Vol.2014-ARC-213 No.26
Vol.2014-HPC-147 No.26
2014/12/10

MaxwellアーキテクチャGPUにおける疑似倍精度演算を用いたDGEMMの実装と評価

椋木 大地[1,a)]　今村 俊幸[1,b)]

概要：NVIDIA が 2014 年にリリースした Maxwell アーキテクチャの GM107・GM204 コア搭載 GPU は，浮動小数点演算の理論ピーク演算性能比が倍精度:単精度=1:32 である．このような環境ではソフトウェアで実装した疑似倍精度演算を用いた方が，倍精度の計算を高速に行える可能性がある．本稿では GM204 コアを搭載する GeForce GTX 980 を対象に，単精度型を 2 個連結して倍精度型を表現し，単精度演算で疑似的な倍精度演算を実現する double-float 演算（DF 演算）を用いて，倍精度行列積を計算する BLAS ルーチンである DGEMM を実装した．その結果，ハードウェアの倍精度演算による通常の DGEMM と比べて，DF 演算を用いた DGEMM は約 2 倍の性能が得られた．

1. はじめに

今日の多くのプロセッサは IEEE 754-2008[1] の binary32（単精度）および binary64（倍精度）に準拠した浮動小数点演算が可能な演算器を搭載している．しかし，倍精度演算と比べて単精度演算が高速に行えるアーキテクチャが多い．NVIDIA の GPU アーキテクチャもその一つであり，2010 年の Fermi アーキテクチャでは倍精度:単精度の性能比が 1:2，2012 年の Kepler アーキテクチャでは 1:3 の性能比を持つ製品が発表されている．一方，2014 年に発表された Maxwell アーキテクチャを採用する GPU として本稿執筆時点で発表されている GM107・GM204 コア搭載製品は，倍精度:単精度の性能比が 1:32 となった．これは GM107・GM204 コアが画像処理を目的とした一般製品向けに設計されており，画像処理には不要な倍精度演算性能が重視されなかったためであると推測できる．Fermi アーキテクチャや Kepler アーキテクチャにおいても，一般製品向けには倍精度:単精度が 1:12 や 1:24 である製品が発表されていた[*1]．

GPU の場合，本来の目的であるコンピュータグラフィクスの計算では倍精度演算を必要としないため，CUDA がサポートされたばかりの初期の製品では倍精度演算器自体が搭載されていなかったが，その後 GPU 上で汎用計算を行うGPU コンピューティングのために倍精度演算機能が強化されるようになったという経緯がある．GPU 以前にメニーコアプロセッサとして注目を集めた Cell Broadband Engine（Cell/B.E.）においても，家庭用ゲーム機である PLAYSTATION3 に搭載されたものは倍精度:単精度の性能比が 1:14 であり，その後高性能計算向けに倍精度演算が強化された製品が登場した．このように，画像処理を含むマルチメディア処理では倍精度演算の需要が低いため，それらの用途向けに設計されたコモディティプロセッサにおいては，倍精度演算をサポートしないか，倍精度と単精度の演算性能に大きく差がある設計がなされることがある．

倍精度演算をサポートしないハードウェアにおいては，いわゆる多倍長精度演算手法を用いることでソフトウェアによって倍精度演算を実現できる．Dekker [2] はある仮数部長の浮動小数点演算でその 2 倍の仮数部長の計算を行う手法を示した．この手法によって倍精度演算を用いて 4 倍精度相当の演算を行う Double-Double 演算（DD 演算）が知られているが，同様にして単精度演算で倍精度相当の演算を行う Double-Float 演算（DF 演算）も可能である．この手法では使用するアルゴリズムにもよるが，GPU でサポートされている Fused Multiply-Add（FMA）命令を活用することで，2 倍長の積和演算（$a \times b + c$）を 16 回の浮動小数点演算命令で実現することができる．したがって，GM107・GM204 コアの Maxwell アーキテクチャ GPU では，演算律速の計算において，ハードウェアによる倍精度演算よりも DF 演算を用いたソフトウェアによる疑似倍精度演算の方が，高い性能が得られる可能性がある．

