Vol.2012-ARC-202 No.37
Vol.2012-HPC-137 No.37
2012/12/14

GPUにおける4倍精度演算を用いた疎行列反復解法の実装と評価

椋木 大地[1,a)]　高橋 大介[2,b)]

概要：疎行列の反復解法として用いられるクリロフ部分空間法は，丸め誤差の影響によって収束までの反復回数が増加したり，収束しなくなるケースがある．このような場合に高精度演算を用いることで収束性を改善できるケースがあることが報告されている．このとき，高精度演算を行うことによる 1 反復あたりの計算時間の増大に対して，反復回数の削減による計算時間の短縮効果が大きければ，求解までの計算時間を短縮できる可能性がある．我々は GPU（Tesla M2050）において Double-Double（DD）演算による 4 倍精度を用いて，クリロフ部分空間法の一つである BiCGStab 法を実装し性能を評価した．GPU 上では 4 倍精度 BiCGStab 法の 1 反復あたりの計算時間が，倍精度の約 1.0–2.2 倍となり，反復回数の削減量によっては，4 倍精度演算を用いることで求解までの計算時間を短縮できる場合が存在した．本稿では GPU 上の疎行列反復解法における 4 倍精度演算の性能と有効性について検討する．

1. はじめに

浮動小数点演算の精度は有限であり，悪条件の問題などで倍精度では計算できない問題や，高い精度の解を得る目的で，倍精度よりも高精度の四則演算を必要とするケースがある．近年では IEEE754-2008[1] において 128bit の浮動小数点型（binary128）が定義されるなど，計算機の発達により，高精度演算に対する需要が高くなっている．

一方で，反復解法では丸め誤差の影響により，収束までに必要な反復回数が理論的に必要とされる回数より増加することがある．このようなケースでは高精度演算を用いることで，倍精度演算と比較してより少ない反復回数で解が得られるケースが存在する．例えば，疎行列の反復解法として用いられる CG（Conjugate Gradient）法などのクリロフ部分空間法は丸め誤差の影響に敏感であり，収束までの反復回数が増加したり収束しないケースがあり，このような場合に高精度演算を用いることで収束性を改善できる場合があることが報告されている [2]．このとき，高精度演算を行うことによる 1 反復あたりの計算時間の増大に対して，反復回数の削減による計算時間の短縮効果が大きければ，求解までの時間を短縮できる可能性がある．すなわち，高精度演算を計算の高速化に用いることが可能であると考えられる．

我々は GPU において DD（Double-Double）演算を用いた 3 倍・4 倍精度の BLAS（Basic Linear Algebra Subprograms）ルーチンを実装し，その性能を評価してきた [3]．DD 演算は倍精度浮動小数点型を 2 個を連結して 4 倍精度浮動小数点型を表現し，2 桁の筆算の原理 で倍精度浮動小数点演算を使用して 4 倍精度浮動小数点演算を行うため，演算コストの大きな処理である．しかし Byte/Flop 比の小さいハイエンド GPU 上では，Level–2 演算である行列ベクトル積（GEMV）が 4 倍精度演算においてもメモリ律速となることが分かっている．CG 法は疎行列ベクトル積（SpMV）といくつかの Level–1 演算で構成される．ここで SpMV は密行列計算よりもメモリアクセスが複雑となるため，その 4 倍精度演算はメモリ律速となる可能性が高く，1 反復あたりの計算時間は，4 倍精度演算が倍精度演算の高々 2 倍程度で済む可能性がある．したがって，4 倍精度演算の使用によって反復回数が倍精度演算と比べて半減するケースでは，4 倍精度演算を使用することで求解までの時間を短縮できる可能性がある．

本稿では，NVIDIA Tesla M2050 によるシングル GPU 環境において，クリロフ部分空間法の一つである BiCGStab 法の 4 倍精度版を実装し，倍精度演算に比べて 4 倍精度演算で反復回数が削減されるケースにおいて，両者の収束までの計算時間を比較してその性能を考察する．そして，GPU 上の疎行列反復解法における 4 倍精度演算の有効性について検討する．

[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

$\boldsymbol{p_0} = \boldsymbol{\tilde{r}_0} = \boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{Ax_0}$
$\rho_0 = (\boldsymbol{\tilde{r}_0}, \boldsymbol{r_0})$
for : $k = 0, 1, 2, ...$ **do**
  $\boldsymbol{v} = \boldsymbol{Ap_k}$
  $\alpha = \rho_k / (\boldsymbol{\tilde{r}_0}, \boldsymbol{v})$
  $\boldsymbol{s} = \boldsymbol{r_k} - \alpha \boldsymbol{v}$
  $\boldsymbol{t} = \boldsymbol{As}$
  $\omega = (\boldsymbol{t}, \boldsymbol{s}) / (\boldsymbol{t}, \boldsymbol{t})$
  $\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + \alpha \boldsymbol{p_k} + \omega \boldsymbol{s}$
  $\boldsymbol{r_{k+1}} = \boldsymbol{s} - \omega \boldsymbol{t}$
  if $||\boldsymbol{r_{k+1}}|| / ||\boldsymbol{b}|| < \epsilon$ **then**
    break
  end if
  $\rho_{k-1} = \rho_k$
  $\rho_k = (\boldsymbol{\tilde{r}_0}, \boldsymbol{r_{k+1}})$
  $\beta = (\rho_k / \rho_{k-1})(\alpha / \omega)$
  $\boldsymbol{p_{k+1}} = \boldsymbol{r_{k+1}} + \beta(\boldsymbol{p_k} - \omega \boldsymbol{v})$
end for

