Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

GPUによる3倍精度浮動小数点演算の検討

椋木大地[†1]　高橋大介[†2]

近年，プロセッサの演算性能向上に対してメモリやネットワークのバンド幅不足が問題となっている．浮動小数点演算において倍精度演算で精度が不足する場合，4倍精度演算を用いることが検討されてきたが，データアクセス量が少なくて済む3倍精度演算が有効となるケースが存在すると考えられる．本研究では3倍精度数を倍精度数と単精度数に分けて格納するDouble+Single型3倍精度型（D+S型）およびD+S型3倍精度演算（D+S型演算）を提案し，GPUによる3倍精度のBLAS（Basic Linear Algebra Subprograms）ルーチンを実装して，その性能をTesla C2050で評価した．D+S型演算にはDouble-Double型4倍精度演算（DD型演算）のアルゴリズムにおいて一部演算を単精度演算で行う手法を実装したが，倍精度数–単精度数の型変換が多発しD+S型演算はDD型演算よりも高コストとなった．そのためBLASの入出力をD+S型で行い，演算にはDD型演算を用いる方式を実装した．Tesla C2050では3倍精度AXPYがCUBLASの倍精度AXPYの約1.57倍の実行時間，3倍精度GEMVが倍精度GEMVの約1.69倍の実行時間となり，それぞれ4倍精度ルーチンよりも高速な性能を示した．本稿ではGPUにおけるD+S型およびD+S型演算の有効性について議論する．

1. はじめに

浮動小数点演算にはその原理上丸め誤差が存在し，科学技術計算では64bitの倍精度演算でも精度が不足する計算が存在する．線形計算の場合，例えばクリロフ部分空間法などの反復法では，丸め誤差の影響で倍精度演算であっても収束が停滞するケースが存在する[1)]．また今後，計算の大規模化による丸め誤差の蓄積が問題となることが予想される．このような背景から，倍精度演算よりも高い精度の演算に対する需要が少なからず存在する．

倍精度演算よりも高い精度の演算として，x86 CPUでは80bit拡張倍精度演算がサポートされている．しかしx86以外のGPUなどのプロセッサにおいてこのような拡張倍精度演算はサポートされておらず，倍精度演算よりも高精度の演算としては，ソフトウェアエミュレーションによる4倍精度演算が比較的よく用いられる．4倍精度浮動小数点型はIEEE754-2008においてbinary128（仮数部112bit）として定義されているが，市販のプロセッサにおいて4倍精度型および4倍精度演算はハードウェア実装はなされていない．

4倍精度浮動小数点演算をソフトウェアエミュレーションで実現する手法として，Double-Double型4倍精度演算（DD型演算）[2)]が知られている．DD型演算では倍精度数を2個用いて4倍精度数を表現し，倍精度演算によって4倍精度演算をエミュレートする．そのため仮数部は$52 \times 52 = 104$bitとなり，IEEE754-2008のbinary128より小さい．我々はこのDD型演算を用いたBLAS（Basic Linear Algebra Subprograms）ルーチンをGPU上に実装し，その性能を評価してきた[3)]．その結果，Byte/Flop比の小さいハイエンドGPU上では，AXPYのようなByte/Flop比の比較的大きな処理を，メモリ律速で実現できることが明らかとなっている．

近年，プロセッサの演算性能向上に対してメモリやネットワークのバンド幅不足が問題となっている．我々は，GPUのようなByte/Flop比の小さい環境では，データアクセス量を節約する目的で，4倍精度演算の代わりに3倍精度演算の使用が有効となるケースが存在すると考えた．倍精度演算では精度が不足するために4倍精度演算を必要としているアプリケーションの中には，実際には3倍精度演算でも演算精度が十分であるケースが存在する可能性があり，メモリ律速のアプリケーションでは，4倍精度演算の代わりに3倍精度演算を用いることで，処理の高速化を図ることができる可能性がある．

本研究ではGPUにおいて，3倍精度浮動小数点数を倍精度浮動小数点数と単精度浮動小数点数に格納する方式で表現し，DD型演算のアルゴリズムを使用して3倍精度浮動小数点演算を実現する方法を検討する．そして基本的な線形代数演算における性能を評価するために，3倍精度のBLASルーチンを実装してその性能を評価し，GPUにおける3倍精度演算の有効性を検討する．

2. 関連研究

3倍精度演算については，1960年代に48bitワードを3個利用して3倍精度演算を実現した事例[4)]があるほか，村上の文献[5)]においても，かつて富士通の大型計算機において3倍精度演算がサポートされていたとの記述がある．しかしx86 CPUおよびGPUにおいてソフトウェアエミュレーションで3倍精度演算を実現した事例は見当たらない．

