Vol.2013-AL-145 No.16
2013/11/7

GPUを用いた並列ソートアルゴリズムの実装と評価

小池 敦[1,a)]　定兼 邦彦[1,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 上でプログラムを高速動作させる場合，コアの計算量だけでなくメモリアクセスについても適切に考慮する必要がある．筆者らは GPU 上のアルゴリズムを評価するためのモデルとして AGPU モデルを提案しており，本モデルでは I/O 計算量を用いることにより，メモリアクセスの評価を行うことができる．また，筆者らは GPU 上での比較ソートに関して，I/O 計算量のオーダーが理論的な下界と一致するアルゴリズムを提案している．本稿ではアルゴリズムを実装し，実 GPU 上で評価を行うことで，本アルゴリズムの有効性を示す．

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[11] 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 [4] 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．[9] 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 [17]．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 上での比較ソートアルゴリズムを扱う．ソートは多くの処理の中で使用される最も基本的な処理の一つであり，これを高速化することは他の多くのアルゴリズムの高速化につながる．そのため，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 [3], [5], [6], [7], [8], [10], [12], [13], [14], [15], [16].

本報告ではまず，2 章で AGPU モデルを説明する．次に，3 章において，AGPU モデル上での比較ソートアルゴリズムの計算量の下界を示し，また，既存の 2 つの比較ソートアルゴリズムを紹介する．1 つはバイトニックソート (CUDA SDK のサンプルプログラム) であり，もう 1 つは GPU-Warpsort [16] である．後述のようにこれらのアルゴリズムの計算量は最適ではない．4 章では，I/O 計算量が最適となる比較ソートアルゴリズムを提案し詳細を説明する．5 章では，提案アルゴリズムの AGPU モデル上での計算量を解析する．6 章で結論を述べる．

[1] 国立情報学研究所
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101–8430, Japan
[a)] koike@nii.ac.jp
[b)] sada@nii.ac.jp

© 2013 Information Processing Society of Japan

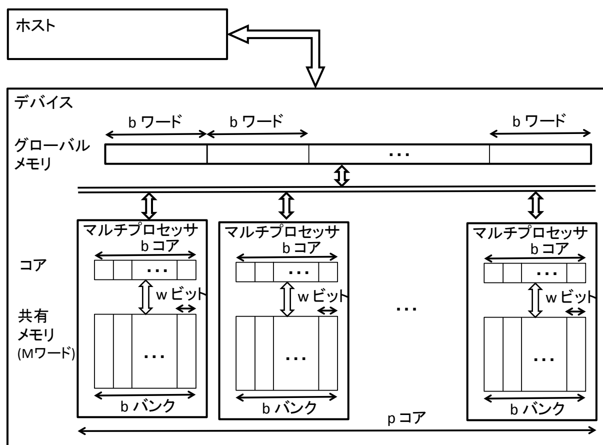

図 1　AGPU モデルのアーキテクチャ

2. AGPU モデル

AGPU モデルは，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．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．その後，I/O モデル [1] との関係について述べる．

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 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すべてのコアは常に同一の命令を実行する．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．

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．同一命令を実行す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る，同一命令を実行する 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．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, w) と記載する．ただし M, 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

まず，アルゴリズムの実行時間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．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，同時に一つのマルチプロセッサからしかアクセスすることができないため，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．使用される単位はビットである．また，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．ただ，M ワード以下であっても共有メモリ使用量が大きい場合はグローバルメモリア

© 2013 Information Processing Society of Japan

クセスの速度 (レイテンシ隠ぺい効果) が小さくなる [18]．しかし，本報告ではその影響については考慮しない．

2.3 I/O モデルとの関係

標準的な *EM (external-memory)* モデル，または *I/O* モデル [1] は，1 つのプロセッサ，M ワードを格納できる 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 ワードからなるブ 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で評価される．このモデルを I/O(B,M) と表す．以下，AGPU モデルと I/O モデルの関係について記載する．

定義 2.1 (等式の定義) 2 つの計算モデル X,Y を考える．モデル Y 上の任意のアルゴリズム A_Y に対し，モデル X 上のアルゴリズム A_X が存在し，任意の入力に対して A_X の I/O 計算量が A_Y の I/O 計算量の α 倍以下である時，$X_{IO} \leq \alpha Y_{IO}$ と書く．$X_{IO} \leq O(1) Y_{IO}$ かつ $Y_{IO} \leq O(1) X_{IO}$ の時，$X_{IO} = Y_{IO}$ と記述する．

