Vol.2016-HPC-155 No.3
2016/8/8

I/O分割による遅延隠蔽を用いたOut-of-coreなGPU Set Intersectionの性能評価 (Unrefereed Workshop Manuscript)

佐藤 仁[1,2,3]　溝手 竜[2,3]　松岡 聡[2,3,1]　小川 宏高[1]

概要：GPU アクセラレータと不揮発性メモリデバイスを考慮した Out-of-core な Set Intersection 手法 ooc_set_intersection 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，対象となる入力レコード列をデバイスメモリに収まるサイズへチャンクに分割し，GPU 上での Set Intersection 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する Set Intersection 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，CPU 上で実行可能なレコード総数の 16 倍，GPU 上で実行可能なレコード総数の 512 倍となる 2^{36} の int64_t 型の整数値からなるランダムでユニークな 2 つの入力レコード (レコード総数は $2^{37}(137,438,953,472)$) に対し，311,583,199 records/sec で動作し，2 ソケット 72 スレッドで動作させた CPU 版の非同期 I/O による Out-of-core な Set InterSection と比較して 1.03 倍の性能を示すことを確認した．また，これらの結果を踏まえ，GPU アクセラレータと不揮発性メモリを活用したビッグデータ処理カーネルの Out-of-core 化に向けて性能上のボトルネックの調査を行い，高速化への指針を得た．

1. はじめに

Set Intersection は，データベース管理システムの JOIN 操作や，ウェブ検索のクエリ処理など，ビッグデータ処理で頻出する重要なカーネルの一つであり，その高速化は常に重要な課題である．とりわけ，GPU アクセラレータは，高い演算性能とメモリバンド幅を活かすことにより，Set Intersection の高速化が可能であることが知られている [1], [2], [6], [14] が，GPU のデバイスメモリの容量 (概ね数十 GB 程度) などにより対象となる入力レコード列の規模が制限されることが問題になる．また，CPU に関してもマルチスレッドや SIMD 処理を活用することで Set Intersection の高速化が可能であることが知られている [7], [8], [12] が，CPU を搭載するホストノードの DRAM の容量などにより対象となる入力レコード列の規模が制限されることが問題となる．一般に，ビッグデータ処理においては大規模なデータセットに対する解析の高速化のために高速な大容量の 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をはじめとして様々な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．

我々これまで GPU アクセラレータと不揮発性メモリを考

[1] 産業技術総合研究所
[2] 東京工業大学
[3] 科学技術振興機構, CREST

慮した Out-of-core なソート手法を提案してきた [11], [15]. これは，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対象となるレコードをデバイスメモリに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ものであり，ホストメモリやデバイスメモリに収まらない大規模なデータセットに対しても良好な結果を得ることを確認している．一方で，同様の手法のソート以外のアルゴリズムへの適用例や性能特性，最適化手法などのは明らかではない．

我々は，GPU アクセラレータと不揮発性メモリデバイスを考慮した Out-of-core な Set Intersection 手法 ooc_set_intersection 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，対象となる入力レコード列をデバイスメモリに収まるサイズへチャンクに分割し，GPU 上での Set Intersection 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 Set Intersection 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，CPU 上で実行可能なレコード総数の 16 倍，GPU 上で実行可能なレコード総数の 512 倍となる 2^{36} の int64_t 型の整数値からなるランダムでユニークな 2 つの入力レコード (レコード総数は $2^{37}(137,438,953,472)$) に対し，311,583,199 records/sec で動作し，2 ソケット 72 スレッドで動作させた CPU 版の非同期 I/O による Out-of-core な Set InterSection と比較して 1.03 倍の性能を示すことを確認した．また，これらの結果を踏まえ，GPU アクセラレータと不揮発性メモリを活用したビッグデータ処理カーネルの Out-of-core 化に向けて性能上のボトルネックの調査を行い，高速化への指針を得た．

2. 関連研究

Set Intersection は 2 つの集合 A, B から共通の集合である $A \cap B = \{x : x \in A \wedge x \in B\}$ を求める操作で，データベース管理システムの JOIN 操作や，ウェブ検索のクエリ処理など，ビッグデータ処理のカーネルとして広く用いられている．最も単純な実装としては，A と B をそれぞれ $[first1, last1)$, $[first2, last2)$ の 2 つ入力レコードとし，$A \cap B$ を $result$ を先頭とする出力レコードとすると，

