Vol.2013-HPC-138 No.5
2013/2/21

GPUにおける高速なCRS形式疎行列ベクトル積の実装

椋木 大地[1,a)]　高橋 大介[2,b)]

概要：疎行列ベクトル積（SpMV）は科学技術計算において多用される重要な基本演算である．本稿ではGPUにおける高速なCRS形式SpMVの実装について報告する．GPUとしてNVIDIA社のKeplerアーキテクチャを対象とし，CUDA5.0環境において実装を行った．従来のFermiアーキテクチャまでのGPUを対象に提案されていた実装手法をベースに，Keplerアーキテクチャで新たにサポートされた機能や仕様変更を活用して，最適化を行った．Keplerアーキテクチャの Tesla K20における性能評価では，CUDA5.0に付属のcuSPARSEにおけるCRS形式の倍精度SpMVルーチンに対して，200種類の行列において，平均で約1.86倍，177種類の行列で性能向上を達成した．

1. はじめに

疎行列ベクトル積（SpMV）は科学技術計算において多用される重要な基本演算である．疎行列は通常，メモリを節約するために，ゼロ要素を省いた非ゼロ要素のみのデータ行列と，その要素の位置を格納したインデックス行列に格納される．そのため，疎行列計算は密行列計算と比べてメモリアクセスが複雑となるほか，疎行列の非ゼロ要素出現パターンは幾通りも存在するため，高速なSpMVの実装には様々な工夫が求められる．

本稿では疎行列格納形式にCRS（Compressed Row Storage）形式[*1]を用いたSpMVのGPUにおける最適化手法について報告する．GPUとしてNVIDIA社のKeplerアーキテクチャGPUを対象とし，NVIDIA社のGPGPU開発環境であるCUDAを用いて実装する．CRS形式は疎行列を行方向に走査し，非ゼロ要素を格納するデータ配列と，そのデータの列番号および各行の先頭位置を格納する2つのインデックス行列を用いる（**図 1**）．CRS形式は古くからCPUにおいて使用されていた手法であり，最も広く普及していると考えられる．

GPUにおいては，疎行列格納形式を工夫することで高速化を達成した事例が多く存在する．例えば2008年頃に行われたBellら[1]によるSpMVのCUDA実装では，ELLとCOOを組み合わせたHYB形式を提案し，CRS形式より高い性能が得られるケースがあることを示している．Bellらの論文以降にも，大島ら[2]によるSegmented Scan形式のCUDA向け最適化，Weizhiら[3]によるブロック化したCRS形式，Xiaowenら[4]によるSIC形式，Matamら[5]によるCRSとELLを組み合わせた方式など，CRS形式以外の格納形式による高速な実装手法が提案されている．また，性能を左右するパラメータが多いSpMVにおいては自動チューニングが有効である．GPUにおけるSpMVの自動チューニングに関する研究も数多く行われており，例えばKubotaら[6]は非ゼロ要素率と非ゼロ要素のばらつきに着目し，格納形式を最適な形式に変換する手法を提案している．

$$\begin{bmatrix} 8&9&0&0&4&5\\ 0&7&5&6&2&0\\ 0&6&0&7&0&0\\ 9&0&0&6&0&2\\ 0&0&2&0&0&0\\ 0&0&7&2&8&0 \end{bmatrix}$$

val = [8, 9, 4, 5, 7, 5, 6, 2, 6, 7, 9, 6, 2, 2, 7, 2, 8]

ind = [1, 2, 5, 6, 2, 3, 4, 5, 2, 4, 1, 4, 6, 3, 3, 4, 5]

ptr = [1, 5, 9, 11, 14, 15, 18]

図 1 CRS形式による疎行列格納

しかしCRS形式以外の格納形式を用いる場合，CRS形式を用いていたアプリケーションのGPU化やCPUとの協調計算においては，CRS形式からの格納形式の変換が必要となる場合がある．また，自動チューニングでは，事前に行列の特徴を調べる必要があるなど，繰り返し同じSpMVを実行するケースでなければ有効とは言えない場合がある．したがって，特に数値計算ライブラリなどにおいては，多種多様な行列に対して平均的に高速な性能を示すCRS形式SpMVの実装が求められる．NVIDIA社のGPU向け疎行列計算ライブラリcuSPARSE[7]においてもCRS形式のSpMVルーチンが存在する．

[1] 筑波大学大学院システム情報工学研究科
[2] 筑波大学システム情報系
[a)] mukunoki@hpcs.cs.tsukuba.ac.jp
[b)] daisuke@cs.tsukuba.ac.jp