本稿では GM204 コアを搭載する Maxwell アーキテク

[1] 理化学研究所計算科学研究機構
[a)] daichi.mukunoki@riken.jp
[b)] imamura.toshiyuki@riken.jp

[*1] GPU コンピューティング用の製品においても倍精度:単精度=1:24 の Kepler アーキテクチャ GK104 コアを搭載した Tesla K10 という製品が存在する

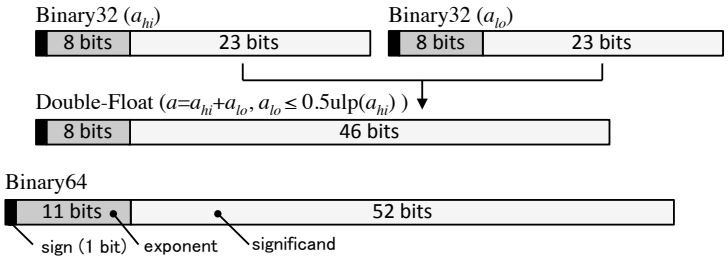

図 1 DF 型と binary64 型

チャの GPU である GeForce GTX 980[3] を対象に，倍精度密行列積 $C = \alpha AB + \beta C$ を計算する BLAS ルーチンである DGEMM を DF 演算を用いて実装し性能を評価する．GEMM は科学技術計算における重要なカーネルの一つであるとともに，$O(N^2)$ のデータに対して $O(N^3)$ の計算を行うため，実装を工夫することで性能は演算律速となり，線形計算における浮動小数点演算性能の指標となる．

GPU においては倍精度演算がハードウェアでサポートされていなかった初期の GPU において DF 演算を実装した事例が存在する [4] [5]．また，DF 演算による DGEMM の実装は，これまでにも行われている DD 演算による疑似 4 倍精度 GEMM の実装 [6][7] と技術的には同じである．しかし，倍精度演算がハードウェアでサポートされていながら，倍精度演算をソフトウェアで実装した方が高速に計算できるという状況は我々の知る限りこれまで見られなかった．本稿はその実例を示し，ソフトウェア実装による倍精度演算の有効性と，ハードウェアがサポートすべき倍精度演算性能について議論することを目的とする．

2. DF 演算の実装

2.1 DF 型フォーマット

DF 演算では 1 つの倍精度浮動小数点数 a を 2 つの単精度浮動小数点数（binary32）a_{hi} と a_{lo} を用いて $a = a_{hi} + a_{lo}$ と表現する（**図 1**）．ただし，$a_{lo} \leq 0.5\mathtt{ulp}(a_{hi})$) として上位桁部 a_{hi} と下位桁部 a_{lo} がオーバーラップしないように正規化して格納する．Binary32 の仮数部長は最上位ビットを常に 1 とみなす正規化（いわゆるケチ表現）により 24 ビット相当であるため，DF 型の仮数部は $24 \times 2 = 48$ ビット相当となり，binary64 型の 53 ビットと比べて 5 ビット小さい．また 1 つの倍精度浮動小数点数を 2 つの単精度数の和で表現するという原理上，指数部は拡張されず binary32 型の 8 ビットのままである．

DF 型は float 型 2 個からなる構造体 cudfreal に格納する．cudfreal は 8 バイトでアラインされるように宣言し，GPU 上では binary64 と同様に 8 バイトの数として扱われる．DF 型の配列は構造体の配列（Array-of-Structure）形式でメモリ上に格納する．