†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

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

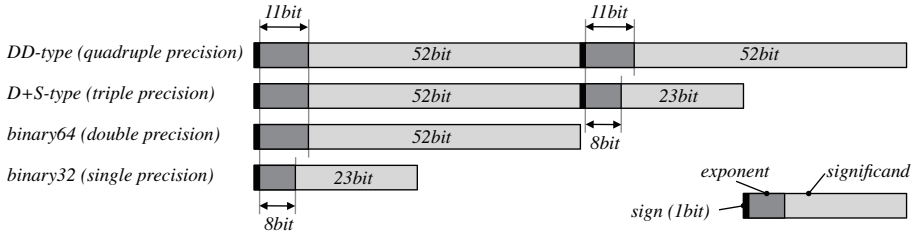

図 1 データフォーマット

4 倍精度演算では前述の DD 型演算が知られており，本稿では 3 倍精度演算に，この DD 型演算のアルゴリズムを用いる．DD 型演算の実装例として知られているのが QD[2] である．QD は CPU 用の 4 倍・8 倍精度演算ライブラリで，DD 型演算と，倍精度数を 4 個連結して 8 倍精度浮動小数点数を表現する Quad-Double 型 8 倍精度浮動小数点演算を行う．これらの演算では倍精度型が持つ浮動小数点型の指数部を，4 倍精度演算のエミュレーションでそのまま利用する．そのため GMP[6] などの高精度浮動小数点演算で必要となる指数部計算が不要であり，比較的単純に実装できるとともに高速に計算できる利点がある．しかし指数部は倍精度型と同じサイズのまま拡張できないため表現できる数の範囲が限られ，8 倍精度を超える精度の実現には適さない．なお，この DD 型演算を使用した CPU 用の BLAS 実装として，XBLAS[7]，MBLAS[8] が存在する．

GPU 上では，Göddeke ら[9] が倍精度演算に対応していない GPU に対して，単精度数を 2 個連結する方式で倍精度数を表現し，単精度演算を用いて倍精度演算をエミュレートしているほか．Thall[10] も同様の方式による倍精度演算と，単精度数を 4 個用いて 4 倍精度数を表現し，単精度演算を用いて 4 倍精度演算をエミュレートする手法を GPU 上に実装している．また Lu ら[11] は QD の GPU 版である GQD を実装している．さらに中里[12]，中田ら[13] は DD 型演算を用いた GEMM を GPU に実装している．これらの研究では GPU の高い演算性能を活用することで，CPU と比べ高速に 4 倍精度演算が実現できることを示している．一方で我々も DD 型演算による AXPY，GEMV，GEMM を GPU 上に実装してきた[3]．

3. 3 倍精度演算

ここでは我々が提案する 3 倍精度浮動小数点型，およびその演算手法について説明する．

$$
\begin{aligned}
&\mathrm{TwoSum}(a^{(d)}, b^{(d)})\{\\
&\quad s^{(d)} = a^{(d)} \oplus b^{(d)}\\
&\quad v^{(d)} = s^{(d)} \ominus a^{(d)}\\
&\quad e^{(s)} = (a^{(d)} \ominus (s^{(d)} \ominus v^{(d)}))^{(s)} \oplus (b^{(d)} \ominus v^{(d)})^{(s)}\\
&\quad \mathrm{return}\ (s^{(d)}, e^{(s)})\\
&\}
\end{aligned}
$$

図 2 TwoSum のアルゴリズム

我々の手法は DD 型の 4 倍精度演算に基づいており，DD 型および DD 型演算についても説明する．

3.1 データフォーマット

図 1 に各精度のデータフォーマットを示す．DD 型では 4 倍精度数 $a^{(q)}$ を 2 つの倍精度数 $a_{hi}^{(d)}$ と $a_{lo}^{(d)}$ によって $a^{(q)} = a_{hi}^{(d)} + a_{lo}^{(d)}$（$a_{lo}^{(d)} <= 0.5\mathrm{ulp}(a_{hi}^{(d)})$）と表す．IEEE 754-2008 の倍精度型（binary64）は仮数部が 52 ビット（ケチ表現により実際には 53 ビット相当）であり，DD 型の仮数部は 104 ビット（同様にケチ表現で 106 ビット相当），十進で約 32 桁の精度となる．

一方，我々の提案する 3 倍精度型では，DD 型と同様の原理で，3 倍精度数を倍精度数と単精度数に格納した．本稿ではこれを Double+Single 型 3 倍精度型（D+S 型）と呼ぶ．D+S 型では 3 倍精度数 $a^{(t)}$ は倍精度数 $a_{hi}^{(d)}$ と単精度数 $a_{lo}^{(s)}$ によって $a^{(t)} = a_{hi}^{(d)} + a_{lo}^{(s)}$ ($a_{lo}^{(s)} <= 0.5\mathrm{ulp}(a_{hi}^{(d)})$) と表す．仮数部は $52 + 23 = 75$ ビット（ケチ表現で 77 ビット），十進で約 24 桁の精度である．

3.2 演算アルゴリズム

D+S 型の演算（D+S 型演算）は，DD 型演算のアルゴリズムを用いて行う．DD 型演算では 4 倍精度数 $a^{(q)}$（$a^{(q)} = a_{hi}^{(d)} + a_{lo}^{(d)}$），$b^{(q)}$（$b^{(q)} = b_{hi}^{(d)} + b_{lo}^{(d)}$）どうしの計算を，2 桁の筆算の原理で計算する．演算は倍精度演算を用いて行われる．一方で D+S 型演算ではアルゴリズム中，単精度数で表現される D+S 型の下位桁の計算において，一部に単精度演算を用いる．以下，本章では，QD ライブラリで用いられている DD 型演算アルゴリズムを基に，一部に単精度演算を用いた D+S 型演算の手法を示す．本章で示すアルゴリズムにおいて単精度数を倍精度数，単精度演算を倍精度演算にしたものが，オリジナルの DD 型演算である．

以下，本稿では通常の四則演算（$\{+, -, \times, \div\}$）に対して，浮動小数点演算を $\{\oplus, \ominus,$

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

```
QuickTwoSum(a^(d), b^(s)){
    s^(d) = a^(d) ⊕ (b^(s))^(d)
    e^(s) = b^(s) ⊖ (s^(d) ⊖ a^(d))^(s)
    return (s^(d), e^(s))
}
```

図 3 QuickTwoSum のアルゴリズム

```
TwoProd(a^(d), b^(d)){
    p^(d) = a^(d) ⊗ b^(d)
    e^(s) = (fma(a^(d) × b^(d) − p^(d)))^(s)
    return (p^(d), e^(s))
}
```

