Vol.2013-HPC-140 No.35
2013/8/2

GPUにおける4倍精度浮動小数点演算を用いたクリロフ部分空間法の高速化

椋木 大地[1,2,a)]　高橋 大介[3,b)]

概要：クリロフ部分空間法の収束性は浮動小数点演算の丸め誤差に影響されることがあり，倍精度演算の代わりに 4 倍精度演算を用いることで，収束までの反復回数を削減できる場合がある．ここで，4 倍精度演算を用いることで 1 反復あたりの実行時間が x 倍に増加したとしても，求解までに必要な反復回数が $1/x$ 倍より少なくなれば，倍精度演算で計算可能な問題においても，4 倍精度演算を用いることで求解を高速化することが可能であると考えられる．本研究ではクロリフ部分空間法の一種である Conjugate Gradient（CG）法および Bi-Conjugate Gradient Stabilized（BiCGStab）法について，4 倍精度浮動小数点演算を用いた実装を Tesla K20X GPU 上に行い，倍精度版の実装と性能を比較した．また，前処理として cuSPARSE ライブラリの単精度，倍精度 ILU(0) 前処理を適用した場合についても検討を行った．本稿では The University of Florida Sparse Matrix Collection から収集した疎行列において 4 倍精度演算を用いることで求解を高速化できた 4 つのケースを示し，反復回数を削減し求解を高速化する手段として，倍精度演算の代わりに 4 倍精度演算を用いる有効性について検討を行う．

1. はじめに

疎行列の反復解法として広く用いられているクリロフ部分空間法の収束性は，浮動小数点演算の丸め誤差に影響し，倍精度演算の代わりに 4 倍精度演算を用いることで，収束性が改善する場合があることが知られている [1]．そのため，倍精度演算のみでは求解できない問題を解くために，4 倍精度演算を用いる場合がある．

一方で，倍精度演算の代わりに 4 倍精度演算を用いることで，1 反復あたりの実行時間が例えば x 倍に増加したとしても，求解までに必要な反復回数が $1/x$ 倍より少なくなれば，4 倍精度演算を用いることで求解を高速化することが可能であると考えられる．我々はこれまでに，GPU においてクリロフ部分空間法の一種である Bi-Conjugate Gradient Stabilized（BiCGStab）法を実装し，4 倍精度版の実装が倍精度版より高速に求解できる場合があることを示した [2]．しかしこの報告において 4 倍精度演算が有効であったケースは，GPU で計算を行うには問題の規模が小さすぎたと考えられるなど，GPU における求解の高速化を目的とした 4 倍精度演算の有効性に関しては，さらなる検討と議論を要した．

本稿では性能評価により規模の大きな疎行列を用い，正定値対称行列向けに Conjugate Gradient（CG）法，非対称行列向けに BiCGStab 法を用いて評価を行った．また前処理として NVIDIA 社の cuSPARSE ライブラリ [3] による単精度，倍精度の ILU(0) 前処理を用いた場合についても検討を行った．そして GPU におけるクリロフ部分空間法について，倍精度演算の代わりに 4 倍精度演算を用いることで求解を高速化することが可能であるかについて議論を行う．

2. 関連研究

我々の知る限りでは，倍精度演算で求解できるケースにおいて，4 倍精度演算や多倍長精度演算を用いることで求解を高速化した事例は報告されていない．しかし収束性の改善を目的として 4 倍精度演算や多倍長精度演算を用いたクリロフ部分空間法の実装例はいくつか存在する．

Hasegawa[1] は 4 倍精度演算を用いた前処理なしの BiCG 法と倍精度の前処理付き BiCG 法の性能をさまざまなアーキテクチャ上で比較している．この研究において 4 倍精度演算を用いた実装が倍精度版より高速に求解できるケースは示されていないが，並列アーキテクチャ上では 4 倍精度演算の使用が並列性の低い前処理と比べて有効となる可能性を示唆している．小武守ら [4] は 4 倍精度演算に対応

