GPUを用いた並列ソートアルゴリズム

小池 敦[1,a)]　定兼 邦彦[2,b)]

概要：GPU を用いた並列比較ソートアルゴリズムを扱う．GPU 向けの高速なソートアルゴリズムとして，Merrill らの提案する高速 Radix ソート [14] が知られている．本論文では，まずその計算量について解析したのち，本アルゴリズムを改良することで新しいアルゴリズムを二つ提案する．一つは MSD Radix ソートであり，これは分散システム等におけるアルゴリズムを設計する際に有効である．もう一つは Splitter-based ソートである．これは比較ソートであるため，キーの性質に依らずに使用することができる．

1. はじめに

プロセッサの動作クロック周波数の向上は限界を迎えており，周波数向上に代わるパフォーマンス向上の手段として並列アーキテクチャが注目されている．GPU (Graphics Processing Unit) は元々はグラフィック処理のための専用プロセッサとして開発された．しかし，非常に高い並列性を持っていることから，グラフィック処理以外にも GPU が使われ始めている．汎用の処理に GPU を使用することは GPGPU (general-purpose GPU) と呼ばれており，安価に超並列環境が構築できることから注目されている．

GPU は多数のコアを用いて効率よく処理を行うため，特殊なアーキテクチャとなっている．GPU プログラミングにおいては，このアーキテクチャを適切に考慮する必要がある．NVIDIA 社は GPGPU のための開発環境として，CUDA[15] を提供しており，CUDA 上で開発することにより，様々な GPU モデル上で動作するプログラムを実装することができる．しかし，最適なパフォーマンスを得るためには，GPU アーキテクチャを適切に考慮してアルゴリズムを設計する必要がある．

逐次アルゴリズムの評価では，RAM(Random Access Machine) モデル上での漸近解析が一般的に行われている．RAM モデルはすべての逐次実行マシンに対する抽象化となっており，RAM モデルを用いて漸近解析を行うことで，デバイスの仕様や入力データの値に依らない汎用的なアルゴリズムの性能を知ることができる．一方，並列実行マシンには，RAM モデルのような共通の抽象化が存在しない．並列アルゴリズムの漸近解析に一般的に使用されているモデルに PRAM モデル [6] があるが，PRAM モデルは GPU アーキテクチャとは大きく異なっており，GPU 向けアルゴリズムの性能を正しく評価できない．[13] では GPU における実際の計算実行時間を精度よくシミュレートすることについて検討されているが，計算実行時間は GPU のモデルに大きく依存するため，GPU 向けアルゴリズムの汎用的な性能評価とならない．筆者らは GPU 向けアルゴリズムを漸近解析するための並列計算モデルとして AGPU モデルを提案している [11]．アルゴリズムの正確な計算量はデバイス仕様に依存するが，AGPU モデル上で解析された計算量の高々定数倍である．AGPU モデルにより，GPU デバイスの仕様や入力データの値に依らない汎用的なアルゴリズムの性能を知ることができる．

本報告では，GPU 上でのソートアルゴリズムを扱う．これまでにも多くの GPU 向けソートアルゴリズムが提案されている [2], [7], [8], [10], [12], [14], [16], [17], [18], [19], [21]. その中で最も高速なもののとして，Merrill らの提案する高速 Radix ソート [14] が挙げられる．本アルゴリズムは LSD Radix ソートをベースとしているが，グローバルメモリアクセス回数の削減およびグローバルメモリアクセスのレイテンシ隠ぺいの両方が適切に考慮されており，従来アルゴリズムを大きく上回るパフォーマンスが得られている．本報告では，まず，AGPU モデルを改良することにより，上記アルゴリズムの計算量の漸近解析を行う．その後，上記アルゴリズムの変形版について 2 つ提案する．一つ目は MSD Radix ソートである．MSD Radix ソートはメモリアクセスが局所的になりやすい事が特徴である．よって分散環境におけるソートアルゴリズムを設計する場合への応用が容易である．二つ目は Spliter-based ソーティング

1 国立情報学研究所アーキテクチャ科学研究系
2 東京大学大学院情報理工学系研究科
a) koike@nii.ac.jp
b) sada@mist.i.u-tokyo.ac.jp