図 4 TwoProd のアルゴリズム

$\otimes, \oslash$ } と表現する．浮動小数点演算は IEEE 754-2008 の最近接偶数丸めモードで行う．アルゴリズム中の上付き添字の (d)，(s) はそれぞれ倍精度数，単精度数を示す．また $(x^{(d)})^{(s)}$ は倍精度数 $x^{(d)}$ の単精度数への型変換を表し，同じ型の変数どうしの演算は変数の型と同じ精度で行う．

DD 型演算は，Dekker[14]，Knuth[15]，Shewchuk[16] による丸め誤差を考慮した浮動小数点演算のアルゴリズムを基にしている．**図 2** の TwoSum は加算における丸め誤差，すなわち $a+b$ の浮動小数点演算結果 $s = \mathrm{fl}(a+b)$ と，浮動小数点演算で生じた丸め誤差 $e = \mathrm{err}(a+b)$ を計算する．一方で**図 3** に示す QuickTwoSum は，同様に $a+b$ における丸め誤差を計算するが，$|a| \geq |b|$ が仮定される場合のみに使用できる．この QuickTwoSum は D+S 型演算において，演算結果が $a_{lo}^{(s)} <= 0.5\mathrm{ulp}(a_{hi}^{(d)})$（DD 型演算の場合，$a_{lo}^{(d)} <= 0.5\mathrm{ulp}(a_{hi}^{(d)})$）を満たすように正規化するために用いられる．

図 4 に示す TwoProd は，$p = \mathrm{fl}(a \times b)$，$e = \mathrm{err}(a \times b)$ として，倍精度乗算における丸め誤差を計算する．このアルゴリズムでは，積和演算 $a \times b + c$ の中間の演算結果を丸め誤差なしの 106 ビットで保持する，倍精度 Fused-Multiply Add（FMA）命令を利用する．$\mathrm{fma}(a \times b - p)$ は FMA 命令を用いた $a \times b - p$ の計算である．FMA 命令を使用しない場合，より複雑な計算が必要となる．

図 2–図 4 の丸め誤差を考慮した浮動小数点演算を用いた D+S 型加算および乗算アルゴリズムを**図 5**，**図 6** に示す．なお QD ライブラリでは DD 型加算において 106 ビットの精度を保証するアルゴリズムと，106 ビットの精度を保証しない代わりに演算量を削減したアルゴリズム（Sloppy アルゴリズム）の 2 種類が実装されている．Sloppy アルゴリズムでは 4 倍精度数の下位桁（$a_{lo}^{(d)}$ と $b_{lo}^{(d)}$）どうしの加算時にキャリーが発生した場合，その分だけ仮数部の下位桁が失われてしまうが，これを考慮しない．本研究では速度を優先するため Sloppy アルゴリズムを使用した．図 5 に示したアルゴリズムは Sloppy アルゴリズムに基づくものである．

```
TripleAdd(a_hi^(d), a_lo^(s), b_hi^(d), b_lo^(s)){
    (c_hi^(d), c_lo^(s)) ← TwoSum(a_hi^(d), b_hi^(d))
    c_lo^(s) ← c_lo^(s) ⊕ (a_lo^(s) ⊕ b_lo^(s))
    (c_hi^(d), c_lo^(s)) ← QuickTwoSum(c_hi^(d), c_lo^(s))
    return (c_hi^(d), c_lo^(s))
}
```

図 5 TripleAdd のアルゴリズム

```
TripleMul(a_hi^(d), a_lo^(s), b_hi^(d), b_lo^(s)){
    (c_hi^(d), c_lo^(s)) ← TwoProd(a_hi^(d), b_hi^(d))
    c_lo^(s) ← c_lo^(s) ⊕ ((a_hi^(d))^(s) ⊗ b_lo^(s)) ⊕ (a_lo^(s) ⊗ (b_hi^(d))^(s))
    (c_hi^(d), c_lo^(s)) ← QuickTwoSum(c_hi^(d), c_lo^(s))
    return (c_hi^(d), c_lo^(s))
}
```

図 6 TripleMul のアルゴリズム

4. GPU による 3 倍精度 BLAS の実装

GPU 上での基本的な線形代数演算における D+S 型および D+S 型演算の有効性を評価するために，3 倍精度の BLAS ルーチンを実装する．まず D+S 型のインタフェースをもつ 3 倍精度 BLAS ルーチンを定義した．そして D+S 型演算を行う関数を実装し，BLAS カーネルにそれらを適用して，3 倍精度の BLAS ルーチンを作成した．実装には NVIDIA 社の GPGPU 開発環境である CUDA を用いた．また CUDA 対応ハードウェアとして Fermi アーキテクチャの NVIDIA Tesla C2050 を対象とした．

4.1 D+S 型ベクトルデータの格納と 3 倍精度 BLAS のインタフェース

D+S 型では 3 倍精度数を倍精度数と単精度数のペアで表現する．したがって，1 つの 3 倍精度数は 1 つの倍精度数と 1 つの単精度数から成る D+S 型の構造体に格納すれば良い．

一方で，BLAS のようなベクトル・行列の演算では，メモリ上に連続に配置された配列データへのアクセスが生じる．CUDA では，同一ワープ内のスレッドがメモリ上の連続領域にアクセスするとき，各スレッドによるメモリアクセスを 1 回のメモリアクセスとして行うコアレスアクセスが行われる．このコアレスアクセスは Fermi アーキテクチャの GPU の

