Vol.2016-HPC-156 No.15
2016/9/16

メモリ使用量を抑えた疎行列疎行列積計算のGPU高速化

長坂 侑亮[1,a)]　額田 彰[1]　松岡 聡[1]

概要：AMG 法など反復解法の前処理において用いられる疎行列疎行列積計算は，ランダムなメモリアクセスによって性能向上が困難であることに加え，出力される行列の非ゼロ要素配置が計算開始時には不明であるという特徴を持つ．GPU での高速化を目的とした既存のアルゴリズムでは，実際に出力行列に必要となるメモリ使用量と比べて多大なメモリを要するため，適用可能な行列が制限されている．適切な場合分けとシェアードメモリの活用によってメモリの使用量を抑えることで広範な行列に対して適用可能であり，かつ更なる高性能化を実現する GPU での疎行列疎行列積計算手法を提案する．様々な特性を持つ 12 個の行列に対して Maxwell 世代 GPU にて性能評価を行い，既存の疎行列計算ライブラリから単精度で最大 4.77 倍，倍精度で最大 3.84 倍の性能向上を達成した．

1. はじめに

シミュレーションなどの数値計算分野やグラフ解析の分野において，多くの要素がゼロ要素で占められている疎行列を用いる計算が行われている．特に，連立一次方程式等の反復解法の前処理として用いられる代数的マルチグリッド（Algebraic Multigrid，AMG）法 [1] や，グラフクラスタリング [2] や幅優先探索 [3] においては，疎行列同士の積演算である疎行列版の行列積計算が行われている．疎行列を扱う際，メモリ使用量や計算量の削減を目的として，非ゼロ要素に関する情報のみを保持するように圧縮を行う．疎行列疎行列積計算においても，入出力疎行列は共に圧縮された形で扱うのが望ましい．しかしながら，出力行列の非ゼロ配置や非ゼロ要素数は入力疎行列の非ゼロ要素のパターンに依存しており，疎行列疎行列積計算を行う前には不明であるため，出力行列に関するメモリ使用の扱いが困難なものとなっている．また，密行列の場合と異なり，疎行列の行列積計算では入力行列要素への間接参照が発生するため，キャッシュミスが頻発しメモリへのアクセスが増加する [4]．出力行列要素に関するメモリアクセスについても規則的ではないため，最終的な出力非ゼロ要素の値を得るには各要素ごとの計算結果を適切にまとめていく必要があり，性能向上を妨げる要因となっている．

グラフィック用途で用いられていた GPU を科学技術計算にも用いることが昨今盛んに行われており，GPU の持つ高い演算性能を活用することによって，これまで多くのアプリケーションや計算カーネルの高速化が成されてきた．疎行列計算についても，GPU への適用は数多く行われており [5–7]，疎行列ベクトル積計算に関しては GPU の持つ高い並列性とメモリバンド幅をより活用した高速化が行われてきた．疎行列疎行列積計算についても GPU への適用は行われている [8] が，多数のスレッドが動作する状態でのメモリ管理は困難であるため，実際に出力行列に必要となるメモリ使用量と比べて多大なメモリを要する．結果として，小さいデバイスメモリ容量しか持たない GPU においては適用可能な行列が制限されるという問題がある．また，性能面に関しても，GPU の持つ種々のメモリと並列性を十分に活用しきれているとは言えず，性能向上の余地が多く残されている．

本研究では，広範な行列データに対する GPU での疎行列疎行列積計算の高速化を目的として，GPU のオンチップシェアードメモリの効果的な活用によりメモリ使用量とメモリアクセス量を抑え，更に，非ゼロ要素数や計算量に応じて行列の各行をグループ分けし，段階的に GPU のリソースの割り当てを行うことによって，既存の疎行列疎行列積計算手法からの更なる高性能化を実現する計算手法を提案する．University of Florida の Sparse Matrix Collection [9] から様々な特性を持つ行列データ 12 個を選出し，既存の疎行列計算ライブラリである cuSPARSE，CUSP，BHSPARSE との比較を行った．Maxwell 世代 GPU にて性能評価を行った結果，単精度で最大 4.77 倍，倍精度で最大 3.84 倍の性能向上を達成した．また，最大メモリ使用量について，実験に用いた全ての行列データに対して，他の疎行列計算ライブラリからの削減に成功しており，単精度において平均

[1] 東京工業大学
Tokyo Institute of Technology
[a)] nagasaka.y.aa@m.titech.ac.jp

© 2016 Information Processing Society of Japan

10.3%，倍精度において平均 14.3%の最大メモリ使用量削減を達成した．

2. 背景

はじめに，疎行列を扱う際にメモリ使用量や計算量の削減を目的として用いられている疎行列格納形式について述べる．次に，逐次での計算処理を踏まえた上での疎行列疎行列積計算について述べ，最後に，GPU での場合を含めた疎行列疎行列積計算処理の問題点を述べる．

2.1 疎行列フォーマット

行列内の要素の多くが処理に必要でないゼロ要素である場合には，ゼロ要素の削除による圧縮を行い，処理に必要となる非ゼロ要素のみを保管する．このような非ゼロ要素の格納方法のことを疎行列フォーマットと呼ぶ．疎行列フォーマットとして広く用いられているものとして Coordinated（COO）と Compressed Sparse Row（CSR）フォーマットがある．COO フォーマットは行列の各非ゼロ要素に関して，値，行インデックス，列インデックスを組として保持する．CSR フォーマットの例を図 1 に示す．CSR フォーマットでは行インデックスを各非ゼロ要素毎に持つかわりに，同じ行インデックスを持つ非ゼロ要素をまとめ，各行の開始インデックス row pointer（配列 rpt）を保持した上で，各非ゼロ要素の列インデックスと値を保持する．CSR フォーマットでは COO フォーマットと比較してメモリ使用量の削減する．