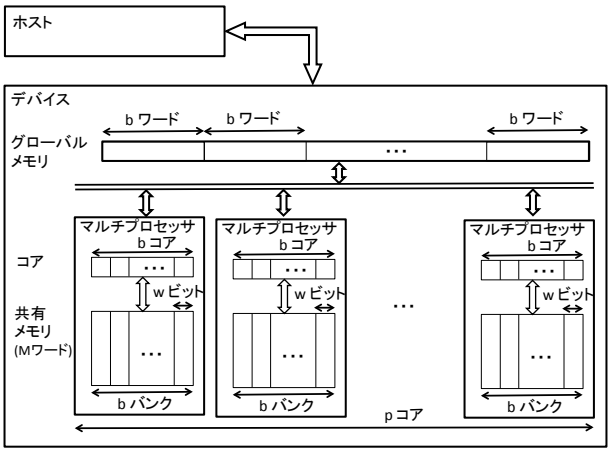

図 1　AGPU モデルのアーキテクチャ

である．Merrill らの高速 Radix ソートは比較ソートでないため，キーの性質によっては，利用できないことがある．そこで，上記の MSD ソートを改良することで比較ソートのアルゴリズムを提案する．提案アルゴリズムは，共に Merrill らの高速 Radix ソートと同様の効率的なメモリアクセスが可能である．

本論文の構成は以下の通りである．2 章で マルチスレッド対応 AGPU モデルについて説明する．次に，3 章において，Merrill らの高速 Radix ソートを紹介し，AGPU モデルでの計算量解析について述べる．4 章では，筆者らの提案する MSD Radix ソートについてアルゴリズムと計算量を説明する．5 章では，筆者らの提案する比較ソートアルゴリズムについて，詳細と計算量を説明する．6 章で結論を述べる．

2. マルチスレッド対応 AGPU モデル

AGPU モデル [11] は，GPU 向けアルゴリズムの設計と評価を行うための並列計算モデルである．AGPU モデルを用いることで，GPU デバイスの詳細仕様に依らない汎用的なアルゴリズム設計と評価を行うことができる．従来の AGPU モデルでは，並行して実行されるスレッド (ワープ) が共有メモリを共有することについて，考慮していなかった．そこで，本論文では AGPU モデルを改良する．まず，AGPU モデルのアーキテクチャを説明した後，GPU 向けアルゴリズムの評価基準について説明する．

2.1 アーキテクチャ

AGPU モデルのアーキテクチャを図 1 に示す．AGPU モデルのアーキテクチャは並列計算を行うためのデバイス (GPU) とデバイスを制御するためのホスト (CPU) の異種混載システムとなっている．デバイスは p 個のコアを備えている．コアのワード長は w ビットであり，コアはワード単位でデータにアクセスする．また，デバイスは k 個のマルチプロセッサで構成されており，各マルチプロセッサは

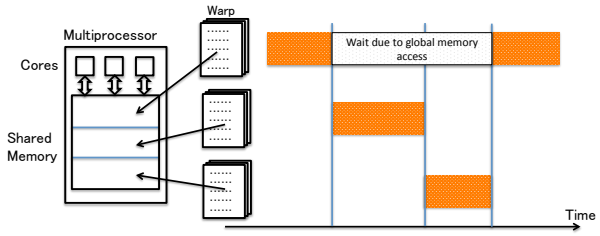

図 2　マルチスレッディングによるレイテンシ隠ぺいの例

b 個のコアを備えている．すなわち $p = kb$ である．マルチプロセッサはホストから起動されたプログラムを個別に実行する．すなわち，マルチプロセッサは他のマルチプロセッサとの通信手段および同期手段を持たない．ホストはすべてのマルチプロセッサの処理完了を待つことにより，マルチプロセッサ間の同期を行うことができる．しかし，マルチプロセッサの処理完了時，共有メモリのデータはすべて削除される．後で参照する必要があるデータはマルチプロセッサの処理終了時にすべてグローバルメモリに書き込む必要がある．

マルチプロセッサ内の b 個のコアは b 個のスレッドに対し，常に同一の命令を実行する．この時，各コアは同一命令を並列に実行するという．1 つのマルチプロセッサ内で並列に処理されるスレッドの集合をワープと呼ぶ．ただし，オペランドに指定されるデータアドレスについてはコアごとに指定することができる．また，命令には実行条件を含めることができ，条件を満たすコアのみ命令を実行させることができる．一方，各コアは複数のスレッドを時分割で切り替えながら同時実行することができる．この時，各コアは複数スレッドを並行に実行するという．言い換えれば，マルチプロセッサは複数ワープを並行に実行することができる．GPU のこの機能をマルチスレッディングと呼ぶ．各マルチプロセッサが並行に実行可能なワープの最大数を C とする．

マルチスレッディングにはグローバルメモリアクセスのレイテンシ（待ち時間）を隠ぺいする効果がある．すなわち，あるワープがグローバルメモリアクセスにより，待ち状態になっている場合に，マルチプロセッサは他のワープを実行することによりコアの使用率を高めることができる．図 2 に具体例を示す．マルチスレッディングは GPU における効率的なメモリアクセスのキーとなる技術である．