2.2 DF 演算の加算・乗算

計算手法の基本原理は 2 つの浮動小数点数のペアを 2 桁の数とみたてた筆算方式の計算アルゴリズムである．ここでは行列積を実装するのに必要な加算および乗算の DF 演算を行うデバイス関数の実装を示す．加算には QD ライブラリ [8] で使用されている Sloppy アルゴリズム（**図 2**），乗算には文献 [9] に示されているアルゴリズム（**図 3**）を使用した．これらのアルゴリズムはガードビットに相当する処理を行っていないため最近接偶数丸めによる 48 ビットの精度を保証しないが，我々が知る限りでは最も少ない命令数で DF 演算を行うことができる．また，乗算アルゴリズムでは FMA 命令を活用する．デバイス関数の実装においては，意図した箇所のみに FMA 命令を使用するために，加減算，乗算，FMA 命令を組み込み関数を使用して記述した．これらの実装により，DF 演算による疑似倍精度積和演算（$a \times b + c$）は，16 回の単精度浮動小数点演算命令で実現できる．

2.3 Binary64 型と DF 型の変換

Binary64 型から DF 型への変換には，1 つの浮動小数点数を仮数部を二分して 2 つの浮動小数点数に格納する SPLIT アルゴリズム [2] を使用する．デバイス関数の実装例を**図 4** に示す．536870913 は $2^{29} + 1$ であり，binary64 型の d と $2^{29} + 1$ の積 t の計算によって，d の仮数部 53bit のうち下位 29bit の情報が失われる．この t を用いて d との減算により d の上位 24bit と下位 29bit が抽出できる．下位 29bit 分は binary32 に格納する際に型変換で下位 5bit 分が丸められる．

一方，DF 型から binary64 型への変換は，DF 型の上位部と下位部をそれぞれ倍精度型に変換した後，たがいに binary64 の倍精度演算を用いて足し合わせるだけである．

3. GEMM の実装

GEMM は $C = \alpha AB + \beta C$ の計算を行う BLAS ルーチンである．BLAS ルーチンとしては行列の転置をサポートするが，今回は行列は転置されていない場合についてのみ実装する．行列は列優先格納とする．また，用途に応じて使い分けることを想定し，インタフェース（ルーチンの引

```
__device__ __forceinline__ cudfreal
DFadd (const cudfreal a, const cudfreal b){
    cudfreal t, c;
    float v;
    t.x = __fadd_rn (a.x, b.x);
    v = __fsub_rn (t.x, a,x);
    t.y = __fadd_rn (
    __fsub_rn (a.x, __fsub_rn (t,x, v)), __fsub_rn (b.x, v));
    t.y = __fadd_rn (t.y, __fadd_rn (a.y, b.y) );
    c.x = __fadd_rn (t.x, t.y);
    c.y = __fsub_rn (t.y, __fsub_rn (c.x, t.x));
    return c;
}
```

図 2 DF 加算

```
__device__ __forceinline__ cudfreal
DFmul (const cudfreal a, const cudfreal b){
    cudfreal c;
    float t, v, e;
    v = __fmul_rn (a.x, b.y);
    t = __fmaf_rn (a.y, b.x, v);
    c.x = __fmaf_rn (a.x, b.x, t);
    e = __fmaf_rn (a.x, b.x, -c.x);
    c.y = __fadd_rn (e, t);
    return c;
}
```

図 3 DF 乗算

```
__device__ __forceinline__ cudfreal
double_to_cudfreal (const double d){
    double t = __dmul_rn (536870913.0, d);
    double h = __dsub_rn (t, __dsub_rn (t, d));
    double l = __dsub_rn (d, h);
    return make_cudfreal ((float)h, (float)l);
}
```

図 4 Binary64 型から DF 型への変換

数のデータ型）と内部の演算方式の違いによって以下の 3 種類の DGEMM を実装する．

- インタフェースは binary64 で演算はハードウェアによる binary64 で行われる DGEMM（一般的な DGEMM）
- インタフェースは DF 型で演算は DF 演算で行われる DGEMM（これを DGEMM-DF(DF) と呼ぶ）
- インタフェースは binary64 で演算は DF 演算で行われる DGEMM（これを DGEMM-D(DF) と呼ぶ）