Algorithm 1 Set Intersection

```
1: while first1 ≠ last1 and first2 ≠ last2 do
2:     if *first1 < *first2 then
3:         ++first1;
4:     else if *first1 > *first2 then
5:         ++first2;
6:     else
7:         *result = *first1;
8:         ++result; ++first1; ++first2;
9:     end if
10: end while
```

Algorithm1 のように記述できる．

Set Intersection はこれまで数多くの研究が行われてきた．代表的なアプローチとしては，Merge ソートの Merge ステップのように，2 つの入力レコードを並列にスキャンし Merge する方法が採られる．この種のナイーブなアプローチの場合，2 つの集合の大きさが同程度であるときは良好な性能を達成するもの，そうではない場合はオーバーヘッドが大きいため，分岐命令をできるだけ排除した手法 [3], [4], [10]，ハッシュを活用した手法 [5] や SIMD 処理を活用したもの [7], [8], [12] など多くのホストメモリの DRAM を対象とした In-core な手法が提案されている．

また，近年では，GPU アクセラレータの高い演算性能やメモリバンド幅を活用するべく，多くの GPU 向けの Set Intersection の手法の提案が行われている [1], [2], [6], [14]. とりわけ，MergePath [6] は，CUDA で記述された C++ 向け STL ライブラリである Thrust にも採用されており広く用いられている．

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], [13] ものの，これらの手法は I/O に関してはブロッキングを行うため，アプリケーションの性能向上は限られる．また，我々はこれまで GPU アクセラレータと不揮発性メモリを対象に非同期 I/O による Out-of-core なソートの高速化に取り組み，良好な結果を得てきた [11], [15]．しかしながら，Set Intersection 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. GPU アクセラレータと不揮発性メモリを考慮した Out-of-core な Set Intersection

3.1 概要

提案手法は，我々がこれまで提案してきた GPU アクセラレータと不揮発性メモリを考慮した Out-of-Core ソート [11], [15] を Set Intersection へ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

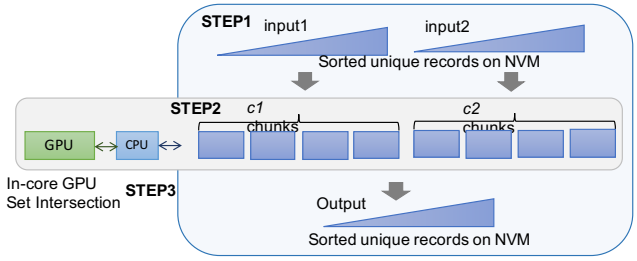

図 1　提案手法 ooc_set_intersection の概要

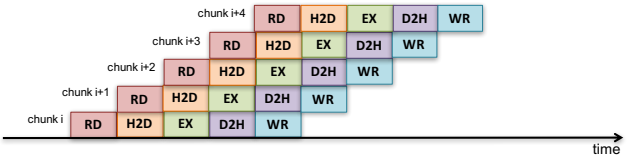

図 2　Out-of-core な Set Intersection のパイプライン実行

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対象となる入力レコードをデバイスメモリに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 間のメモリ間データ転送，GPU 上での Set Intersection の実行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 Set Intersection を行う．

図 1 に提案手法である ooc_set_intersection の概要を示す．大まかな手順としては以下の通りである．

Step1　入力は型 T のソート済みのユニークな 2 つのレコード ($input1$ と $input2$ とする) として，不揮発性メモリがバイト単位でアドレス指定可能 (Byte-addressable) である場合はその領域に，ストレージとしてアクセスする場合はファイルとして置かれる．レコード数はレコードの総サイズを型のサイズ $sizeof(T)$ で割ることによって取得できる．

Step2　入力レコード $input1$, $input2$ に対して，GPU のデバイスメモリの空き容量を考慮してチャンクサイズを定め，チャンクサイズを基に複数のチャンクへ分割する．

Step3　2 つの入力レコード $input1$, $input2$ のチャンク毎に，不揮発性メモリからホストメモリを経由してデバイスメモリへデータ転送を行い，GPU 上で Set Intersection を行う．その後，結果を出力レコードとして不揮発性メモリ上に置く．