[*1] NVIDIAのcuSPARSEやGPU関連の論文ではCSR（Compressed Sparse Row）と呼ばれていることが多いが，本稿ではCRSに呼称を統一する．

本稿では，これまでに Fermi アーキテクチャまでの GPU を対象に提案されていた実装手法をベースに，Kepler アーキテクチャで新たにサポートされた機能や仕様変更を活用して最適化を行うことで，高速化を実現した．その結果，Kepler アーキテクチャの Tesla K20 において，CUDA5.0 に付属の cuSPARSE における CRS 形式の倍精度 SpMV ルーチンに対して，200 種類の行列において平均で約 1.86 倍，177 種類の行列で性能向上を達成した．

2. 関連研究

GPU における CRS 形式 SpMV の実装としては，Bell ら [1] の論文において 2 種類の実装方式が提案されている．Bell らは行列の 1 行あたりの計算（すなわち $\boldsymbol{y} = \boldsymbol{Ax}$ におけるベクトル $\boldsymbol{y}$ の 1 要素の計算）を 1 スレッドで行い，列方向にスレッドをマッピングする CRS-scalar 方式と，1 行あたりの計算に複数のスレッドを割り当てる CRS-vector 方式を提案している．CRS-scalar 方式は CPU コードからの変更が最も少ないシンプルな実装手法であるが，複数スレッドによるメモリの連続領域アクセス（コアレスアクセス）が行えないため，GPU に適した実装であるとは言い難い．これに対して CRS-vector 方式はコアレスアクセスを可能とする．Bell らは 1 行あたり 32 スレッドを割り当てている．

CRS-vector 方式では，行あたりの非ゼロ要素数が 32 より少ない場合には，1 行あたりの計算スレッド数が少ない方が効率が高いことがある．Baskaran ら [8] は 1 行を計算するスレッド数を 32 から 16 とした実装を行っているほか，Guo ら [9] も行列の特性によって 1 行を計算するスレッド数を 16 と 32 に切り替えることが有効であることを示唆している．また，ElZein ら [10] も，行列の行あたりの平均非ゼロ要素数を指標として用いることで，CRS-vector と CRS-scalar の切り替えを行っている．さらに，Reguly ら [11] は，CRS-vector 方式における 1 行を計算するスレッド数を，行あたりの非ゼロ要素数の平均値によって 1, 2, 4, 8, 16, 32 に切り替える手法が有効であることを示している．ここで 1 行を計算するスレッド数が 1 である場合は CRS-scalar に相当すると言える．本稿ではこの Reguly らの実装をベースとする．また，類似のアイディアとして，Yoshizawa ら [12] は，行あたりの非ゼロ要素数の最大値に着目し，起動スレッド数を 1, 2, 4, 8, 16, 32 の中から選択する自動チューニング手法を提案している．

3. Kepler アーキテクチャ GPU

Kepler アーキテクチャは NVIDIA が 2012 年に発表した GPU アーキテクチャである．本稿ではこの Kepler アーキテクチャを採用する Tesla K20 をターゲットに実装を行った．

Kepler アーキテクチャについては NVIDIA が公開しているホワイトペーパー [13] に概要がまとめられている．一世代前の Fermi アーキテクチャからの大きな変更点として，Fermi アーキテクチャで SM と呼ばれていたストリーミングマルチプロセッサが SMX として大きく更新された点が挙げられる．Fermi アーキテクチャの SM では 32 個の CUDA コアが搭載されていたが，SMX ではその数が 192 個に増加した．これに伴いマルチプロセッサあたりの最大ワープ数，スレッド数，スレッドブロック数などが増加している．また，Max X Grid Dimension（1 グリッド内に定義できる x 方向のスレッドブロック数）が 65,535 から 2,147,483,647 に増加した．これによって，ベクトル演算におけるインデックス計算において，ループを用いたアドレス計算を行わずに，スレッドとスレッドブロックの ID のみで計算を行えるようになる場合がある．また，32 ビットレジスタ数も 65,536 本に倍増し，Kepler アーキテクチャからは 1 スレッドが扱えるレジスタ数も 63 本から 255 本に増加しているほか，ワープスケジューラが改良され，倍精度命令の実行効率が Fermi と比べて向上している，としている．

一方で，Kepler アーキテクチャで新たにサポートされた機能のなかで，CRS 形式 SpMV の実装に恩恵があると考えられるものとして，シャッフル命令が挙げられる．シャッフル命令はワープ内のスレッド間でデータを共有するための命令である．従来，ワープ内のスレッド間でデータを共有するためには共有メモリを利用する必要があった．シャッフル命令を用いることで，ワープ内の他のスレッドが持つデータにアクセスすることができる．シャッフル命令として，任意のスレッドのデータ参照（`__shfl`），n 個右側シフト（`__shfl_up`），n 個左側シフト（`__shfl_down`），バタフライ（XOR）交換（`__shfl_xor`）の 4 つの命令がサポートされている．