デバイスは 2 種類のメモリを備えている．1 つ目はグローバルメモリである．これは低速であるが大容量であり，すべてのマルチプロセッサおよびホストからアクセス可能である．グローバルメモリは b ワードごとのブロックに分割されている．マルチプロセッサ内の全コアが同一ブロックにアクセスする時，1 回のメモリアクセスで全コア分のデータにアクセスすることができる．これはコアレッシングと呼ばれており，処理時間に大きな影響を与える．

一方，コアが複数の異なるブロックにアクセスする時は，各ブロックのに対して 1 回のアクセスが必要となる．2 つ目は共有メモリである．各マルチプロセッサは内部に容量 M ワード ($b \leq M$) の共有メモリを備えている．これは高速であるが低容量である．また，マルチプロセッサ内部のコアからのみアクセス可能である．共有メモリは b 個のバンクから構成されており，マルチプロセッサ内の b 個のコアのそれぞれが異なるバンクにアクセスする時，単位時間でデータにアクセスできる．一方，複数のコアが同一のバンクにアクセスする時は，処理がシリアライズされる．これはバンクコンフリクトと呼ばれており，これも処理時間に大きな影響を与える．

以上で定義される計算モデルを AGPU(p, b, M, C, w) と記載する．ただし M, C, w については，省略される場合がある．

2.2 アルゴリズムの評価基準

AGPU モデルではアルゴリズムを計算量，メモリ使用量，多重度を使用して評価する．以下では，計算量とメモリ使用量について説明し，多重度については，次節で説明する．

まず，アルゴリズムの計算量を評価する基準として，時間計算量と I/O 計算量を使用する．時間計算量は，各マルチプロセッサで実行されるプログラムの命令発行数である．マルチプロセッサが複数のワープを並行に実行する場合は，すべてのワープの命令発行数の合計値となる．共有メモリへのアクセスでバンクコンフリクトが発生する場合，コンフリクト数に応じた時間が時間計算量に加算される．また，グローバルメモリへのアクセスについては，b ワードのブロックに対する書き込みまたは読み込みの時間計算量を 1 とする．マルチプロセッサごとに命令発行数が異なる場合には，最も多い発行数を時間計算量とする．I/O 計算量については，上記で説明したグローバルメモリアクセス回数のすべてのマルチプロセッサでの合計値とする．I/O 計算量を時間計算量とは別に評価する理由は，グローバルメモリアクセス処理に要する時間が他の処理に比べて大きくなるためである．また，グローバルメモリに対しては，同時にアクセスできるマルチプロセッサの数が限られているため，アクセス回数については，すべてのマルチプロセッサでの合計値とする．

次に，メモリ使用量を評価する基準として，グローバルメモリ使用量と共有メモリ使用量を使用する．共有メモリ使用量は各マルチプロセッサで使用されるメモリ使用量の最大値とする．大規模データを扱う場合，グローバルメモリ使用量を少なくすることは特に重要である．また，共有メモリ使用量は M ワード以下にする必要がある．また，共有メモリ使用量は次節で説明する多重度にも影響する．

2.3 マルチスレッディングの効果

2.1 節で述べた通り，マルチスレッディングは GPU におけるメモリアクセスのキーとなる技術である．しかし，I/O 計算量の値はマルチスレッディングの効果とは無関係であるため，マルチスレッディングの効果を I/O 計算量を用いて評価することはできない．本節ではマルチスレッディングの効果を評価する値として多重度を導入する．

マルチスレッディングの効率を上げるためには，マルチプロセッサに割り当てるワープ数を増やせば良い．1 つのマルチプロセッサに C 個のワープを割り当てる時，マルチスレッディングの効果は最も高くなる．

また，十分な数のワープが生成されている時，GPU ではユーザの設定に関わらず，1 つのマルチプロセッサにより多くのワープを割り当てようとする．これにより，マルチスレッディングの効果を高めることができる．しかし，マルチプロセッサに常に C 個のワープが割り当てできるとは限らず，割当数は共有メモリ使用量に依存する．マルチプロセッサ内のすべてのワープは同一の共有メモリを使用するため，全ワープでの共有メモリ使用量の合計値が共有メモリサイズを超えることはできない．