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

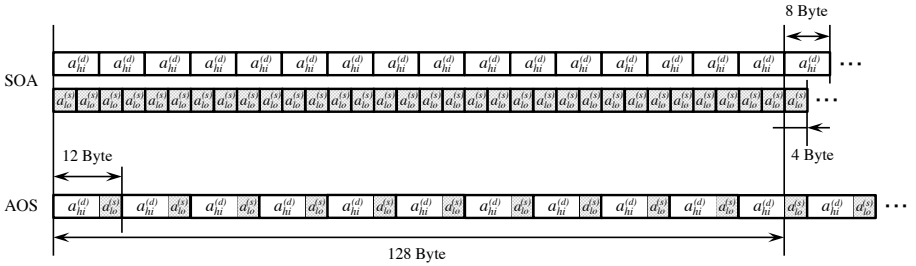

図 7 AOS 方式と SOA 方式による D+S 型ベクトルデータの格納

表 1 D+S 型演算および DD 型演算に必要な命令数と Tesla C2050 における合計必要サイクル数

命令	D+S 型演算		DD 型演算	
	加算	乗算	加算	乗算
add.rn.f32（単精度加算）	4	3	0	0
mul.rn.f32（単精度乗算）	0	2	0	0
add.rn.f64（倍精度加算）	7	2	11	5
mul.rn.f64（倍精度乗算）	0	1	0	3
fma.rn.f64（倍精度 FMA）	0	1	0	1
cvt.f64.f32（単精度→倍精度型変換）	1	1	0	0
cvt.f32.f64（倍精度→単精度型変換）	3	4	0	0
合計必要サイクル数	26	23	22	18

場合，128Byte のメモリアラインメントが満たされている場合に最大効率で行われる．しかし D+S 型の構造体は 8Byte の倍精度数と 4Byte の単精度数から成るため，1 つの D+S 型の構造体は 12Byte となる．この D+S 型の構造体を用いて配列を確保した場合，128Byte のメモリアラインメントを満たせないケースが発生する．

そのため，D+S 型のベクトルデータをそれぞれ独立に確保した倍精度数の配列と単精度数の配列に格納する方式の方が，GPU において効率的なメモリアクセスが行えると考えられる．本稿では D+S 型構造体の配列を用いて 3 倍精度ベクトルデータを格納する方式を Array of Structures（AOS：構造体の配列）方式と呼び，一方で 3 倍精度ベクトルデータを倍精度数の配列と単精度数の配列に分けて格納する方式を Structure of Arrays（SOA：配列の構造体）と呼ぶ．**図 7** に AOS 方式と SOA 方式の概要を示す．

今回，3 倍精度 BLAS のインタフェースとして，AOS 方式に対応したインタフェースと，SOA 方式に対応したインタフェースを定義し，両方の方式を実装した．

4.2 D+S 型演算関数の実装

3 章に示した D+S 型演算を行う関数を CUDA のデバイス関数として実装した．デバイス関数はコンパイル時にインライン展開されるため，関数呼び出しのオーバーヘッドは存在しない．

表 1 に D+S 型演算に必要な命令数と，Tesla C2050 における必要サイクル数を示す．比較のため DD 型演算の場合も示した．Tesla C2050 では単精度演算命令が 1 サイクル実行，倍精度演算命令が 2 サイクル実行であり，単精度演算が倍精度演算の 2 倍高速である．しかし倍精度数–単精度数の型変換に 2 サイクルを要する．そのため，型変換が必要となる D+S 型演算は DD 型演算よりも必要サイクル数が増加してしまった．なお，コンシューマ向けの製品である GeForce シリーズでは，倍精度演算命令に 8 サイクルを要するので，単精度演算の性能は倍精度演算の性能の 8 倍である．しかし型変換に要するサイクル数も同様に 8 サイクルとなるため，加算＋乗算において D+S 型演算で 169 サイクル，DD 型演算で 160 サイクルが必要となり，依然として D+S 型演算の方が高コストとなる．

4.3 BLAS カーネルの実装

Level 1–3 BLAS の代表的なルーチンとして Level 1 BLAS の AXPY ($y = \alpha x + y$)，Level 2 BLAS の GEMV ($y = \alpha Ax + \beta y$)，Level 3 BLAS の GEMM ($C = \alpha AB + \beta C$) の，実数版ルーチンを実装した．転置行列のサポートなど BLAS の仕様に準拠した実装を行ったが，AXPY，GEMV においてベクトルの incx，incy が 1 ではない場合の演算は実装していない．

AXPY，GEMV ではスレッドを 1 次元で起動してベクトル y の 1 要素ずつを各スレッドが計算し，GEMM ではスレッドを 2 次元で起動して，行列 C を計算する．GEMV，GEMM では，グローバルメモリに対して高速なスクラッチパッドメモリである共有メモリを用いたブロッキングを行った．GEMV のカーネルを**図 8**，GEMM のカーネルを**図 9** に示す．これらの図において，黒色で塗りつぶされた領域を共有メモリに格納しブロッキングする．NT はスレッドブロックあたりのスレッド数，BLK は GEMM におけるブロッキングサイズを示す．各スレッドブロックは矢印方向に内積計算を行う．AXPY，GEMV では NT=128 とした．GEMM では BLK=16，NT=16 とした．これら NT・BLK のサイズは GPU におけるスレッド実行単位やメモリアクセス単位を考慮し，ハンドチューニングにより決定した．