1　筑波大学大学院システム情報工学研究科
2　日本学術振興会特別研究員 DC
3　筑波大学システム情報系
a)　mukunoki@hpcs.cs.tsukuba.ac.jp
b)　daisuke@cs.tsukuba.ac.jp

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
for : $k = 1, 2, ...$ **do**
  solve $\boldsymbol{M}\boldsymbol{z}_{k-1} = \boldsymbol{r}_{k-1}$
  $\rho_{k-1} = \langle \boldsymbol{r}_{k-1}, \boldsymbol{z}_{k-1} \rangle$
  if $k = 1$ **then**
    $\boldsymbol{p}_1 = \boldsymbol{z}_0$
  else
    $\beta_{k-1} = \rho_{k-1}/\rho_{k-2}$
    $\boldsymbol{p}_k = \boldsymbol{z}_{k-1} + \beta_{k-1}\boldsymbol{p}_{k-1}$
  end if
  $\boldsymbol{q}_k = \boldsymbol{A}\boldsymbol{p}_k$
  $\alpha_k = \rho_{k-1}/\langle \boldsymbol{p}_k, \boldsymbol{q}_k \rangle$
  $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$
  $\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{q}_k$
  if $||\boldsymbol{r}_k||/||\boldsymbol{r}_0|| < \epsilon$ **break**
end for

図 1 CG 法

した反復法ライブラリ Lis を実装している．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についても検討している．Furuichi ら [5] は 4 倍精度演算を使用した Generalized Conjugate Residual（GCR）法を NEC SX-9 上に実装している．この研究では前処理付き解法において，多くの実行時間を要する前処理以外の箇所に 4 倍精度演算を用いている．その結果，4 倍精度演算を使用することによるわずかな実行時間の増大で，収束性を改善することに成功している．Saito ら [6] も 4 倍精度演算を行う Scilab ツールボックスを実装した上で，GCR 法において 4 倍精度演算を適用し収束性が改善することを示している．また，部分的な 4 倍精度演算の使用にも言及している．Kouya[7] は MPFR/GMP ベースの多倍長精度計算ライブラリ BNCpack において疎行列ベクトル積を実装し，BiCGStab 法などのクリロフ部分空間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るが，廣川ら [8] は GPU において GMP を CUDA に移植した CUMP を用いたクリロフ部分空間法の実装を行っている．

3. 4 倍精度演算を用いた CG 法と BiCGStab 法

クリロフ部分空間法の一種である CG 法（**図 1**）および BiCGStab 法（**図 2**）[9] は連立一次方程式 $Ax = b$ を計算する反復解法である．CG 法は正定値対称行列向けの解法であり，BiCGStab 法は非対称行列向けに拡張し安定化したものである．クリロフ部分空間法の収束は係数行列の固有値分布に依存し，係数行列の特性を改良するために前処理を適用する場合が多い．図 1 および図 2 において M は前処理行列であり，$M = I$ の場合に前処理なしのアルゴリズムが導出できる．本稿では前処理なしの場合と前処理ありの場合の両方を取り上げる．

