

GPU アクセラレータと不揮発性メモリを考慮した外部 ソート (Unrefereed Workshop Manuscript)

佐藤 仁^{1,2} 溝手 竜^{1,2} 松岡 聡^{1,2}

概要：GPU アクセラレータと不揮発性メモリを考慮した外部ソートアルゴリズム `xtr2sort` (extreme external sort) を提案する。GPU の高い演算性能とメモリバンド幅を活かし、不揮発性メモリ、ホストメモリ、デバイスメモリ間のデータ移動に伴う遅延を隠蔽するために、不揮発性メモリ上のソートの対象となるレコードをデバイスメモリの収まるサイズへチャンクに分割し、チャンク毎にパイプラインで不揮発性メモリへの I/O 操作、CPU-GPU 間のメモリ転送、GPU 上でのソート処理を非同期に行うことで、デバイスメモリやホストメモリの容量を超えたサイズのレコードに対しても高速なソートを行う。提案手法を 2-way の Intel Xeon E5-2699 v3 2.30GHz (18 コア)、NVIDIA Tesla K40 を搭載した 1 台のサーバで評価した結果、Linux Asynchronous I/O(`libaio`) を用いたノンブロッキング I/O による提案手法の実装において、CPU 上で実行可能なレコード数の 4 倍、GPU 上で実行可能なレコード数の 64 倍となる 25.6×10^9 の `int64_t` 型の整数値からなるレコードに対し、78,121,548 records/sec で動作し、2 ソケット 72 スレッドで動作させた CPU 版のノンブロッキング I/O による out-of-core ソートと比較して 2.16 倍の性能を示すことを確認した。これらから、GPU アクセラレータを用いた Out-of-core な処理に向けて、不揮発性メモリを組み合わせ I/O のチャンク化と遅延隠蔽を行うことが良好な手法であることが伺える。

1. はじめに

ソートは、データベース管理システム、ビッグデータミドルウェアフレームワーク、大規模データ処理を伴う科学技術計算アプリケーションなど様々なソフトウェアで基本的な処理であり、その高速化は常に重要な課題である。ソートアルゴリズムの性能は理論的には計算量により決まるが、その実効性能は実行されるマシンの特性を活かした実装技術に依存することが多い。とりわけ、マルチコア CPU やメニーコア GPU など昨今のモダンなプロセッサを搭載したマシン上でのソートの高速化では、コア数を増やしてスレッド並列性を上げる、SIMD 処理によりデータ並列性を上げる、in-core でかつオンチップメモリを活用しイレギュラーなメモリアccessを避ける、などの手法が必要となる [7]。しかし、ヘルスケア、システム生物学、ソーシャル・ネットワーク、ビジネスインテリジェンス、スマートグリッドなどビッグデータアプリケーションでは大規模なデータセットの解析の高速化のために大容量の DRAM を必要とする一方で、容量あたりの導入コストの高さや

消費電力の高さ、また、将来の計算機アーキテクチャに向けてはプロセッサのコア数の増大や DRAM を構成する半導体の集積度の限界などにより利用可能なコアあたりの DRAM のバンド幅・容量が少なくなることが問題となる。

一方、昨今、DRAM と比較して低バンド幅と高レイテンシだが大容量で低コストという特性を持ったフラッシュなどに代表される不揮発性メモリデバイスが登場し、明示的なデータ移動が必要であるもののバンド幅を必要としない処理に伴うデータセットを積極的に不揮発性メモリへオフロードすることで、アプリケーションが必要とするバンド幅と容量を稼ぐなどの活用が期待されている。また、同様の状況は、マルチコア CPU とメニーコア GPU の関係にも当てはまり、GPU のデバイスメモリを超えるような大容量のメモリを必要とするアプリケーションでは、GPU の持つ高い演算性能とメモリバンド幅を活用するために、演算性能やメモリバンド幅を必要としない処理を伴うデータセットを積極的にホストメモリへオフロードする必要がある。ソートにおいても、GPU のデバイスメモリと不揮発性メモリを組み合わせることにより、GPU のデバイスメモリや CPU のホストメモリの容量を超えるような巨大なデータセットに対する高速化が実現できると考えられる。

¹ 東京工業大学

² 独立行政法人科学技術振興機構, CREST

しかし、多階層のメモリを対象とした実装の煩雑さや、不揮発性メモリへの I/O や CPU-GPU 間のメモリ転送の隠蔽手法、また、最新のデバイスを対象とした最適化手法・性能特性は明らかではない。