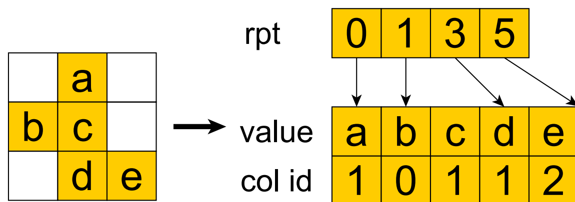


図 1: CSR フォーマット

また，疎行列ベクトル積計算の高速化等を目的として様々な疎行列フォーマットが提案されており [10,11]，加えて，GPU 等のメニーコアプロセッサでの疎行列ベクトル積計算の高速化を実現する疎行列フォーマットも多数提案されている [12,13]．このような疎行列フォーマットにでは，通常 CSR や COO フォーマットで保管されていたものからの変換が必要となる．繰り返し疎行列計算を行う反復解法などでは変換時間を十分に小さいとみなすことが可能であるため，フォーマット変換による疎行列計算の高速化は有用であるが，フォーマットの変換時間に対して疎行列計算の実行時間が小さい場合にはフォーマット変換によってかえって実行時間が増加する点に注意する必要がある．

Algorithm 1 Pseudocode of SpGEMM

```
set matrix C to ∅
for all a_i* in matrix A do
  for all a_ik in row a_i* do
    for all b_kj in row b_k* do
      value ← a_ik b_kj
      if c_ij ∈ c_i* then
        insert (c_ij ← value) to c_i*
      else
        c_ij ← c_ij + value
      end if
    end for
  end for
end for
```

2.2 疎行列疎行列積計算

逐次にて実行される基本的な疎行列疎行列積計算手法について述べる．入力の行列として A, B が与えられた時，疎行列疎行列積計算は $C = AB$ となるような C を出力する．入出力行列ともに疎行列であり，各行列は疎行列フォーマットを用いて保管されている．CSR 形式で格納されている場合には，各非ゼロ要素は行単位で保管されるため，疎行列疎行列積計算は行を単位として計算を行う．CSR 形式での疎行列疎行列積計算を対象とし，行を単位として処理を行う疎行列疎行列積計算の例を記す．Algorithm 1 に疎行列疎行列積計算の擬似コードを示す．a_{ij} は行列 A の i 行目 j 列目の成分，a_{i*} は行列 A の i 行目を表す．行列 A の各行，つまり出力行列 C の各行の計算を順次行う．A の各非ゼロ要素の列インデックスに対応する B の行の各非ゼロ要素を読み，出力行列 C の各非ゼロ要素の計算を行う．

疎行列疎行列積計算は密行列積計算と異なり三つの問題がある．一つは行列要素への間接参照である．Algorithm1 にあるように，行列 B へのアクセスは行列 A の各非ゼロ要素によって決定される．そのため，行列 B の各非ゼロ要素へのメモリアクセスは間接参照となり，多くのキャッシュミスが発生する可能性がある．二つ目の問題は，出力行列 C の非ゼロ配置や非ゼロ要素数が事前に明らかでない点である．そのため，はじめに非ゼロ要素数を数え，後に必要となるメモリを確保し，計算を行う方法を採用する，もしくは，出力行列の保管に求められるメモリ使用量を超える量を確保した上で計算を行う方法を採用する必要がある．前者では特に計算コストの増加，後者ではメモリ使用量が増加するという問題が生じる．三つ目は，出力行列の各非ゼロ要素の挿入や値の足し合わせといった操作に関するものである．出力行列の状態が疎であるため，出力行列の各要素の足し合わせに関する適切な手法の構築が困難であり，性能面でのボトルネックとなっている．

GPU の持つ高い並列性と計算能力を活用した疎行列疎行列計算の高速化も行われているが，GPU においては三つの問題に加えて，並列化を行った際にどのようにスレッド間の負荷均衡を得るかという問題が生じる．疎行列疎行列積

計算では各行毎に大きく計算量やメモリアクセス量が異なる場合があるため，適切にスレッドを配置する必要がある．Bell らによって，疎行列疎行列積計算を GPU に適した計算に分割して行うことで負荷均衡を達成する ESC アルゴリズムが提案された [1]．ESC アルゴリズムは expansion，sorting，contraction の三つの段階から成り立っており，はじめに，行列積計算によって得られるすべての中間積，つまり各非ゼロ要素毎の積のリストを作成する．次に，中間積のリストを列インデックスと行インデックスをもとに並び替えを行い，最後に同じ行インデックスと列インデックスを持つ中間積をまとめることで出力行列の非ゼロ要素の行と列のインデックス，値を得る．ESC の各種法は GPU の高い並列性を活用することが可能であるが，大量の中間積データを扱う必要があるため性能が低い．また，中間積のリストをメモリに保管するため，メモリの使用量が増大するという問題がある．これに対して，Dalton らは GPU のシェアードメモリなどを効率的に活用することによって，ESC アルゴリズムの性能向上を図っている [14]．

3. 提案

疎行列疎行列積計算の GPU 高速化のための手法が提案されてきているが，出力行列の非ゼロ要素数に対して非常に多くのメモリを必要とするため，メモリ容量が小さい GPU では適用可能な行列が制限される．本研究ではシェアードメモリをより効率的に活用したハッシュテーブルを用い，さらに適切なグループ分けを行うことによって，既存手法からの更なる高性能化を達成し，かつメモリ使用量を抑えた疎行列疎行列積計算手法を提案する．