補題 2.2

$$\mathrm{AGPU}_{IO}(p,b,M) = \mathrm{AGPU}_{IO}(b,b,M)$$

証明. $\mathrm{AGPU}_{IO}(b,b,M)$ はマルチプロセッサを 1 つのみ有しており，$\mathrm{AGPU}_{IO}(p,b,M)$ は $\mathrm{AGPU}_{IO}(b,b,M)$ と同一のマルチプロセッサを $k = p/b$ 個有している．明らかに $\mathrm{AGPU}_{IO}(b,b,M)$ 上のアルゴリズムと同じ I/O 計算量を持つ $\mathrm{AGPU}_{IO}(p,b,M)$ 上のアルゴリズムが存在する．そこで，$\mathrm{AGPU}_{IO}(p,b,M)$ 上のアルゴリズムに対応する $\mathrm{AGPU}_{IO}(b,b,M)$ 上のアルゴリズムについて考える．2 章で説明したように，ホストはマルチプロセッサの同期を行うことができる．そこで同期ごとにフェーズを区切って考える (ホストによる同期が一回もない場合はプログラムの開始から終了までが 1 フェーズとなる)．あるフェーズにおいて，$\mathrm{AGPU}_{IO}(p,b,M)$ の各マルチプロセッサが処理する合計 k 個のタスクを 1 つのマルチプロセッサがシーケンシャルに処理することを考える．アーキテクチャの制約により，マルチプロセッサ間では一切の通信ができない．また，タスク終了時に共有メモリのデータは消去される．よって，1 つのマルチプロセッサでシーケンシャルに処理する場合でも，k 個のマルチプロセッサが並列に処理する場合と同じ処理を行うことができる．I/O 計算量はすべてのマルチプロセッサのグローバルメモリアクセス命令の合計であるので，k 個のマルチプロセッサが行う処理を 1 つのマルチプロセッサがシーケンシャルに処理するように変更しても，I/O 計算量は変らない．すべてのフェーズにおいて，これが言えるため，$\mathrm{AGPU}(p,b,M)$ 上の任意のアルゴリズムは $\mathrm{AGPU}(b,b,M)$ を使用して，同一の I/O 計算量でシミュレートできる． □

もし同期時に共有メモリが消去されないのであれば，マルチプロセッサは次のフェーズのためのデータを共有メモリに保持しておく必要があるため，1 つのマルチプロセッサで k 個のマルチプロセッサの処理をシミュレートすることはできない．よって，この場合には上記の補題は成り立たない．もしシミュレートする場合には kM ワードの共有メモリが必要である．しかし，AGPU モデルおよび実際の NVIDIA 社 GPU では同期時，共有メモリの内容が消去される．これは，プログラミング時，マルチプロセッサが共有メモリの仮想アドレスを使用することに起因する．これにより，GPU モデルに依存しないプログラミングができるようになっている．

定理 2.3 (I/O モデルとの I/O 計算量の関係)

$$\mathrm{AGPU}_{IO}(p,b,M) = \mathrm{I/O}_{IO}(b,M)$$

証明. $\mathrm{AGPU}(b,b,M)$ は M ワードの共有メモリを備えたマルチプロセッサを 1 つのみ有する．$\mathrm{I/O}(b,M)$ の内部メモリは $\mathrm{AGPU}(b,b,M)$ の共有メモリに相当し，両者の容量はともに M ワードである．$\mathrm{AGPU}(b,b,M)$ はコアレッシングの機能を持っており，$\mathrm{I/O}(b,M)$ と同様に 1 回のグローバルメモリアクセスで b ワードを取得することができる．よって，$\mathrm{AGPU}_{IO}(b,b,M) = \mathrm{I/O}_{IO}(b,M)$．よって，$\mathrm{AGPU}_{IO}(p,b,M) = \mathrm{AGPU}_{IO}(b,b,M) = \mathrm{I/O}_{IO}(b,M)$ となる． □

3. 既存の比較ソートアルゴリズム

3.1 計算量の下界について

$\mathrm{AGPU}(p,b,M)$ 上での比較ソートの時間計算量および I/O 計算量の下界について考察する．

時間計算量については，並列処理を行わない場合の下界が $\Omega(n \log n)$ であり，$\mathrm{AGPU}(p,b,M)$ のコア数は p なので，自明な下界は $\Omega\left(\frac{n}{p}\log n\right)$ となる．

I/O 計算量については，$\mathrm{I/O}(b,M)$ での計算量下界が $\Omega(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$ である [1] ことから，定理 2.3 より，$\mathrm{AGPU}(p,b,M)$ での I/O 計算量下界も $\Omega(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$ となる．