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\tilde{\boldsymbol{r}} = \boldsymbol{r}_0$
for : $k = 1, 2, ...$ **do**
  $\rho_{k-1} = \langle \tilde{\boldsymbol{r}}, \boldsymbol{r}_{k-1} \rangle$
  if $\rho_{k-1} = 0$ **method fails**
  if $k = 1$ **then**
    $\boldsymbol{p}_k = \boldsymbol{r}_{k-1}$
  else
    $\beta_{k-1} = (\rho_{k-1}/\rho_{k-2})(\alpha_{k-1}/\omega_{k-1})$
    $\boldsymbol{p}_k = \boldsymbol{r}_{k-1} + \beta_{k-1}(\boldsymbol{p}_{k-1} - \omega_{k-1}\boldsymbol{v}_{k-1})$
  end if
  solve $\boldsymbol{p}_k = \boldsymbol{M}\hat{\boldsymbol{p}}$
  $\boldsymbol{v}_k = \boldsymbol{A}\hat{\boldsymbol{p}}$
  $\alpha_k = \rho_{k-1}/\langle \tilde{\boldsymbol{r}}, \boldsymbol{v}_k \rangle$
  $\boldsymbol{s} = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{v}_k$
  if $||\boldsymbol{s}||/||\boldsymbol{r}_0|| < \epsilon$ **then**
    $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \hat{\boldsymbol{p}}$
    break
  end if
  solve $\boldsymbol{s} = \boldsymbol{M}\hat{\boldsymbol{s}}$
  $\boldsymbol{t} = \boldsymbol{A}\hat{\boldsymbol{s}}$
  $\omega = \langle \boldsymbol{t}, \boldsymbol{s} \rangle / \langle \boldsymbol{t}, \boldsymbol{t} \rangle$
  $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \hat{\boldsymbol{p}} + \omega_k \hat{\boldsymbol{s}}$
  $\boldsymbol{r}_k = \boldsymbol{s} - \omega_k \boldsymbol{t}$
  if $||\boldsymbol{r}_k||/||\boldsymbol{r}_0|| < \epsilon$ **break**
  if $\omega = 0$ **break**
end for

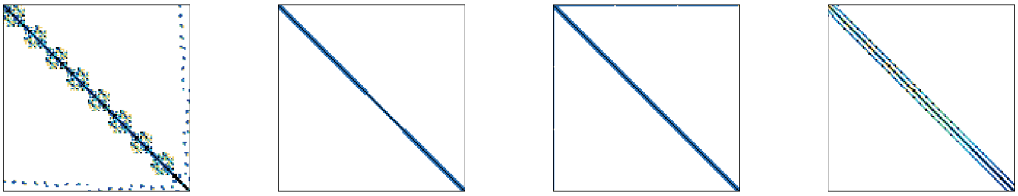
図 2 BiCGStab 法

4 倍精度版の実装においては，収束判定のためのノルム計算を除くすべての倍精度浮動小数点演算を，4 倍精度演算に変更する．ノルム計算は収束に無関係であるため，倍精度で計算する．また，入力行列 A とベクトル b は倍精度で保持する．一方，ベクトル x と他の浮動小数点データはすべて 4 倍精度で保持する．前処理付き解法については，cuSPARSE ライブラリの提供するルーチンを用いて ILU(0) 前処理を行う．ILU(0) 前処理は係数行列 A の非ゼロ要素パターンを保持したまま不完全 LU 分解を行うもので，クリロフ部分空間法の前処理として広く用いられている．ILU(0) 前処理は $A \approx M = LU$ として不完全 LU 分解によって係数行列 A を下三角行列 L および上三角行列 U を用いて近似し，そして前進代入および後退代入を用いて $M^{-1}Ax = M^{-1}b$ を計算する．cuSPARSE が提供する前処理ルーチンは単精度，倍精度のみであるが，前処理は近似計算に相当するものであるため，精度を落として計算することが可能である．本稿では 4 倍精度演算を用いた解法においても，単精度および倍精度の前処理を適用する．同様のアプローチは Furuichi ら [5] の論文にも見られる．

なお，本研究では対称行列を対象とした CG 法において，疎行列ベクトル積および前処理において，対称性を生かした実装は行っていない．

表 1 実験に用いた疎行列の特性

	bmwcra_1	bone010	rajat31	raefsky3
# of rows	148,770	986,703	4,690,002	21,200
# of nonzeros	10,641,602	47,851,783	20,316,253	1,488,768
Structure	symmetric	symmetric	unsymmetric	unsymmetric
Positive definite	yes	yes	no	no
Application	structural problem	model reduction problem	circuit simulation problem	computational fluid dynamics problem