提案手法は大きく二段階で行われる．はじめに出力行列の各行の非ゼロ要素数を求めることで，出力行列全体の非ゼロ要素数を算出する．次に出力行列の格納に必要となるメモリ量のみを確保した後，出力行列の値を計算する．このように二段階で行うことによって，必要とするメモリ使用量を最小限に抑えることが可能となる．また，GPU のリソースを更に効果的に活用するためには，中間積数や非ゼロ要素数によって主要な処理をグループ分けを行った．このような多段階処理による疎行列疎行列積計算手法の概略を図 2 に示す．非ゼロ要素数を求めるための処理である（1）から（4）が第一段階，実際に出力行列の計算を行う（5），（6）が第二段階となる．

以下では，グループ分け，非ゼロ要素数算出，出力行列の計算の三処理について詳細を述べ，最後に Maxwell GPU に対する各グループでの最適なパラメータ設定を記す．なお，A, B を入力行列，C を出力行列として，行う処理は $C = AB$ とし，全ての行列は CSR 形式で格納されているものとする．

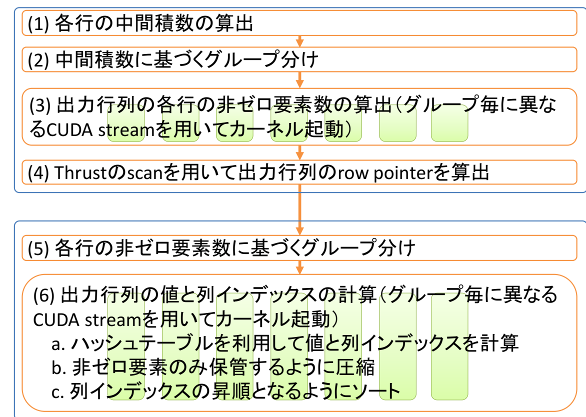


図 2: 提案手法の処理の流れ

3.1 グループ分け

疎行列疎行列積計算では各行の処理量に差があるため，並列化の方法によっては負荷不均衡が生じる可能性がある．負荷不均衡への対策に加え，シェアードメモリでの処理を多くのスレッドで同時実行することを目的として，中間積もしくは出力行列の非ゼロ要素数に応じてグループ分けを行う．図 2 の（2）のグループ分けの段階では各行の非ゼロ要素数が不明であるため，（1）において基準となる各行の中間積数を求めた後，各行を 7 つのグループに分ける．次に（5）のグループ分けでは（3）で算出した非ゼロ要素数を基に各行を 7 つのグループに分ける．グループ分けの際，行列全体での並び替えは処理コストが大きいため，行番号とグループを対応させた配列を作成することによって，グループ分けを行う．本提案手法ではこの行番号を保管しておくための配列が GPU での追加のメモリ使用となる．なお，行毎の非ゼロ要素数の最大が十分に小さい場合には，全ての行が中間積数が少ない行として同一のグループに割り当てられるため，グループ分けの処理を簡略化し，グループ分けのコストを軽減している．

3.1.1 中間積数の算出

グループ分けにおいて基準となる中間積数の算出方法について述べる．各行での中間積数は出力行列の非ゼロ要素数の上限となる．Algorithm 2 に各行の中間積数を求める処理を示す．この処理に必要となるのは A の row pointer と列インデックス，B の row pointer のみであるため，疎行列疎行列積計算全体の処理時間に対して短時間で完了することが可能である．

Algorithm 2 Count the number of intermediate products of i-th row

$prod_num \leftarrow 0$
for $j = rpt_A[i]$ to $rpt_A[i+1]$ **do**
  $prod_num \leftarrow rpt_B[col_A[j]+1] - rpt_B[col_A[j]]$
end for

Algorithm 3 Count the number of non-zero elements of i-th row by 1WARP/ROW

```
tid ← threadIdx%warp
for j ← rpt_A[i] to rpt_A[i+1] stride 32 do
    col ← col_A[j+tid]
    for k ← 0 to 32 do
        dest ← __shfl(col, k)
        for g ← rpt_B[dest]+tid to rpt_B[dest+1] stride 32 do
            //hash operation
        end for
    end for
end for
```

3.2 非ゼロ要素数の算出

入力行列 A の各行に着目し，各非ゼロ要素の列インデックスに対応した B の各行を読み込み，重複を考慮しつつ出力行列の対応する行の非ゼロ要素数を数える．本提案では入力行列へのメモリアクセスを GPU に適した形とした上で，ハッシュテーブルを用いて出現した列インデックス要素の管理を行い，重複を考慮しつつ非ゼロ要素数を数える．中間積数や非ゼロ要素数に応じて必要十分なハッシュテーブルサイズを設定することが GPU リソースの効果的な利用に繋がるため，グループによって異なる設定の CUDA カーネルを用いる．本手法では各カーネルが同時に実行されることによって GPU 内のリソースの利用率を向上させることを目的として，グループ数と等しい CUDA Stream を生成し，各 stream にそれぞれのカーネルを割り当てている．

3.2.1 スレッド配置とメモリアクセスの最適化

入力行列の双方に対するコアレスなメモリアクセスと負荷不均衡の改善を実現するスレッド配置とメモリアクセス方法として二通りの手法，1WARP/ROW と 1TB/ROW を提案する．なお，TB は Thread Block を表す．

1WARP/ROW では A の各行に 1WARP，そして A と B の各非ゼロ要素にスレッドを割り当て，メモリアクセスを行うという．Algorithm 3 に出力行列の非ゼロ要素数算出時のメモリアクセスを示す．同 WARP 内のスレッド間でレジスタの値の受け渡しを行う機能である warp shuffle を用いて，WARP 内のスレッド間で入力行列 A の要素の交換を行うことによって，入力行列 A，B のそれぞれに対してコアレスなメモリアクセスを実現している．なお，1WARP/ROW は 32 スレッドと少ないスレッドで各行の処理を行うため，中間積や非ゼロ要素数が小さいグループに対して用いる．また，各スレッドがそれぞれ数え上げた非ゼロ要素数を warp shuffle を用いて足し合わせることによって各行の非ゼロ要素数を算出している．