GPU 上では binary64 を使わずにすべて DF 型でデータを保持し続ける場合には DGEMM-DF(DF) を用いれば良いが，DGEMM 以外では binary64 を使うという場合には DGEMM-D(DF) が必要である．DGEMM-D(DF) は一般的な DGEMM とインタフェース互換であり，差し替えるだけで動作する．DGEMM-D(DF) ではグローバルメモリからデータを読み込むタイミングで double_to_cudfreal（図 4）を呼び，binary64 型から DF 型へ変換する．

行列積は行列が $N \times N$ であるとき $O(N^2)$ のデータに対して $O(N^3)$ の計算を行うが，同じデータへ複数回のアクセスが発生するため，データの再利用が一切行われない状態では $O(N^3)$ のアクセスが発生する．そこでブロッキングを行って高速な記憶領域にあるデータの再利用性を高めることにより，低速なメモリへのアクセス回数を削減することが最も基礎的な最適化手法である．CUDA による GEMM の実装に関する研究はいくつか行われている [10][11][12] が，いずれも高速なオンチップメモリである共有メモリを活用したブロッキングが用いられている．

行列積 $C = AB$ において，行列 A：$M \times K$，行列 B：$K \times N$，行列 C：$M \times N$ であるとき，ブロッキングを行わない場合のメモリアクセス回数は $2MNK$ 回であるが，行列 A, B, C にそれぞれ $BM \times BK$，$BK \times BN$，$BM \times BN$ のブロッキングを行うと，$MNK(1/BM + 1/BN)$ 回となる．$M = N = K$，$BM = BN$ とするとメモリアクセス回数は $2N^3$ 回から $2N^3/BN$ 回となり，つまりブロッキングによってメモリアクセス回数を $1/BN$ に減らすことができる．

3.1 理論ピーク演算性能

適切なブロッキングサイズを決定するために，まず GeForce GTX 980 における理論ピーク演算性能を見積もる．GeForce GTX 980 にはマルチプロセッサ（SMM）が 16 基搭載されている．1 つのマルチプロセッサには 128 個の単精度浮動小数点ユニットと 4 個の倍精度浮動小数点ユニットが搭載されている．コアクロックは動的に変化するが，ピーク時の 1216MHz における理論ピーク浮動小数点演算性能は，単精度が $16 \times 128 \times 1.216 \times 2 = 4980.736$GFlops，倍精度が $16 \times 4 \times 1.216 \times 2 = 155.648$GFlops と計算できる．これらの式の $\times 2$ は積和演算器による $a \times b + c$ の 2 演算を意味する．

一方，DF 演算の場合，DF 演算の積和演算 1 回が単精度浮動小数点演算命令 16 個で構成されるため，DF 演算の積和演算における理論ピーク性能は $16 \times 128 \times 1.216 \times 2/16 = 311.296$GFlops となる．なお本稿では DF 演算による倍精度浮動小数点演算 1 回も 1Flops と数える．

3.2 一般的な DGEMM の場合

行列積の最内側ループは $a \times b + c$ の積和演算で占められる．積和演算 2Flops あたり 3 オペランドを要するため，64bit 倍精度では $(3 \times 8)/2 = 12$Bytes/Flop である．一方，