(a) bmwcra_1 (b) bone010 (c) rajat31 (d) raefsky3

図 3 実験に用いた疎行列の非ゼロ要素パターン

4. 実装

本稿では性能比較のため，倍精度版と 4 倍精度版の両方を実装する．実装には NVIDIA 社の GPGPU 開発環境である CUDA を用いた．また，GPU として，同社の Kepler アーキテクチャ GPU（Compute Capability 3.5）をターゲットとした．

4.1 CG 法と BiCGStab 法の実装

GPU における前処理付き CG 法と BiCGStab 法の実装にあたっては Naumov の文献 [10] を参考にした．DOT（$r = \langle x, y\rangle$），AXPY（$y = \alpha x + y$）などのベクトル演算および疎行列ベクトル積（sparse matrix-vector multiplication：SpMV）（$y = Ax$）を GPU カーネル関数として実装し，スカラ値の計算は CPU 側で行う．SpMV や AXPY などの BLAS ルーチンは NVIDIA 社のライブラリである cuSPARSE および CUBLAS [11] において倍精度版ルーチンが提供されており，これらを用いて CG 法や BiCGStab 法の実装が可能である．しかしこれらのソースコードは非公開であり，同一のアルゴリズムにおいて浮動小数点演算の精度の違いのみによる影響を確認するために，本稿では 4 倍精度版，倍精度版の両方を必要とする計算については，独自にカーネル関数の実装を行った．また，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，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．

実装したカーネル関数のなかでも SpMV は CG 法，BiCGStab 法において一般に最も実行時間を要する処理である．本稿の実装では，疎行列の格納形式として，Compressed Row Storage（CRS）フォーマット（CSR：Compressed Sparse Row とも呼ばれる）を用いた．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．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 における CRS 形式 SpMV の実装としては，Bell ら [12] による 1 行あたりの計算に 32 スレッドを割り当てる CRS-vector 方式が知られている．一方，Reguly ら [13] 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．我々はこの Reguly らの手法をベースに，Kepler アーキテクチャ GPU からサポートされた 48KB リードオンリーデータキャッシュやシャッフル命令などを活用することで，性能向上を図った [14]．

前処理付き解法では，cuSPARSE ライブラリが提供する単精度，倍精度の ILU(0) 前処理ルーチンを用いた．倍精度 ILU(0) の場合，反復部分では cuSPARSE のルーチン `cusparseDcsrsv_solve()` が 4 回呼ばれる．このルーチンは疎行列の下三角または上三角行列を前進代入または後退代入で解くものである．

4.2 4 倍精度浮動小数点演算

4 倍精度浮動小数点演算は GPU においてハードウェア実装されていないため，我々は Double-Double（DD）演算を用いて 4 倍精度演算を行った．DD 演算は Dekker[15], Bailey ら [16] の手法に基づくもので，4 倍精度浮動小数点数 $a^{(q)}$ を 2 つの倍精度浮動小数点数 $a_{hi}^{(d)}$ と $a_{lo}^{(d)}$ を用いて，$a^{(q)} = a_{hi}^{(d)} + a_{lo}^{(d)}$（$|a_{lo}^{(d)}| \leq 0.5\mathrm{ulp}(a_{hi}^{(d)})$）として表現する．そして 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 倍精度浮動小数点演算を行う．DD 型において指数部は double 型倍精度の 2 倍であるが，仮数部は拡張されず double 型と同様であり，IEEE 754-2008 [17] にお

表 2 "bmwcra_1" の実験結果

Implementation	# of iter.	Execution time [sec]		$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$
		1 iter.	Total	
DP-CG	18442	1.30E-03	24.0	6.26E-08
QP-CG	10077	2.14E-03	21.6	2.06E-09
DP-CG+SP-ILU0	3010	0.0189	56.8	3.13E-08
QP-CG+SP-ILU0	2959	0.0197	58.3	2.06E-09
DP-CG+DP-ILU0	2191	0.0193	42.3	2.31E-08
QP-CG+DP-ILU0	1387	0.0201	27.9	2.06E-09