1TB/ROW では A の各行に 1 スレッドブロック，A の各非ゼロ要素に 1WARP，B の各非ゼロ要素にスレッドを割り当てる．Algorithm 4 に出力行列の非ゼロ要素数算出時のメモリアクセスを示す．多くのスレッドによって各行の処理を行うため，1TB/ROW は中間積や非ゼロ要素数が大きい行のグループに対して，効果的である．グループ毎に 1WARP/ROW と 1TB/ROW を使い分けることによって，負荷均衡を向上させることが可能となる．各スレッドで数えられた非ゼロ要素数は 1WARP/ROW と同様に WARP 内で warp shuffle を用いて reduction を行ったのち，スレッドブロック内の各 WARP での和をシェアードメモリを用いて reduction を行うことによって各行の非ゼロ要素数を算出する．

Algorithm 4 Count the number of non-zero elements of i-th row by 1TB/ROW

```
tid ← threadIdx%warp
wid ← threadIdx/warp
wnum ← blockDim/warp
for j ← rpt_A[i]+wid to rpt_A[i+1] stride wnum do
    dest ← col_A[j]
    for k ← rpt_B[dest]+tid to rpt_B[dest+1] stride 32 do
        //hash operation
    end for
end for
```

Algorithm 5 Hash Algorithm

```
(Hash table is initialized : table[] ← −1)
key ← col_B
hash ← (col_B * HASH_SCAL)%table_size
nz ← 0
while true do
    if table[hash] = key then
        break
    else if table[hash] = −1 then
        old ← atomicCAS(table + hash, −1, key)
        if old = −1 then
            nz ← nz + 1
            break
        end if
    else
        hash ← (hash + 1)%table_size
    end if
end while
```

3.2.2 ハッシュテーブルの構築

疎行列疎行列積計算において，同じ行インデックスと列インデックスを持つ中間積を一つの非ゼロ要素としてまとめる処理は，アクセスや処理が規則的で無いことから，実行時間を要する処理となっている．疎行列疎行列積計算の高速化においては，この処理を改良することが重要であり，本提案ではハッシュテーブルを用いて高速化の実現を図っている．(3) 出力行列の非ゼロ要素数の算出だけでなく (6) 値の計算においてもハッシュテーブルを用いる．本提案でのハッシュアルゴリズムを Algorithm 5 に示す．ハッシュテーブルには列インデックスを格納することから，-1 を代入することで初期化を行っている．ハッシュ値は列インデックスに定数 $HASH_SCAL$ を掛け，ハッシュテーブルサイズで割った余りとしている．ハッシン

表 1: Maxwell GPU におけるグループ毎のパラメータ設定
Shared memory は倍精度計算時での 1 スレッドブロックあたりの最大シェアードメモリ使用量を表す．

（2）中間積数	（5）非ゼロ要素数	Assignment	Thread Block size	Shared memory	TB / SMM
0-256	0-128	1 WARP / row	512	24KB / TB	4
257-512	129-256	1 WARP / row	256	24KB / TB	4
513-1024	257-512	1TB / row	128	6KB / TB	16
1025-2048	513-1024	1TB / row	256	12KB / TB	8
2049-4096	1025-2048	1TB / row	512	24KB / TB	4
4097-8192	2049-4096	1TB / row	512	48KB / TB	2
8193-	4097-	1TB / row	512	48KB / TB	2, 4

グは linear probing に基づいており，ハッシュが衝突した際には一つ隣のエントリを確認し，値の格納が可能になるまで繰り返す．また，複数のスレッドが同時にハッシュテーブルの同一エントリへアクセスする可能性があるため，compare-and-swap（atomicCAS）を用いて排他的に処理を行う．

ハッシュテーブルのサイズは中間積数や非ゼロ要素数に応じて決定しており，中間積数もしくは非ゼロ要素数が最も大きい行のグループ以外に関してはシェアードメモリに載るサイズとしている．最も中間積数の大きいグループに関しては，二段階で非ゼロ要素数の算出を行う．はじめに一つ小さいグループと同様のパラメータ設定によってシェアードメモリを用いて実行する．この際，テーブルサイズを超える非ゼロ要素数が検出された行に関しては，その行に関する処理を終了し，別途行のインデックスを記録する．このパラメータ設定での処理が全て終了した後，算出が終わっていない行についてはグローバルメモリを用いて十分に大きいハッシュテーブルを構築することで，非ゼロ要素数の算出を行う．中間積数は見積もりの上限値であり，最終的なゼロ要素数に比べて非常に大きくなるケースが多いため，実際には多くの処理を高速なアクセスが可能なシェアードメモリで完結させることが可能になる．この結果，処理速度の優位性に加えて，デバイスメモリの使用量を最小限に抑えることが可能になる．