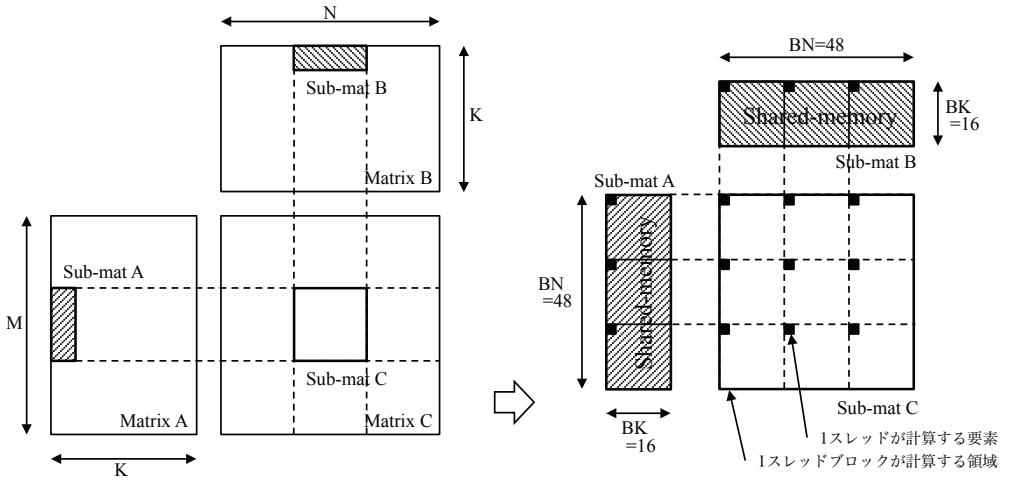

図 5 DF 演算を用いた DGEMM カーネルの設計

グローバルメモリの実測バンド幅は 180GB/s 程度であった．倍精度演算性能が約 156GFlops であるから，要求されるデータ供給バンド幅は $156 \times 12 = 1872$GB/s となる．180GB/s のグローバルメモリでデータを供給するためには，$BN \geq 1872/180 = 10.4$ で，$BN \geq 11$ 程度のブロッキングが必要であると考えられる．

一方，対象 GPU のキャッシュラインサイズは 128Bytes で 64bit 倍精度型 16 要素分に相当し，グローバルメモリへはこの倍数単位で連続アクセスが行われることが望ましい．そこで $BN \times BK$ のブロッキングにおいて $BK = 16$ として，行列 A と B の両方へキャッシュラインサイズ単位で連続アクセスを行う．よって，ブロッキングは $BN \times 16$，BN は 11 以上の 16 の倍数が候補となる．ここでスレッドブロックを $BK = 16$ にあわせて 16×16 スレッドの 2 次元構成とし，1 スレッドが $(BN/16) \times (BN/16)$ 要素を計算することにする．これにより $(BN/16) \times (BN/16)$ のレジスタブロッキングが可能となる．

BN は大きいほど共有メモリとレジスタを使用し，1 マルチプロセッサに同時に起動できるスレッドブロック数（アクティブブロック数）が減少する．CUDA ではアクティブブロック数を増やすことでスレッドブロックレベルでのストール（主にスレッドブロック内でのスレッド同期に起因する）を隠蔽する．アクティブブロック数は経験的に最低でも 3–4 程度が望ましく，今回は最低 4 を確保する．対象 GPU では 1 マルチプロセッサあたりの共有メモリ量が 96KB，1 スレッドブロックが使用できる最大量が 48KB である．一方で，1 マルチプロセッサあたり 32bit レジスタは 65536 本利用できるが，1 スレッドブロックが $16 \times 16 = 256$ スレッドであるから，アクティブブロック数 4 を確保するためには 1 スレッドあたりレジスタ数を $65536/256/4 = 64$ 以下とする必要がある．これらの条件を満たす BN の最大値は 64 であった．つまり，64×16 の共有メモリブロッキングを行い，スレッドブロックが 16×16 であるため，1 スレッドが 4×4 要素を計算する．

3.3 DF 演算を用いた DGEMM の場合

DGEMM-DF(DF) と DGEMM-D(DF) についても，一般的な DGEMM の場合と同様にしてブロッキングサイズを決定することができる．まず，DF 演算の理論ピーク性能は約 311GFlops であるため，要求されるデータ供給バンド幅は $311 \times 12 = 3732$GB/s である．180GB/s のグローバルメモリでデータを供給するためには，$BN \geq 3732/180 \approx 21$ 程度のブロッキングが必要であると考えられる．