4.4 DD 型演算を用いた 3 倍精度 BLAS ルーチンの検討

表 1 に示したように，Tesla C2050 において D+S 型演算は DD 型演算よりも必要サイク

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

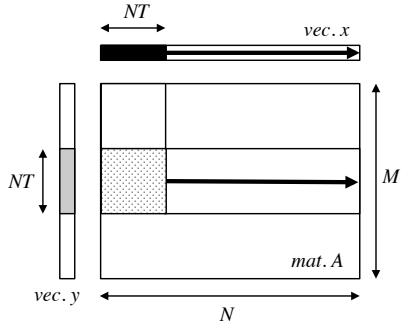

図 8 GEMV カーネル

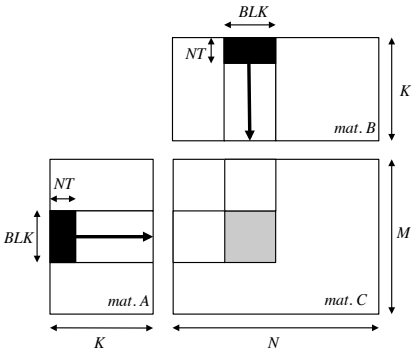

図 9 GEMM カーネル

ル数が多い．そのため，D+S 型のインタフェースを持つ 3 倍精度 BLAS の実装としては，入出力データは D+S 型であるが，BLAS の内部の演算を DD 型演算で行う方式が最も高速であると考えられる．つまりメモリ上の D+S 型のデータに対して DD 型演算を用いて計算し，例えば内積計算などにおいて，中間演算結果もすべて DD 型で格納して，最終的な計算結果を D+S 型に型変換してメモリ上へ返すという方式である．この方式は，入出力データは倍精度型であるが内部の演算において DD 型演算を行う XBLAS と同様の方式である．今回は D+S 型演算を用いた 3 倍精度 BLAS と DD 型演算を用いた 3 倍精度 BLAS の両方を実装した．

5. 3 倍・4 倍精度 BLAS の理論性能

5.1 理論ピーク演算性能

Tesla C2050 における D+S 型演算および DD 型演算の理論ピーク演算性能を算出した．AXPY，GEMV，GEMM はそのほとんどが積和演算で構成されるため，積和演算の場合の理論ピーク演算性能を示す．まず倍精度演算の理論ピーク演算性能は，1.15[GHz]×14[SM]×32[CUDACore]×(2[Flop]/2[Cycle]) = 515.2[GFlops] となる．ここで 2[Flop]/2[Cycle] は Mul+Add（2Flop）が倍精度 FMA 命令（2 [Cycle]）で計算されることを示している．

D+S 型演算の場合，1 回の D+S 型演算を 1TriFlop と定義すると，表 1 より D+S 型積和演算（2TriFlop）に必要なサイクル数は 26+23 = 49 [Cycle] である．したがって，D+S 型演算の理論ピーク演算性能は，1.15[GHz]×14[SM]×32[CUDACore]×(2[TriFlop]/49[Cycle]) ≈ 21.03[GTriFlops] となる．

表 2 BLAS ルーチンの Byte/Flop，Byte/TriFlop，Byte/QuadFlop 比

	AXPY	GEMV	GEMM
倍精度 [Byte/Flop]	12	4	16/N
3 倍精度 [Byte/TriFlop]	18	6	24/N
4 倍精度 [Byte/QuadFlop]	24	8	32/N

一方で DD 型演算の場合，1 回の DD 型演算を 1QuadFlop と定義すると，表 1 より DD 型積和演算（2QuadFlop）に必要なサイクル数は 22+18 = 40 [Cycle] であるから，DD 型演算の理論ピーク演算性能は，1.15[GHz]×14[SM]×32[CUDACore]×(2[QuadFlop]/40[Cycle]) = 25.76[GQuadFlops] となる．

5.2 Byte/Flop 比による性能予測

続いて，プロセッサの処理能力を示す Byte/Flop 比と，BLAS ルーチンの Byte/Flop 比を算出し，BLAS ルーチンがメモリ律速となるか演算律速となるかを示す．例えば BLAS ルーチンの Byte/Flop 比が GPU の Byte/Flop 比を上回る場合，性能はメモリ性能に律速され，理論ピーク演算性能に近い性能は得られない可能性が高い．

ところで，D+S 型演算では演算に倍精度演算と単精度演算が混合しており，さらに型変換のコストも無視できないため，先に求めた TriFlops で表す理論ピーク演算性能を用いて，Byte/Flop の代わりに Byte/TriFlop を計算することにする．DD 型演算の場合も同様に Byte/QuadFlop を求めた．Tesla C2050 における Byte/Flop，Byte/TriFlop，Byte/QuadFlop 比は，理論ピークメモリバンド幅 144[GB/s]，および 5.1 節で算出した理論ピーク演算性能から，以下のように計算できる．

- 倍精度演算：144[GB/s]/515.2[GFlops] ≈ 0.28[Byte/Flop]
- D+S 型演算：144[GB/s]/21.03[GTriFlops] ≈ 6.85[Byte/TriFlop]
- DD 型演算：144[GB/s]/25.76[GQuadFlops] ≈ 5.59[Byte/QuadFlop]

さらに，入力データが $N \times N$ の正方行列および大きさ N のベクトルの場合の，各 BLAS ルーチンの Byte/Flop，Byte/TriFlop，Byte/QuadFlop 比は**表 2** のようになった．例として倍精度の GEMM の場合の算出法を説明する．GEMM では $2N^3 + 3N^2$ 回の演算とグローバルメモリに対する $4N^2$ 回のロード・ストアが発生する．簡単のために演算回数の $O(N^2)$ の項を無視して考えると，乗算と加算の出現比率は 1 : 1 となる．倍精度型 1 要素が 8Byte であるから，$(4N^2 \times 8)$[Byte]/$2N^3$[Flop] = $16/N$[Byte/Flop] となる．なお，この Byte/Flop 比は同じデータに対するメモリ参照が 1 度しか行われない場合の理想値であ