また，従来から存在したテクスチャキャッシュを改良した，リードオンリーデータキャッシュが利用可能となった．リードオンリーデータキャッシュはカーネル実行中に値が変わらないデータに対してのみ適用される 48KB のキャッシュである．Fermi アーキテクチャまでの場合，グローバルメモリのデータをテクスチャとしてマッピングすることで，テクスチャキャッシュを利用することが可能であった．Kepler アーキテクチャからは容量が大幅に増え，さらにグローバルメモリからのロード時には”`const __restrict__`” 修飾子を追加するだけで，コンパイラが自動で管理を行うようになった．

4. 実装

本章では Kepler アーキテクチャにおける CRS 形式 SpMV の実装について説明する．後に性能比較対象として cuSPARSE を取り上げるため，cuSPARSE と同じ $\boldsymbol{y} = \alpha\boldsymbol{Ax} + \beta\boldsymbol{y}$ の計算を行う．また，演算精度は倍精度と

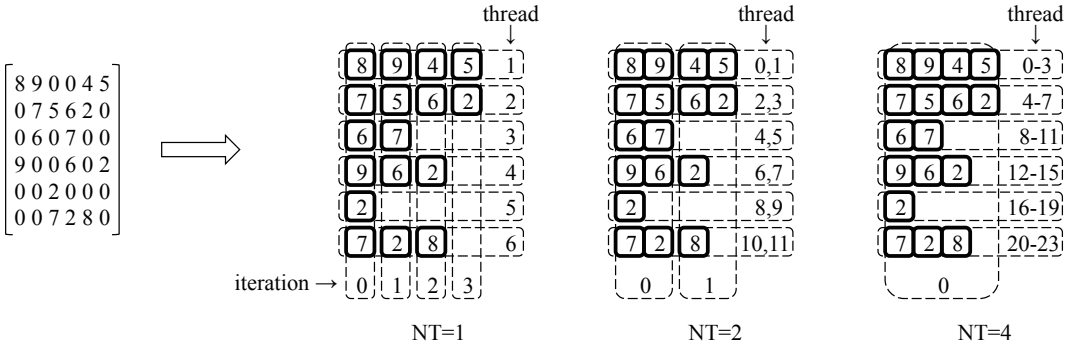

図 2 CRS-vector 方式におけるスレッド割り当て（図 1 の疎行列に対して）

```
__global__ void SpmvKernel_NT
   (int m, double alpha, double* a_val, int* a_ptr,
    int* a_idx, const double* __restrict__ x,
    double beta, double* y)
{
   unsigned int t;
   unsigned int tx = threadIdx.x;
   unsigned int tid = blockDim.x * blockIdx.x + tx;
   unsigned int rowid = tid / NT;
   unsigned int lane  = tid % NT;
   double val;
   int val_hi, val_lo;
   if (rowid < m) {
      val = 0.0;
      for (i = a_ptr[rowid] + lane;
                    i < a_ptr[rowid + 1]; i += NT) {
         val += a_val[i] * x[a_idx[i]];
      }
      for (i = NT / 2; i > 0; i = i >> 1) {
         val_hi = __double2hiint(val);
         val_lo = __double2loint(val);
         val += __hiloint2double(
                       __shfl_xor(val_hi, i, 32),
                       __shfl_xor(val_lo, i, 32));
      }
      if (lane == 0) {
         y[rowid] = alpha * val + beta * y[rowid];
      }
   }
}
```

図 3 カーネルコード（NT には 1, 2, 4, 8, 16, 32 のいずれかの数が入り，2 つ目の for 文をループアンローリングしている）

```
int Spmv
   (char trans, int m, int n, double alpha,
    double* a_val, int* a_ptr, int* a_idx, double* x,
    double beta, double* y, int nonzeros)
{
 int NT, ntx, nbx;
 float nnzrow = (float)nonzeros/(float)m;
 NT = max(1, min(32, (int)pow(2.,ceil(log2(nnzrow)))));
 ntx = NTX;
 nbx = m / (ntx / NT) + ((m % (ntx / NT)) != 0);
 dim3 threads (ntx);
 dim3 grid (nbx);
 if (trans == 'N') {
    if (NT == 32) {
       cudaFuncSetCacheConfig
           (SpmvKernel_32, cudaFuncCachePreferL1);
       SpmvKernel_32 <<< grid, threads >>>
           (m, alpha, a_val, a_ptr, a_idx, x, beta, y);
    }
    else if (NT == 16) {
    ・・・
    else if (NT == 2) {
    ・・・
    else {
    ・・・
}
```