3.2 バイトニックソート

バイトニックソート [2] はソーティングネットワークを用いた並列比較ソートアルゴリズムであり，複数の比較器を用いて並列に計算することが特徴である．図 2 に入力要素数 8 の場合のバイトニックソートの例を示す．入力列の要素数を n とすると，フェーズ i では，$i-1$ 回のステージの処理を行うことにより，長さ 2^{i+1} のソート列を $n/2^{i+1}$ 個生成する．各ステージでは $n/2$ 個の比較器が並列に 1 回の比較処理を行う．フェーズ 0 からフェーズ $\log n - 1$ まで行うことでソート列を得ることができる．ソーティング

© 2013 Information Processing Society of Japan

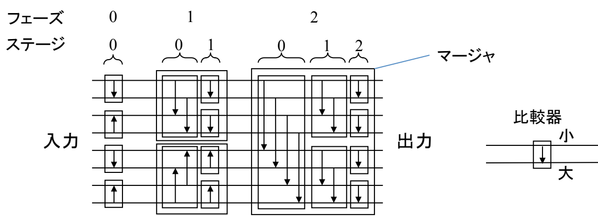

図 2　バイトニックソート

ネットワーク上でのバイトニックソートの計算量（各要素に対する比較演算回数）は $\log^2 n$ となる．GPU 上でバイトニックソートを行う場合について，AGPU(p, b, M) を用いて計算量を解析すると，時間計算量は $\mathcal{O}\left(\frac{n}{p}\log^2 n\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}\log^2\left\lceil\frac{n}{M}\right\rceil\right)$ となる [19]．時間計算量，I/O 計算量ともに下界よりも大きくなっている．これは，GPU でソーティングネットワークをシミュレートする際，入力サイズが大きいフェーズでは処理が非効率になるためである．

3.3　GPUWarpSort

GPU-WarpSort [16] はバイトニックソートとマージソートを組み合わせたソートアルゴリズムである．GPUWarpSort では，基本的には 2 つのソート列のマージを繰り返すことで出力を得る．しかし，2 ソート列のマージ処理の中でバイトニックソートを使用する．

GPU-WarpSort は後述の提案アルゴリズムの $d = 2$ の場合に相当する．後述の結果を用いることで，時間計算量は $\mathcal{O}\left(\frac{n}{p}\log\frac{n}{b}\log b\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}\log\frac{n}{b}\right)$ となる．GPUWarpSort の時間計算量は下界の $\mathcal{O}(\log b)$ 倍以内である．しかし，I/O 計算量は下界よりも大きくなっている．

4.　提案アルゴリズム

GPUWarpSort では, I/O 計算量が下界よりも大きくなっている．そこで本報告では I/O 計算量が最適となるアルゴリズムを提案する．

4.1　メインアイデア

GPUWarpSort では，ソート列を 2 つずつマージしており，2 つのソート列のマージの度に各要素に対して 2 回のグローバルメモリアクセス（リードとライト）が発生している．本報告では d 個のソート列を一気にマージすることで，マージの回数を減らすことを考える．8 個のソート列をマージする場合の例を図 3 に示す．$d = 8$ とすると，GPUWarpSort では入力データの各要素に対して 6 回のグローバルメモリアクセスが発生するのに対し，提案アルゴリズムでは 2 回のグローバルメモリアクセスしか発生しない．

4.2　処理の概要

本アルゴリズムでは，まず入力列を大きさ b ごとに分割する．分割された入力列のそれぞれを基本ブロックと呼ぶ．次に各基本ブロックをバイトニックソートを用いてソートする．次に各基本ブロックを d 個ずつマージする．マージしてできた列をサブアレイと呼ぶことにする．そして，サブアレイが 1 つになるまで，サブアレイのマージを繰り返す．

GPUWarpSort と同様にマージの処理は 3 つのフェーズに分けられる．サブアレイの個数が多い時は通常のマージを行う（この処理を「前半」の処理と呼ぶ）が，サブアレイの個数が少なくなったら，複数のピボット値を用いてサブアレイに対してデータの振り分けを行い（「データ振り分け」処理），振り分けられたデータごとにマージを行う（「後半」の処理）．これはサブアレイの数が少なくなることで処理が割り当てられないマルチプロセッサが発生することを防ぐための処理である．