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

表 3 評価環境

CPU	AMD Opteron 6134 ×2 sockets (2.3GHz, 8-Cores, Hyper-Threading disabled)
RAM	16 GB (DDR3)
OS	CentOS 6.0 (x86-64, kernel: 2.6.32-71.29.1.el6)
GPU	Tesla C2050 (3GB, GDDR5, ECC-enabled)
CUDA	CUDA SDK 4.0, CUDA Driver 4.0
Compiler	gcc 4.4.4 (-O3), nvcc 4.0 (-O3)

り，性能の目安に過ぎない．

6. 性能評価

実装した 3 倍精度 BLAS の性能を評価する．まず予備評価として，4.1 節で取り上げた，D+S 型ベクトルデータの格納方式に AOS 方式を用いた場合と，SOA 方式を用いた場合の性能を比較する．続いて 4.4 節に示した，D+S 型演算を用いた 3 倍精度 BLAS と，内部で DD 型演算を行う 3 倍精度 BLAS の性能を比較する．最後に，最も性能が良い実装を用いた 3 倍精度 BLAS と，倍精度，4 倍精度 BLAS の性能を比較する．

6.1 評価手法

評価環境を**表 3** に示す．入力に用いた行列およびベクトルは 0–1 の一様乱数で初期化された $N \times N$ の正方行列（列優先格納）および大きさ N のベクトルである．AXPY の α，GEMV，GEMM の α，β についても同様に乱数で初期化した．3 倍・4 倍精度乱数の生成には QD ライブラリの DD 型乱数生成関数 dd_rand を使用し，D+S 型へは型変換を行った．

BLAS ルーチンの実行時間の測定では，同じルーチンを最低 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．実行時間は PCI Express による GPU–CPU 間のデータ転送時間は含まれていないカーネル実行時間である．また実行時間と BLAS ルーチンの理論演算量をもとに，1 秒間に計算した 3 倍精度演算回数を TriFlops として算出した．なお DD 型演算を用いた 3 倍精度 BLAS は，演算自体は DD 型演算であり 4 倍精度演算が行われているが，本稿では 3 倍精度の BLAS として扱い，TriFlops を用いて性能を示す．

6.2 AOS 方式と SOA 方式の比較

まず D+S 型ベクトルデータの格納方式として，AOS 方式と SOA 方式を用いた場合の性能を比較した．**図 10**–**図 12** に結果を示す．なお 3 倍精度 BLAS の内部演算には D+S 型演算を使用している．いずれのルーチンにおいても，SOA 方式の方が高速となった．AXPY，

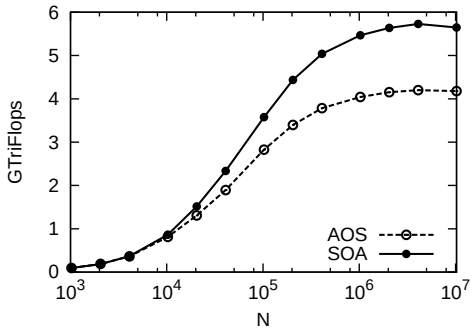

図 10 D+S 型演算を使用した 3 倍精度 AXPY における AOS 方式と SOA 方式の性能

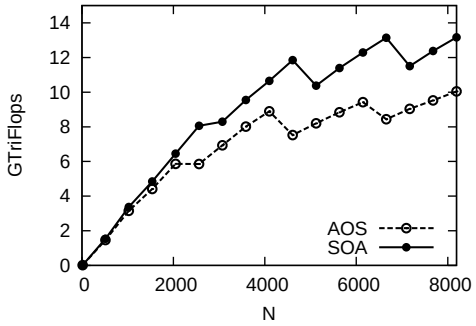

図 11 D+S 型演算を使用した 3 倍精度 GEMV における AOS 方式と SOA 方式の性能

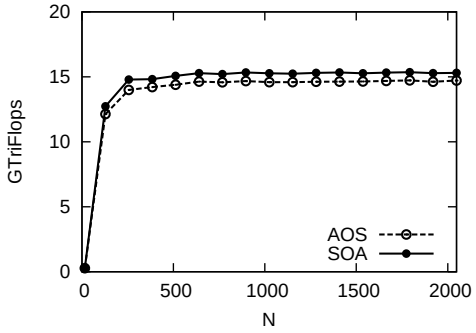

図 12 D+S 型演算を使用した 3 倍精度 GEMM における AOS 方式と SOA 方式の性能

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

GEMV では 1.3 倍程度の速度差が生じた．Byte/TriFlop 比から明らかにメモリ律速である AXPY や，GPU と BLAS ルーチンの Byte/TriFlop 比がほぼ等しい GEMV では，メモリアクセスの性能が演算性能に及ぼす影響が大きいと考えられる．一方で GEMM は Byte/Flop 比から演算律速であると考えられる．そのため AOS 方式と SOA 方式によるメモリアクセス性能の違いが，演算性能に及ぼす影響がわずかとなった．

これらの結果から，以降の実験では SOA 方式を用いた場合の性能について議論する．

6.3 D+S 型演算と DD 型演算の比較

続いて，D+S 型のインタフェースを採用する 3 倍精度 BLAS において，内部の演算に D+S 型演算を使用した場合と，DD 型演算を使用した場合の性能を比較した．**図 13**–**図 15** に結果を示す．前節の結果から D+S 型ベクトルデータの格納方式には SOA 方式を使用した．