図 4 ホストコードの一部

する．

本稿では，Reguly らの提案する，CRS-vector 方式における 1 行を計算するスレッド数（NT）を，行あたりの非ゼロ要素数の平均値によって NT=1, 2, 4, 8, 16, 32 の中から切り替える手法を用いる．この手法は cuSPARSE の SpMV ルーチンと比較すると非ゼロ要素数を与える引数が 1 つ増加するが，非ゼロ要素数は疎行列を CRS 形式で格納した際に明らかなパラメータであるから，SpMV ルーチンをコールする前に行列を改めて走査する必要はない．

CRS-vector 方式では，複数スレッドを用いて行方向に内積を計算する．**図 2** に，NT=1, 2, 4 とした場合のスレッドマッピングの概念図を示す．“iteration”は行方向のループである．また，最後には 1 行を計算する複数スレッド内で総和を計算する．NT は NT = max(1, min(32, (int)pow(2,ceil($\log_2$(nnzrow)))) で与える．このとき，ワープ内の総和計算であれば同期が不要となるため，NT は最大で 32 としている．

本稿ではさらに Kepler アーキテクチャ向けに最適化を行った．具体的には，(1) リードオンリーデータキャッシュの使用，(2) 最外側ループの削除，(3) シャッフル命令の使用である．**図 3** に本研究で実装した SpMV の GPU カーネルコード，**図 4** にカーネルを呼び出すホストコードの一部を示す．なお，実際には総和計算部分（図 4 における 2 つ目の for 文）において，反復回数が計算スレッド数 NT によって決定するため，ループアンローリングを行ってい

る．以降では，Kepler アーキテクチャ向けに行った 3 つの最適化手法について説明する．

4.1 リードオンリーデータキャッシュの使用

Kepler アーキテクチャからサポートされた 48KB のリードオンリーデータキャッシュを利用する．GPU カーネル関数においてリードオンリーとなる引数に `const` および`__restrict__`を加える．これによりコンパイラがリードオンリーデータキャッシュの利用を自動的に管理するようになる．このリードオンリーデータキャッシュは L1 キャッシュとは独立したパスでアクセスするため，L1 キャッシュの負荷およびキャッシュ容量を節約できる．今回 SpMV において，データの再利用が行われ，キャッシュの効果が高いと考えられるベクトル $\boldsymbol{x}$ を，リードオンリーデータキャッシュ適用の対象とした．

4.2 最外側ループの削除

Kepler アーキテクチャの GPU では，x 次元方向のグリッドサイズの最大値 MaxGridDimX（x 次元方向に定義できるスレッドブロック数）が，従来の最大 65,535 から 2,147,483,647 へと拡大された．これによって，ベクトルのインデックスをスレッド ID から計算する場合に，インデックスとアドレスを一対一で対応することで，アドレス計算が不要となる場合がある．

CRS-vector 方式の実装において，計算可能な疎行列の行数の最大値 RowMax は，RowMax=MaxGridDimX×BlockDim.x/NT で求められる．BlockDim.x はスレッドブロックにおける x 方向のスレッド数であり，本稿における実装では BlockDim.x=128 が最適であった．また RowMax は NT=32 のときに最小となる．したがって，Fermi アーキテクチャの GPU では RowMax=65,535×128/32=262,140 であり，ベクトル長が 262,140 を越える計算を行うためには，スレッド ID からアドレスを再計算し処理をループさせる必要があった．また，総和計算で共有メモリを使用する場合にスレッド同期を必要とした[*2]．一方，Kepler アーキテクチャの GPU では，計算可能な要素数は RowMax=2,147,483,647×128/32=8,589,934,588 となる．これは単精度のベクトル 1 本であっても 32GB 分に相当し，現行 GPU のメモリサイズが高々数 GB 程度であることを考えると，SpMV ルーチンにおいてサポートする計算可能な次元数としては十分なサイズであると言える．したがって，アドレスの再計算を伴う最外側ループを削除することが可能となる．

[*2] 共有メモリを volatile で宣言することで同期を使用せず実装できるが，我々の実験では同期を使用した実装と比べて，性能低下が認められた．

4.3 シャッフル命令の使用

CRS-vector 方式の実装では，1 行あたりの計算スレッド数 NT が 2–32 の場合に，ワープ内の複数スレッドによる総和計算が発生する．この際に，従来は共有メモリを用いてワープ内で総和を計算していた．Kepler からサポートされたシャッフル命令を利用することで，共有メモリを用いることなく実装することができる．今回，バタフライ（XOR）交換（`__shfl_xor`）を使用して総和を計算した．シャッフル命令は 32bit データの移動のみをサポートしており，64bit データの移動は行えない．本稿では倍精度演算を行うカーネルにおいて，double 型を 2 つの int 型に変換した上で，シャッフル命令を 2 回発行し，2 つの int 型を再度 double 型へ変換している．シャッフル命令ではストアからロードまでが 1 ステップで実行されるため，ロード・ストアをそれぞれ行う必要があった共有メモリを用いた実装と比べ，実行性能の向上が期待できる．