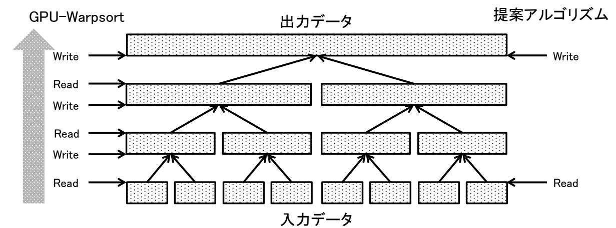

図 3　提案アルゴリズムのメインアイデア

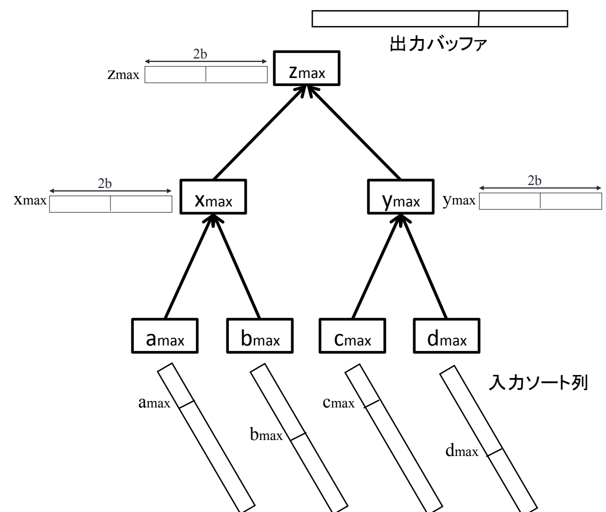

図 4　マージ処理に用いるヒープ

4.3　複数のサブアレイのマージ処理

d 個のサブアレイをマージして 1 つのサブアレイを出力する処理について説明する．提案アルゴリズムでは，この処理を繰り返し行うことにより全入力に対するマージを行う．この処理は 1 つのマルチプロセッサを用いて行われる．

まず，この処理のために用いるヒープについて説明する．図 4 に $d = 8$ の時のヒープの例を示す．ヒープは d 個の葉を持ち，それぞれは入力サブアレイへのポインタを保持している．葉は後述する規則で入力データを親に渡す．内部

節点のそれぞれは共有メモリ中に大きさ $2b$ のバッファを持っている．内部節点の数は $d-1$ 個なので，共有メモリ使用量の合計は $2b(d-1)$ である．バッファ内の要素は常にソートされた状態であるようにする．根を除く内部節点は後述する規則でバッファ内のデータを親に渡す．根は後述する規則でバッファ内のデータを出力サブアレイに書き込む．全節点はキーを持っている．キーの値は自身のバッファ (葉の場合は入力サブアレイ) から親 (根の場合は出力サブアレイ) にデータを移動した際の最終要素の値とする．バッファ，サブアレイはソート済みなので，最終要素は移動するデータの最大値である．後述の規則によりキーはヒープ条件を満たしている．また，ヒープの各節点には index が付けられており，根は 1，節点 i の左の子は $2i$，右の子は $2i+1$ である．

次に Heapify の処理について説明する．Heapify(i) は index i の節点のバッファに対し，子のバッファから b データを移動する処理である．節点 i のバッファに格納されているデータの要素数が b 以下の際に呼び出すことができる．節点 i はキーが小さい方の子から先頭の b データを移動させ，その子のキーの値を移動した最終要素の値に更新する．そして自バッファについて，元からあるデータと取得したデータをマージする．その後，同様の処理をデータ取得先の子に対しても行う．これを葉に到達するまで繰り返す．

次にマージの処理手順について説明する．まず，Heap の初期化として，index の大きい内部節点から順に Heapify を行う．そして，自節点のキーの値をマイナス無限大（正確には入力データのどの要素と比較してもそれより小さくなる値）に設定する．その後，根のノードから b データを出力サブアレイに書き込み，根に対して，Heapify を行う．これをすべてデータが出力し終わるまで繰り返す．

根のバッファの先頭 b データは常に出力バッファに未出力のデータのうち最小の b データが格納されている．よって，出力バッファには正しいソート列が出力される．これを説明するために，まず，子が葉であるような内部節点を考える．Heapify の処理で子からデータを移動する前の左の子のキーを $\alpha_{\max}$，右の子のキーを $\beta_{\max}$ とし，$\alpha_{\max} < \beta_{\max}$ だったとすると，その内部節点のバッファに格納されている b データの最終要素は $\beta_{\max}$ である．Heapify の規則に従い，左の子から b データを内部節点のバッファに移動すると，未出力の（内部節点の親に移動させる前の）データのうち，最小の b 要素は必ずサイズ $2b$ の内部節点バッファの中にある（右の子の未読み込みのデータが最小の b 要素に含まれることはない）．よって，このバッファをソートすることで，バッファ内の $2b$ 要素のうちの最小の b 要素が先頭の b 要素の中に含まれるようになる．すべての内部節点において同様の議論ができる．よって，この方法により正しくマージが行われることが言える．

バッファのサイズは $2b$ から b に減らすことが可能である．上述の説明では，Heapify は指定ノードから子の方向に処理を行っていくが，処理を行う節点は予め分かるため，葉から順に指定ノードの方向に処理を行うことができる．このようにすると内部バッファに b データ以上のデータを保持する必要がなくなり，バッファサイズは b で十分となる．