3.3 出力行列の計算

出力行列 C の計算は大きく三つの処理から構築されている．一つ目の処理では，C の非ゼロ要素数算出と同様にハッシュテーブルを用いて，C の列インデックスの算出と値の計算を行う．基本的には非ゼロ要素数の算出処理と同様であり，追加される処理である各入力行列の値要素へのアクセスに関しては列インデックス要素へのメモリアクセス最適化手法をそのまま用い，ハッシュテーブル上での出力行列の値計算に関しては値用のテーブルを用意した．複数のスレッドが同時に同じ要素への計算を行う可能性があるため，値の計算は atomic 処理を用いて行う．各行に関して計算が終了すると，各ハッシュテーブルは出力行列の非ゼロ要素として計算がなされた要素とエントリがなかった要素で構成された状態となる．二つ目の処理では，エントリがなかった要素を削除し，非ゼロ要素のみとなった状態へ圧縮を行う．最後の三つ目の処理では，非ゼロ要素を列インデックスの昇順となるようにソートする．出力行列の各非ゼロ要素にスレッドを割り当て，同じテーブル内の全非ゼロ要素を参照し，担当する非ゼロ要素が何番目に小さいかを計算する．この結果を元に，グローバルメモリの対応する箇所に計算結果を書き出すことで処理を終える．出力行列の計算に関する三つの処理は各グループごとに一つのカーネルで終えることが出来，また，ハッシュテーブルがシェアードメモリに載る場合には全ての処理をシェアードメモリ上で完結させることが可能となる．

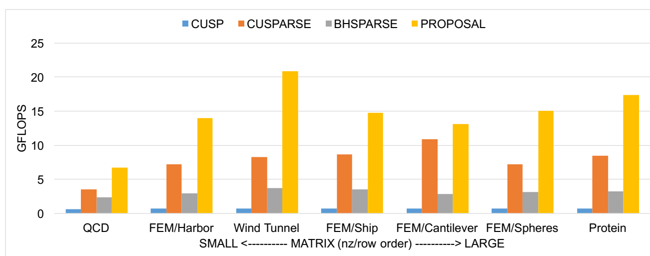
3.4 グループ毎のパラメータ設定

負荷均衡の改善とシェアードメモリの更なる活用を目的として，Maxwell GPU において最適となるようなパラメータの設定を行った．表 1 に（2）と（5）のグループ分けの処理で基準とする中間積数と非ゼロ要素数の範囲と，各グループに対応するパラメータを記す．（2）中間積数や（5）非ゼロ要素数の上限値がそれぞれハッシュテーブルのサイズとなっており，例えば最も中間積数等が小さいグループに関しては（2）ではサイズ 256 の，（5）ではサイズ 128 のハッシュテーブルを構築する．この時，1 スレッドブロックあたりでは 16 のハッシュテーブルが構築されており，倍精度計算時においては 1 スレッドブロックあたり 24KB のシェアードメモリが用いられることになる．Maxwell GPU は各 SMM あたり最大 2048 スレッド割り当て可能であり，96KB のシェアードメモリを有している．シェアードメモリへの atomic 操作が発生するため，非ゼロ要素数等が大きいグループを除いて，処理を効率よく行うために各 SMM には 4 つ以上のスレッドブロックが割り当てられるようパラメータ設定を行っている．なお，最も中間積数が大きいグループに関しては，(3) の非ゼロ要素数算出の処理においてはじめに一つ小さいグループと同様のパラメータ設定で行うため，shared メモリを使用する．

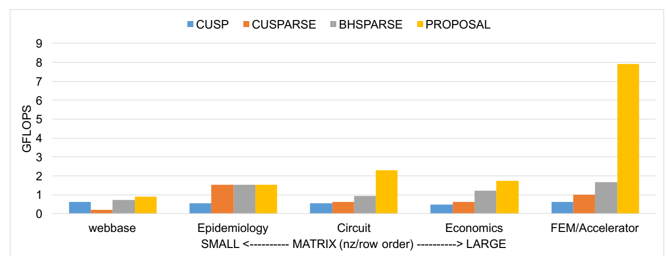
© 2016 Information Processing Society of Japan

表 2: 実験に用いた疎行列

Name	Row	Non-zero	Nnz/row	Max nnz	Intermediate product of A^2	Nnz of A^2
Protein	36,417	4,344,765	119.3	204	555,322,659	19,594,581
FEM/Spheres	83,334	6,010,480	72.1	81	463,845,030	26,539,736
FEM/Cantilever	62,451	4,007,383	64.2	78	269,486,473	17,440,029
Wind Tunnel	217,918	11,634,424	53.4	180	626,054,402	32,772,236
FEM/Harbor	46,835	2,374,001	50.7	145	156,480,259	7,900,917
QCD	49,152	1,916,928	39	39	74,760,192	10,911,744
FEM/Ship	140,874	7,813,404	55.5	102	450,639,288	24,086,412
Economics	206,500	1,273,389	6.2	44	7,556,897	6,704,899
Epidemiology	525,825	2,100,225	4	4	8,391,680	5,245,952
FEM/Accelerator	121,192	2,624,331	21.7	81	79,883,385	18,705,069
Circuit	170,998	958,936	5.6	353	8,676,313	5,222,525
webbase	1,000,005	3,105,536	3.1	4700	69,524,195	51,111,996



(a) High-Throughput Matrices



(b) Low-Throughput Matrices

図 3: 単精度での疎行列疎行列積計算性能

4. 性能評価

提案手法と既存の疎行列疎行列積計算ライブラリとの性能比較を行った．性能比較は行列の自乗計算の Flops 値を元に行っており，Flops 値は中間積数を 2 倍したものを実行時間で割ったものである．実験に用いた行列は University of Florida の Sparse Matrix Collection [9] のものであり，GPU での疎行列計算での評価でしばしば用いられている疎行列データ [5] の中から正方行列である 12 種類を選出した．表 2 に用いた行列データを記す．行列によって行列サイズ，非ゼロ要素数，非ゼロ要素のばらつきなど様々である．行あたりの平均非ゼロ要素数（Nnz/row）を基準として，表 2 に記した行列データセットの内，上位 7 つを High-Throughput Matrices，下位 5 つを Low-Throughput Matrices とする．特徴的な行列として，webbase は Nnz/row に対して，行毎の非ゼロ要素数の最大（Max nnz）が大きく，行ごとの非ゼロ要素数の偏りが大きい行列であるといえる．このような行列に対する GPU での疎行列疎行列積計算はスレッド間の負荷不均衡が発生しやすい．また，中間積数は実際に出力される行列の非ゼロ要素数に対して非常に大きいことから，中間積数に応じてメモリ使用量が増大する手法では，非常に多くの多くのメモリを使用することが予想される．