表 3 "bone010" の実験結果

Implementation	# of iter.	Execution time [sec]		$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$
		1 iter.	Total	
DP-CG	23383	7.77E-03	181.6	7.77E-08
QP-CG	11108	0.0127	140.6	1.38E-09
DP-CG+DP-ILU0	5745	0.0597	343.0	3.77E-08
QP-CG+DP-ILU0	4498	0.0646	290.6	1.38E-09

* SP-ILU0 の場合は反復回数上限に到達

いて定義されている binary128 型と異なる.

実装手法は我々の先行研究 [18] と同様に，DD 演算の乗算と加算を行うデバイス関数を実装した．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．グローバルメモリ上における 4 倍精度データの格納には CUDA において定義されている double 型 2 個からなるベクトル型である double2 型を使用し，1 つの double2 型に 1 つの DD 型の上位部と下位部を格納した．CPU 上で処理するスカラ値の計算における 4 倍精度演算には，QD ライブラリ [16] 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．

5. 性能評価

5.1 評価手法

倍精度版および 4 倍精度版の CG 法および BiCGStab 法について，前処理なしの場合，単精度 ILU(0) 前処理を適用した場合，倍精度 ILU(0) 前処理を適用した場合の求解までの実行時間を調べた．実行時間は反復部分のみを測定した．評価には Kepler アーキテクチャの NVIDIA Tesla K20X（6GB GDDR5，ECC-enabled）および CUDA 5.0 を用いた．GPU が接続されているホストマシンは Intel Xeon E5-2609 (2.40GHz)，16 GB DDR3 メモリ，CentOS 6.4（kernel: 2.6.32-358.2.1.el6.x86_64）である．コンパイラは nvcc 5.0（-O3 -arch sm_35）および gcc 4.4.6（-O3）を用いた．nvcc のコンパイラオプション "-arch sm_35" は Kepler アーキテクチャ向けの機能を利用するためのものである．

本稿では CG 法，BiCGStab 法についてそれぞれ 2 つずつ，合計 4 つの行列の結果を示す．実験に用いた疎行列の特性を**表 1**，非ゼロ要素パターンを**図 3** に示す．これらの疎行列は The University of Florida Sparse Matrix Collection[19] から取得したもので，非ゼロ要素数が 1,000,000 以上の実数の正方行列のなかから，4 倍精度演算の使用が有効であったケースを探したものである．正定値対称行列に対して CG 法，非対称行列に対して BiCGStab 法を用いた．実験時の条件はすべての測定において共通であり，右辺ベクトル $b = (1, 1, ..., 1)^T$，$x_0 = 0$，収束判定 $\epsilon = 10^{-12}$，最大反復回数 30,000 回である．

なお，本稿では以降，前処理なしの CG 法において倍精度演算のみを用いたものを "DP-CG"，4 倍精度演算を用いたものを "QP-CG" と表記した．また，単精度の ILU(0) 前処理を用いたものについては "+SP-ILU0"，倍精度の場合には "+DP-ILU0" を加えて表した．BiCGStab 法についても同様に，例えば，4 倍精度演算を用いた倍精度前処理付き BiCGStab 法は "QP-BiCGStab+DP-ILU0" と表している．

5.2 結果

CG 法における "bmwcra_1" と "bone010" の実験結果を**表 2** と**表 3** にそれぞれ示す．"bmwcra_1" および "bone010" においては，前処理なし，DP-ILU0 ありの場合において 4 倍精度演算を用いることで求解が高速化された．SP-ILU0 を適用した場合，"bmwcra_1" では DP-ILU0 よりも反復回数，求解までの実行時間が増加し，さらに DP-ILU0 を用いた場合と比べて SP-ILU0 を用いた場合に，4 倍精度演算を用いることによる反復回数の削減効果が少ないことがわかる．また "bone010" において SP-ILU0 を適用した場合には，倍精度，4 倍精度の場合ともに設定した上限反復回数 30,000 回以内に収束しなかった．これらの 2 つの問題においては ILU(0) 前処理を用いるよりも，前処理なし