4.4 前半の処理

すべてのサブアレイに対して，d 個ずつをマージする処理を 1 ステップと呼ぶ．前半の処理ではマージを s_0 ステップ繰り返す．

1 ステップの処理を説明する．入力サブアレイは d 個ごとに入力列集合としてまとめられ，そのそれぞれを 1 つのマルチプロセッサを用いてマージする．入力列集合の数がマルチプロセッサ数よりも大きい場合には，各マルチプロセッサは複数の入力列集合を処理する．d 個のサブアレイのマージ方法については，4.3 節で説明した方法を用いる．

4.5 データ振り分け処理

データ振り分け処理は以下の手順で行われる．

Step.1: 各サブアレイからピボット値を取り出す

Step.2: 各サブアレイから取り出されたピボット値列をマージして一つのソート列にする．

Step.3: 上記のピボット列を用いて各サブアレイを振り分ける．振り分けられた各領域をブロックと呼ぶ．

Step.1 ではすべてのサブアレイから同数のピボット値が取り出される．また．各サブアレイにおいて，ピボット値は等間隔で取り出される．Step.2 では，4.4 と同様の処理をピボット列に対して行う．これをすべてのピボット列がマージされるまで続ける．Step.3 では，4.4 と同様の処理をマージされたピボット列とサブアレイに対して行う．マージされた列の中におけるピボット列の各 index から自身より左にあるピボットの数を引いたものが分割された各ブロックの開始 index となる．

4.6 後半の処理

前半の処理において，ℓ 個のサブアレイが残ったとする．サブアレイ長を h とすると $h\ell = n$ である．ピボット値により振り分けられた領域のそれぞれをブロックと呼ぶ．各サブアレイごとのブロック数を u，各サブアレイから抽出するピボットの数を ρ とすると，$u = \rho\ell + 1$ となる．$\ell \leq u$ となることに注意する．また，u は $u < n/b$ となるようにする．

後半の処理ではピボット値によって同じ領域に割り当てられてたブロックに対し，マージ処理を行い 1 つにする．そのために d 個ずつブロックをマージし，同じ領域内のブロックが 1 つになるまでこれを繰り返す．すべてのブロックに対しマージ処理を行い，ブロック数を $1/d$ にする処理

を1ステップと呼ぶ．以下，1ステップの処理について説明する．まず，入力ブロックのサイズを合計することにより出力の各ブロックのサイズを計算する．そして，各出力ブロックの出力アドレスを計算する．各マルチプロセッサは一つのヒープを用いて d 個のブロックのマージを行う．もし，出力のブロック数がマルチプロセッサ数よりも大きいときはマルチプロセッサは複数回この処理を行う．各マルチプロセッサに対するブロックの割当については下記が成り立つ．

補題 4.1 1ステップのマージ処理について，すべての入力ブロックに対して i 番目の入力列の長さを w_i と書き，一つのマルチプロセッサが一つのヒープを使用してマージする際の入力列の集合を c_j とする．また，index x のマルチプロセッサが処理する入力列集合の集合を C_x とする．一つのサブアレイに対するブロック数 u が $u > \ell k + 1$ を満たす時，下記を満たすような C_x が存在する．

$$\max_x \left(\sum_{c_j \in C_x} \sum_{i \in c_j} w_i \right) < \frac{2n}{k}$$

証明. 各サブアレイから抽出されるピボット数は $\frac{u-1}{\ell}$ であり，その際のピボットの間隔 w_p は $w_p = \frac{n}{\ell\left(\frac{u-1}{\ell}+1\right)} < \frac{n}{u-1}$ となる．これが，後半開始時のブロック長の最大値となる．入力列集合のブロック長合計値 $\sum_{i \in c_j} w_i$ の最大値を $w_{\max}$ とおくと，後半のどのステップにおいても，$w_{\max}$ は w_p の ℓ 倍以下となるので，$w_{\max} < \frac{n\ell}{u-1} < \frac{n\ell}{\ell k} = \frac{n}{k}$ となる．また，すべてのブロック長が同じ大きさでかつ，すべてのマルチプロセッサが同じ数のブロックを処理すると仮定した際の，1マルチプロセッサが処理するブロック長の合計は，$\frac{n}{k}$ となる．

さて，各マルチプロセッサは，すべてのブロックの処理が終わるまで，未処理のブロックを処理し続ける．ここで，あるマルチプロセッサが α が存在して，処理ブロック長合計が $\frac{2n}{k}$ 以上になるとすると，ブロック長合計が平均未満，すなわち $\frac{n}{k}$ 未満となるマルチプロセッサ β が存在する．この場合，マルチプロセッサ α の入力列集合の一つを代わりにマルチプロセッサ β が処理することにする．入力列集合のブロック長合計の最大値は $\frac{n}{k}$ なので，このようにしても，マルチプロセッサ β の処理ブロック長合計は $\frac{2n}{k}$ 未満となる．これを繰り返すことにより，すべてのマルチプロセッサの処理ブロック長を $\frac{2n}{k}$ 未満にすることができる．以上により証明された． □