一般的な DGEMM の場合と同様に $BK = 16$ としてアクティブブロック数が 4 となる BN を計算すると，BN の最大値は 48 であった．共有メモリの使用量は一般的な DGEMM の場合と同じであるが，一般的な DGEMM と比べると DF 演算ではレジスタを多く使用するため，レジスタ使用量の制約で確保できるブロッキングサイズが小さくなった．DGEMM-D(DF) では binary64 型と DF 型の変換コードが追加されるが，DGEMM-DF(DF) とレジスタ使用量は同一であった．**図 5** に DF 演算を用いた DGEMM（DGEMM-DF(DF) と DGEMM-D(DF)）の実装を示す．48×16 の共有メモリブロッキングに対して 16×16 のスレッドブロックで計算し，1 スレッドが 3×3 要素を計算する．

4. 性能評価

4.1 評価方法

GeForce GTX 980 を搭載するホストマシンの環境は，CPU：Intel Core i7-4790 (3.60GHz，4 コア)，メモリ：16 GB，OS：CentOS 7.0（3.10.0-123.8.1.el7.x86_64）である．GeForce GTX 980 には負荷と温度に応じてクロックを動的に変化させる機能が搭載されているが，この機能を無効にする方法がわからなかったため，今回は有効の状態

で測定している．CUDA は Version 6.5[*2]，コンパイラは nvcc V6.5.16，gcc 4.8.2 で，nvcc のコンパイルオプションは “-arch=sm_52 -O3” とした．”-arch=sm_52” は GM204 向けのコード生成を行うオプションである．

$C = \alpha AB + \beta C$ において，行列は大きさ $N \times N$ の正方行列とし，N は 128 から 8192 まで 128 ずつ測定した．入力データは α，β も含めて rand 関数で初期化した．測定対象のルーチンは 5 回実行し，最も良い値から Flops 値を計算した．なお Flops 値は DF 演算による疑似倍精度浮動小数点演算回数 1 回も 1Flop とカウントする．

比較対象として代表的な CUDA による BLAS 実装である，CUBLAS 6.5[13] と MAGMA 1.5.0[14] の SGEMM・DGEMM の性能も示す．MAGMA はビルド時にアーキテクチャを指定することで最適化されたコードを利用できるが，Maxwell 向けのものは用意されていなかったため，Kepler 向けのビルドを行った．

4.2 結果

SGEMM の性能を**図 6**，DGEMM の性能を**図 7** に示す．また，SGEMM および DGEMM の理論ピーク演算性能と実ピーク性能および実行効率を**表 1** に示す．ハードウェアによる倍精度演算を用いた通常の DGEMM と比べて，DGEMM-DF(DF) は約 2 倍のピーク性能が得られた．

CUBLAS の SGEMM および各 DGEMM は 89%前後の実行効率が得られているが，本稿で示した実装による DGEMM および DGEMM-DF(DF) においてもほぼ同等の実行効率が得られた．一方, DGEMM-D(DF) の実行効率は最大で約 71.6%となった．DGEMM-D(DF) と DGEMM-DF(DF) のコード上の違いは D-DF 変換があるかないかのみであり，スレッドブロックやブロッキングサイズの設定も同一である．また，共有メモリの使用量とレジスタ使用量も同一であったため，ワープ占有率やマルチプロセッサあたりのアクティブブロック数も同一である．したがって，DGEMM-D(DF) の実行効率の低下は binary64 型–DF 型の変換の影響のみに由来するものであると考えられる．Binary64 型から DF 型への変換においてはスループットが単精度命令の 1/32 である倍精度命令が 6 個必要であるため大きなコストを要している．

なお，DF 演算を用いた DGEMM は問題サイズが大きいところでピーク性能から約 1 割の性能低下が見られる．NVIDIA Management Library（NVML）でコア温度を取得しながら計測したところ問題サイズによらず約 80 度に到達すると性能が低下していたため，温度上昇によるクロック低下が一因として考えられる．一方，CUBLAS の SGEMM は緩やかな性能低下が見られるものの 80 度には到達しておらず，現時点では他の原因も排除できない．