5. 性能評価

5.1 方法

評価環境には，GPU として Kepler アーキテクチャの NVIDIA Tesla K20 を用いた．ホストの CPU は Intel Xeon E3-1230 3.20GHz, OS は CentOS 6.3 (kernel: 2.6.32-279.14.1.el6.x86_64）であり，CUDA5.0 (Driver Version: 304.54)，コンパイルは nvcc 5.0（-O3 -arch sm_35）および gcc 4.4.6 (-O3）で行った．なお”-arch sm_35”は Kepler アーキテクチャでサポートされた機能を利用するために必要なコンパイルオプションである．

性能は GPU カーネル関数のみの実行時間をもとに計算した Flops 値で評価する．CPU ホスト側との PCI-Express による通信時間は実行時間に含めていない．正確に測定するために GPU カーネル関数を最低 3 回以上，かつ実行時間が 1 秒以上となるように繰り返し実行し，実行時間の平均から性能を求めた．性能比較対象として，CUDA5.0 に付属する NVIDIA 社が提供する疎行列計算ライブラリ cuSPARSE の性能も測定する．

性能評価に用いる疎行列は，The University of Florida Sparse Matrix Collection[14] の数ある行列の中から 200 種類を選んだ．ただし，行列はすべて実数の正方行列であり，選択にあたっては行列の一辺の長さと非ゼロ要素数のどちらかが異なるように選択した．行列の一辺の長さは 1,813–5,558,326 で，非ゼロ要素数は 4,257–117,406,044 である．疎行列の性質については 6 章において，性能とともに議論する際に示す．また，$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$ を計算する際に疎行列 $\boldsymbol{A}$ 以外の値はすべて乱数で初期化している．