多重度はこれらの効果を見積もるために導入される．AGPU(p, b, M, C) 上で設計されたアルゴリズムについて，共有メモリ使用量を m，マルチプロセッサごとの使用ワープ数を c とすると，多重度 $\mathcal{M}$ は $\mathcal{M} := Mc/m$ と定義される．これは CUDA のオキュパンシに対応する値であるが，多重度は AGPU モデルのパラメータを使用して計算することができる．多重度の値が C 以下の時，値が大きいほどマルチスレッディングの効果が大きくなるが，C より大きくしても効果は大きくならない．共有メモリ使用量が大きく，かつ，マルチプロセッサへの割当ワープ数が小さい場合に多重度の値は小さくなり，マルチスレッディングの効果が小さくなる．

3. 既存の高速 Radix ソートの解析

本章では，Merrill らの高速 Radix ソート [14] について，AGPU モデルを用いて，アルゴリズムの概要説明と計算量の漸近解析を行う．

3.1 アルゴリズムの概要

Merrill らの高速 Radix ソート [14] は LSD Radix ソート [3] に分類される．すなわち，最下位桁から最上位桁の方向に順に各桁の値のみを用いてソートを行う．各桁のソートが安定の時，本ソートアルゴリズムは正しく動作する．彼らのアルゴリズムでは，各桁は $r = 2^d$ 個の数字（基数）で表現されるものとする．

次に各桁のソート処理について説明する．マルチプロセッサ数 ($k = p/b$) が 4，$r = 4$ の場合の例を，図 3 に示す．図 3 において，基数 r_1, r_2, r_3, r_4 は $r_1 < r_2 < r_3 < r_4$

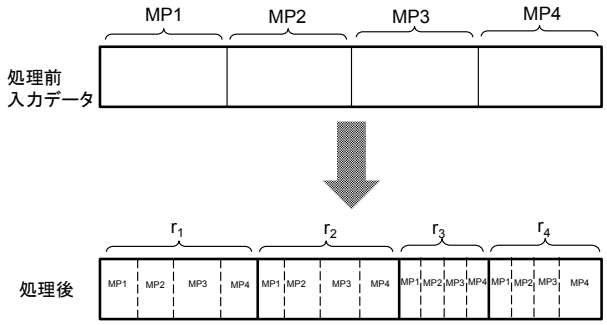

図 3　各桁のソート処理

を満たすものとする．各マルチプロセッサは入力データの連続する要素を均等に割り振られ，担当するデータを処理する．出力は図 3 の下図の様になる．まず，最小の基数 r_1 について各マルチプロセッサの担当する入力データのうち，r_1 に属するものが出力され，次に r_2, r_3, r_4 の順に同様に出力される．各マルチプロセッサが並列に処理を行うためには，処理したデータをどこに出力するかについて，あらかじめ計算しておく必要がある．そこで，各桁の処理を以下の 3 ステップで構成することにする．

(1) Bottom-level Reduction
(2) Top-level Scan
(3) Bottom-level Scan/Scatter

本節では，図 3 処理後データの 16 個の領域のそれぞれをブロックと呼ぶ事にする．Bottom-level Reduction では，マルチプロセッサごとに各基数に属する要素の数を計算する．これにより，図 3 下図の 16 個のブロックそれぞれのサイズがわかる．この計算は，Harris らの提案する GPU 向け高速 Reduction アルゴリズム（Cascading アルゴリズム）[9] を用いて行うことができる．次に，Top-level Scan では，各ブロックの先頭アドレスを計算する．これは Bottom-level Reduction で得られた各ブロックの要素数を格納した配列に対して Prefix Scan の処理を行うことで得る事ができる．GPU 向けの Prefix Scan アルゴリズムとして Tree-based アルゴリズム [20] が知られている．次に，Bottom-level Scan/Scatter では，入力のそれぞれの要素について適切なアドレスへのデータコピーを行う．前のフェーズにおいて，各ブロックの先頭アドレスが分かっているため，各マルチプロセッサは並列に処理を行うことができ，マルチプロセッサ間の情報交換は不要である．以下，各マルチプロセッサの処理を説明する．この処理を効率良く行うため，彼らは *“multi-scan”* という方法を提案している．

multi-scan では Dotsenko らの提案する高速 prefix scan アルゴリズム（Matrix-based アルゴリズム） [4] を使用する．Matrix-based アルゴリズムでは各マルチプロセッサは担当する入力データをサイズ ab の小ブロックに分割し (b はマルチプロセッサ内のコア数，a はチューニングパラメータ)，各小ブロックをシーケンシャルに処理する．a の値については，大きいほど時間計算量が小さくなるものの，マルチスレッディングの効率が下がる事が知られている．筆者らはこの事について，AGPU モデルを用いて解析を行っている [22].

