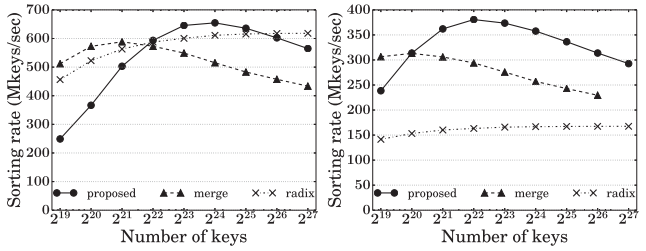
最後のマージでは，データ分割時に行ったバケットのアラインメントを元に戻す必要がある．そのためには，マージ結果を元のバケット開始アドレスに出力すればよい．しかし，単純にこの処理を行うと，3.1.4 節で述べたように完全なコアレスドアクセスにはならない．これは，1 warp 中のスレッドのアクセスが 128 バイト境界を跨いでしまっているためである．この問題を解決するために，128 バイト境界前後の書き込みを別々に扱う．すなわち，128 バイト境界前の出力は必要なスレッドだけが行い，境界後は 1 番目のスレッドから順に出力を行うように割り当てる（図 5）．これによって，境界後の出力は全てコアレスドアクセスの条件を満たすようにできる．

4. 評価実験

本節では，3 節で提案したアルゴリズムの性能を実験により評価する．実験に用いた GPU は Tesla C2050 (14 SM, 32 SP@1.15GHz, メモリ 3GB, ECC on) である．また OS として Ubuntu 12.04.3 LTS 64 bit を，CUDA toolkit はバージョン 5.5 を利用した．まず，4.1 節で，提案アルゴリズムと既存アルゴリズムとの比較を行う．4.2 節以降では，提案アルゴリズムの詳細な分析をしていく．具体的には，4.2 節でバケット数を変化させた場合の性能の変化を示し，4.3 節で最適化の効果について検証する．

4.1 既存アルゴリズムとの比較

ここでは，提案アルゴリズム，Baxter のマージソート [2]，Merrill の基数ソート [8] の三つの性能を比較する．データとして，一様分布によって生成した 32-bit 整数配列と 64-bit 整数配列を用いる．データサイズを変化させ，ソートの性能を計測した結果を図 6 に示す．ここで性能は，同じ分布に基づき 10 種類のデータを作成し，それらをソートしたときの性能の平均とした．分割数 k は，予備実験をもとに，最適な値を設定した．(詳細は 4.2 節で示す．) なお，メモリ不足などで正常に動作しなかった箇所は，図では省



(a) 32-bit 整数　　(b) 64-bit 整数

図 6: 一様分布によって生成したデータをソートしたときの性能．縦軸は性能（一秒間にどれだけソートできたか）を示している．

いている．

マージソートで用いるタイルサイズは手動で適切な値を設定した．具体的には，32-bit の場合 1408，64-bit の場合 2944 とした．また，Modern GPU には segmented sort という，複数バケットを同時にソートするカーネルが用意されている．そのため，提案アルゴリズムのデータ分割を行ったあとに，segmented sort を用いることでソート処理が実現できる．しかし，我々の予備実験では，この実装の性能は通常のマージソートとそれほど大きな違いを示さなかった．そのため，ここでは通常のマージソートを比較対象とする．基数ソートのライブラリでは，用いる GPU に応じて，一度のパスで調べるビット数を自動的に適切な値に設定される．今回の実験に用いた GPU の場合，一度のパスで調べるビット数は 5 ビットとなった．