5.2 結果

NVIDIA 社が提供する疎行列計算ライブラリである cuSPARSE（CUDA5.0 付属）と，本稿における実装の性能を

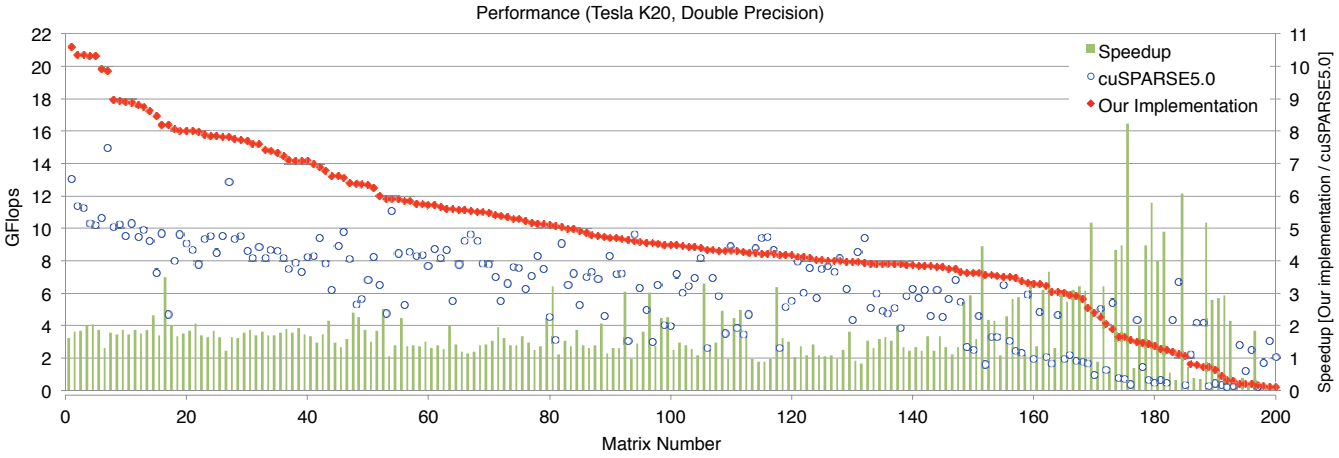

図 5 cuSPARSE5.0 と比較した性能

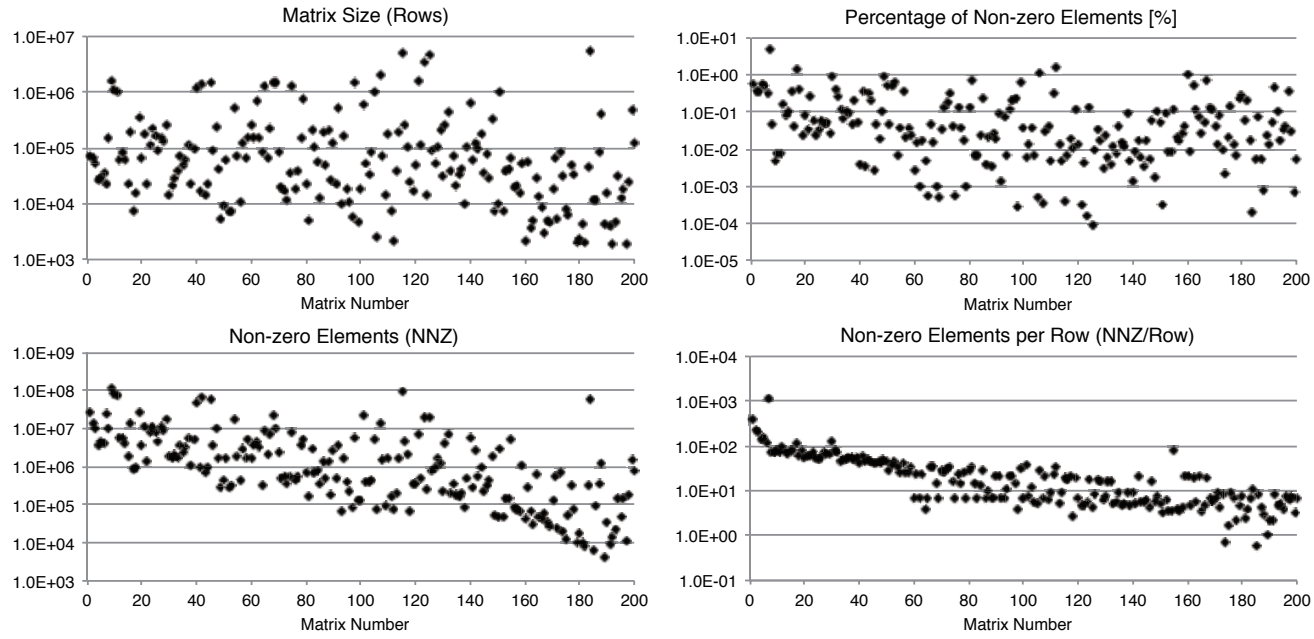

図 6 実験に用いた疎行列の特性

比較した．**図 5** に結果を示す．cuSPARSE および我々の実装の Flops 値（左軸）とともに，cuSPARSE に対する相対性能（右軸）を示す．データは本稿の実装の Flops 値が高いものから順にソートしている．横軸の Matrix Number は 200 種類の行列の種類を示しており，図 5 の左から順に番号を付与している．以降，本稿に掲載する図の横軸は，この図 5 の横軸に対応する．

cuSPARSE に対して，200 種類の行列において平均で約 1.86 倍，177 種類の行列で性能向上が得られた．また，最大で約 8.21 倍高速な性能を示した．特に実行性能が低い図右方のケースにおいて，cuSPARSE に対する性能向上が大きいことが分かる．一方で，23 種類の行列では性能が cuSPARSE より低く，最も性能が低下したものでは，cuSPARSE の約 0.08 倍となった．

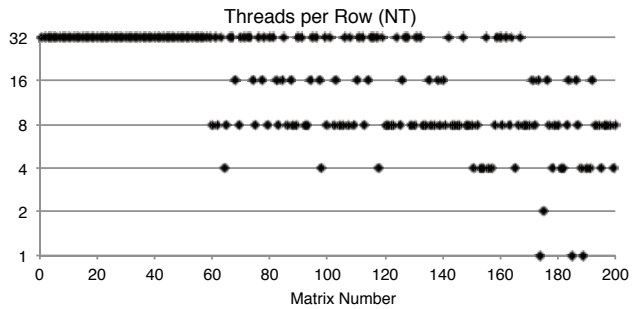

図 7 1 行あたりの計算スレッド数（NT）

6. 考察

本章では行列の特性と性能の関係および各種最適化手法の効果について議論する．**図 6** に 200 種類の疎行列の行数（Rows），非ゼロ要素数（NNZ），非ゼロ要素率および一行あたりの非ゼロ要素数（NNZ/Row）をプロットしたものを示