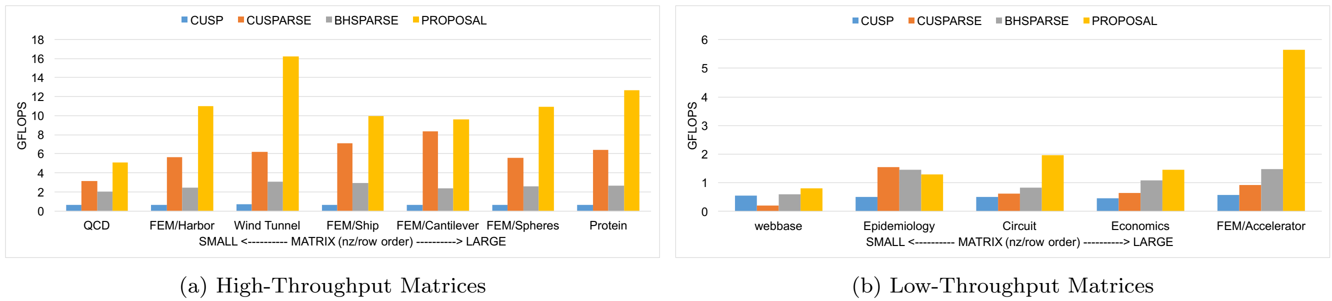
実験には NVIDIA の Quadro M6000 GPU を用いた．GPU は最大で 317GB/sec の帯域幅である容量 12GByte のメモリを持つ．これに加えて，Maxwell 世代の GPU では 24 の SMM で 2MByte の L2 キャッシュを共有しており，各 SMM には同じデータパスである L1 キャッシュと read-only キャッシュを有する．コードは CUDA8.0RC で実装されている．なお，CPU は Intel Core i7-4770@3.40GHz であり，Cent OS Linux 7.0.1406 のもと実行された．

4.1 計算性能

疎行列疎行列積計算の性能比較として，cuSPARSE [19]，CUSP [20]，BHSPARSE [16] という三種類の疎行列計算ライブラリを用いた．BHSPARSE は行あたりの非ゼロ要素数に偏りがあるような行列に対して高速化を図ったライブラリである．

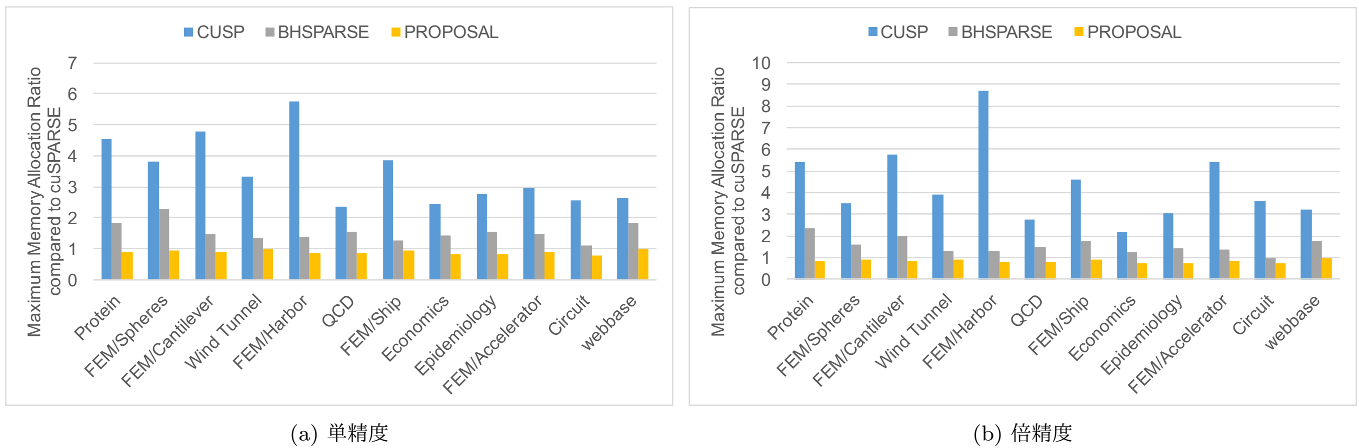
図 3 に単精度での疎行列疎行列積計算性能を示す．CUSP は High-Throughput，Low-Throughput Matrices ともに同様の性能値を示している．CUSP は ESC アルゴリズムを採用しているため，行列のサイズや非ゼロ配置によらず性能が一定となる．次に cuSPARSE と BHSPARSE について．BHSPARSE は Low-Throughput Matrices 内の行列について，cuSPARSE を上回る性能を示している．一方 High-Throughput Matrices では cuSPARSE が良い性能を示している．我々の提案手法は今回用いた疎行列データ全てについて，三種類の疎行列計算ライブラリを大きく上回る性能値を示した．シェアードメモリを効果的に活用した

© 2016 Information Processing Society of Japan



(a) High-Throughput Matrices　　(b) Low-Throughput Matrices

図 4: 倍精度での疎行列疎行列積計算性能



(a) 単精度　　(b) 倍精度

図 5: 疎行列疎行列積計算時における最大メモリ使用量

ハッシュテーブルの構築による High-Throughput Matrices での性能向上に加え，グループ分けと適切なスレッド配置による負荷不均衡の改善によって，Low-Throughput Matrices に対しても高い優位性を示した．我々の提案手法は CUSP，cuSPARSE，BHSPARSE とそれぞれ比較して，最大で 28.9 倍，7.9 倍，5.7 倍の，平均で 14.2 倍，2.8 倍，3.6 倍の性能向上を，最も良いライブラリと比較した場合には最大 4.77 倍の性能向上を達成した．