図 1 BiCGStab 法のアルゴリズム

2. GPU における疎行列反復解法の実装

本稿では疎行列反復解法として非対称行列用のクリロフ部分空間法である BiCGStab 法を実装する．本稿では前処理は実装していない．GPU は NVIDIA 社の Fermi アーキテクチャ GPU を対象とし，同社の GPGPU 開発環境である CUDA を用いた．なお，4 倍精度版とともに，性能比較を行うための倍精度版の実装も行う．

2.1 BiCGStab 法の実装

BiCGStab 法のアルゴリズムを**図 1** に示す．BiCGStab 法において反復のループ内で行われるベクトル演算は，SpMV（$\boldsymbol{y} = \boldsymbol{Ax}$）が 2 回，DOT（$r = (\boldsymbol{x}, \boldsymbol{y})$）が 4 回，AXPY（$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$）が 3 回，AXPYZ（$\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$）が 2 回，XPAY（$\boldsymbol{y} = \boldsymbol{x} + \alpha\boldsymbol{y}$）が 1 回，そして収束判定のために 2 ノルムを求める NRM2 が 1 回である．SpMV 以外はすべてベクトル同士の計算を行う Level-1 演算であるため，計算時間の大半を Level-2 演算である SpMV が占める場合が多い．

今回，これらのベクトル演算をカーネル関数として実装し，スカラ値の計算は CPU 側で行った．収束判定の計算は倍精度で足りるため，倍精度版と 4 倍精度版の両者ともに CUBLAS[4] の DNRM2 関数を用いた．AXPYZ と XPAY 以外のベクトル演算については，NVIDIA 社が提供するライブラリ（CUBLAS，cuSPARSE[5]）において倍精度版実装が提供されている．しかし，本稿では倍精度版についても独自の実装を行い，その倍精度版をベースに 4 倍精度版を実装した．これは NVIDIA 製ライブラリのソースコードが公開されておらず，倍精度版と 4 倍精度版で同一のアルゴリズムによる実装を行うことが困難となるからである．BiCGStab 法を構成するベクトル演算のうち，SpMV と DOT については，総和計算において計算順序が異なると計算結果が異なるケースがあり，これによって収束までの反復回数が変動する可能性がある．また SpMV は実装アルゴリズムによって，疎行列の形状（非ゼロ要素の分布など）に対する性能特性が大きく変化することがある．そのため，倍精度版と 4 倍精度版のアルゴリズムが異なると，演算精度以外の理由に起因する反復回数の変化が生じる可能性がある．なお，本稿では我々の行った倍精度版実装の性能の妥当性を示すために，NVIDIA 製ライブラリとの性能比較結果を 3.1 節で示す．

```
__global__ void SpMV_CRS_kernel
   (int m, double alpha,
   double* a_val, int* a_ptr, int* a_idx,
   double* x, double beta, double* y)
{
   int tx = threadIdx.x;
   int tid = blockDim.x * blockIdx.x + tx;
   int rowid = tid / NUM_T;
   int lane  = tid % NUM_T;
   int maxrow = MAX_BLK * NUM_TH / NUM_T;
   __shared__ double vals[NUM_TH];
   while (rowid < m){
      vals[tx] = 0.0;
      for (int i = a_ptr[rowid] + lane;
                 i < a_ptr[rowid+1]; i += NUM_T)
         vals[tx] = a_val[i] * x[a_idx[i]] + vals[tx];
      for (i = NUM_T/2; i > 0; i = i >> 1) {
         if (lane < i)
            vals[tx] += vals[tx + i];
      }
      if (lane == 0)
         y[rowid] = alpha * vals[tx] + beta * y[rowid];
      if (m > maxrow)
         __syncthreads ();
      rowid += gridDim.x * blockDim.x / NUM_T;
   }
}
```

図 2 疎行列ベクトル積のカーネルコード

2.2 疎行列ベクトル積の実装

本節では疎行列ベクトル積の実装手法について示す．疎行列は通常，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，この格納手法と疎行列の形状によって SpMV の性能が大きく異なることが知られている．さらに GPU においては，たとえ格納形式が同一であっても実装手法によって大きく性能が異なることが，吉澤らの研究 [6] から知られている．

本稿では疎行列の格納形式として，最も一般的な CRS（Compressed Row Storage）形式を用いた．実装アルゴリズムは Bell ら [7] の実装に基づく手法を用いた．この手法では，行列 1 行の計算（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）に，複数のスレッドを割り当て，