multi-scan では入力の各データの出力アドレスを計算するため，基数ごとにワープを生成する．各ワープは担当する基数に属する入力データをスキャンすることにより，それらの出力先アドレスを計算する．この計算を行うために Dotsenko らの提案する高速 prefix scan アルゴリズムを使用する．最後にデータを指定のアドレスに出力するが，グローバルメモリアクセス回数を減らすため，連続するアドレスに出力されるデータを一旦共有メモリの連続する領域に書き込んだのち，出力される．これにより，グローバルメモリへのコアレスアクセスが行われやすくなる．

3.2　計算量の解析

本アルゴリズムの各桁の処理は基数ごとの Prefix Scan の処理に Scatter（データ出力）処理を追加したものとなっている．Scatter 処理で共有メモリにデータをコピーする際にバンクコンフリクトが発生するため，時間計算量は Prefix Scan よりも大きくなる．しかし，$r = a$ とすると，I/O 計算量は Prefix Scan と同様になる．また，入力をビット長を w とすると，各桁の処理は $w/\log r$ 回繰り返される．

入力要素数 n がコア数 p よりも十分大きい時アルゴリズム全体の計算量はアルゴリズム全体の計算量は，表 1 のようになる．

4.　MSD Radix ソート

本章では 3 章で紹介した Merrill らの高速 Radix ソート [14] を変更することにより，高速な MSD Radix ソートアルゴリズムを提案する．MSD Radix ソートは，最上位桁から最下位桁の方向に順に各桁の値のみを用いてソートを行う．MSD Radix ソートは複数 GPU による Radix ソートを設計する場合などへの応用が容易である．また，5 章の Splitter-based ソートは本章のアルゴリズムを変更したものである．

4.1　アルゴリズムの概要

本章以降では，各桁のソート処理をフェーズと呼ぶ．最初のフェーズは最上位桁に対して Merrill らの高速 Radix ソート [14] と同様の処理を行う．次のフェーズに関しては，Merrill らの Radix ソートを修正する必要がある．なぜならば，MSD から LSD 方向への Radix Sort では，前フェーズで基数により区切られた領域を別々に処理する必要があるためである．

2 フェーズ目以降では，各領域へのマルチプロセッサの割当は図 4 のように行う．まず，各マルチプロセッサに対

表 1　入力をビット長 w の整数としたときの，アルゴリズム全体の計算量

	I/O 計算量	時間計算量	多重度
LSD Radix ソート	$\mathcal{O}\left(\frac{nw}{b \log r}\right)$	$\mathcal{O}\left(\frac{nw}{p \log r}\left(r + \frac{\log b}{r}\right)\right)$	$\mathcal{O}\left(\frac{M}{rb}\right)$

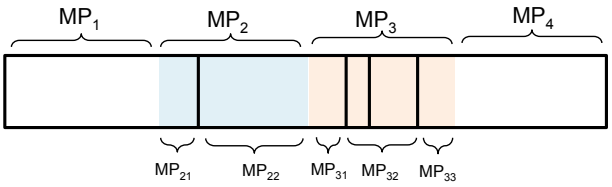

図 4　マルチプロセッサへの要素の割当

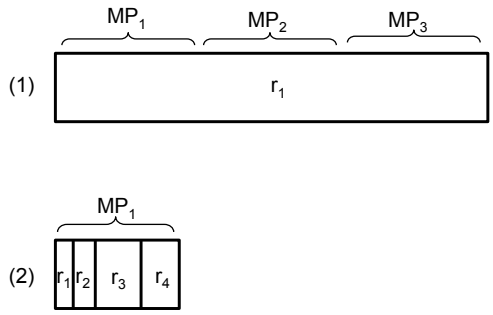

図 5　基数領域と分割領域の関係

し，均等に領域を割り当てる．そして，割当領域に複数の基数領域がある場合には，以下のようにさらに領域を分割する．

- 複数領域のうち最も左の基数領域
- 複数領域のうち最も右の基数領域
- それ以外の領域

このようにしても分割された領域の数は高々 3 倍にしかならない．

次に，分割された領域を各マルチプロセッサに同じ数だけ分配する．この時，各マルチプロセッサに割り当てられる要素数は均等に分配した場合の高々 3 倍となる．

また，分割された領域と基数領域の関係は必ず以下のどちらかになる．図 5 に具体例を示す．

(1) 一つの基数領域が複数の分割領域で構成される
(2) 複数の基数領域で一つ分割領域が構成される

すなわち，(2) の場合の除けば，基数領域の区切りは必ず分割領域の区切りとなっている．

図 5 の (1) の場合，マルチプロセッサは最初のフェーズと同様の各桁のソート処理を行う．(2) の場合は，マルチプロセッサはシーケンシャルに各基数領域を処理する．