そこで、我々は GPU アクセラレータと不揮発性メモリを考慮した外部ソートアルゴリズム *xtr2sort* (extreme external sort) を提案する。GPU の高い演算性能とメモリバンド幅を活かし、不揮発性メモリ、ホストメモリ、デバイスメモリ間のデータ移動に伴う遅延を隠蔽するために、不揮発性メモリ上のソートの対象となるレコードをデバイスメモリの収まるサイズへチャンクに分割し、チャンク毎にパイプラインで不揮発性メモリへの I/O 操作、CPU-GPU 間のメモリ転送、GPU 上でのソート処理を非同期に行うことで、デバイスメモリやホストメモリの容量を超えたサイズのレコードに対しても高速なソートを行う。

提案手法を CPU が Intel Xeon E5-2699 v3 2.30GHz (18 コア) 2 ソケット、DRAM が DDR4-2133 128GB、SSD が Huawei ES3000 v1 PCIe SSD 2.4TB、GPU が NVIDIA Tesla K40 (GDDR5 メモリ 12GB) からなる 1 台のサーバで実験したところ、Linux Asynchronous I/O (libaio) を用いたノンブロッキング I/O による提案手法の実装において、CPU 上で実行可能なレコード数の 4 倍、GPU 上で実行可能なレコード数の 64 倍となる 25.6×10^9 の *int64_t* 型の整数値からなるレコードに対し、78,121,548 records/sec で動作し、2 ソケット 72 スレッドで動作させた CPU 版のノンブロッキング I/O による out-of-core ソートと比較して 2.16 倍の性能を示すことを確認した。これらから、GPU アクセラレータを用いた Out-of-core な処理に向けて、不揮発性メモリを組み合わせ I/O のチャンク化と遅延隠蔽を行うことが良好な手法であることが伺える。

2. 関連研究

GPU アクセラレータの高い演算性能とメモリバンド幅を活用するべく、GPU Radix Sort [5]、GPU Quick Sort [2]、GPU Sample Sort [4]、GPU Tera Sort (Bitonic Sort) [3]、GPU Merge Sort [7] など GPU を対象とした様々なアルゴリズムが提案されている。しかし、これらのアルゴリズムは in-core なソートアルゴリズムであるため、GPU のデバイスメモリの容量を超えるような巨大なデータセットは扱うことができない。

GPU のデバイスメモリの容量を超えるデータセットのソートを目的とした Out-of-core なソートアルゴリズムとしては、サンプルソート及びマージソート由来の CPE Sort (CUDA-based Parallel External Sort) [6] や GPUMemSort [11] が挙げられる。我々も GPUMemSort を拡張したオルタナティブな実装 [8] を開発している。これらのアルゴリズムは、本質的には類似しており、GPU アクセラレータの高い演算性能とメモリバンド幅を活かし CPU-GPU 間

のデータ転送の際のバンド幅の要求を低く抑えるために、ソート対象のレコードをスプリッタにより GPU のデバイスメモリの容量に収まるようなバケットへ分類し、バケット単位で GPU へのデータ転送と GPU 上でのソート処理を行い、最終的な結果をマージする。しかし、これらのアルゴリズムは CPU のホストメモリと GPU のデバイスメモリとの 2 つのメモリ階層間の Out-of-core のソート処理に特化している。また、多階層のメモリを考慮したソートアルゴリズムの提案としては Bender ら [1] のものがあるものの GPU アクセラレータや不揮発性メモリは考慮されていない。

近年、アクセラレータからの I/O 手法の抽象化に関する研究がいくつか提案されている [9]、[10] もの、ソートアルゴリズムの実装手法を考慮した上での不揮発性メモリへの I/O を含めた CPU-GPU 間のメモリ転送の隠蔽手法・コデザイン、また、最新のデバイスを対象とした最適化手法・性能特性は明らかではない。

3. *xtr2sort*

3.1 アルゴリズムの概要

提案手法は、Ye らにより提案されたサンプルソート由来の GPU 向け Out-of-core なソートアルゴリズムである GPUMemSort [11] やそのオルタナティブな実装 [8] をベースに、不揮発性メモリを利用してデバイスメモリやホストメモリの容量を超えたサイズのレコードへも対応するように拡張したものである。GPU の高い演算性能とメモリバンド幅を活かし、不揮発性メモリ、ホストメモリ、デバイスメモリ間のデータ移動に伴う遅延を隠蔽するために、不揮発性メモリ上のソートの対象となるレコードをデバイスメモリの収まるサイズへチャンクに分割し、チャンク毎にパイプラインで不揮発性メモリへの I/O 操作、CPU-GPU 間のメモリ転送、GPU 上でのソート処理を非同期に行うことで、デバイスメモリやホストメモリの容量を超えたサイズのレコードに対しても高速なソートを行う。