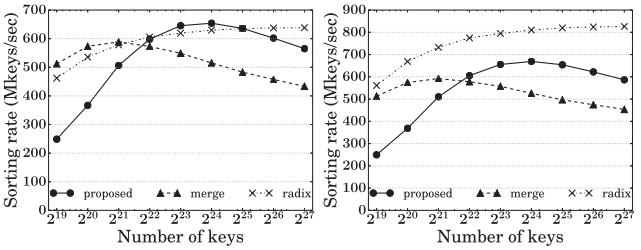
マージソートと比較して，データサイズが十分に大きい場合には，提案アルゴリズムが良い性能を示している．例えば 32-bit 整数の場合（図 6(a)），データサイズが 2^{22} 以上のときに，提案アルゴリズムがマージソートより高速となる．特に，キー数が 2^{25} のとき，提案アルゴリズムはマージソートの 1.32 倍の性能となった．また，64-bit 整数が対象の場合（図 6(b)），最大でマージソートの 1.39 倍となり，32-bit 整数の場合より多少有効性が高い結果となった．これは，64-bit 整数の方が必要メモリバンド幅が大きいため，データ分割によるマージ回数の削減がより有効に働くからと考えられる．一方，データが小さい場合には提案アルゴリズムよりマージソートが高速となった．これは，逆に分割をしすぎて GPU の性能を十分に引き出せない状態となってしまうためと考えられる．このような場合には，マージソートをそのまま利用したほうが良い．

基数ソートと比較すると，32-bit 整数のソートの場合（図 6(a)），キー数が 2^{22} から 2^{25} の間では，提案アルゴリズムが基数ソートと同等かそれ以上の性能を示している．一方，キー数が増加するにつれて，基数ソートが優位となっていく．基数ソートは計算量がデータサイズに比例するため，データが大きくなっても性能は劣化せずに済む．しかし，マージソートの計算量は $O(n \log n)$ であるため，データサイズが 2 倍になると処理時間は 2 倍超となるため，性能は徐々に低下していくことになる．64-bit 整数の場合（図 6(b)）では，2.2 節で触れたとおり基数ソートの性能は大幅に劣化しており，提案アルゴリズムが高速となる．特に，キー数が 2^{22} のときに，基数ソートと比較して提案アルゴリズムが 2.33 倍の性能を達成している．

次に，提案アルゴリズムとマージソートのより詳細な比較を行う．表 1 に，それぞれの計算時間の内訳と主なカーネルの呼び出し回数を示す．この表から，提案アルゴリズムのマージ回数は 6 回であり，マージソートの 14 回と比較して 8 回削減できていることがわかる．ここでは分割数を $k = 128 = 2^7$ としているが，提案アルゴリズムとマージソートのマージ回数の差が 8 と，7 より多くなっている．これは，それぞれのタイルサイズの設定が異なるためである．また，提案アルゴリズムのタイル内ソートが少し遅くなっているが，これは 3.2 節で述べたとおり，必要なブロック数

表 1: 計算時間の内訳とカーネル呼び出し回数（32-bit，2^{25} キー）

カーネル	提案アルゴリズム		マージソート [2]	
	時間 (ms)	回数	時間 (ms)	回数
タイル内ソート	16.5	1	13.7	1
マージ	22.4	**6**	56.5	**14**
分割点の選択	0.8	1		
バケットサイズの計算	4.4	1		
Prefix sum	0.3	1		
転置	0.3	1		
データの再配置	8.5	1		
合計	53.2		70.2	



(a) 正規分布（平均 2^{30}，標準偏差 2^{26}）　　(b) 指数分布（$\lambda = 10^{-3}$）

図 7: 一様分布以外の分布を利用したときのソート性能比較．

が多少増加してしまうためと考えられる．データ分割は 14.3 ms で実現できており，全体の処理時間は提案アルゴリズムが 17 ms 速い結果となった．

最後に，一様分布以外の分布を用いて生成した 32-bit 整数キーの配列を用いて性能比較を行う．図 7 に，正規分布と指数分布を利用した結果を示す．正規分布の場合（図 7(a)）は一様分布のときとほぼ同じ結果となった．一方，指数分布を用いたとき（図 7(b)）は，提案アルゴリズムとマージソートには大きな変化がなかったが，基数ソートの性能が大幅に向上した．これは，指数分布では多くのキーが小さな値になるためと考えられる．この場合，上位ビットのほとんどが 0 となり，基数ソートではソートが早い段階で完了する．そのため，一様分布の場合と比較して，性能が向上した．