表 1 疎行列の特性と収束までの反復回数

行列	一辺の長さ	非ゼロ要素数	非ゼロ要素率 [%]	反復回数 倍精度	反復回数 4 倍精度	反復回数 4 倍精度/倍精度
atmosmodl	1489752	10319760	0.0005	279	266	0.95
wathen120	36441	565761	0.0426	338	332	0.98
memplus	17758	126150	0.0400	2448	2288	0.93
SiO2	155331	11283503	0.0468	3203	2125	0.66
epb1	14734	95053	0.0438	605	545	0.90
mario001	38434	206156	0.0140	4423	3059	0.69
CO	221119	7666057	0.0157	4225	2164	0.51
crankseg_2	63838	14148858	0.3472	7835	3702	0.47
add20	2395	17319	0.3019	822	743	0.90
coupled	11341	98523	0.0766	3916	2474	0.63
adder_trans_01	1814	14579	0.4431	299	205	0.69
circuit_2	4510	21199	0.1042	741	469	0.63
dc2	116835	766396	0.0056	2539	1607	0.63
Pd	8081	13036	0.0200	229	161	0.70
dw2048	2048	10114	0.2411	2495	1774	0.71
TSOPF_RS_b9_c6	7224	54082	0.1036	1319	488	0.37

この複数スレッド内で共有メモリを用いて総和計算を行い，ベクトル $\boldsymbol{y}$ の一要素を求める．本稿における実装では，行列 1 行あたり 8 スレッドを割り当てた．これは幾つかの行列について実験を行った結果，8 スレッドにした場合に平均的に良好な性能が得られたからである．SpMV のコードを**図 2** に示す．BiCGStab 法が必要とするのは $\boldsymbol{y} = \boldsymbol{Ax}$ の計算であるが，cuSPARSE の SpMV ルーチンと同様に $\boldsymbol{y} = \alpha\boldsymbol{Ax} + \beta\boldsymbol{y}$ を計算する実装とした．このコードにおいて NUM_T は行列 1 行あたりのスレッド数を表している（本稿における実装では NUM_T=8 である）．また，NUM_TH は 1 スレッドブロックあたりのスレッド数であり，今回は 128 とした．MAX_BLK は 1 グリッドの各次元に指定できる最大スレッドブロック数であり，65535 である．

2.3 4 倍精度版の実装

4 倍精度浮動小数点演算には Double-Double（DD）演算を用いた．DD 演算は Dekker[8]，Bailey ら [9] の手法に基づくもので，倍精度浮動小数点型（double 型）を 2 個を連結して 4 倍精度浮動小数点型を表現し，2 桁の筆算の原理で，倍精度浮動小数点演算のみを使用して 4 倍精度演算をソフトウェア的に行う手法である．

実装手法は我々の先行研究 [3] と同様である．4 倍精度版の実装にあたっては，まず倍精度版のプログラムを作成し，四則演算を DD 演算による 4 倍精度演算に置き換えた．DD 演算は 4 倍精度乗算・加算を行うデバイス関数として実装した．グローバルメモリ上における 4 倍精度データの格納は，DD 型の上位部と下位部を格納する倍精度行列をそれぞれ別に確保する方式とした．これは我々の先行研究 [3] において，SoA（Structure of Arrays）レイアウトとしている手法である．1 つの DD 型変数を倍精度型 2 個の構造体に格納する AoS（Array of Structures）レイアウトを採用しなかった理由は，4 倍精度版の SpMV およびその他の Level–1 演算において，SoA レイアウトと AoS レイアウトにおける性能の優劣が認められず，かつ収束判定を行う際のノルム計算において CUBLAS の DNRM2 関数を用いる際に，DD 型から倍精度型へのキャストを必要とせず，DD 型の上位部に対してこの関数を用いるだけで済むからである．

なお，カーネル関数において，起動スレッド数などのパラメータは倍精度版と 4 倍精度版ですべて同一とし，違いは演算精度のみとなるようにした．これは総和計算の際に計算順序が変わり，演算結果に差異が生じることを防ぐためである．また，BiCGStab 法における α や β などのスカラー値の計算はカーネル関数として実装せず，CPU 上で計算している．その際の CPU 上における 4 倍精度演算には QD ライブラリ [9] を使用した．この QD ライブラリは本稿で実装したものと同様のアルゴリズムによる DD 演算を用いて 4 倍精度演算を行う．

3. 性能評価

本稿で性能評価に用いた疎行列を**表 1** に示す．これらの行列は The University of Florida Sparse Matrix Collection[10] から取得したもので，以下のようにして選択した．

- 数ある行列の中から行列の大きさや非ゼロ要素数が異なる 208 種類の疎行列を入手（ただし，すべて実数の正方行列で，バイナリデータや整数値のみの行列は除いている）
- そのなかで本稿で実装した前処理なしの倍精度