図 1 に提案アルゴリズムの概要を示す。入力型 T の未ソートのレコードからなるファイルとして不揮発性メモリ上に置かれる。従って、レコード数 n はファイルのサイズ f を型のサイズ $sizeof(T)$ で割ることにより取得できる。また、出力も同様にソート済みのレコードからなるファイルとして不揮発性メモリ上に置かれる。サンプルソートでは、入力レコードを c 個のチャンクへ分割するために、 $c-1$ 個のランダムに選択されたスプリッタを導入する。提案手法では、各チャンクが GPU のデバイスメモリに収まることが必要であり、入力レコードのサイズと各パイプラインが使用する GPU のデバイスメモリの容量を考慮してチャンク数 c を予め決定する。詳細な *xtr2sort* のアルゴリズム

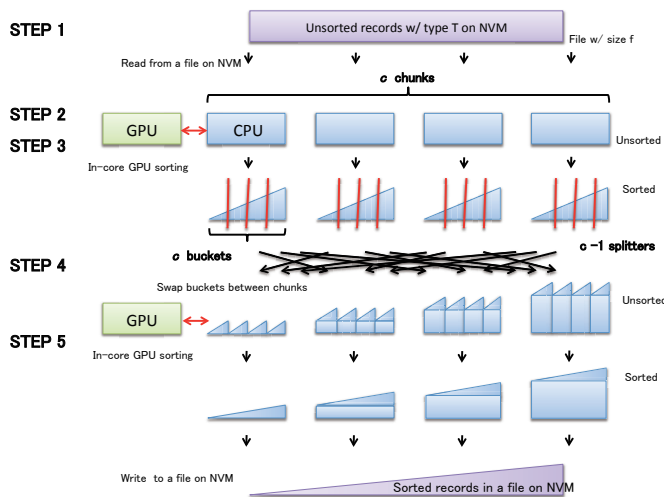


図 1 xtr2sort のアルゴリズムの概要

の手順は以下のとおりである。

Step1 GPU のデバイスメモリの容量に収まるように不揮発性メモリ上の入力となるレコードを均一なレコード数からなる c 個のチャンクに分割する。

Step2 不揮発性メモリ上の入力となるレコードからランダムに s 点を選択し、不揮発性メモリから読み込んでソートし、その中からスプリッタとなる $c-1$ 個のレコードを選択する。この際、 $(k+1) \cdot s/c$ 番目の点を選択し、その値を $splitters[k]$ とする。ただし、 $k \in [0, c)$ とする。

Step3 チャンク毎に不揮発性メモリから GPU のデバイスメモリへデータを転送し、GPU 向けの in-core ソートアルゴリズムを用いて GPU 上でソートを行う。その後、 c 個のバケットに分割する。この際、 k 番目のバケット内のレコードの値の最大値が $splitters[k]$ 以下となるようにする。実際には、ファイルのオフセットの位置とバケット内のレコード数を記録する。最終的に、 c 個のバケットに分割されたソート済みのチャンクは不揮発性メモリに書き戻される。

Step4 k 番目のチャンクに含まれるレコードの最大値が $splitters[k]$ 以下になるように c 個のチャンク間でバケットの交換を行う。実際には、バケットが保持するファイルのオフセットの位置とバケット内のレコード数を交換する。

Step5 バケット交換を行なったチャンク毎に不揮発性メモリから GPU のデバイスメモリへデータを転送し、GPU 向けの in-core ソートアルゴリズムを用いて GPU 上でソートを行う。Step4 ではファイルのオフセット位置とレコード数のみを交換しているため、ファイル上の分散した位置に存在するバケット毎に読み出しチャンクを再構成する必要がある。その際にファイルへの細粒度な I/O が発生する。その後、ソート済みのチャンクを不揮発性メモリへ書き戻す。

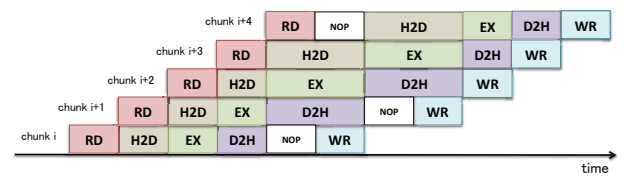


図 2 ブロッキング I/O を用いた場合の 5 段階パイプライン実行

3.2 メモリ階層間のデータ移動の遅延隠蔽

3.2.1 概略

xtr2sort では、前節の Step3 や Step5 で導入されるような異なるメモリ階層間のデータ転送の際にいかにレイテンシを隠蔽するかが重要な課題となる。そのために、入力レコードを分割した各チャンクに対して、不揮発性メモリに対する I/O 操作、デバイスメモリに対するデータ転送、GPU 上でのソート処理を非同期にパイプラインで実行することにより実現する。ホストメモリからデバイスメモリへのデータ転送と GPU 上でのソート処理をオーバーラップさせるために、CUDA Stream によるストリームに対して `cudaMemcpyAsync` 関数による非同期データ転送と CUDA カーネルによる非同期実行を行う。この際、CPU-GPU 間で効率のよいデータ転送を達成するためには、ホストメモリ上でページロックされたメモリ領域 (pinned memory) が必要となる。一方、不揮発性メモリに対する I/O については、`pread` や `pwrite` によるブロッキング I/O を用いる手法、Linux Asynchronous I/O (libaio) によるノンブロッキング I/O を用いる手法が考えられる。