3.2　メモリ階層間のデータ移動の遅延隠蔽

提案手法では，3.1 節の Step3 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遅延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 $input1$, $input2$ を各々分割したチャンクに対して，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 Set Intersection 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．

まず，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 Set Intersection 処理をオーバーラップさせるために，CUDAStream 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行うためには，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 をオーバーラップさせるためには，Linux Asynchronous I/O (libaio) による非同期 I/O を用いる手法がある．ただし，Linux Asynchronous I/O で非同期 I/O を行うためにはファイルを O_DIRECT フラグをつけて open する必要があり，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 bytes) 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．

提案手法では，これらの手法を組み合わせることで，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 上での Set Intersection 処理の全てを完全にオーバーラップする．このために，ホストメモリ上にページロックされたメモリ領域 (pinned memory) で，かつ，論理ブロックサイズ (512 bytes) でメモリアライメントされたメモリ領域を確保する必要がある．今，GPU のデバイスメモリの空き容量を gpu_mem_free とし，GPU のデバイスメモリの空き容量に対して 1 つのチャンクが使用するバッファのサイズの割合を $memrate$ とする．ページロックかつメモリアライメントされたバッファ上のチャンク 1 つあたりの最大レコード数 $max\_num\_records\_per\_chunk$ は，GPU のデバイスメモリの空き容量に $memrate$ を掛け，それを型のサイズ $sizeof(T)$ で割り，論理ブロックサイズ $block\_size$ (= 512 bytes) と型のサイズの最大公約数の倍数に丸めることで算出することができる．具体的には次のようにする．

$max\_num\_records\_per\_chunk :=$
$RoundDown(gpu\_mem\_free * memrate / sizeof(T),$
$GCD(sizeof(T), block\_size))$ ただし，$RoundDown(a, b)$ は a を超えない最大の b の倍数を返す．すなわち，$RoundDown(a, b) = a - (a \bmod b)$ である．

3.3　Out-of-core な Set Intersection のパイプライン実行

図 2 に Out-of-core な Set Intersection のパイプライン実行の概略を示す．パイプライン中の各ステージは，不揮発性メモリ上のデータ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 Set Intersection 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へデータの書き込みを行う WR ステージの計 5 段からなる．

いま，2 つの入力レコード $input1$, $input2$ が各々 $c1$, $c2$ 個のチャンクへ分割されており，各チャンクをそれぞれ $chunk1[i1]$ $(1 \leq i1 \leq c1)$, $chunk2[i2]$ $(1 \leq i2 \leq c2)$ と表記する．基本的な方針としては，$chunk1[i1]$ と $chunk2[i2]$ に対して Set Intersection を実行し，その後，処理したチャンクを進めることでパイプライン全体を進める．ただし，正しい順序でチャンクを進めなければ整合性のある結果を得ることができないため，チャンク内のレコードの末尾の値の大小関係を調べることにより，$chunk1[i1]$ と $chunk2[i2]$ のどちらのチャンクを読み進めるかを決定する．具体的には，$chunk1[i1]$ 内のレコードの末尾の値 $chunk1\_last$ と $chunk2[i2]$ 内のレコードの末尾の値 $chunk2\_last$ を比較した結果，$chunk1\_last < chunk2\_last$ である場合，添字 $i1$ を 1 つ増加することで $chunk1[i1]$ を読み進め，$chunk1\_last > chunk2\_last$ である場合，添字 $i2$ を 1 つ増加することで $chunk2[i2]$ を読み進める．$chunk1\_last = chunk2\_last$ である場合，両方の添字 $i1$, $i2$ を各々 1 つ増加することで $chunk1[i1]$ と $chunk2[i2]$ の両方を読み進める．これらのパイプライン実行手法と 3.2 節で述べたメモリ階層間のデータ移動の遅延隠蔽手法を組み合わせることにより，高速な Out-of-core な Set Intersection のパイプライン実行を実現する．

4. 予備実験

提案手法の有効性を明らかにするために ooc_set_intersection について，レコード数に対するスケーラビリティ，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った．比較対象となる実装は以下のものとした．

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 Set Intersection を行う実装．

in-core-cpu(n) ホストメモリ上のレコードに対して，GNU libc++の Parallel Mode 拡張により CPU 上でスレッド数 n で Set Intersection を行う実装．