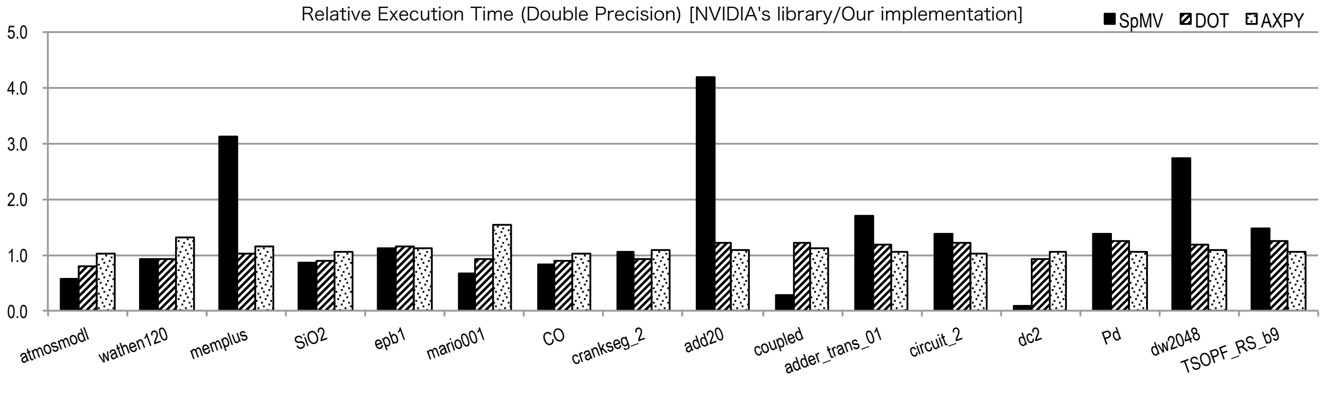

図 3 Tesla M2050 における我々の実装を基準とした NVIDIA 製ライブラリの相対実行時間（倍精度演算）

BiCGStab で 10,000 反復以内に収束したものが 51 種類（収束判定 $\epsilon = 1e-12$，$\boldsymbol{b} = (1...1)^T$, $\boldsymbol{x}_0 = (0...0)^T$ の条件）

- さらに 4 倍精度 BiCGStab で反復回数が減少したものが 30 種類（残り 14 種類は反復回数が最大約 1.3 倍に増加，7 種類は反復回数に変化なし）
- 本稿ではその 30 種類のうち，4 倍精度 BiCGStab によって倍精度と比べて収束までの時間が短くなった 8 種類と，そうでないケースのなかから似た傾向のものどうしを除いた 8 種類の合計 16 種類を取り上げる

なお，表 1 は，4 倍精度 BiCGStab 法で収束するまでのトータルの計算時間が長いものから順にソートしている．以降に示す図表は，すべてこの順にソートしている．また，本稿における「収束」とは，上記と同様に収束判定 $\epsilon = 1e-12$，$\boldsymbol{b} = (1...1)^T$, $\boldsymbol{x}_0 = (0...0)^T$ の条件で反復を行い，10,000 反復以内に収束したケースである．これは倍精度・4 倍精度ともに共通である．

性能評価には GPU として Fermi アーキテクチャの NVIDIA Tesla M2050 を用いた．ホストの CPU は Intel Xeon E5630 (2.53GHz, 4-core) ×2，OS は CentOS 6.3 であり，CUDA は 5.0 (Driver Version: 304.54)，コンパイルは gcc 4.4.6（-O3）および nvcc 5.0（-O3 -arch sm_20）で行った．本研究はシングル GPU における GPU カーネルの実行時間のみを対象に議論を行う．そのためホストとの通信時間は計測に含めていない．BiCGStab 法の性能は図 1 における 3 行目以降の反復部分のみの実行時間を測定し，1，2 行目の初期値計算の時間は除いている．また $||\boldsymbol{b}||$ は予め計算しているため反復中では計算しない．正確に測定するために，収束するまでを 3 回測定し，実行時間の平均値を求めた．また，SpMV などのカーネル関数のみの実行時間測定では，カーネルを最低 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，実行時間の平均を求めた．

3.1 予備評価：倍精度版実装の妥当性

まず 4 倍精度版のベースとなる倍精度版の実装の妥当性を示すために，倍精度演算における NVIDIA 製ライブラリと我々の実装の性能比較を行った．**図 3** に我々の実装に対する NVIDIA 製ライブラリ（CUDA5.0 に含まれる cuSPARSE および CUBLAS）の SpMV，DOT，AXPY の実行時間を示す．縦軸は我々の実装の実行時間を 1 とした時の NVIDIA 製ライブラリによる倍精度演算の実行であり，1 を下回る場合に NVIDIA 製ライブラリの方が高速であることを表している．DOT，AXPY は対応する疎行列の一辺の長さに対する計算を行った結果である．

SpMV は行列の種類によって高速である場合や低速である場合があるが，我々の実装は 16 種類の行列において平均で約 1.4 倍高速であった．また DOT，AXPY についてはCUBLAS と比較して大差はなく，妥当な性能であると言える．なお AXPYZ，XPAY については AXPY と同じ，ベクトル同士の加算およびスカラ倍の計算で，実装手法も同一であり，性能もほぼ同一であった．

3.2 倍精度と 4 倍精度の性能比較

次に，倍精度 BiCGStab と 4 倍精度 BiCGStab の性能比較を行った．**図 4** に，倍精度 BiCGStab を基準とした 4 倍精度 BiCGStab の収束までのトータルの計算時間および 1 反復あたりの計算時間を示す．結果は収束するまでのトータルの計算時間が長いものから順にソートしている（本稿では，すべての表と図においてこの順にソートしている）．

図 4 において “add20” から右の 8 個は，4 倍精度演算を用いることで倍精度演算と比べて収束までのトータルの計算時間が短くなったケースである．この 8 種類は，倍精度 BiCGStab で 10,000 反復以内に収束したもの 51 種類中の 8 種類であり，数多くあるケースであるとは言えないが，実際に倍精度演算の代わりに 4 倍精度演算を用いることで，求解までの計算時間を短縮できるケースが存在することが示された．一方で，1 反復あたりの計算時間は倍精度の約 1.0–2.2 倍であり，16 種類の平均では約 1.5 倍であっ