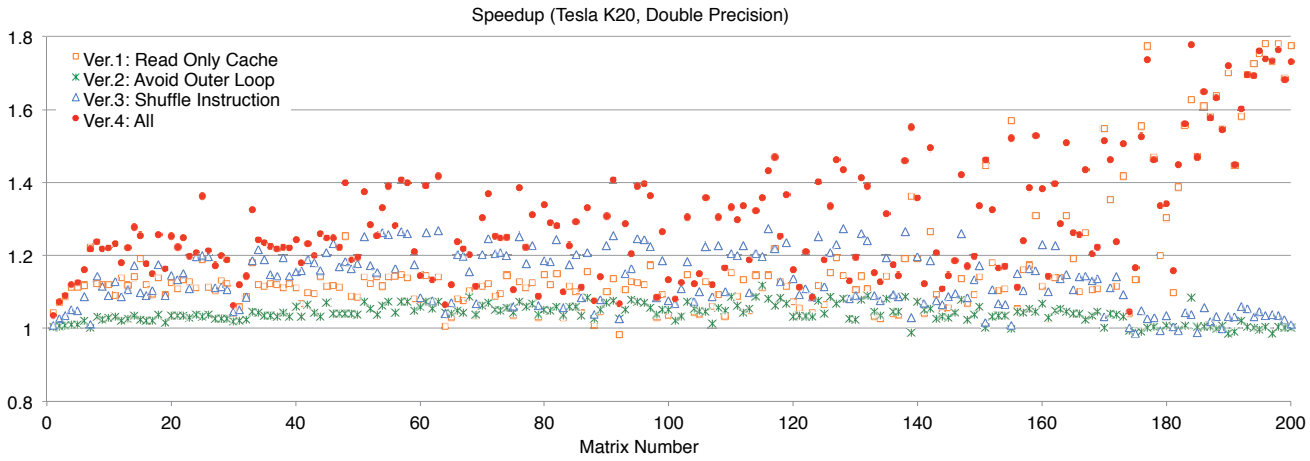

図 8 各種実装手法の効果（Ver. 0 に対する相対性能）

す．横軸の Matrix Number は図 5 と対応する．これらの中で Flops 値と相関があるものとして NNZ/Row があり，NNZ/Row が大きいほど性能が高いと言える．**図 7** に 1 行あたりの計算スレッド数 NT を示す．NT は NNZ/Row に応じて決定しているため，NNZ/Row と NT は関連する．NT が大きいほどコアレスアクセスとなり，メモリアクセス効率が高くなるため，高い性能が得られていると推測できる．一方で，cuSPARSE に対する相対性能については，cuSPARSE のソースコードが公開されていないため，詳細な検討は行えない．しかし NNZ や NNZ/Row が小さい行列において，高い性能向上を示しているものがあるが，逆に大きく性能が低下しているケースがある．これは行によって非ゼロ要素数がばらつくようなケースにおいて，最適な NT の選択を誤っている可能性が考えられる．

次に，本研究で用いた Kepler アーキテクチャ向けの各種実装手法の効果を確認するために，以下の 5 つの実装を作成して性能を比較した．

- Ver. 0：Ver. 1–3 のいずれも適用していない Fermi 向け最適化
- Ver. 1：リードオンリーデータキャッシュを使用
- Ver. 2：最外側ループの削除
- Ver. 3：シャッフル命令の使用
- Ver. 4：Ver. 1–3 のすべてを適用した Kepler 向け最適化

Ver. 0 は比較対象となる原型で，Fermi アーキテクチャ向けの最適化を行った実装である．Ver. 1–3 は Ver. 0 に対して Kepler アーキテクチャから利用可能となった手法を各々一つずつ適用した状態である．なお Ver. 1 以外では，ベクトル x の読み込み時において，グローバルメモリのデータをテクスチャとしてマッピングする Fermi アーキテクチャまでに用いられていた手法でテクスチャキャッシュを利用している．Ver. 1 はその代わりにリードオンリーデータキャッシュを用いたものである．Ver. 4 は図 5 において性能を示した Kepler 向け実装であり，図 3 および図 4 に示した実装と同一である．

図 8 に 200 種類の疎行列に対する Ver. 0 を基準とした Ver. 1–4 の性能を示す．最終的に Kepler アーキテクチャ向けに最適化を行った Ver. 4 では，Fermi アーキテクチャ向け実装の Ver. 0 と比較して，200 種類の行列の平均で約 1.29 倍（最大約 1.78 倍，最低約 1.04 倍）の性能向上が得られた．いずれの手法においても，わずかな性能低下（最大で Ver. 1 における約 0.98 倍）が確認されたケースがあるものの，ほぼすべての行列で性能が向上しており，有効性が確認できた．また，行列番号が 170–200 番付近の行列では，リードオンリーデータキャッシュの効果が高い一方で，他の手法の効果は小さい．これらの行列は図 6 において 1 行あたりの非ゼロ要素数が小さい行列であることが分かる．そのためキャッシュ適合性が高く，リードオンリーデータキャッシュが有効に働いたと考えられる．一方で NT が小さく総和計算部分の反復が少ないため，総和計算部分のボトルネックが元々小さく，シャッフル命令を用いることによる効果が小さかったと考えられる．