メモリ律速である AXPY では D+S 型演算と DD 型演算の性能差は確認できない．また GEMV でも性能差はわずかである．一方で演算律速の GEMM では $N = 2,048$ で約 1.26 倍の性能差が生じた．これは 5.1 で示した D+S 型演算の理論ピーク演算性能 21.03GTriFlops と DD 型演算の 25.76GQuadFlops の差である約 1.22 倍に相当する．なお $N = 2,048$ において，D+S 型演算を用いた 3 倍精度 GEMM の性能は理論ピーク演算性能の約 73%，一方で DD 型演算を用いた 3 倍精度 GEMM は約 75%の性能であった．

これらの結果から，以降の実験では 3 倍精度ルーチンにおいて SOA 方式かつ DD 型演算を使用した場合の性能を示す．

6.4 倍精度・4 倍精度 BLAS との性能比較

最後に，3 倍精度ルーチンの実装方法として最も良い性能を示した，SOA 方式および DD 型演算を用いた 3 倍精度ルーチンの性能を，CUBLAS 4.0[17] の倍精度ルーチンおよび我々の実装した 4 倍精度ルーチンとの性能を比較した．4 倍精度ルーチンは本稿で示した 3 倍精度ルーチンと同様の手法で実装を行った．また DD 型配列は，CUDA 独自のベクトル型である double2 型の配列を使用した．double2 型は double 型 2 要素から成る 16Byte の型であり，AOS 方式に相当する．

図 16 に倍精度 AXPY の実行時間を 1 としたときの各精度の AXPY の実行時間を示す．AXPY は Byte/Flop 比より倍精度，3 倍精度，4 倍精度 AXPY すべてがメモリ律速であると考えられる．$N = 10,240,000$ において，3 倍精度 AXPY が倍精度 AXPY の約 1.57 倍の実行時間，4 倍精度 AXPY は倍精度 AXPY の約 2.05 倍の実行時間となった．3 倍精度 AXPY では内部で 4 倍精度 AXPY と同一の DD 型演算を行っているが，メモリ律速であ

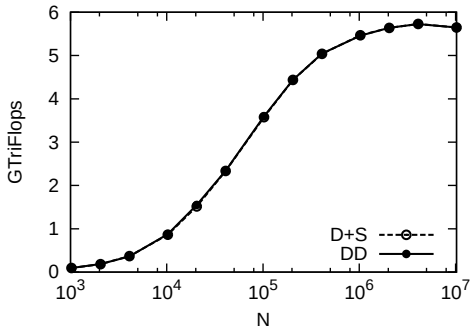

図 13 D+S 型演算と DD 型演算を使用した 3 倍精度 AXPY（SOA 方式）の性能

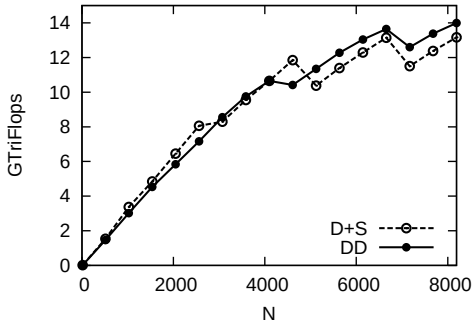

図 14 D+S 型演算と DD 型演算を使用した 3 倍精度 GEMV（SOA 方式）の性能

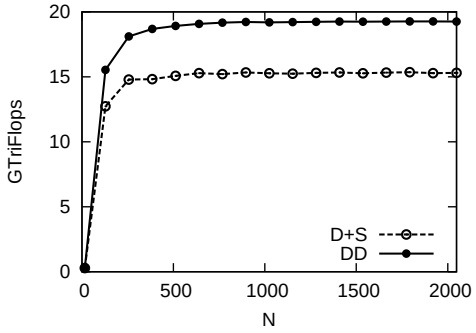

図 15 D+S 型演算と DD 型演算を使用した 3 倍精度 GEMM（SOA 方式）の性能

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

るため，性能が入出力データの型のサイズに依存する結果となった．また $N < 10,000$ では，倍精度，3 倍精度，4 倍精度 AXPY の性能差がほとんど存在しない．演算を行わない空のカーネルの実行時間を測定した結果，$N < 10,000$ における各精度の AXPY の実行時間とほぼ同じ実行時間であったことから，カーネル生成コストが原因であると考えられる．

図 17 に倍精度 GEMV の実行時間を 1 としたときの各精度の GEMV の実行時間を示す．Byte/Flop 比から倍精度，4 倍精度 GEMV はメモリ律速であると考えられる．一方 3 倍精度 GEMV は，GPU と BLAS ルーチンの Byte/TriFlop 比がほぼ等しく，演算律速に傾く可能性がある．$N = 8,192$ において，3 倍精度が倍精度の約 1.69 倍，4 倍精度が倍精度の約 2.07 倍の実行時間となった．GEMV では AXPY とは対照的にグラフが右肩下がりとなった．この結果は CUBLAS の倍精度 GEMV の実行時間を基準とした実行時間であるため，我々の実装した GEMV はサイズの小さなベクトル・行列に対するチューニングが十分でない可能性が考えられる．