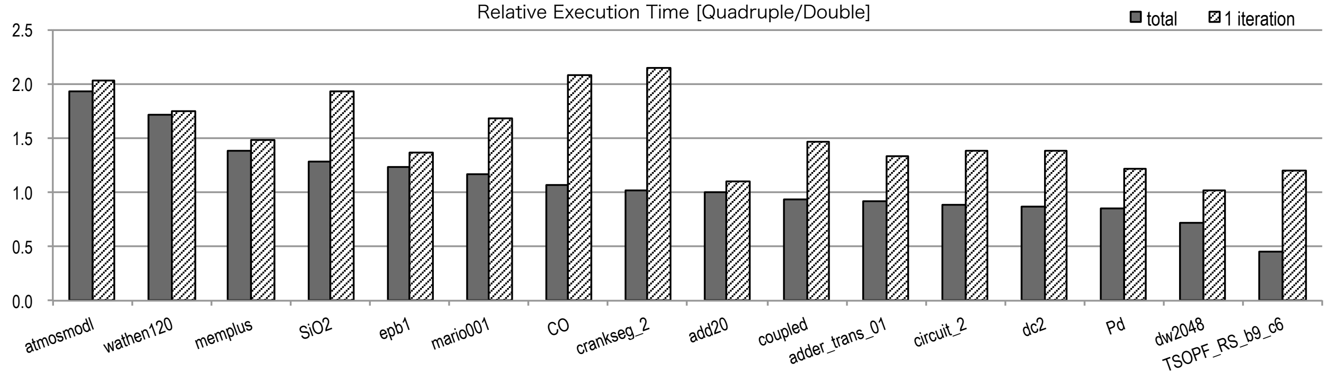

図 4　Tesla M2050 における BiCGStab 法の相対実行時間（倍精度の実行時間を 1 とする）

た．また，右方の行列ほど，1 反復あたりの計算時間が倍精度と比較してあまり増加しておらず，また表 1 に示した反復回数を参照すると，4 倍精度を用いることによる反復回数の減少量が多い傾向にあることがわかる．

今回の実験において，倍精度演算の代わりに 4 倍精度演算を用いることで収束までの計算時間を短縮できたか，同程度の時間となったケースは，大きく 2 つの場合に分けることができる．一つ目は，1 反復あたりの計算時間が約 2 倍程度に増加した一方で，反復回数が半分近くに減少したケース（“SiO2”，“CO”や “crankseg_2”）である．二つ目は，反復回数はあまり減少していないが，1 反復あたりの計算時間が 4 倍精度と倍精度であまり変わらないケース（“add20”，“dw2048”など）である．ここで性能の観点から興味深いのは，1 反復あたりの計算時間である．今回の 16 種類の行列において，4 倍精度の 1 反復あたりの計算時間は倍精度の約 1.0–2.2 倍であった．次章ではこの倍精度演算に対する 4 倍精度演算の計算時間に関して考察を行う．

4.　考察

今回の実験から，倍精度の代わりに 4 倍精度を用いることで収束までの計算時間を短縮できる可能性があるケースは，1 反復あたりの計算時間が約 2 倍程度に増加した一方で反復回数が半減するようなケースと，反復回数はあまり減少していないが，1 反復あたりの計算時間が 4 倍精度と倍精度であまり変わらないケースの 2 つに分類できる．どのような行列においてどれぐらいの反復回数が削減できるかということについては数理的な考察が必要であるが，ここでは数理的な要因を除いて，性能の観点，すなわち倍精度演算に対する 4 倍精度演算の実行時間に関して考察を行う．また，本稿では前処理の実装を行っていないが，疎行列の反復解法では反復回数を削減する方法として前処理を用いるのが一般的であり，前処理を考慮した場合の 4 倍精度演算の有効性についても検討を行う．

4.1　4 倍精度演算の倍精度演算に対する相対コスト

DD 演算は $a \times b + c$ の積和演算が 20 回の倍精度演算で構成される [3]．BiCGStab 法を構成する SpMV とその他のベクトル演算の大半の処理が積和演算であることから，4 倍精度演算に要する演算コストは理論上，倍精度演算の約 20 倍と考えることができる．しかし 4 倍精度 BiCGStab 法の 1 反復あたりの実行時間は最大でも倍精度の約 2.2 倍であった．

まず，1 反復あたりの実行時間の内訳を**図 5** に示す．一部の行列では 1 反復あたりの実行時間において，必ずしも SpMV だけが実行時間の大半を占めるわけではないことがわかる．これは行列のサイズが小さい場合や非ゼロ要素率が低い場合に，SpMV の実行時間が相対的に短くなるためであると考えられる．続いて**図 6** に，BiCGStab 法を構成する主なカーネル関数（SpMV，DOT，AXPY）の，倍精度に対する 4 倍精度の相対実行時間を示す．4 倍精度の実行時間は倍精度に対して，SpMV で約 1.4–2.5 倍，DOT で約 1.3–2.1 倍，AXPY で約 1.0–3.2 倍であった．GPU におけるこれらの演算において，4 倍精度の実行時間が倍精度の，(1) 2 倍になるケース，(2) 1 倍（変わらない）ケース，(3) 2 倍を上回るケース，の 3 パターンに分けてその理由を考察する．