表 4　“rajat31”の実験結果

Implementation	# of iter.	Execution time [sec]		$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$
		1 iter.	Total	
DP-BiCGStab	17743	0.0155	274.2	2.32E-10
QP-BiCGStab	6680	0.0293	195.9	6.89E-13

* 前処理ありの場合は収束せず

表 5　“raefsky3”の実験結果

Implementation	# of iter.	Execution time [sec]		$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$
		1 iter.	Total	
DP-BiCGStab+SP-ILU0	530	0.0361	19.11	2.24E-12
QP-BiCGStab+SP-ILU0	552	0.0364	20.11	1.03E-12
DP-BiCGStab+DP-ILU0	165	0.0376	6.20	1.34E-12
QP-BiCGStab+DP-ILU0	153	0.0378	5.79	7.64E-13

* 前処理なしの場合は収束せず

の解法で 4 倍精度演算を用いた方が高速に求解することができた．しかし倍精度版，4 倍精度版ともに，前処理を適用すると前処理なしの場合と比べて 1 反復あたりの実行時間が大幅に増加し，求解までの実行時間も増加した．1 反復あたりの実行時間については，2 つの問題ともに，前処理なしの場合に 4 倍精度版は倍精度版の約 1.6 倍であったが，DP-ILU0 を行った場合には 4 倍精度版が倍精度版の約 1.0–1.1 倍程度となった．

次に，BiCGStab 法における “rajat31”と “raefsky3”の結果を**表 4** と**表 5** にそれぞれ示す．“rajat31”では前処理なしの場合に 4 倍精度演算を用いることで反復回数が半分以下となり，求解が高速化された．しかし SP-ILU0 および DP-ILU0 を用いた場合には反復の計算途中で NaN が発生して計算が停止し収束しなかった．一方，“raefsky3”においては前処理なしの場合に DP-BiCGStab では反復途中で $\rho = 0$ により停止し，QP-BiCGStab では上限反復回数の 30,000 回まで反復させても収束しなかった．しかし SP-ILU0 および DP-ILU0 を用いることで収束した．この例では DP-ILU0 を適用した場合には 4 倍精度演算を用いることで反復回数が減少し，求解までの実行時間も減少しているが，SP-ILU0 を適用した場合には，逆に反復回数が増加している．

なお，これらの 4 つのケースすべてにおいて，4 倍精度演算を用いることで解の精度は向上していることがわかる．

6. 考察

6.1　4 倍精度演算の計算コスト

Tesla K20X GPU 上の CG 法と BiCGStab 法における 4 倍精度演算のコストを明らかにするために，前に示した 4 つのケースにおける，SpMV，DOT，AXPY の 4 倍精度および倍精度ルーチンの性能を**表 6** に示す．倍精度演算の性能は “Flops”を用いるが，4 倍精度演算の性能は 1 秒あたりの DD 演算回数を示す “DDFlops”を用いて示す．倍精度と 4 倍精度の演算性能比（DP:QP）は SpMV において約 1.6–1.7:1，DOT において約 1.2–2.3:1，AXPY において約 1.3–2.0:1 であった．