以上を LSD まで繰り返す事で，Radix sort を行うことができる．

計算量は Merrill らの LSD Radix ソートと同様になる．

5. Splitter-based ソート

4 章の入力データ振り分け処理を，基数を用いずにピボット集合を用いて行うように変更する．ピボット集合を用いる場合，ピボット集合の選び方が問題となる．ピボット集合により振り分けられた各集合のサイズに偏りが有る場合，より多くのフェーズが必要となり，計算量が大きくなる．

本章では，Aggarwal ら [1] の I/O 計算量最適な Distribution ソートのアイデアを用いて適切にピボット集合を選択することについて検討する．ただし，このアルゴリズムをそのまま適用すると，多重度が小さくなり．マルチスレッディングの効果が下がるので，多重度を大きく保てるように工夫する．

以下では，まず Aggarwal らの Distribution ソートについて説明した後，筆者らの提案するアルゴリズムについて説明する．

5.1 Aggarwal らの Distribution ソートについて

Aggarwal ら [1] の Distribution ソートも，1 フェーズの処理を繰り返すことでソートを行う．ただし，各フェーズは以下の 2 ステップからなる．

(1) ピボット集合を算出する
(2) ピボット集合を用いて，データを分割する

Radix ソートにおける各桁の処理と比較すると，ピボット集合を算出する処理が追加されている．(2) のデータ振り分け処理については，MSD Radix ソートと同様のアルゴリズムを用いることができるので，以降では (1) について説明する．

Aggarwal らの Distribution ソートは I/O モデルと呼ばれる計算モデル上で設計されている．標準的な I/O モデルは 1 つのプロセッサ，M 要素を格納できる 1 つの内部メモリおよび 1 つの外部メモリ (ディスク) から構成される．プロセッサは単位時間あたりに外部メモリの連続した b レコードからなるブロックにアクセスすることができる．アルゴリズムはブロックの転送回数 (I/O 計算量) で評価される．本モデルを I/O(b, M) と記述する．

Aggarwal らの Distribution ソートは，外部メモリに保存された要素数 n の入力データに対し，最適な I/O 計算量でソートを行う．図 6 に処理の流れを示す．まず，入力を要素数 M のメモリロードに分割する．そして，各メモリロードをソートする．次に，各メモリロードから間隔 $S/4$ ごとにピボット集合を取り出す（S の値は最後に決める）．本論文ではこのピボットをローカルピボットと呼ぶことにする．次に全メモリロードからのローカルピボットを結合する．これを本論文ではローカルピボット集合と呼ぶことにする．最後にこの集合から等間隔に S 個のピボットを取り出す．すなわち，i 番目のグローバルピボットはローカルピボット集合の中で $4iN/S^2$ 番目に小さい要素となる．グ

ローバルピボット集合の抽出は線形時間セレクション [5] を S 回行うことでできる．この時，I/O 計算量も入力サイズに線形となるようにできる．

次に index i のグローバルピボットの入力データ中での rank (何番目に小さいか) の値を検討する．ローカルピボット集合の中で，このグローバルピボット以下の値を持つものは（自身も含めて）i 個であり，ローカルピボットのピボット間隔が $S/4$ なので，取りうるランク値の最小値は $rank(i) \geq \frac{4iN}{S^2} \cdot \frac{S}{4} = \frac{iN}{S}$ となる．また，ランク値は以下の値よりは小さくなる．$rank(i) < \frac{iN}{S} + \frac{N}{M}\frac{S}{4} < \frac{N}{S}\left(i + \frac{1}{4}\right)$

これより，グローバルピボットによる振り分け後の各領域のサイズは $\frac{5}{4}\frac{N}{S}$ 未満となる．ここで，$S = \sqrt{\frac{M}{b}}$ とすると，フェーズの回数は高々 $\mathcal{O}\left(\log_{\frac{4}{5}S} \frac{N}{b}\right) = \mathcal{O}\left(\frac{\log_{\frac{M}{b}} \frac{N}{b}}{\log_{\frac{M}{b}} \frac{4}{5}\sqrt{\frac{M}{b}}}\right) = \mathcal{O}\left(\log_{\frac{M}{b}} \frac{N}{b}\right)$ となる．各フェーズでの I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\right)$ なので，合計の I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}} \frac{N}{b}\right)$ となり，これは下界 [1] と一致する．