図 4 に倍精度での疎行列疎行列積計算性能を示す．基本的に単精度での評価と同様の傾向を示しており，既存の疎行列計算ライブラリからの大幅な性能向上を達成した．CUSP，cuSPARSE，BHSPARSE とそれぞれ比較して，最大で 23.9 倍，6.2 倍，5.3 倍の，平均で 11.3 倍，2.4 倍，3.2 倍の性能向上を，最も良いライブラリと比較した場合には最大 3.84 倍の性能向上を達成した．単精度での結果と比較して，倍精度では性能向上率がわずかに低下している．出力行列の計算の際に用いる atomic での加算について，倍精度での atomicAdd がサポートされていないため atomicCAS を用いたことによるものと考えられる．なお，Pascal 世代の GPU では倍精度での atomicAdd がサポートされるため，Pascal 世代 GPU では我々の提案手法の更なる性能向上が期待出来る．

4.2 メモリ使用量

本提案手法では，疎行列疎行列積計算の高速化をメモリの使用量を抑えて実現することを目的としている．各ライブラリと提案手法について，最大メモリ使用量の評価を行った．図 5 に単精度と倍精度における cuSPARSE に対する各手法の最大メモリ使用量の割合を示す．いずれの精度，行列において，我々の提案手法は最大メモリ使用量を最も小さくすることに成功している．提案手法は cuSPARSE ライブラリに対して，単精度では平均 10.3%，倍精度では平均 14.3%の最大メモリ使用量削減を達成した．我々の提案手法を用いることで限られたデバイスメモリ容量しか持たない GPU において，サイズの大きい行列に関する疎行列疎行列積計算を GPU にて高速に行うことが可能であることを示している．特に Low-Throughput Matrices に関しては BHSPARSE が性能的には cuSPARSE に対して優位であったが，二倍近くのメモリを使用している．これに対して，提案手法は Low-Throughput Matrices について BHSPARSE よりも高速にかつ，最大で 57%のメモリ使用量削減を達成した．

5. 関連研究

GPU 向けの疎行列疎行列積計算に関する研究がこれまで多く行われている．

© 2016 Information Processing Society of Japan

Gremse らによって，GPU 内で同時に処理が行われる単位である 32 スレッドの集合 WARP 内で，効率的に中間積の足し合わせを行う手法が提案された [15]．入力行列 A の各行に WARP の一部を割り当て，各スレッドが担当する非ゼロ要素に対応する入力行列 B の行の持つ非ゼロ要素のテーブルを作成する．作成したテーブルについてスレッド間で merge 処理を行っていくことによって，出力行列の対応する行の非ゼロ要素の値と列インデックスを得る．WARP 内で足し合わせの処理を行うことが出来るため高速に行うことが出来る一方で，処理が行えるようにするために行列を分解する必要があり，メモリ使用量や追加の変換コストが生じる．

Liu らによって，グラフデータ等の各行の非ゼロ要素数が不均衡な行列データでの疎行列疎行列積計算の GPU 高速化を図った手法が提案されており [16]，BHSPARSE ライブラリで採用されている．疎行列疎行列積計算は非ゼロ要素数や中間積数によって各行毎の処理時間が大きく異なる．各行の中間積数に応じて bin を作成し，bin 毎に適切な中間積の足し合わせアルゴリズムを採用することでロードバランスの改善を図っている．足し合わせのアルゴリズムとしては heap method，bitonic ESC method，merge method などを採用している．また，CPU での疎行列疎行列積計算性能との比較も行っており，GPU での疎行列疎行列積計算の優位性を示している．本提案でのグループ分けの目的は，スレッドの割り当てによる並列性の改善やハッシュテーブルのサイズを調整することによるシェアードメモリの使用量の調整であるという点で異なる．

Anh らによって，ロードバランスと中間積の足し合わせ処理の双方の改善を図った疎行列疎行列積計算手法 Balanced Hash が提案された [17]．中間積の計算で必要となる要素のインデックスのペアである worklist を作成することで，ロードバランスの改善を図っている．なお，複数の行を中間積数を元にまとめた上で，一つのスレッドブロックを割り当てるという形式をとっている．中間積の足し合わせ処理に関してはハッシュテーブルを用いている．シェアードメモリ上に固定サイズのハッシュテーブルを作成しており，ハッシュが衝突した場合には後で処理するものとして別途キューに保管する．一通り計算が終了した後，ハッシュテーブル内の要素をメモリへ書き出し，別途キューに蓄えられた要素がある場合にはハッシュテーブルをリフレッシュした後，改めて計算を行う．この処理をキューが空になるまで行う．処理の多くをシェアードメモリで行うことが出来るため，高速な疎行列疎行列積計算を実現できている．一方で，worklist はメモリに一度保管するため，多くのメモリ量を必要としており，また，ハッシュが衝突した際に用いるキューに関してもメモリに保管するため，同様に追加のメモリ使用とメモリアクセスが発生する．逆に非ゼロ要素数が小さい場合には必要以上にシェアードメモリを使用することになるため，GPU リソースの使用率が低下する．本手法ではグループごとに異なるサイズを持つハッシュテーブルを構築することによって，非ゼロ要素数が小さい場合にはより多くのスレッドブロックを GPU の各 SM に割り当て，リソースを最大限活用することが可能になり，非ゼロ要素数が大きい場合には多くの処理をシェアードメモリ上で完結させせられるため，高速に処理を行うことが可能となる．