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 ooc_set_intersection と同様の手法で CPU-GPU 間の非同期データ転送，GPU 上での Set Intersection の非同期実行を行う実装．GPU 上の in-core な Set Intersection には Thrust ライブラリを用いる．

out-of-core-cpu(n)+psync 3 節で述べた提案手法と同様の手法であるが，GPU による Set Intersection や CPU-GPU 間のデータ転送は行わず，GNU libc++の Parallel Mode 拡張により CPU 上でスレッド数 n で Set Intersection を行う実装．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．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(n)+libaio 3 節で述べた提案手法と同様の手法であるが，GPU による Set Intersection や CPU-GPU 間のデータ転送は行わず，GNU libc++の Parallel Mode 拡張により CPU 上でスレッド数 n で Set Intersection を行う実装．ただし，ファイル I/O は Linux Kernel の Asynchronous I/O による非同期 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．

ooc_set_intersection+psync 3 節で述べた提案手法による実装．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．

ooc_set_intersection 3 節で述べた提案手法による実装．

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/sec, 逐次書き込み I/O が 2.8GB/sec, 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 7.5，Thrust は 1.8.2, ファイルシステムは xfs 3.1.1 を用いた．

対象となるデータセットは，int64_t のソート済みのユニークな整数列で，以下のように生成した．

Step1 十分な長さのユニークでランダムな int64_t の整数列からなる初期のレコードを用意する．今回の実験では，128×10^9 records (=1TiB) を用意した．

Step2 selectivity, difference, 及び，入力レコード input1 のサイズを定め，入力レコード input1, input2, 及び，出力レコード output のサイズを決定する．ここで，selectivity は，2 つの入力レコードのうちのレコード数が少ないもののレコード数に対する出力レコード output のレコード数の割合を表し，difference は，2 つの入力レコードのレコード数の差を表す．今回の実験では，difference は全て 0 (すなわち同サイズのレコード) とした．

Step3 初期のレコードから output を共通部分として持つような連続したレコード input1, input2 を切り出す．

Step4 入力レコード input1, input2, 及び，出力レコー

ド output をそれぞれソートし，input1, input2 を Set Intersection に与える入力レコードとする．また，出力レコード output は Set Intersection の実行結果と比較し正当性の検証のためのデータとした．

4.1 Set Intersection のスループット

図 3 に selectivity を 50%としたときの各実装でのレコード数に対する Set Intersection のスループットの結果を示す．x 軸はレコード数 [records] を表し，y 軸はスループット [10^6 records/sec] を表す．ここで，スループットは入力レコードの総数 (すなわち，input1 と input2 のレコード数を足し合わせたもの) を実行時間で割ったものであると定義する．レコードの総数が 2^{28} 以下を対象とした Set Intersection では，GPU 上の In-core な Set Intersection の実装である in-core-gpu が最速でレコード総数が 2^{28} $(268,435,456)$ のとき 594,595,425 records/sec を示したが，今回使用した GPU デバイスのメモリの容量は 12GB 程度なので，2^{28} より大きいレコード数に対しては GPU 上のみで Set Intersection 処理を実行することができない．同様に，2^{28} より大きく 2^{33} 以下のレコード数に対してはホストメモリ上の全レコードを対象とした Out-of-core な GPU の Set Intersection の実装である out-of-core-gpu が最速でレコード総数が 2^{33} $(8,589,934,592)$ のとき 874,786,861 records/sec を示したが，今回の実験環境では 2^{33} 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，それ以上のレコード数に対しては out-of-core-gpu や in-core-cpu の実装で Set Intersection を行うことができない．一方，Out-of-core CPU な Set Intersection の実装はレコード総数が 2^{33} 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総数の 16 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総数の 512 倍である 2^{37} とした場合) においても Set Intersection 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 Linux Kernel の Asynchronous I/O を用いた非同期 I/O と GPU の非同期データ転送を組み合わせた GPU 上の Set Intersection 実装 ooc_set_intersection が最良の結果を示し，2^{37} (137,438,953,472) のレコード総数に対して 311,583,199 records/sec で動作することを確認した．しかし，この結果は CPU を用いた同様の実装を 2 ソケット 72 スレッドで動作させた場合と比較して 1.03 倍の性能でほぼ同等であり，著しく優位な結果は示さなかった．これは，GPU により Set Intersection がアクセラレーションされる一方で，レコードの総数が多いため不揮発性メモリへの I/O が全体性能上のボトルネックになっているためであると考えられる．