3.2.2 ブロッキング I/O による遅延隠蔽手法

図 2 にブロッキング I/O を用いた場合のパイプライン実行の概略を示す。パイプラインのステージは、不揮発性メモリ上のファイルからホストメモリへ読み込みを行う RD ステージ、ホストメモリからデバイスメモリへ非同期データ転送を行う H2D ステージ、GPU 上で CUDA カーネルの非同期実行によりソート処理を行う EX ステージ、デバイスメモリからホストメモリへ非同期データ転送を行う D2H ステージ、そして、ホストメモリから不揮発性メモリ上のファイルへ書き込みを行う WR ステージの計 5 段階からなる。このうち、H2D ステージ、EX ステージ、D2H ステージは各々異なる CUDA Stream(最大 3 本)で動作する。また、RD ステージ、H2D ステージで読み込み I/O 用のバッファが計 2 つ、D2H ステージ、WR ステージで書き込み I/O 用のバッファが計 2 つ存在し、各バッファのサイズはチャンクの最大サイズである。ブロッキング I/O を用いた場合、実装が簡便になる利点があるものの、図中の nop (no operation) のように読み込み・書き込み I/O がブロックするという欠点がある。ただし、この場合においても、CPU-GPU 間のデータ転送や GPU 上のソート処理をオーバーラップすることは可能である。

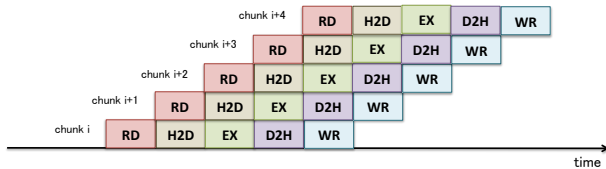


図3 ノンブロッキングI/Oを用いた場合の5段パイプライン実行 (STEP3)

3.2.3 ノンブロッキングI/Oによる遅延隠蔽手法

一方、ノンブロッキングI/Oを用いた場合、I/O操作によるブロックが発生しないため、不揮発性メモリに対する読み込みI/O・書き込みI/O、及び、CPU-GPU間のデータ転送やGPU上のソート処理の全てを完全にオーバーラップすることが可能であるが、libaioでノンブロッキングI/Oを行うためにはファイルをO_DIRECTフラグをつけてopenする必要がある、また、ホストメモリ上のバッファ、ファイルのオフセット、転送サイズなどが論理ブロックサイズ(512バイト)のアライメントである必要があるため、実装が煩雑になるという欠点がある。CPU-GPU間の非同期データ転送と組み合わせる場合、バッファとして確保されるページロックされたメモリ領域はアライメントされているため、ファイルのオフセット、転送データサイズを考慮すれば基本的にはそのままメモリ領域を利用することが可能である。実際、STEP3では、各チャンクが均一なレコード数であるため、論理ブロックサイズ単位で連続して読み込みI/O・書き込みI/Oを行うことが可能であり、図3に示すように5段のステージからなるパイプライン処理で各ステージを完全にオーバーラップすることが可能である。しかし、STEP5では、ファイルの読み出し時にファイル上の分散した位置に存在するパケット毎に読み出しチャンクを再構成する必要がある、ナイーブに実装した場合、バッファ、ファイルオフセット、転送サイズなどを論理ブロックサイズのアライメントにすることが困難である。また、ファイルの書き込み時も同様に、各チャンクサイズは均一ではなく異なり、任意の位置へファイルの書き込みが発生するため、ナイーブに実装した場合、ファイルへの書き込み位置のオフセット、バッファ、転送サイズなどを論理ブロックサイズのアライメントにすることが困難である。このため、提案手法では、libaioによる読み込みI/O・書き込みI/OのためのバッファとCPU-GPU間のデータ転送のバッファを別途用意し、明示的にバッファ間でコピーを行うことで対処する。図4にその処理の概要を示す。RDステージでは、不揮発性メモリ上のファイルからlibaioにより読み込みI/O用のバッファヘデータの読み込みを行う。1つの読み込みI/O用のバッファはパケットの最大サイズからなる c 個のサブバッファで構成される。次に、R2HステージでCPU-GPU間データ転送用のバッファヘデータのコピーを行う。H2Dステージ、EXステー