6. 結論

反復解法の前処理として用いられる AMG 法やグラフアルゴリズムにおいて，性能のボトルネックとなっている疎行列疎行列積計算の性能向上を図ることは極めて重要な課題である．また，スーパーコンピュータで広く用いられている GPU などのメニーコアプロセッサを用いた疎行列疎行列積計算の高速化が行われているものの，メモリアクセスやシェアードメモリの活用において高速化の余地が多く残されており，さらに，既存手法では多くのメモリを使用するため，限られたデバイスメモリ容量しか持たない GPU では実行可能な行列が制限されるという問題がある．我々は適切な場合分けとシェアードメモリの活用によってメモリの使用量を抑えることで広範な行列に対して適用可能であり，かつ既存の疎行列計算ライブラリから更なる性能向上を達成する GPU 向け疎行列疎行列積計算手法を構築した．性能評価として，既存の疎行列計算ライブラリである cuSPARSE，CUSP，BHSPARSE から単精度において最大 4.77 倍，倍精度において最大 3.84 倍の性能向上を達成した．また，メモリの使用量については，他の全ての疎行列計算ライブラリからの削減を図ることに成功しており，単精度において平均 10.3%，倍精度において平均 14.3%の最大メモリ使用量削減を達成した．

今後の課題として，今回提案した手法の更なる改良が挙げられる．非ゼロ要素数や中間積数が小さい場合においてはハッシュテーブルを用いた手法ではなく，特化した手法を構築する必要があると考えられる．加えて，我々の提案する疎行列疎行列積計算手法が他のメニーコアプロセッサ，例えば AMD の Radeon GPU においても有用であるかを確認する必要があると考えられる．なお，AMD の GPU については warp shuffle 等の機能が無いため，適宜アルゴリズムの修正が必要である．また，今回はベンチマークとして University of Florida の疎行列を用いたが，AMG 法などで実際に生成される行列などで評価を行うことで，実アプリケーションにおける有用性を確認する必要があると考えられる．

謝辞　本研究の一部は科学技術振興機構戦略的創造研究推進事業「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」による．

参考文献

[1] Bell, N., Dalton, S. and Olson, L. N.: Exposing fine-grained parallelism in algebraic multigrid methods, *SIAM Journal on Scientific Computing*, Vol. 34, No. 4, pp. C123–C152 (2012).

[2] Van Dongen, S.: Graph clustering via a discrete uncoupling process, *SIAM Journal on Matrix Analysis and Applications*, Vol. 30, No. 1, pp. 121–141 (2008).

[3] Buluç, A. and Gilbert, J. R.: The Combinatorial BLAS: Design, implementation, and applications, *International Journal of High Performance Computing Applications* (2011).

[4] Sulatycke, P. D. and Ghose, K.: Caching-Efficient Multithreaded Fast Multiplication Of Sparse Matrices, *Proceedings 12the International Parallel Processing Symposium*, IEEE Computer Society, pp. 117–123 (1998).

[5] Bell, N. and Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM, p. 18 (2009).

[6] Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarath, S. and Sadayappan, P.: Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications, *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 781–792 (online), DOI: 10.1109/SC.2014.69 (2014).

[7] Greathouse, J. and Daga, M.: Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format, *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 769–780 (online), DOI: 10.1109/SC.2014.68 (2014).

[8] Matam, K., Indarapu, S. R. K. B. and Kothapalli, K.: Sparse matrix-matrix multiplication on modern architectures, *2012 19th International Conference on High Performance Computing (HiPC)*, IEEE, pp. 1–10 (2012).

[9] Davis, T.: The University of Florida Sparse Matrix Collection.

[10] Nishtala, R., Vuduc, R. W., Demmel, J. W. and Yelick, K. A.: When cache blocking of sparse matrix vector multiply works and why, *Applicable Algebra in Engineering, Communication and Computing*, Vol. 18, No. 3, pp. 297–311 (2007).

[11] Karakasis, V., Goumas, G. and Koziris, N.: Perfomance Models for Blocked Sparse Matrix-Vector Multiplication kernels, *ICPP'09: International Conference on Parallel Processing 2009.*, IEEE, pp. 356–364 (2009).

[12] Yan, S., Li, C., Zhang, Y. and Zhou, H.: yaSpMV: Yet Another SpMV Framework on GPUs, *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14 (2014).

[13] Kreutzer, M., Hager, G., Wellein, G., Fehske, H. and Bishop, A. R.: A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units, *SIAM Journal on Scientific Computing*, Vol. 36, No. 5, pp. C401–C423 (online), DOI: 10.1137/130930352 (2014).

[14] Dalton, S., Olson, L. and Bell, N.: Optimizing sparse matrix—matrix multiplication for the gpu, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 41, No. 4, p. 25 (2015).

[15] Gremse, F., Hofter, A., Schwen, L. O., Kiessling, F. and Naumann, U.: GPU-accelerated sparse matrix-matrix multiplication by iterative row merging, *SIAM Journal on Scientific Computing*, Vol. 37, No. 1, pp. C54–C71 (2015).

[16] Liu, W. and Vinter, B.: An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data, *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 370–381 (online), DOI: 10.1109/IPDPS.2014.47 (2014).

[17] Anh, P. N. Q., Fan, R. and Wen, Y.: Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication, *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, Vol. 1, New York, NY, USA, ACM, pp. 36:1–36:12 (online), DOI: 10.1145/2925426.2926273 (2016).

[18] Bell, N. and Hoberock, J.: Thrust: A Productivity-Oriented Library for CUDA (2011).

[19] NVIDIA: NVIDIA CUDA Sparse Matrix library (cuSPARSE).

[20] Dalton, S., Bell, N., Olson, L. and Garland, M.: Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations Ver.0.5.1 (2014).

© 2016 Information Processing Society of Japan