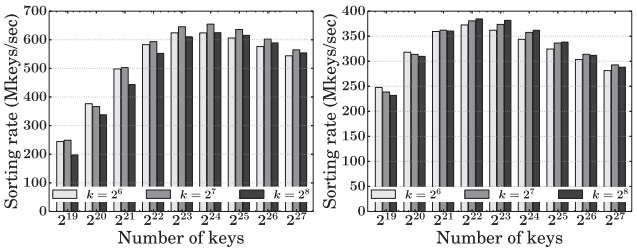
4.2 バケット数の違いによる性能変化

本節では，バケット数 k の違いによる性能の変化について調査する．前節と同様に一様分布を用いてデータを生成し，バケット数 k を $2^6, 2^7, 2^8$ としたときの性能を図 8 に示す．この図から，32-bit 整数では $k = 2^7$ が最適な値となることがわかる．64-bit 整数では，$k = 2^7$ と $k = 2^8$ の場合が良い性能を示しており，データサイズが 2^{22}–2^{25} では $k = 2^8$ が，それより大きい場合には $k = 2^7$ が優位となる．本研究で行った実験では，大きなデータに対して有効な $k = 2^7$ を採用した．

4.3 最適化の効果

最後に，3.1.3 節で導入した二つの最適化の効果を検証する．具体的には，prefix sum の計算後の行列の転置と，バケット開始位置のアラインメントである．後者の最適化には，3.2 節で述べた，最後のマージにおける最適化も含めている．図 9 に，最適化を適用した際の性能向上率を示す．この図では，二つの最適化を適用していない場合の性能を 1 としている．

図から，二つの最適化のうち，アラインメントがより効果的であることがわかる．これは二つの最適化の影響範囲の大きさの違いによるものと考えられる．アラインメントは，データ分割以降全体に影響を与えるが，転置はデータの再配置にしか影響しない．データ分割以降は全体の処理時間の半分以上を占める部分であるため，アラインメントの効果が大きい結果となった．ただし，図 9(a) におけるキー数 2^{27} のように，転置のみを適用した場合より，同時に最適化を施した方が転置がより効果的となることがある．これは以下のように考えられる．最適化をしていない場合，再配置の



(a) 32-bit 整数　　(b) 64-bit 整数

図 8: 異なるバケット数を用いた場合の性能比較.

カーネルは，入力と出力の両方でコアレスドアクセスの条件を満たさない状態である．そのため，転置によって入力はコアレスドアクセスになるが，出力は変わらないため，こちらがボトルネックとなる．そこで，アラインメントを行うことで，出力もコアレスドアクセスになり，問題が解消される．

また，キー数が増加するにしたがって，性能向上率も上昇していくことが見てとれる．さらに，32-bit 整数より 64-bit 整数の場合に最適化が有効であることもわかる．これは，データが大きくなるにつれて，要求メモリバンド幅が増加していくためである．二つの最適化はどちらも，コアレスドアクセスに関するものであり，これらが有効に機能していることがわかる．一方，データが小さい場合には，逆に最適化によって遅くなってしまうこともある．これは，最適化の導入によって発生するオーバーヘッドによるものと考えられる．例えば図 9(a) における 2^{19} では，マージの回数が 1 回のみであり，データも小さいため，最適化によって性能が劣化する結果となった．

5. おわりに

本研究では，GPU 上の高効率なソートアルゴリズムを提案した．提案アルゴリズムはサンプルに基づくデータ分割と協調的マージによってソートを行う．はじめにデータ分割を行うことで，マージソートで行われるマージの回数を減らし，必要メモリバンド幅の削減を図った．さらに，コアレスドアクセスが最大限発生するように，prefix sum の行列の転置とバケットのアラインメントの二つの最適化を導入した．その後，一つのカーネルで全てのバケット内のマージを行う，協調的マージによってバケット内のソートを完了する．実験により，データサイズが十分大きい場合に，マージソートより高速化できていることを示した．また，一様分布に基づく 32-bit 整数の配列を用いた場合には，基数ソートと同等の性能となることもわかった．さらに，64-bit 整数の場合には，基数ソートの 2 倍以上の性能となった．

今後の課題としては，データ分割方法の改良がある．今回はデータ分割の回数を 1 回に限定していたが，これを 2 回以上行うことで必要なメモリアクセスをさらに削減できると考えられる．また，現在の分割方法では atomic 関数を用いているため，ソート結果の安定性は保証されていない．この問題を改善することも検討する予定である．