次に，入出力レコードのサイズの違いによる Set Intersection のスループットに与える影響を調べるために，selectivity を 0%, 25%, 50%, 75%, 100%と変化させて実験

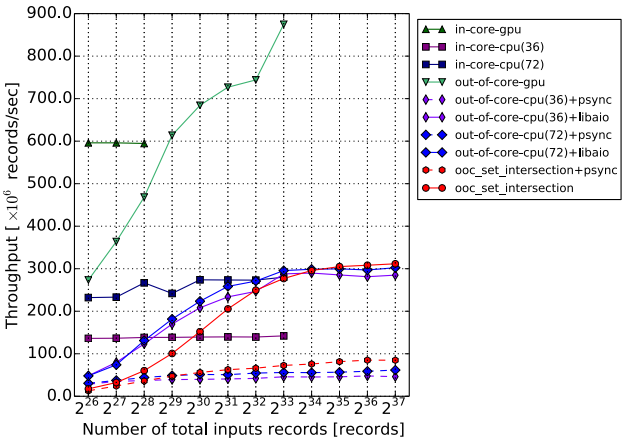

図 3　各実装における Set Intersection のスループット

を行った．selectivity は，2 つの入力レコードのうちのレコード数が少ないもののレコード数に対する出力レコード output のレコード数の割合を表す．レコードの総数を 2^{34} $(17,179,869,184)$ としたときの各実装での Set Intersection スループットの結果を図 4 に示す．selectivity が 0%の場合が最も良い場合で結果を出力せず，selectivity が 100%の場合は最も悪い場合で 2 つの入力レコードのうちのレコード数が少ないものと同一のレコードを結果として出力する．Linux Kernel の Asynchronous I/O による非同期 I/O を用いた手法 (libaio) では selectivity が増加するにつれ，Set Intersection のスループットが単調に低下していくことを確認した．一方で，selectivity が 100%においても著しい性能低下は示さなかった．具体的には，selectivity が 100%の場合においても，提案手法 (ooc_set_intersection) はブロッキング I/O(psync) を用いた手法 (ooc_set_intersection+psync) と比較して 3.87 倍の性能を示した．このことから，少なくとも不揮発性メモリの活用において非同期 I/O による遅延隠蔽が有効であると考えられる．

4.2 メモリ階層間のデータ移動の遅延隠蔽

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，提案手法と Out-of-core な CPU の Set Intersection の実装 (out-of-core-cpu) の実行時間の内訳 [msec] を図 5 に示す．各々，I/O に関して pread, pwrite によるブロッキング I/O を用いたもの (psync) と，Linux Kernel の Asynchronous I/O を用いた非同期 I/O を用いたもの (libaio) 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 Set Intersection，非同期 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．psync による実装では実際に I/O を行っている時間が大部分を占めるのに対し，libaio による実装は I/O が非同