[*2] CUDA 6.5 には GeForce GTX 970/980 のサポートが加えられた Version 6.5.19 が別に公開されており，これを使用した．

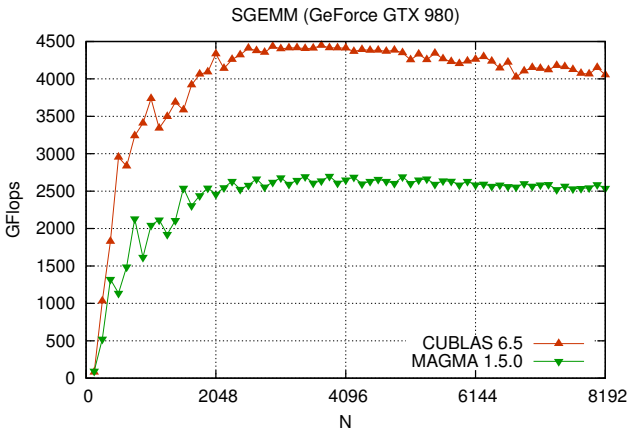

図 6 SGEMM の性能

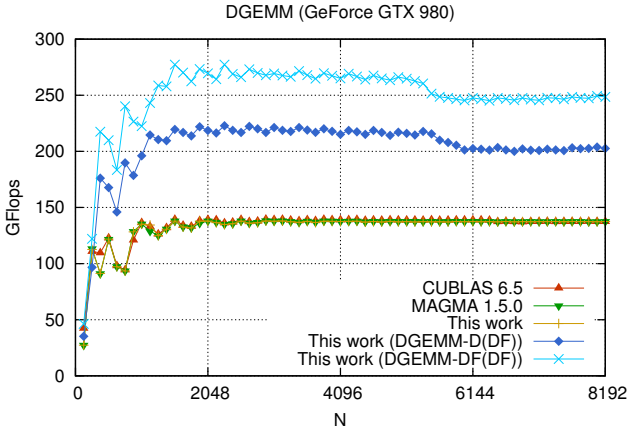

図 7 DGEMM の性能

5. まとめ

本稿では倍精度:単精度の理論ピーク演算性能比が 1:32 である Maxwell アーキテクチャ GPU において，ソフトウェアにより疑似的な倍精度演算を行う DF 演算を用いて，倍精度行列積を計算する BLAS ルーチンである DGEMM を実装した．積和演算における DF 演算の理論ピーク演算性能は単精度演算の 1/16 であり，GeForce GTX 980 における性能評価では，ハードウェアによる倍精度演算を用いた通常の DGEMM と比べて，DF 演算を用いた DGEMM は約 2 倍のピーク性能を達成した．DF 型はハードウェアによる倍精度演算（binary64）と比べて精度が若干低下することや表現できる数の範囲が小さいといった問題があるが，GEMM のような積和演算を中心とする演算律速の計算を倍精度演算性能が単精度の 1/16 を下回るような環境で行う場合には，速度上のメリットが期待できると言える．

現時点では GeForce GTX 980 と同価格帯で高速な倍精度演算性能を備えた GPU 製品が存在するため，DF 演算の必要性は高いとは言えない．しかし，初期の CUDA 対応 GPU や Cell/B.E. がそうであったように，非常に高速な単精度演算性能を持ちながら，倍精度演算機能を搭載し