4.1.1　4 倍精度の実行時間が倍精度の 2 倍になるケース

4 倍精度の実行時間が倍精度の 2 倍になるケースは，演算がメモリ律速であり，4 倍精度型が倍精度型の 2 倍のデータサイズであるため，データアクセスに 2 倍の時間を要するからであると考えられる．GPU と演算の Byte/Flop 比を検討すると，Tesla M2050 の Byte/Flop 比は 515.2GFlops の理論ピーク演算性能に対して理論メモリバンド幅が 144GB/s であるため，倍精度で約 0.3 Byte/Flop である．また 4 倍精度（DD 演算）は 20 倍の倍精度演算を要するため，“Flops” と同様に 1 秒あたりに行える DD 演算の理論ピーク演算性能を DDFlops とすると，$515.2/20 = 25.76$[GDDFlops] となる．したがって，Byte/Flop 比は約 5.6Byte/DDFlop と表せる．一方，もっとも Byte/Flop 比が小さくなる SpMV

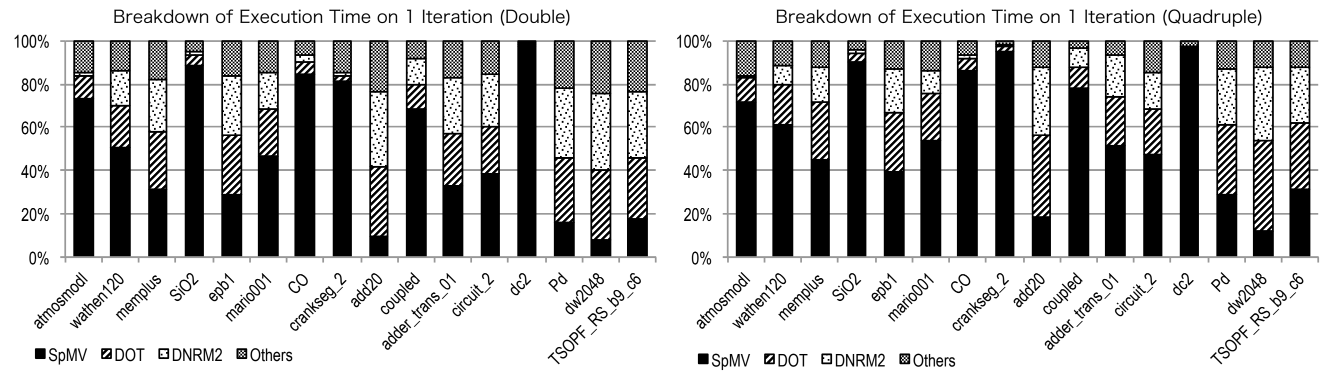

図 5　Tesla M2050 における 1 反復あたりの実行時間の内訳（左：倍精度，右：4 倍精度）

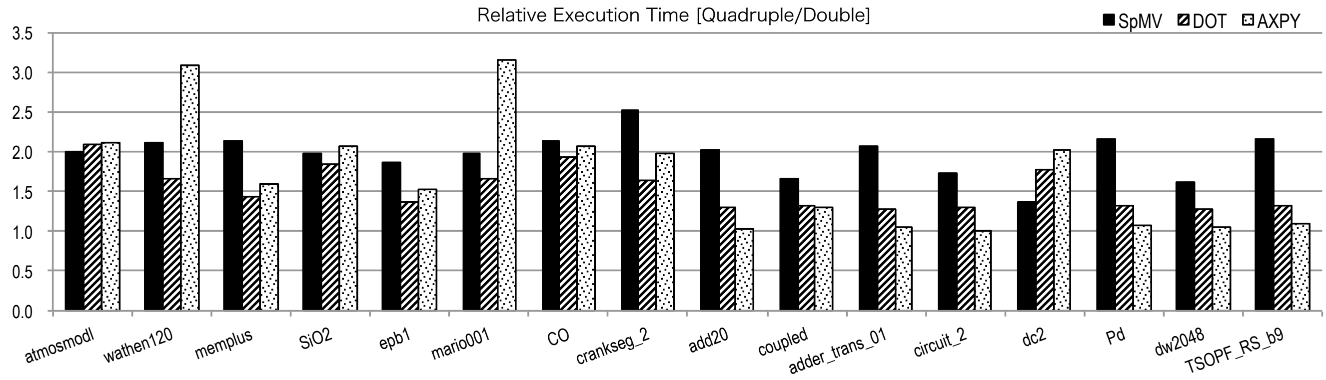

図 6　Tesla M2050 における BiCGStab 法を構成する主なカーネル関数の相対実行時間（倍精度の実行時間を 1 とする）

の Byte/Flop 比は非ゼロ要素率によって変化するが，今回の実験に用いた 16 個の行列では，インデックス行列を考慮しない場合でも倍精度で最低約 4.0Byte/Flop，4 倍精度で約 8.1 Byte/DDFlop である．したがって Byte/Flop 比から Tesla M2050 上では倍精度・4 倍精度ともに BiCGStab 法はメモリ律速であると判断できる．