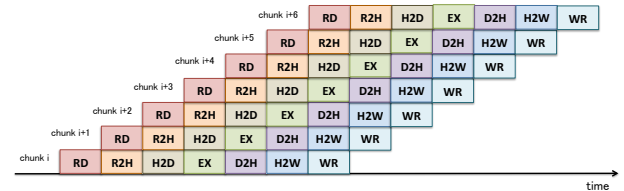


図4 ノンブロッキングI/Oを用いた場合の7段パイプライン実行 (STEP5)

ジ、D2Hステージで、各々異なる3本のCUDA StreamによりCPU-GPU間のデータ転送とGPU上でのソートの処理を非同期に行い、H2Wステージでlibaioによる書き込みI/O用のバッファヘデータのコピーを行う。この際、アライメントされたファイルの書き込みになるようにバッファの調整を行う。書き込みI/O用のバッファのサイズはチャンクの最大サイズである。最後に、WRステージでlibaioにより不揮発性メモリ上のファイルヘデータの書き込みを行う。R2HステージとH2Wステージではメモリコピーの際のブロックを避けるため別途スレッドを起動して非同期に実行している。libaioによる読み込みI/O・書き込みI/Oのためのバッファは、それぞれ、RDステージとR2Hステージ、H2DステージとWRステージで重複するため、各々2つずつ存在する。同様に、CPU-GPU間のデータ転送のバッファも、CPUからGPUへのデータ転送時のR2HステージとH2Dステージ、GPUからCPUへのデータ転送時のD2HステージとH2Wステージでそれぞれ重複するため、各々2つずつ存在する。

4. 予備実験

提案手法の有効性を明らかにするためにxtr2sortについて、レコード数に対するスケラビリティ、メモリ階層間データ移動の遅延隠蔽の効果についての実験を行なった。比較対象となるソートアルゴリズムの実装は以下のものとした。

xtr2sort+libaio 3節で述べたxtr2sortによる実装。ただし、ファイルI/OはlibaioによるノンブロッキングI/Oを行う。GPUのin-coreソートにはthrustライブラリを用いる。

xtr2sort+pio 3節で述べたxtr2sortによる実装。ただし、ファイルI/Oはpread, pwriteによるブロッキングI/Oを行う。GPUのin-coreソートにはthrustライブラリを用いる。

out-of-core-cpu+libaio(n) 3節で述べたxtr2sortと同様のアルゴリズムであるが、GPUによるソート処理やCPU-GPU間のデータ転送は行わず、CPUによるGNU libcppのParallel Mode拡張を用いてスレッド数 n でソートを行う実装。ただし、ファイルI/OはlibaioによるノンブロッキングI/Oを行う。また、CPU上で利用可能なメモリ容量はGPUの場合と同様

に 12GB としている。

out-of-core-cpu+pio(n) 3節で述べた xtr2sort と同様のアルゴリズムであるが、GPU によるソート処理や CPU-GPU 間のデータ転送は行わず CPU による GNU libc++ の Parallel Mode 拡張を用いてスレッド数 n でソートを行う実装。ただし、ファイル I/O は read, pwrite によるブロッキング I/O を行う。また、CPU 上で利用可能なメモリ容量は GPU の場合と同様に 12GB としている。

out-of-core-gpu ファイル I/O は行わず、ホストメモリ上のレコードに対して 3節で述べた xtr2sort と同様のアルゴリズムで CPU-GPU 間の非同期データ転送、GPU 上でのソートの非同期実行を行う実装。GPU の in-core ソートには thrust ライブラリを用いる。

in-core-cpu(n) ホストメモリ上のレコードに対して、CPU による GNU libc++ の Parallel Mode 拡張を用いてスレッド数 n でソートを行う実装。

in-core-gpu ホストメモリ上のレコードに対して、CPU-GPU 間の同期データ転送を行い、thrust ライブラリにより GPU 上で in-core ソートを行う。

ソートの対象となるデータは一樣乱数で生成された int64_t 型の整数値を用いた。

実験環境は、CPU が Intel Xeon E5-2699 v3 2.30GHz (18 コア) 2 ソケット、DRAM が DDR4-2133 128GB、SSD が PCI-e Gen3 x16 のスロットに接続された Huawei ES3000 v1 PCIe SSD 2.4TB、GPU が PCI-e Gen3 x16 のスロットに接続された NVIDIA Tesla K40 (GDDR5 メモリ 12GB) からなる 1 台のサーバである。SSD の性能は、逐次読み込み I/O が 3.2GB/s、逐次書き込み I/O が 2.8GB、4KB のランダム読み込み I/O が平均 760,000、4KB のランダム書き込み I/O が平均 240,000 であった。ソフトウェアに関しては、OS は Linux 3.19.8、gcc は 4.4.7、CUDA は 7.0、thrust は 1.8.1、ファイルシステムは xfs を用いた。