これらのルーチンは主に積和演算（$a \times b + c$）で構成されるが，GPU において積和演算における理論ピーク演算性能は DP:QP=20:1 となる [2]．しかしこれらのルーチンは GPU 上で性能がメモリ律速となっているため，実際の演算性能比は DP:QP=2:1 程度となったと考えられる．メモリ律速であることは Byte/Flop 比から明らかである．例として 4 倍精度 SpMV の場合，$(8 + 4) \times$NNZ [Bytes] / $(2\times$NNZ) [DDFlop] $= 6.0$ [Bytes/DDFlop] である（ただし NNZ $\gg$ N とする．また，4 倍精度 SpMV において入力行列は倍精度である）．一方，Tesla K20X GPU の理論ピーク演算性能は倍精度演算において 1.31TFlops であり，メモリの理論バンド幅は ECC 無効時に公称 250GB/s であるが，ECC 有効時の実バンド幅は約 170GB/s であった．4 倍精度演算の理論ピーク演算性能が倍精度演算性能の 1/20 であるから，1.31[TFlops]/20 $=$ 65.5[GDDFlops] となる．したがって，GPU の Bytes/DDFlop 比は $170/65.5 \approx 2.6$ であり，Tesla K20X において 4 倍精度 SpMV がメモリ律速であることがわかる．

例えば “raefsky3”では AXPY や DOT において DP:QP の性能比が 1.2–1.3:1 程度であるが，これはカーネルの実行効率が問題サイズや演算精度によって変化することや，問題サイズが小さい場合におけるカーネル起動コストの影響などが原因であると考えられる [2]．また SpMV においては演算精度に関わらず一律のコストとなるインデクス配列の扱いが存在するほか，本研究では入力行列が倍精度であったことなどから，実際には演算性能の DP:QP が 2:1 を下回り 1.6–1.7:1 程度となったと考えられる．

6.2　4 倍精度演算の有効性

Byte/Flop 比が十分に小さい GPU においては，SpMV，DOT，AXPY は 4 倍精度演算を行っても性能がメモリ律速となるため，これらの 4 倍精度版の実行時間は倍精度

表 6 倍精度，4 倍精度 SpMV，DOT，AXPY の性能（ただし 4 倍精度 SpMV の入力行列は倍精度である）

Problem	DP [GFlops]			QP [GDDFlops]		
	SpMV	DOT	AXPY	SpMV	DOT	AXPY
bmwcra_1	20.93	5.48	12.44	12.55	3.57	6.85
bone010	20.79	13.20	14.75	13.02	6.31	7.47
rajat31	9.34	16.13	14.98	5.46	6.98	7.52
raefsky3	19.96	1.14	7.90	12.17	0.94	6.11

の 2 倍程度，場合によってはそれ以下で実現できる．したがって，前処理なしの解法においては，4 倍精度演算の使用によって反復回数が倍精度版の半分程度になるケースにおいて，4 倍精度演算を用いることで求解までの実行時間を短縮できる可能性がある．

一方，ILU(0) 前処理を適用した場合においても，4 倍精度演算を用いることで反復回数が削減され，求解を高速化できるケースが存在した．本稿で示した事例では，前処理に要する時間が 1 反復あたりの実行時間の多くを占めた．そして，前処理として倍精度版，4 倍精度版ともに同じ精度の前処理を用いたため，1 反復あたりの実行時間は倍精度版，4 倍精度版でほぼ等しくなった．このようなケースでは，4 倍精度演算によって反復回数がわずかに減少した場合においても，反復回数の削減分だけの実行時間の短縮が見込める．

また，“bmwcra_1” および “bone010” においては，ILU(0) 前処理を適用するよりも，前処理なしの解法に 4 倍精度演算を適用した方が高速であった．しかし，そもそも倍精度の解法において前処理なしの場合よりも前処理を用いた場合の方が求解に時間を要したため，結果として今回の事例では ILU(0) 前処理は有効ではなかった．ILU(0) 前処理は前進代入および後退代入の処理において処理の依存関係があるため並列化が難しく，GPU において非効率であった可能性がある．しかし一般に効果的な前処理は並列化困難な処理を含んでおり，GPU においては並列性の低い前処理を用いるよりも，4 倍精度演算を用いるほうが高速に解ける場合が存在する可能性がある．また “rajat31” では，ILU(0) 前処理を用いても収束しなかった．これらの結果から，収束性を改善する手段として，あるいは求解を高速化する手段として，前処理だけではなく 4 倍精度演算の活用が有効となるケースが存在すると考えられる．