上記のような割当を見つけるためには，単に全ブロックを各マルチプロセッサに順番に割り当てていけばよい．ただし，処理ブロック長の合計が $\frac{2n}{b}$ 以上となる場合にはそのマルチプロセッサにはブロックを割り当てず，次のマルチプロセッサに割り当てる．上記の証明により，マルチプロセッサにブロックを割り当てることによりマルチプロセッサの処理ブロック長の合計が $\frac{2n}{b}$ 以上となる場合，そのブロックを必ず他のマルチプロセッサに割り当てることができる．提案アルゴリズムではこのような割当を用いることとする．

5. 提案アルゴリズムの計算量解析

5.1 基本ブロックのバイトニックソート

まず，グローバルメモリ上の1つの基本ブロックを1つのマルチプロセッサを用いてソートする時の計算量について説明する．基本ブロックはまず共有メモリにコピーされ，共有メモリ上でソートされた後，グローバルメモリに出力される．よって，I/O 計算量は2となる．各ステージの時間計算量は $\mathcal{O}(1)$ であり，ステージ数が $\mathcal{O}\left(\log^2 b\right)$ であるので，合計の時間計算量は $\mathcal{O}\left(\log^2 b\right)$ となる．

次にサブアレイのマージ処理の中で使用されるバイトニックソートについて解析する．共有メモリ上にある2つのソート済基本ブロックを1つのマルチプロセッサを用いてマージする．この場合，バイトニックソートの最後のフェーズのみ行えば良いので，ステージ数は $\mathcal{O}(\log b)$ となる．共有メモリ内の各バンクには高々2つの入力データが格納される．よって，共有メモリアクセスで発生するバンクコンフリクトの数は最大でも2となる．よって，時間計算量は $\mathcal{O}(\log b)$ となる．I/O 計算量は0である．

5.2 前半の計算量の解析

サブアレイを d 個ずつマージし，サブアレイの数を $1/d$ にする処理を1ステップと呼ぶ．まず，d 個のサブアレイを1つのマルチプロセッサでマージする処理の計算時間を解析する．s 回目のステップを考えると，サブアレイ長は $d^{(s-1)}b$ となる．

まず，1つのマルチプロセッサが d 個の入力列をマージする処理の計算量について説明する．ヒープの初期化については時間計算量は $\mathcal{O}(d \log b)$，I/O 計算量は基本ブロックの読み込みを d 回行うので，$\mathcal{O}(d)$ である．ヒープからキー最小の b データを取り出す処理の計算量については，時間計算量は heap の高さの分だけ，基本ブロックのバイトニックソートを行うので，$\mathcal{O}(\log d \log b)$ であり，I/O 計算量は b データの読み込みと書き込みを1回ずつ行うので，合計で2となる．マージを完了するためには，この処理を $d\left\lceil \frac{d^{s-1}b}{b} \right\rceil = d^s$ 回行えばよい．よって，1つのマルチプロセッサで d 個の入力列をマージする処理について，時間計算量は $\mathcal{O}\left(d \log b + d^s \log d \log b\right) = \mathcal{O}\left(d^s \log b \log d\right)$，IO 計算量は $\mathcal{O}\left(d + 2d^s\right) = \mathcal{O}\left(d^s\right)$ となる．

1ステップの処理全体では，入力サブアレイ数が $\left\lceil \frac{n}{d^{s-1}bd} \right\rceil = \left\lceil \frac{n}{d^s b} \right\rceil$ であり，それを $k = p/b$ 個のマルチプロセッサを用いて処理するので，時間計算量は $\mathcal{O}\left(\left\lceil \left\lceil \frac{n}{d^s b} \right\rceil \frac{1}{k} \right\rceil d^s \log b \log d\right) = \mathcal{O}\left(\left(\frac{n}{p} + d^s\right) \log b \log d\right)$,

I/O 計算量は $\mathcal{O}\left(\left\lceil \frac{n}{d^s b} \right\rceil d^s\right) = \mathcal{O}\left(\frac{n}{b}\right)$ となる．

最後に，前半の処理の合計の計算量を考える．前半の処理では s_0 回のステップの処理を行うので，時間計算量は $\mathcal{O}\left(\sum_{s=1}^{s_0}\left(\frac{n}{p}+d^s\right)\log b \log d\right) = \mathcal{O}\left(\log b \log d\left(\frac{n}{p}s_0 + d^{s_0+1}\right)\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}s_0\right)$ となる．