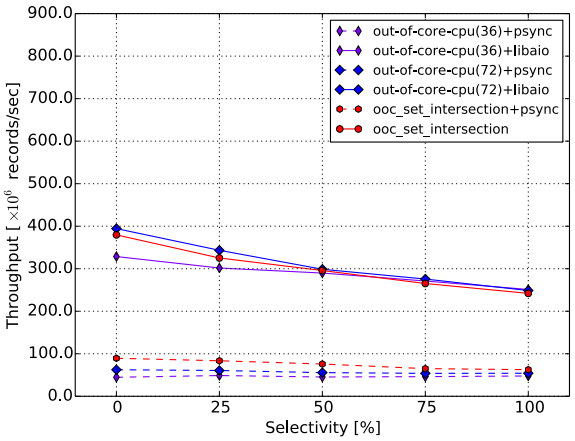

図 4　入出力レコードのサイズ (selectivity) の違いによる Set Intersection のスループット

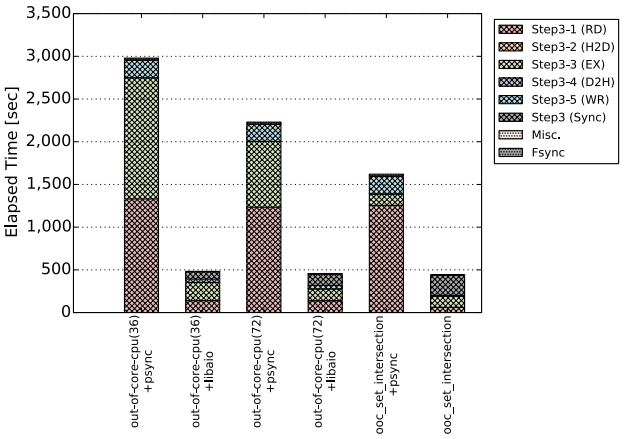

図 5　提案手法 ooc_set_intersection と Out-of-core な CPU の Set Intersection(out-of-core-cpu) の実行時間の内訳

期に行われるのでみかけの実行時間が削減されている一方，Sync の実行時間が増えており，処理のオーバーラップの効果があることが確認できる．CPU(out-of-core-cpu) と GPU(ooc_set_intersection) での Set Intersection の実行時間の違いは EX 及び一部 Sync の項に含まれるが libaio 版においては著しい性能差はなかった．今回の Set Intersection の実装では既存の Thrust ライブラリに含まれる Set Intersection の実装を用いているが，GPU の実装の内部に同期処理が含まれておりこれが律速となり実行時間が顕在化している．また，CPU の Set Intersection 実装 (out-of-core-cpu) において，非同期 I/O 版 (libaio) とブロッキング I/O 版 (psync) の Set Intersection の実行時間は EX の項に含まれるが，psync 版が著しく実行時間が大きくなることを確認した．これは，今回用いた psync 版の実装ではチャンク数の決定の際にメモリのアライメントを行っておらず libaio 版と同数か少ないチャンク数となる場合があり，チャンクサイズの違いやメモリアクセス性能が Set Intersection の実行時間に影響を与えたことが考えられる．従って，理想的な場合においても，libaio 版における EX ステージの実行時間と同等程度になると予測される．

また，提案手法である ooc_set_intersection において，Step3 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 6 に示す．図中の赤いラインはそれぞれのステージでの実行時間の分布の中央値を表す．不揮発性メモリ上の入力レコードの読み込みを行う RD ステージ，及び，出力レコードの書き込みを行う WR ステージに実行時間の大部分が占められており，続いて，GPU 上で Set Intersection を行う EX ステージの実行時間，及び，CPU-GPU 間のデータ転送の実行時間である H2D, D2H ステージに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．実際，EX ステージの実行時間の分布の中央値を 1 としたとき，その他のステージの実行時間の分布の中央値は，RD ステージは 87,983 倍，H2D ステージは 18,722 倍, D2H ステージは 8,957 倍，WR ステージは 91,826 倍となっている．これは，GPU により Set Intersection が十分に高速化されているため，残りのデータ移動の部分が支配的になり律速になっているためである．このため，今回の実験環境の設定では，GPU での EX ステージの実行時間が非常に高速で，CPU-GPU 間のデータ転送を伴う H2D ステージ及び D2H ステージの実行時間がボトルネックになっており，CPU での EX ステージの実行時間が H2D ステージ及び D2H ステージの実行時間より大きくならない限りは，GPU による Set Intersection の高速化は見込めず，さらに，不揮発性メモリを考慮した場合は，CPU，GPU のいずれにせよ EX ステージの性能は，I/O を行う RD ステージ，WR ステージで性能が律速される．このことから，更なる高速な不揮発性メモリデバイスの利用や I/O の最適化だけでなく，Set Intersection において不必要なデータにはアクセスしないなどのアルゴリズムレベルでの改善などが必要であると考えられる．