4.1 ソートのスループット

図 5 に各ソートアルゴリズムの実装でのレコード数に対するスループットの結果を示す。x 軸はレコード数 [10^9 records] を表し、y 軸はスループット [records/sec] を表す。 0.4×10^9 以下のレコード数を対象としたソートでは、GPU 上の in-core なソートアルゴリズムの実装が (in-core-gpu) が圧倒的に高速であったが、今回使用した GPU のデバイスメモリの容量は 12GB 程度なので 0.4×10^9 より多いレコードに対してはソートを行うことができない。同様に、 0.4×10^9 より大きく 3.2×10^9 以下のレコード数を対象としたソートでは、2 ソケット 72 スレッドを用いた in-core-cpu によるソートアルゴリズム実装 (in-core-cpu(72)) が最も高速であったが、今回の実験環境では、 6.4×10^9 までしかホストメモリにレコードを載せることができず、それ以上のレ

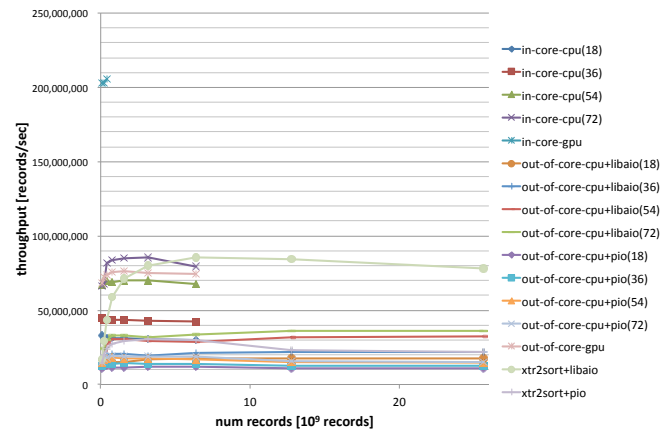


図 5 各ソートアルゴリズムの実装でのレコード数に対するスループット

コード数に対しては in-core-cpu によるソートアルゴリズム実装ではソートを行うことができない。一方、out-of-core なソートアルゴリズムの実装はレコード数が 6.4×10^9 を超えた場合 (例えば、ホストメモリに載せることができる最大のレコード数の 4 倍、デバイスメモリに載せることができる最大のレコード数の 64 倍である 25.6×10^9 とした場合) においてもソートを行うことが可能である。その中でも、我々の提案手法である libaio を用いたノンブロッキング I/O による xtr2sort アルゴリズム (xtr2sort+libaio) の実装が最も高速であり、 25.6×10^9 のレコードに対して 78,121,548 records/sec で動作することを確認した。これは、CPU を用いた out-of-core ソートで最も高速であった libaio を用いたノンブロッキング I/O による out-of-core-cpu によるソートアルゴリズムの実装を 2 ソケット 72 スレッドで動作させた場合 (out-of-core-cpu+libaio(72)) と比較して 2.16 倍の性能を示している。ただし、out-of-core-cpu によるソートアルゴリズムの実装もホストメモリの利用の改善など更なる最適化の余地がある。

4.2 メモリ階層間のデータ移動の遅延隠蔽の効果

メモリ階層間のデータ移動の際にどの程度遅延を隠蔽できているのかを確認するために、図 6 に提案手法と out-of-core-cpu のソートアルゴリズムの実装の実行時間の内訳 [ms] を示す。各々、I/O に関して read, pwrite によるブロッキング I/O を用いたもの (pio) と、libaio によるノンブロッキング I/O (libaio) を用いたものを記載している。ここで、非同期処理 (CPU-GPU 間データ転送、GPU 上のソート処理、ノンブロッキング I/O) の場合は処理を投入するまでの時間を計測し、実際に処理が終了するまでの時間は sync の項に現れる。out-of-core-cpu のソートアルゴリズムの実装では実際にソートを行なっている時間が大部分を占めるのに対し、out-of-core-gpu のソートアルゴリズムの実装では、ソートの処理は GPU 上で非同期に行われるため実行時間が削減されている一方、sync の時間が増え