また，いずれの手法においても，従来のプログラムより記述をシンプルにすることができた．特にリードオンリーデータキャッシュは，従来のテクスチャキャッシュを利用するための複雑な記述を行うことなく利用でき，さらに性能も向上しているため，有益であると考えられる．結論として，これらの 3 つの手法は CRS 形式 SpMV の性能向上に寄与したと同時に，ベースとなる Fermi アーキテクチャ向けの実装と比べて容易な記述で高性能を達成できたと言える．

7. まとめ

本稿では Kepler アーキテクチャの GPU を対象に，高速な CRS 形式 SpMV の実装を行った．Fermi アーキテクチャまでの GPU を対象に提案されていた実装手法をベースに，Kepler アーキテクチャで新たにサポートされたリードオンリーデータキャッシュ，シャッフル命令，x 次元方

向のグリッドサイズの最大値の拡張による最外側ループの削除を行うことで，高速な実装を実現した．その結果，Kepler アーキテクチャの Tesla K20 における性能評価では，CUDA5.0 に付属の cuSPARSE における CRS 形式の SpMV に対して，倍精度演算で 200 種類の行列において平均で約 1.86 倍，177 種類の行列で性能向上が得られた．本稿で用いた Kepler アーキテクチャ向けの実装手法は，高速な行列演算プログラムをより容易に実装するために有効であったと言える．本稿で示した各種の実装手法は他の GPU プログラムにおいても適用できると考えられる．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」による．

参考文献

[1] Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA, *NVIDIA Technical Report*, No. NVR-2008-004 (2008).
[2] 大島聡史, 櫻井隆雄, 片桐孝洋, 中島研吾, 黒田久泰, 直野健, 猪貝光祥, 伊藤祥司：Segmented Scan 法の CUDA 向け最適化実装, 情報処理学会研究報告, Vol. 2010-HPC-126, No. 1, pp. 1–7 (2010).
[3] Xu, W., Zhang, H., Jiao, S., Wang, D., Song, F. and Liu, Z.: Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU, *Proc. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2012)*, pp. 231–235 (2012).
[4] Feng, X., Jin, H., Zheng, R., Hu, K., Zeng, J. and Shao, Z.: Optimization of Sparse Matrix-Vector Multiplication with Variant CSR on GPUs, *Proc. IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS 2011)*, pp. 165–172 (2011).
[5] Matam, K. and Kothapalli, K.: Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU, *Proc. International Conference on Parallel Processing (ICPP 2011)*, pp. 612–621 (2011).
[6] Kubota, Y. and Takahashi, D.: Optimization of Sparse Matrix-Vector Multiplication by Auto Selecting Storage Schemes on GPU, *Proc. 11th International Conference on Computational Science and Its Applications (ICCSA 2011)*, Part II, Lecture Notes in Computer Science, No. 6783, pp. 547–561 (2011).
[7] NVIDIA Corporation: cuSPARSE Library (included in CUDA Toolkit), https://developer.nvidia.com/cusparse.
[8] Baskaran, M. M. and Bordawekar, R.: Optimizing Sparse Matrix-Vector Multiplication on GPUs, *IBM Research Report*, Vol. RC24704 (2009).
[9] Guo, P. and Wang, L.: Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs, *Proc. International Conference on Computational and Information Sciences (ICCIS 2010)*, pp. 1154–1157 (2010).
[10] El Zein, A. H. and Rendell, A. P.: Generating Optimal CUDA Sparse Matrix Vector Product Implementations for Evolving GPU Hardware, *Concurrency and Computation: Practice and Experience*, Vol. 24, pp. 3–13 (2012).
[11] Reguly, I. and Giles, M.: Efficient sparse matrix-vector multiplication on cache-based GPUs, *Proc. Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (InPar 2012)*, pp. 1–12 (2012).
[12] Yoshizawa, H. and Takahashi, D.: Automatic Tuning of Sparse Matrix-Vector Multiplication for CRS format on GPUs, *Proc. 15th IEEE International Conference on Computational Science and Engineering (CSE 2012)*, pp. 130–136 (2012).
[13] Corporation, N.: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (2012).
[14] Davis, T. and Hu, Y.: The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.