4.1.2　4 倍精度の実行時間が倍精度と同じになるケース

4 倍精度の実行時間が倍精度と同じになるケースは，カーネルの起動時間や疎行列のインデックス行列の取り扱いなど，演算精度によらず一律に要するコストが全体の実行時間に対して支配的となっている場合が考えられる．我々の先行研究 [3] では，ベクトル長の短いところではカーネルの起動コストが実行時間に対して大きくなるため，倍精度と 3 倍精度の性能差が小さくなることが分かっている．図 6 と表 1 に示した行列の大きさを考慮すると，4 倍精度と倍精度で実行時間の差が小さいケースは，そうでないケースと比べて問題サイズが小さいことがわかる．また，SpMV では非ゼロ要素にアクセスするためのアドレスを格納したインデックス行列へのアクセスが必須となる．これは演算精度に関わらず 32bit 整数型の配列へのアクセスを要する．さらに，インデックス行列を介した行列の各要素へのアクセスは，GPU が不得意とする非連続アクセスとなる場合が多く，メモリアクセス律速であるが，実際にはバンド幅律速ではなく，疎行列へのデータアクセス命令律速となっている可能性が高い．

4.1.3　4 倍精度の実行時間が倍精度の 2 倍を上回るケース

“wathen120”，“mario001”では，4 倍精度 AXPY の実行時間が倍精度の約 3 倍である．これはベクトル長に対する性能効率が，倍精度と 4 倍精度で異なることに起因すると考えられる．“wathen120”，“mario001”の大きさは，それぞれ N=36,441，N=38,434 である．AXPY はベクトル長が短いところでは 4 倍精度と倍精度の性能はほぼ変わらないが，十分なベクトル長があれば，最終的には 4 倍精度が倍精度の約 2 倍の実行時間となる．しかし N=37,000 近辺においては，4 倍精度に比べて倍精度の実行効率が低く，性能差が開いている．

4.2　前処理について

疎行列の反復解法では反復回数を削減する方法として前処理を用いるのが一般的である．現段階では前処理の実装を行っていないが，ここでは前処理を行った場合でも 4 倍精度が有効となる可能性があるのかについて検討する．行列の特性によって適切な前処理は異なるが，前処理として最も一般的である fill-in なしの不完全 LU 分解（ILU(0)）を適用した場合に，倍精度と 4 倍精度で反復回数がどのように変化するのかを，CPU 向けの疎行列反復解法ライ

表 2 反復法ライブラリ Lis による BiCGStab 法の収束までの反復回数（“–”は収束しなかったもの）

行列	前処理なし		ILU(0) 前処理あり	
	倍精度	4 倍精度	倍精度	4 倍精度
atmosmodl	253	265	110	110
wathen120	328	356	22	22
memplus	2956	2182	170	1345
SiO2	2376	2000	564	336
epb1	584	546	113	112
mario001	5129	2969	-	-
CO	3988	1940	511	312
crankseg_2	6801	3757	639	523
add20	895	799	981	2785
coupled	3456	2456	-	-
adder_trans_01	351	188	-	-
circuit_2	740	523	60	61
dc2	2458	1530	813	266
Pd	233	166	31	27
dw2048	2463	1793	-	3912
TSOPF_RS_b9_c6	1361	483	-	-

ブラリである Lis（a Library of Iterative Solvers for linear systems）[11] を用いて調べた．

表 2 は，Lis Version 1.2.65 を使用して，倍精度・4 倍精度の前処理なし・ILU(0) 前処理ありの BiCGStab 法の収束までの反復回数を調べた結果である．実装の違いによる計算順序の違いや，CPU と GPU では FMA（Fused Multiply-Add）命令の有無によって計算結果が異なるため，我々の実装と比べると反復回数に不一致があるが，議論に影響する大差は生じていない．さて，この結果から ILU(0) を行ったケースでは，倍精度と 4 倍精度で反復回数が変わらなくなるケース，4 倍精度を用いることで反復回数が増加してしまうケースも見られるが，前処理を行わない場合と同様に 4 倍精度を用いることで反復回数が大きく減少しているケースも見られる．“SiO2”，“dc2”では前処理を行った方が 4 倍精度を用いることによる反復回数の削減量が大きい．また ILU(0) では収束しなくなるケースも見られる．

この結果から，ILU(0) を適用した倍精度 BiCGStab 法に対して，ILU(0) を適用した 4 倍精度 BiCGStab あるいは前処理なし 4 倍精度 BiCGStab が有効となるケースが存在することがわかる．行列の特性によって適切な前処理は異なるため，この結果は断片的なものにすぎないが，求解速度の高速化にあたって，適切な解法，前処理の選択とともに，4 倍精度演算の使用も検討の余地があると考えられる．

5. 関連研究

長谷川ら [2], [12] は反復解法において 4 倍精度および任意精度の高精度演算を用いることで収束性が改善することを示している．小武守ら [13] は 4 倍精度演算に対応した反復法ライブラリ Lis を実装している．実装においては SSE2 命令を用いた最適化実装や，最初に倍精度演算で計算を行い，途中から 4 倍精度演算を用いる手法を検討している．また，菱沼ら [14] は Lis への適用を目的とした DD 演算を用いた Level–1 演算の AVX を用いた高速化を行っているが，反復法への組み込みや疎行列ベクトル積の実装はまだなされていない．一方，齊藤ら [15] は 4 倍精度演算を行うための Scilab ツールボックス QuPAT の実装を行い，GCR 法への適用例を示した．また，部分的に 4 倍精度を用いる手法の検討を行っている [16]．幸谷 [17] は MPFR/GMP ベースの多倍長精度計算ライブラリ BNCpack に疎行列ベクトル積を実装し，疎行列ベクトル積および BiCG，GPBiCG 法に適用した場合の性能評価を行っている．