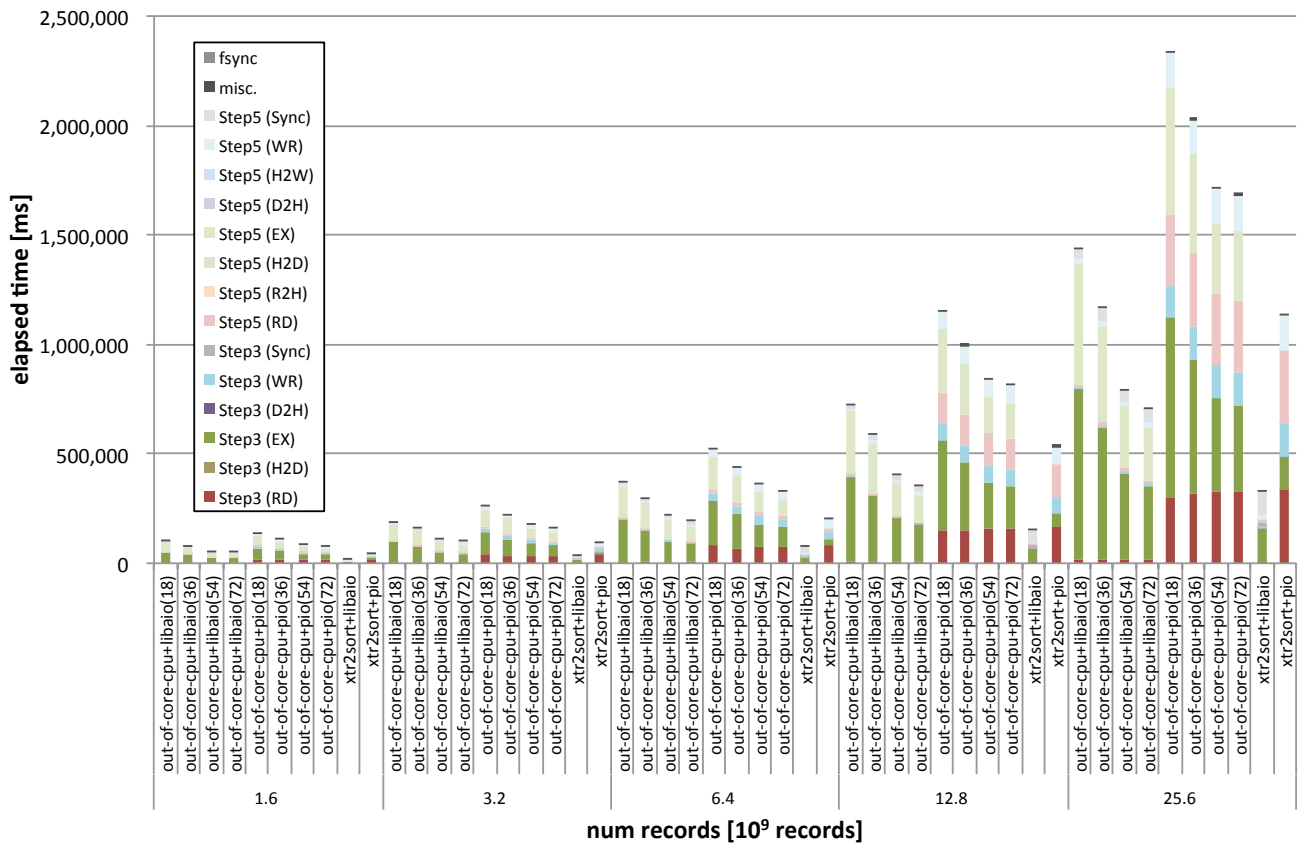


図 6 提案手法と out-of-core-cpu のソートアルゴリズムの実装の実行時間の内訳 [ms]

ており、処理のオーバーラップの効果があることが確認できる。ブロッキング I/O とノンブロッキング I/O の比較に関しては、xtr2sort+pio や out-of-core-cpu+pio のソートアルゴリズムの実装においては読み込み I/O・書き込み I/O の実行時間が顕在化しており隠蔽されていない一方で、xtr2sort+libaio や out-of-core-cpu+libaio のソートアルゴリズムの実装においては読み込み I/O・書き込み I/O の実行時間が隠蔽されており、ノンブロッキング I/O の効果が高いことを確認した。

また、libaio を用いたノンブロッキング I/O による提案手法 (xtr2sort+libaio) において、Step5 の各パイプライン処理の各ステージの実行時間の分布を図 7 に示す。ファイルの読み込みを行う RD ステージ、ファイルの書き込みを行う WD ステージに実行時間の多くが占められており、続いて、実際に GPU 上でソートを行う EX ステージの実行時間の多くが費やされている傾向があることを確認した。このことから、現状では I/O がボトルネックとなっており、更なる高速な不揮発メモリデバイスの利用や I/O の最適化などが必要であると考えられる。