5.2 提案アルゴリズムの概要

$I/O(b, M)$ 上で設計された任意のアルゴリズムに対して，同じ I/O 計算量を持つ AGPU(p, b, M) 上のアルゴリズムが存在する [11]．しかし，前節のアルゴリズムを AGPU モデル上で実装する場合，多重度が 1（最小値）になってしまう．そこで，アルゴリズムを改良し，多重度を大きくすることを考える．

前節のアルゴリズムを AGPU(p, b, M, C) 上で動作させる際に多重度が最小値 1 になる原因は，ローカルピボット抽出時に行うメモリロードのソート処理である．そこでメモリロード全体に対するソート処理を行う事無しにローカルピボットの抽出処理を行うようにする．

基本的なアイデアは，ピボット集合を用いてメモリロードの領域分割処理を繰り返すことで，メモリロードをサイズ b 以下のチャンクに分割することである．

一つのメモリロードは一つのマルチプロセッサによって処理される．まず，メモリロードを b ワードからなる基本ブロックに分割し，マルチプロセッサはすべての基本ブロックに対し共有メモリを用いてソートを行う．次に，各基本ブロックにおいて，$S/4$ 要素ごとにローカルピボットを抽出する．すると合計で $4M/S = 4b \cdot S$ 個のピボットが取り出せる．ここから，S 個のグローバルピボットを取り出す（i 番目に抽出されるグローバルピボットは全ピボットの中で $4bi$ 番目に小さい要素となる）．これは，線形時間セレクション処理 [5] を S 回することで実現できる．1 回のセレクション処理での I/O 計算量は $\mathcal{O}(S)$ なので，S 個のセレクション算出処理での合計 I/O 計算量は $S \cdot S = M/b$ となる．このグローバルピボット集合を用いて，メモリロードの分割を行う（4 章と同様の方法を用いる）と，各領域のサイズは高々 $\frac{5}{4}\frac{M}{S}$ となる．上記の処理を $\log_{4S/5} \frac{M}{b} = \mathcal{O}(1)$ 回行うことで，メモリロードを $\mathcal{O}(b)$ のチャンクに分割することができる．最後に分割されたチャンクをシーケンシャルにチェックしていくことで，メモリロードから $4bS$ 個のローカルピボットを抽出する．

I/O 計算量について考察する．1 回の分割での I/O 計算量はメモリロードにつき $\mathcal{O}(M/b)$ なので，入力全体では $\mathcal{O}(N/b)$ である．よって全フェーズ合計の I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}} \frac{N}{b}\right)$ となり，下界と一致する．また，アルゴリズム全体の計算量は表 2 のようになる．

6. 結論

本論文では，まず，Merrill らの高速 Radix ソートの計算量を AGPU モデルを用いて解析した後，それを変更することで 2 つのアルゴリズムを提案した．一つ目は MSD Radix ソートであり，二つ目は Splitter-based ソートである．MSD Radix ソートは Merrill らの Radix ソートと同様の計算量を持っている．Splitter-baed ソートは漸近的な計算量がキーのサイズ等に依存しないため，キーのサイズが大きいような場合にも適している．

今後は，提案アルゴリズムを実装し，実計算時間を評価したい．また，複数 GPU デバイスを備えた環境において，本アルゴリズムを用いた高速化を試みたい．また，Merrill らの高速 Radix ソート [14] についても更なる高速化を検討したい．

参考文献

[1] Aggarwal, A. and Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems, *Commun. ACM*, Vol. 31, No. 9, pp. 1116–1127 (online), DOI: 10.1145/48529.48535 (1988).

[2] Capannini, G., Silvestri, F., Baraglia, R. and Nardini, F.: Sorting using bitonic network with CUDA, *Proceedings of the 7th Workshop on LSDS-IR* (2009).

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition (2009).

[4] Dotsenko, Y., Govindaraju, N. K., Sloan, P.-P., Boyd, C. and Manferdelli, J.: Fast scan algorithms on graphics processors, *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, New York, NY, USA, ACM, pp. 205–213 (online), DOI: 10.1145/1375527.1375559 (2008).

[5] Floyd, R.: Permuting Information in Idealized Two-Level Storage, *Complexity of Computer Computations* (Miller, R., Thatcher, J. and Bohlinger, J., eds.), The IBM Research Symposia Series, Springer US, pp. 105–109 (1972).

[6] Fortune, S. and Wyllie, J.: Parallelism in random access machines, *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, New York, NY, USA, ACM, pp. 114–118 (online), DOI: 10.1145/800133.804339 (1978).