5.2.1 前半の処理をサブアレイ数が 1 個になるまで継続するときの計算量

後半の処理を行わず，前半の処理をサブアレイ数が 1 個になるまで（つまり最後まで）継続するとき，ステップ数は $s_0 = \mathcal{O}\left(\log_d \frac{n}{bd}\right)$ となる．よって，時間計算量は，$\mathcal{O}\left(\log b \log d\left(\frac{n}{p}\log_d \frac{n}{bd} + dd^{s_0}\right)\right) = \mathcal{O}\left(\frac{n}{p}\log b \log \frac{n}{b} + \frac{n}{b}\log b \log d\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}\log_d \frac{n}{b}\right)$ となる．

時間計算量は第二項の影響で大きくなる．提案アルゴリズムでは途中で後半の処理に切り替えることで，第二項が大きくなることを回避している．

5.3 データ振り分け処理の計算量の解析

データ振り分けの各処理について計算量を解析する．

まず，各サブアレイからピボット値を取り出す処理の計算量については，時間計算量は，$\mathcal{O}\left(\frac{n}{p}\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}\right)$ となる．次に，各サブアレイから取り出されたピボット値列をマージする処理について考える．全ピボットの要素数を $u-1$，残ったサブアレイ数を ℓ とする．ただし，$u\ell \leq \frac{n}{b}$ となるようにする．前半の処理と同等のマージ処理をこのデータに対して行うと時間計算量は $\mathcal{O}\left(\log b\left(\frac{u}{p}\log\frac{u}{b} + \frac{u}{b}\log d\right)\right)$，I/O 計算量は $\mathcal{O}\left(\frac{u}{b}\log_d \frac{u}{b}\right)$ となる．

次に，上記のピボット列を用いて各サブアレイを振り分ける処理については，前半のマージ処理と同等の解析を行うことにより，時間計算量は $\mathcal{O}\left(\left(\frac{n}{b\ell}+\frac{u}{b}\right)\frac{\ell}{k}\log b\right) = \mathcal{O}\left(\left(\frac{n}{p}+\frac{u\ell}{p}\right)\log b\right) = \mathcal{O}\left(\frac{n}{p}\log b\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n+u}{b}\right) = \mathcal{O}\left(\frac{n}{b}\right)$ となる．

以上より，データ振り分け処理全体では，時間計算量は $\mathcal{O}\left(\log b\left(\frac{n}{p}+\frac{u}{p}\log\frac{u}{b}+\frac{u}{b}\log d\right)\right)$，I/O 計算量は $\mathcal{O}\left(\frac{n}{b}+\frac{u}{b}\log_d\frac{u}{b}\right)$ となる．

後半の解析後に分かるように，ほとんどの値は他の処理の計算量の合計よりも小さくなる．唯一小さくならない可能性があるのが，ピボット列のマージ処理の時間計算量である．これは「前半のマージ処理」のアルゴリズムを使用しているためである．しかし，後述するように u と ℓ の値を適切に取ることにより，無視できるようになる．

5.4 後半の計算量の解析

まず，後半の t ステップ目を考える．i 番目の入力列の長さを w_i と書き，一つのマルチプロセッサが一つのヒープを使用してマージする際の入力列の集合を c_j とする．また，index x のマルチプロセッサが処理する入力列集合の集合を C_x とする．また，すべての c_j の集合を B_t とする．この時，$|B_t| = \frac{\ell}{d^{t-1}}\frac{1}{d}u$，$\sum_{c_j\in B_t}\sum_{i\in c_j} w_i = n$ が成り立つ．

u は $u > \ell k + 1$ を満たすものとして，補題 4.1 を用いて計算量を計算すると，計算量は以下のようになる．

時間計算量：

$$\mathcal{O}\left(\max_x\left(\sum_{c_j\in C_x}\sum_{i\in c_j}\left\lceil w_i\frac{1}{b}\right\rceil \log b \log d\right)\right)$$
$$= \mathcal{O}\left(\max_x\left(\sum_{c_j\in C_x}\sum_{i\in c_j}\left(\frac{w_i}{b}+1\right)\log b \log d\right)\right)$$
$$= \mathcal{O}\left(\frac{n}{p}\log b \log d\left(1+\frac{u}{d^{s_0+1}d^{t-1}}\right)\right)$$

I/O 計算量：

$$\mathcal{O}\left(\sum_{c_j\in B_t}\sum_{i\in c_j}\left\lceil w_i\frac{1}{b}\right\rceil\right)$$
$$= \mathcal{O}\left(\sum_{c_j\in B_t}\sum_{i\in c_j}\left(\frac{w_i}{b}+1\right)\right)$$
$$= \mathcal{O}\left(\frac{n}{b}\left(1+\frac{u}{d^{s_0+1}d^{t-1}}\right)\right)$$