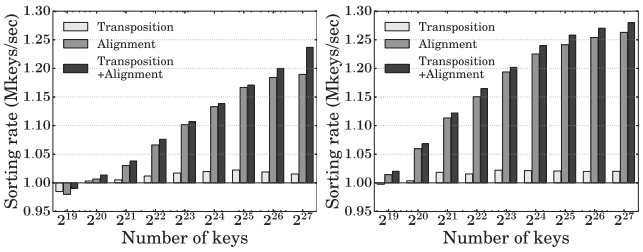
[謝辞]

本研究は，JSPS 科研費（24240015），文部科学省委託事業「実社会ビッグデータ利活用のためのデータ統合・解析技術の研究開発」，および「エクサスケール計算技術開拓による先端学際計算科学教育研究拠点の充実」事業の支援によるものである．

[文献]

[1] K. E. Batcher, “Sorting networks and their applications,” *Proc. AFIPS Spring Joint Computer Conference*, pp. 307–314, 1968.

[2] S. Baxter, “Modern GPU,” `http://nvlabs.github.io/moderngpu/`, (last accessed Feb. 2014).



(a) 32-bit 整数　　(b) 64-bit 整数

図 9: 転置（`Transposition`）とアラインメント（`Alignment`）を適用したときの性能向上率.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, “A Comparison of Sorting Algorithms for the Connection Machine CM-2,” *Proc. 3rd Annual ACM Symp. Parallel Algorithms and Architectures (SPAA)*, pp. 3–16, 1991.

[4] G. Graefe, “Implementing Sorting in Database Systems,” *ACM Comput. Surv.*, vol. 38, no. 3, article 10, Sep. 2006.

[5] O. Green, R. McColl, and D. A. Bader, “GPU Merge Path: A GPU Merging Algorithm,” *Proc. 26th ACM Int’l Conf. Supercomputing (ICS)*, pp. 331–340, 2012.

[6] D. E. Knuth, “The Art of Computer Programming, Volume 3: Sorting and Searching,” Second Edition, Addison-Wesley, 1998.

[7] N. Leischner, V. Osipov, and P. Sanders, “GPU Sample Sort,” *Proc. 24th IEEE Int’l Parallel & Distributed Processing Symp. (IPDPS)*, pp. 1–10, 2010.

[8] D. Merrill, “CUB,” `http://nvlabs.github.io/cub/`, ver. 1.1.1, 2013.

[9] D. Merrill and A. Grimshaw. “High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing,” *Parallel Process. Lett.*, vol. 21, no. 2, pp. 245–272, Jun. 2011.

[10] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, “Merge Path - Parallel Merging Made Simple,” *Proc. 26th IEEE Int’l Parallel & Distributed Processing Symp. Workshops & PhD Forum (IPDPSW)*, pp. 1611–1618, 2012.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU Computing,” *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[12] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, “A novel sorting algorithm for many-core architectures based on adaptive bitonic sort,” *Proc. 26th IEEE Int’l Parallel & Distributed Processing Symp. (IPDPS)*, pp. 227–237, 2012.

[13] S. Yan, G. Long, and Y. Zhang, “StreamScan: Fast Scan Algorithm for GPUs without Global Barrier Synchronization,” *Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, pp. 229–238, 2013.

小澤 佑介 **Yusuke KOZAWA**

筑波大学大学院システム情報工学研究科博士後期課程在学中．2013 筑波大学大学院システム情報工学研究科博士前期課程修了，修士（工学）．GPU を用いた並列データ処理技術の研究開発に従事．データベース学会学生会員．

天笠 俊之 **Toshiyuki AMAGASA**

筑波大学システム情報系准教授．データ工学，データベース，Web マイニング等の研究に従事．日本データベース学会，情報処理学会，ACM 各会員．電子情報通信学会，IEEE 各シニア会員．

北川 博之 **Hiroyuki KITAGAWA**

筑波大学システム情報系教授，ならびに計算科学研究センター教授併任．理学博士（東京大学）．データベース，データマイニング，情報検索等の研究に従事．著書「データベースシステム」（オーム社），「The Unnormalized Relational Data Model」（共著，Springer-Verlag）等．情報処理学会フェロー，電子情報通信学会フェロー，日本データベース学会会長，ACM，IEEE-CS，日本ソフトウェア科学会，各会員．