これらの研究はすべて CPU 上で行われたものであり，GPU における報告はされていなかった．また，長谷川らの研究では，演算性能に対してデータ転送性能が十分でない大規模分散並列環境では，4 倍精度演算がメモリ律速となり，倍精度演算や前処理に対しても性能面で優位となるケースが生まれる可能性について言及しているが，実際に倍精度演算と比べて 4 倍精度演算を使用することで高速化が達成できるケースは示されていなかった．

6. まとめと今後の課題

本稿では，疎行列反復解法である BiCGStab 法において 4 倍精度演算を用いることで，倍精度演算を用いた場合と比べて，求解までの計算時間を短縮できるケースがあることを GPU における実装において示した．GPU において 4 倍精度 BiCGStab 法の 1 反復あたりの計算時間は，最大でも倍精度の約 2 倍程度となることがわかった．したがって，収束までの反復回数が半減すれば，4 倍精度と倍精度の収束までの計算時間はほぼ等しくなる．また，1 反復あたりの計算時間が 4 倍精度と倍精度でほとんど変わらないケースも見られた．このようなケースでは，わずかな反復回数の減少であっても，4 倍精度の使用が高速化に結びつく可能性がある．1 反復あたりの 4 倍精度の計算時間が倍精度の約 1–2 倍となる理由は，GPU が高い浮動小数点演算性能を持つ一方で Byte/Flop 比が小さいことや，カーネル起動のコストなどの理由で小さい問題サイズに対して計算効率が上がらないことに起因すると考えられる．GPU のアーキテクチャ的な特性によって，4 倍精度演算が有効となるケースが生まれた結果であると言える．

本稿で示した 4 倍精度演算の有効事例は，幾つかの条件が重なったケースであり，すべてのケースに対して有効であるとは言えない．また，前処理を考慮した実験も不可欠である．しかし効果の大きな前処理は並列性が低い場合があり，大規模並列環境では前処理の代わりに高精度演算の使用が有効となるケースも考えられる．さらに GPU クラスタなどの環境では，ノード間・GPU 間の通信が性能上のボトルネックとなりうる可能性が高く，今回の事例以上に，

4 倍精度演算が有効となる可能性がある．反復解法は行列の特性によって，解法や前処理を使い分ける必要があるが，4 倍精度演算もそれらと並ぶ一つのツールとして利用できる可能性がある．また，解法，前処理，演算精度の有効性は計算機のアーキテクチャによって異なると言える．さらに，すべての演算を 4 倍精度で行うのではなく，部分的に用いる方法なども検討の余地がある．今後は大規模並列環境における実アプリケーションを対象に，4 倍精度演算の有効的な使用法について検討を行う予定である．

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] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–58 (2008).
[2] Hasegawa, H.: Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov Subspace Methods, *Proc. SIAM Conference on Applied Linear Algebra (LA03)* (2003).
[3] Mukunoki, D. and Takahashi, D.: Implementation and Evaluation of Triple Precision BLAS Subroutines on GPUs, *Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2012), The 13th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12)*, pp. 1378–1386 (2012).
[4] NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit), https://developer.nvidia.com/cublas.
[5] NVIDIA Corporation: cuSPARSE Library (included in CUDA Toolkit), https://developer.nvidia.com/cusparse.
[6] 吉澤大樹，高橋大介：GPU における CRS 形式疎行列ベクトル積の自動チューニング，情報処理学会研究報告，Vol. 2012-HPC-135, No. 31, pp. 1–6 (2012).
[7] Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA, *NVIDIA Technical Report*, No. NVR-2008-004 (2008).
[8] Dekker, T. J.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).
[9] Bailey, D. H.: QD (C++/Fortran-90 double-double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/.
[10] Davis, T. and Hu, Y.: The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.
[11] Lis: a Library of Iterative Solvers for linear systems: http://www.ssisc.org/lis/.
[12] 長谷川秀彦，小武守恒：演算精度をかえれば見えてくる線形方程式の世界，計算工学講演会論文集， Vol. 13, pp. 713–716 (2008).
[13] 小武守恒，藤井昭宏，長谷川秀彦，西田　晃：反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化，情報処理学会論文誌. コンピューティングシステム，Vol. 1, No. 1, pp. 73–84 (2008).
[14] 菱沼利彰，浅川圭介，藤井昭宏，田中輝雄，長谷川秀彦：反復法ライブラリ向け倍々精度演算の AVX を用いた高速化，情報処理学会研究報告， Vol. 2012–HPC–135, No. 16, pp. 1–6 (2012).
[15] Saito, T., Ishiwata, E. and Hasegawa, H.: Development of Quadruple Precision Arithmetic Toolbox QuPAT on Scilab, *Proc. International Conference on Computer Science and Applications 2010 (ICCSA 2010)*, Springer-Verlag, pp. 60–70 (2010).
[16] Saito, T., Ishiwata, E. and Hasegawa, H.: Analysis of the GCR method with mixed precision arithmetic using QuPAT, *Journal of Computational Science*, Vol. 3, No. 3, pp. 87–91 (2012).
[17] 幸谷智紀：多倍長疎行列積を用いた Krylov 部分空間法の性能評価，情報処理学会研究報告， Vol. 2012–HPC–133, No. 25, pp. 1–6 (2012).