[7] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: high performance graphics co-

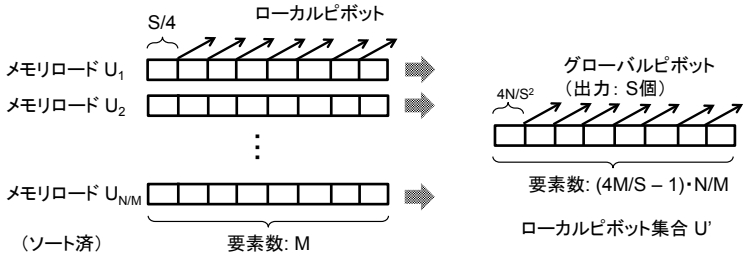

図 **6** Distribution ソートのピボット算出処理の流れ

表 **2** 提案アルゴリズム（Splitter-based ソート）の計算量 $\left(S = \sqrt{\frac{M}{b}}\right)$

	I/O 計算量	時間計算量	多重度
(下界)	$\Omega\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\Omega\left(\frac{N}{p}\log N\right)$	-
Splitter-based ソート	$\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}\left(\frac{N}{p}\left(S+\frac{\log b}{S}\right)\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}(S)$
従来の I/O 最適アルゴリズム [11]	$\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}\left(\frac{N}{p}\log\frac{N}{b}\log b\right)$	$O(1)$

processor sorting for large database management, *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, New York, NY, USA, ACM, pp. 325–336 (online), DOI: 10.1145/1142473.1142511 (2006).

[8] Greß, A. and Zachmann, G.: GPU-ABiSort: optimal parallel sorting on stream architectures, *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, Washington, DC, USA, IEEE Computer Society, pp. 45–45 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1898953.1898980⟩ (2006).

[9] Harris, M.: Optimizing Parallel Reduction in CUDA (2008).

[10] Khorasani, E., Paulovicks, B. D., Sheinin, V. and Yeo, H.: Parallel implementation of external sort and join operations on a multi-core network-optimized system on a chip, *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, Berlin, Heidelberg, Springer-Verlag, pp. 318–325 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2075416.2075446⟩ (2011).

[11] Koike, A. and Sadakane, K.: A Novel Computational Model for GPUs with Application to I/O Optimal Sorting Algorithms, *2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, pp. 614–623 (online), DOI: 10.1109/IPDPSW.2014.72 (2014).

[12] Kolonias, V., Voyiatzis, A. G., Goulas, G. and Housos, E.: Design and implementation of an efficient integer count sort in CUDA GPUs, *Concurr. Comput. : Pract. Exper.*, Vol. 23, No. 18, pp. 2365–2381 (online), DOI: 10.1002/cpe.1776 (2011).

[13] Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P. and Srinathan, K.: A performance prediction model for the CUDA GPGPU platform, *High Performance Computing (HiPC), 2009 International Conference on*, pp. 463 –472 (online), DOI: 10.1109/HIPC.2009.5433179 (2009).

[14] Merrill, D. and Grimshaw, A.: High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing, *Parallel Processing Letters*, Vol. 21, No. 02, pp. 245–272 (online), DOI: 10.1142/S0129626411000187 (2011).

[15] NVIDIA Corporation: NVIDIA CUDA C Programming Guide version 4.2 (2012).

[16] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: Fast in-place sorting with CUDA based on bitonic sort, *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, Berlin, Heidelberg, Springer-Verlag, pp. 403–410 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1882792.1882841⟩ (2010).

[17] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: A Novel Sorting Algorithm for Many-core Architectures Based on Adaptive Bitonic Sort, *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, Washington, DC, USA, IEEE Computer Society, pp. 227–237 (online), DOI: 10.1109/IPDPS.2012.30 (2012).

[18] Satish, N., Harris, M. and Garland, M.: Designing efficient sorting algorithms for manycore GPUs, *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, Washington, DC, USA, IEEE Computer Society, pp. 1–10 (online), DOI: 10.1109/IPDPS.2009.5161005 (2009).

[19] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D. and Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, ACM, pp. 351–362 (online), DOI: 10.1145/1807167.1807207 (2010).

[20] Sengupta, S., Harris, M. and Garland, M.: Efficient parallel scan algorithms for GPUs, *Technical Report NVR-2008-003*, NVIDIA (2008).

[21] Ye, X., Fan, D., Lin, W., Yuan, N. and Ienne, P.: High performance comparison-based sorting algorithm on many-core GPUs, *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–10 (online), DOI: 10.1109/IPDPS.2010.5470445 (2010).

[22] 小池 敦, 定兼 邦彦: AGPU モデルにおけるマルチスレッディングの効果, 総合大会 COMP 学生シンポジウム DS-1-13, 電子情報通信学会 (2013).