最終ステップ数 t_0 は $t_0 = \mathcal{O}\left(\log_d \ell\right)$ となることから，後半の計算量合計は以下のようになる．

時間計算量：

$$\mathcal{O}\left(\sum_{t=1}^{t_0}\frac{n}{p}\log b \log d\left(1+\frac{u}{d^{s_0+1}d^{t-1}}\right)\right)$$
$$= \mathcal{O}\left(\frac{n}{p}\log b \log d\left(t_0+\frac{u}{d^{s_0+1}}\sum_{t=1}^{\log_d n}\frac{1}{d^{t-1}}\right)\right)$$
$$= \mathcal{O}\left(\frac{n}{p}\log b \log d\left(t_0+\frac{u}{d^{s_0+1}}\right)\right)$$

I/O 計算量：

$$\mathcal{O}\left(\sum_{t=1}^{t_0}\frac{n}{b}\left(1+\frac{u}{d^{s_0+1}d^{t-1}}\right)\right)$$
$$= \mathcal{O}\left(\frac{n}{b}\left(t_0+\frac{n}{d^{s_0+1}}\right)\right)$$

5.5 ソートの全計算量の解析

前半終了時のサブアレイ数 ℓ については $\ell = \mathcal{O}\left(\frac{n}{bd^{s_0+1}}\right)$ が成り立つ．また，t_0 については $t_0 = \log_d \frac{n}{b} - s_0$ が成り立つ．また，$n = \Omega\left(b^2\right)$ としてソート全体の計算量を計算すると以下のようになる．

時間計算量：

$$\mathcal{O}\left(\frac{n}{p}\log b \log\frac{n}{b} + \log b \log d\left(\frac{n}{b\ell}+\frac{u\ell}{k}+\frac{u}{b}\right)\right)$$

I/O 計算量：

$$\mathcal{O}\left(\frac{n}{b}\log_d\frac{n}{b}+u\ell\right)$$

5.6 最適なパラメータについて

ℓ と u の値を適切に設定することにより，時間計算量，I/O 計算量それぞれの第二項以降を消去することを考える．

ここで，$\ell = k$，$u = \frac{n/b}{\ell}$ とする．ℓ, u は $\ell \leq u$ を満たす必要があるが，n が十分に大きければこれを満たしている．また，d の値は共有メモリサイズにより制約される．ヒープのためのバッファを共有メモリに格納する際の d の最大値を取り，$d = \mathcal{O}\left(M/b\right)$ とする．この時，計算量は以下の

ようになる．
時間計算量：

$$\mathcal{O}\left(\frac{n}{p}\log b\log\frac{n}{b}+\log b\log d\left(\frac{n}{p}+\frac{n}{p}\right)\right)=\mathcal{O}\left(\frac{n}{p}\log b\log\frac{n}{b}\right)$$

I/O 計算量：

$$\mathcal{O}\left(\frac{n}{b}\log_d\frac{n}{b}+\frac{n}{b}\right)=\mathcal{O}\left(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b}\right)$$

よって，上記のように設定することで時間計算量は最適値の $\mathcal{O}(\log b)$ 倍，時間計算量は最適値となる．

6. 結論

本報告では，GPU 向けの比較ソートアルゴリズムについて，AGPU モデル上での計算量の下界を示した後，I/O 計算量が下界と一致するアルゴリズムを提案した．時間計算量についても下界の $\log b$ 倍となっている．今後は CUDA での実装と評価を行う予定である．

謝辞　この研究の一部は文部科学省科学研究費 (基盤 A 23240002) の援助を受けた．

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] Batcher, K. E.: Sorting networks and their applications, *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), New York, NY, USA, ACM, pp. 307–314 (online), DOI: 10.1145/1468075.1468121 (1968).

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

[4] 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).

[5] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: high performance graphics coprocessor 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] Merrill, D. G. and Grimshaw, A. S.: Revisiting sorting for GPGPU stream architectures, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, New York, NY, USA, ACM, pp. 545–546 (online), DOI: 10.1145/1854273.1854344 (2010).

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

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 小池 敦, 定兼 邦彦：GPU のための並列計算モデル，コンピュテーション研究会 IEICE-COMP2012-42，電子情報通信学会 (2012).

[18] 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).

[19] 小池 敦, 定兼 邦彦, Hoa Vu：AGPU モデルでの並列ソートアルゴリズムの計算量について，コンピュテーション研究会 IEICE-COMP2013-13，電子情報通信学会 (2013).