表 1 GeForce GTX 980 における GEMM の性能

ルーチン	理論ピーク性能	実ピーク性能（問題サイズ）	実行効率
SGEMM (CUBLAS)	4981 GFlops	4448 GFlops (N=3712)	89.3 %
SGEMM (MAGMA)	4981 GFlops	2699 GFlops (N=3840)	54.2 %
DGEMM (CUBLAS)	155.6 GFlops	139.4 GFlops (N=2944)	89.6 %
DGEMM (MAGMA)	155.6 GFlops	138.2 GFlops (N=2944)	88.8 %
DGEMM	155.6 GFlops	138.0 GFlops (N=2944)	88.7 %
DGEMM-D(DF)	311.3 GFlops	222.8 GFlops (N=2304)	71.6 %
DGEMM-DF(DF)	311.3 GFlops	277.3 GFlops (N=1536)	89.0 %

ない，あるいは単精度演算性能と比べて倍精度演算性能が著しく低いコモディティプロセッサが今後も登場する可能性は否定できない．今回，実際にソフトウェアによる倍精度演算がハードウェアによる倍精度演算よりも高速に行えるハードウェアが出現したことは興味深く，今後，安価な一般向け製品においては単精度と倍精度の演算性能差が大きく開いた傾向が続く可能性も考えられる．

一方で，単精度演算のみが高速なコモディティプロセッサを活用する技術として，必要な箇所にのみ倍精度演算を用いる混合精度演算が注目されるようになった．混合精度演算はデータアクセス量の削減による高速化およびそれにともなう電力性能の改善といった効果も期待されている．このような混合精度演算手法の発展は，必要とされる倍精度と単精度の演算性能比のバランスを変える可能性がある．プロセッサが単精度性能に対して倍精度性能をどれだけサポートすべきかについては，ハードウェアの製造コストや電力性能などのハードウェア側の観点，アプリケーション側の需要とともに，混合精度演算やソフトウェアによる倍精度演算の性能・有効性も考慮した議論が必要である．我々は今後，数値計算ライブラリにおいて DF 演算の対応を進めることや，倍精度が必要な計算においてソフトウェアによる倍精度演算だけでも十分な性能が得られる事例を示すことで，倍精度演算器の必要性や計算に必要な演算精度に関する議論を深めたいと考えている．

謝辞 本研究は JSPS 特別研究員奨励費（課題番号 251290）の助成によるものである．

参考文献

[1] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–58 (2008).
[2] Dekker, T.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).
[3] NVIDIA Corporation: Whitepaper NVIDIA GeForce GTX 980 Featuring Maxwell, http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (2014).
[4] Graça, G. D. and Defour, D.: Implementation of float-float operators on graphics hardware, *Proc. 7th Conference on Real Numbers and Computers (RNC7)*, pp. 23–32 (2006).
[5] Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation, *ACM SIGGRAPH 2006 Research Posters* (2006).
[6] Nakata, M., Takao, Y., Noda, S. and Himeno, R.: A Fast Implementation of Matrix-matrix Product in Double-double Precision on NVIDIA C2050 and Application to Semidefinite Programming, *Proc. 3rd International Conference on Networking and Computing (ICNC 2012)*, pp. 68–75 (2012).
[7] 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. 41, pp. 66–77 (2013).
[8] Hida, Y., Li, X. S. and Bailey, D. H.: QD (C++/Fortran-90 double-double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/.
[9] Nagai, T., Yoshida, H., Kuroda, H. and Kanada, Y.: Fast Quadruple Precision Arithmetic Library on Parallel Computer SR11000/J2, *Proc. 8th International Conference on Computational Science, Part I*, ICCS '08, pp. 446–455 (2008).
[10] Nath, R., Tomov, S. and Dongarra, J.: An Improved MAGMA GEMM for Fermi GPUs, *University of Tennessee Computer Science Technical Report*, No. UT-CS-10-655 (2010).
[11] Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y. and Sun, N.: Fast Implementation of DGEMM on Fermi GPU, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, No. 35, pp. 1–11 (2011).
[12] Lai, J. and Seznec, A.: Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs, *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pp. 1–10 (2013).
[13] NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit), https://developer.nvidia.com/cublas.
[14] University of Tennessee: Matrix Algebra on GPU and Multicore Architectures (MAGMA), http://icl.cs.utk.edu/magma/.