5. まとめと今後の課題

ビッグデータアプリケーションにおいてソーティングは重要な処理の一つである。巨大なデータセットのソーティ

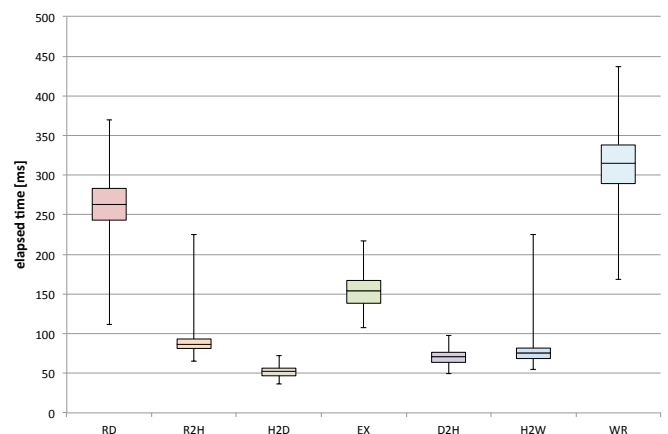


図 7 xtr2sort+libaio のソートアルゴリズムの実装における各パイプライン処理の各ステージの実行時間の分布

ングは GPU アクセラレータにより高速化することが期待できるが、GPU のデバイスメモリの容量やホストノードの DRAM の容量などによりソート対象のデータセットの規模が著しく制限されることが問題となる。本稿では、GPU アクセラレータと不揮発性メモリを考慮した外部ソートアルゴリズム xtr2sort (extreme external sort) を提案し、GPU のデバイスメモリ、CPU のホストメモリを超えるようなサイズのデータセットのソートに対しても、GPU の高い演算性能とメモリバンド幅を活かし、不揮発性メモ

リ, ホストメモリ, デバイスメモリ間のデータ移動に伴う遅延を隠蔽することが可能であることを示した。今後の課題としては, 更なるパイプライン処理の最適化 (特にノンブロッキング I/O など) や, GPU アクセラレータを用いた Out-of-core な処理に向けて, 同様の手法の一般化やその他のビッグデータカーネルへの適用などが挙げられる。

謝辞 本研究の一部は JST CREST「ポストペタスケールシステムにおける超大規模グラフ最適化基盤」, 「EBD: 次世代の年ヨットバイト処理に向けたエクストリームビッグデータの基盤技術」, 及び, JSPS 科研費 26540050 の助成を受けたものである。

参考文献

- [1] Bender, M. A., Berry, J. W., Hammond, S., Hemmert, K., McCauley, S., Moore, B., Phillips, C. A., Resnick, D. and Rodrigues, A.: Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation, *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS2015)* (2015).
- [2] Cederman, D. and Tsigas, P.: GPU-Quicksort: A practical Quicksort algorithm for graphics processors, *Journal of Experimental Algorithmics*, Vol. 14, pp. 4:1.4—4:1.24 (online), DOI: 10.1145/1498698.1564500 (2009).
- [3] Gray, J.: GPU TeraSort: high performance graphics co-processor sorting for large database management, *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data (SIGMOD'06)*, pp. 325–336 (2006).
- [4] Leischner, N., Osipov, V. and Sanders, P.: GPU Sample Sort, *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–10 (2010).
- [5] 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).
- [6] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead, *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010*, pp. 1–8 (online), DOI: 10.1109/IPDPSW.2010.5470833 (2010).
- [7] 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)*, pp. 351–362 (online), DOI: 10.1145/1807167.1807207 (2010).
- [8] Shirahata, K., Sato, H. and Matsuoka, S.: Out-of-core GPU Memory Management for MapReduce-based Large-scale Graph Processing, *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 221–229 (2014).
- [9] Si, M. and Ishikawa, Y.: Design of direct communication facility for many-core based accelerators, *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, Vol. 1, pp. 924–929 (online), DOI: 10.1109/IPDPSW.2012.113 (2012).
- [10] Silberstein, M., Ford, B., Keidar, I. and Witchel, E.: GPUfs: Integrating a file system with GPUs, *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 32, pp. 485–498 (online), DOI: 10.1145/2451116.2451169 (2013).
- [11] Ye, Y., Du, Z., Bader, D. A., Yang, Q. and Huo, W.: GPUMemSort: A High Performance Graphics Co-processors Sorting Algorithm for Large Scale In-Memory Data, *GSTF International Journal on Computing*, Vol. 1, No. 2, pp. 23–28 (online), DOI: 10.5176/2010-2283.1.2.34 (2011).