7. まとめと今後の課題

本稿では 4 倍精度演算を用いた CG 法と BiCGStab 法を GPU 上に実装し，Tesla K20X GPU において倍精度版と 4 倍精度版の性能を比較した．そして，倍精度演算だけで求解可能なケースにおいても，倍精度演算の代わりに 4 倍精度演算を用いることで，求解までの実行時間が短縮できるケースを示した．また cuSPARSE ライブラリを用いて，単精度および倍精度の ILU(0) 前処理を適用した場合においても，4 倍精度演算を用いることで実行時間が短縮できるケースがあることを示した．

一方で，本研究はいくつかの検討課題を残している．まず，前処理と 4 倍精度演算の有効性を比較するにあたっては，問題および GPU に適した前処理を選択し，その上で比較する必要があると言える．また，前処理に限らず，疎行列の格納形式や，解法そのものについても検討が必要である．このほか，本稿では 4 倍精度演算が有効な事例を 4 つ示したが，実問題においてそのようなケースがどれぐらいの割合で存在するか，また具体的にどのようなケースにおいて 4 倍精度演算が有効であるかを明らかにする必要がある．

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」ならびに JSPS 特別研究員奨励費（課題番号 251290）の助成を受けたものである．

参考文献

[1] Hasegawa, H.: Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov Subspace Methods, *Proc. SIAM Conference on Applied Linear Algebra (LA03)* (2003).

[2] 椋木大地，高橋大介：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，情報処理学会研究報告，Vol. 2012–HPC–137, No. 37, pp. 1–8 (2012).

[3] NVIDIA Corporation: cuSPARSE Library (included in CUDA Toolkit), https://developer.nvidia.com/cusparse.

[4] 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).

[5] Furuichi, M., May, D. and Tackley, P.: Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic, *Journal of Computational Physics*, Vol. 230, No. 24, pp. 8835–8851 (2011).

[6] 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).

[7] Kouya, T.: A Highly Efficient Implementation of Multiple Precision Sparse Matrix-Vector Multiplication and Its Application to Product-type Krylov Subspace Methods, *International Journal of Numerical Methods and Applications*, Vol. 7, pp. 107–119 (2012).

[8] 廣川祐太，藤田宜久，伊東拓，生野壮一郎：CUMP を用いた多倍長精度演算による Krylov 部分空間解法の GPU に

よる高速化，情報処理学会研究報告， Vol. 2013-HPC-139, No. 7, pp. 1–6 (2013).

[9] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and der Vorst, H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA (1994).

[10] Naumov, M.: Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, *Technical Report and White Paper*, (online), available from ⟨https://developer.nvidia.com/content/incomplete-lu-and-cholesky-preconditioned-iterative-methods-using-cusparse-and-cublas⟩ (2011).

[11] NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit), https://developer.nvidia.com/cublas.

[12] Bell, N. and Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, *NVIDIA Technical Report*, Vol. NVR-2008-004 (2008).

[13] 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).

[14] Mukunoki, D. and Takahashi, D.: Optimization of Sparse Matrix-Vector Multiplication for CRS Format on NVIDIA Kepler Architecture GPUs, *Proc. The International Conference on Computational Science and Its Applications (ICCSA 2013)*, Lecture Notes in Computer Science, No. 7975, Springer Berlin Heidelberg, pp. 211–223 (2013).

[15] Dekker, T. J.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).

[16] Bailey, D. H.: QD (C++/Fortran-90 double-double and quad-double package), http://crd.lbl.gov/˜dhbailey/mpdist/.

[17] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–58 (2008).

[18] 椋木大地，高橋大介：GPU における 3 倍・4 倍精度浮動小数点演算の実現と性能評価，情報処理学会論文誌. コンピューティングシステム， Vol. 6, No. 1, pp. 66–77 (2013).

[19] Davis, T. and Hu, Y.: The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.