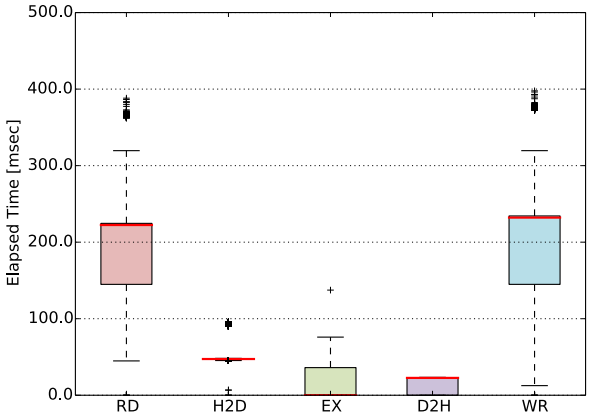

図 6　提案手法 ooc_set_intersection における各パイプライン処理の実行時間の分布

5. まとめと今後の課題

Set Intersection は，ビッグデータアプリケーションにおいて重要な処理の一つである．巨大なデータセットに対する Set Intersection 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 の容量などにより対象となる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 Out-of-core な Set Intersection として ooc_set_intersection 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に対しても，GPU アクセラレータを活用して Set Intersection の処理が可能であることを示した．また，GPU アクセラレータと不揮発性メモリを活用した Set Intersection をはじめとするビッグデータカーネルの Out-of-core 処理に向けて性能上のボトルネックを示し，高速化のための指針を得た．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，及び，正確な性能モデリングを行い，それらを基盤にして，その他のビッグデータカーネルへの適用，不揮発性メモリへのアクセスに親和性のあるアルゴリズムの開発などを行うことが挙げられる．

謝辞　本研究の一部は JST CREST「EBD：次世代の年ヨッタ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」，「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」，及び，JSPS 科研費 26540050 の助成を受けたものである．

参考文献

[1] Amossen, R. R. and Pagh, R.: A New Data Layout for Set Intersection on GPUs, *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011*, pp. 698–708 (2011).

[2] Ao, N., Zhang, F., Wu, D., Stones, D. S., Wang, G., Liu, J. and Lin, S.: Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units, *Proceedings of the VLDB Endowment*, Vol. 4, No. 8, pp. 470–481 (2011).

[3] Bille, P., Pagh, A. and Pagh, R.: Fast Evaluation of Union-Intersection Expressions, *Proceedings of the 18th International Conference on Algorithms and Computation (ISAAC'07)*, pp. 739–750 (2007).

[4] Demaine, E. D., Ldpez-ortiz, A. and Munro, J. I.: Adaptive Set Intersections , Unions , and Differences, *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA'00)*, pp. 743–752 (2000).

[5] Ding, B. and K, A. C.: Fast Set Intersection in Memory, *Proceedings of the VLDB Endowment*, Vol. 4, No. 4, pp. 255–266 (2011).

[6] Green, O., Mccoll, R. and Bader, D. A.: GPU Merge Path: A GPU Merging Algorithm, *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 331–340 (2012).

[7] Inoue, H., Ohara, M. and Taura, K.: Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions, *Proceedings of the VLDB Endowment*, Vol. 8, No. 3, pp. 293–304 (2014).

[8] Lemire, D., Boytsov, L. and Kurz, N.: SIMD Compression and the Intersection of Sorted Integers, *Software - Practice and Experience*, Vol. 46, No. 6, pp. 723–749 (2016).

[9] Matsuo, Y., Shimosawa, T. and Ishikawa, Y.: A file I/O system for many-core based clusters, *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '12*, NeW York, USA, ACM Press, pp. 1 – 8 (2012).

[10] Sanders, P. and Transier, F.: Intersection in Integer Inverted Indices, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pp. 71–83 (2007).

[11] Sato, H., Mizote, R. and Matsuoka, S.: Out-of-core Sorting Acceleration using GPU and Flash NVM, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), Technical Posters*, pp. 78:1–78:2 (2015).

[12] Schlegel, B., Willhalm, T. and Lehner, W.: Fast Sorted-Set Intersection using SIMD Instructions, *Proceedings of the International Workshop on Acceclerating Data Management Systems using Modern Processor and Storage Architectures (ADMS)*, pp. 1–8 (2011).

[13] Silberstein, M., Ford, B., Keidar, I. and Witchel, E.: GPUfs: Integrating a file system with GPUs, *Proceedings of the 18th international conference on Architectural Pupport for Programming Languages and Operating Systems (ASPLOS'13)*, pp. 485–498 (2013).

[14] Wu, D., Zhang, F., Ao, N., Wang, F., Liu, X. and Wang, G.: A Batched GPU Algorithm for Set Intersection, *I-SPAN 2009 - The 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pp. 752–756 (2009).

[15] 佐藤仁，　溝手竜，　松岡聡：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，情報処理学会研究報告 Vol.2015-HPC-150 No.24，pp. 24:1–24:7 (2015).