図 18 の倍精度 GEMM の実行時間を 1 としたときの各精度の GEMM の実行時間を示す．GEMM は Byte/Flop 比から倍精度，3 倍精度，4 倍精度 GEMM のいずれも演算律速である．3 倍精度 GEMM は 4 倍精度 GEMM と同一の DD 型演算を行っているが，3 倍精度 GEMM の方が 4 倍精度 GEMM よりも低速な結果となった．$N = 2,048$ において，4 倍精度 GEMM は理論ピーク演算性能の約 86%の性能が得られているが，3 倍精度 GEMM は約 75%の性能であり，3 倍精度 GEMM の実行効率が低い．カーネル設計自体に改善の余地がある可能性が高い．また，倍精度演算の理論ピーク演算性能 515.2GFlops と，DD 型演算の理論ピーク演算性能 25.76GQuadFlops の差は 20 倍であるが，$N = 2,048$ において，4 倍精度ルーチンの実行時間は倍精度ルーチンの約 15.4 倍である．これは CUBLAS の倍精度 GEMM の実行効率が約 58%と低いため[18] である．

7. ま と め

本稿では 3 倍精度浮動小数点数を倍精度浮動小数点数と単精度浮動小数点数に分けて格納する D+S 型を提案し，BLAS カーネルに適用してその性能を Tesla C2050 を用いて評価した．D+S 型演算には DD 型演算のアルゴリズムにおいて一部演算を単精度で行う D+S 型演算を検討したが，倍精度数–単精度数の型変換が多発し，それがボトルネックとなって DD 型演算よりも高コストとなった．そのため現状では 3 倍精度の BLAS ルーチンとして，入出力のインタフェースは D+S 型であるが，演算には DD 型演算を用いる方式での実装が最も高速である．また D+S 型ベクトルデータを CUDA で扱う際には，D+S 型が 12Byte

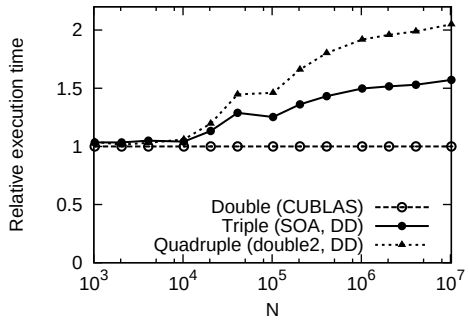

図 16 倍精度 AXPY の実行時間を 1 としたときの 3 倍・4 倍精度 AXPY の実行時間

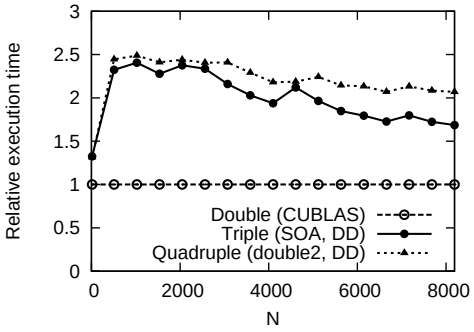

図 17 倍精度 GEMV の実行時間を 1 としたときの 3 倍・4 倍精度 GEMV の実行時間

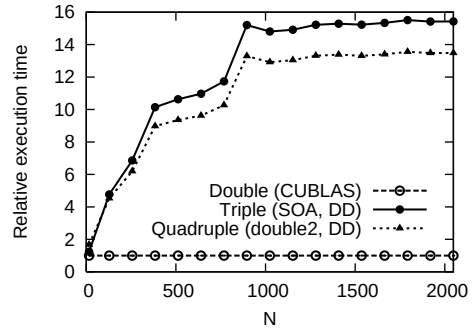

図 18 倍精度 GEMM の実行時間を 1 としたときの 3 倍・4 倍精度 GEMM の実行時間

Vol.2011-ARC-197 No.23
Vol.2011-HPC-132 No.23
2011/11/29

であるため，D+S 型構造体の配列では CUDA の 128Byte のメモリアラインメントを満たすことが難しいため，D+S 型ベクトルを倍精度配列と単精度配列にそれぞれ個別に格納するべきであることがわかった．

性能評価では Tesla C2050 において，3 倍精度 AXPY が CUBLAS の倍精度 AXPY の約 1.57 倍の実行時間となった．また 3 倍精度 GEMV も倍精度 GEMV の約 1.69 倍の実行時間で実現できた．これは 3 倍精度ルーチンが倍精度ルーチンの 1.5 倍のメモリバンド幅を必要とし，AXPY や GEMV がメモリ律速であったためである．そのため倍精度ルーチンの 2 倍のメモリバンド幅を必要とする 4 倍精度ルーチンと比べ，実行時間が減少した．一方で演算律速の GEMM では 4 倍精度 GEMM よりも 3 倍精度 GEMM の実行時間が増加した．

倍精度で精度が不足する場合，従来は 4 倍精度演算の使用が検討されてきたが，我々は 3 倍精度演算が有効となるケースが少なからず存在するのではないかと考えている．近年，プロセッサの演算性能向上に対してメモリやネットワークのバンド幅が不足し，システムの Byte/Flop 比がより小さくなっている．3 倍精度型の表現手法および演算手法には検討の余地があると考えられるが，4 倍精度演算に比べて演算コストが高コストであったとしても，Byte/Flop 比の低い環境では，疎行列計算などのメモリ律速の計算において，データアクセス量が少なくて済む 3 倍精度演算の使用が有効となる可能性がある．今後はさらに効率的な 3 倍精度演算手法の検討と，実アプリケーションにおいて 4 倍精度演算に比べ 3 倍精度演算が有効となるケースを示すことが課題である．

謝辞 本研究の一部は，文部科学省科学研究費補助金「新学術領域研究」（課題番号 22104003）による．

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) Bailey, D.H.: QD (C++ / Fortran-90 double–double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/.
3) 椋木大地, 高橋大介: GPU による 4 倍・8 倍精度 BLAS の実装と評価, 2011 年ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集, pp.148–156 (2011).
4) Ikebe, Y.: Note on triple-precision floating-point arithmetic with 132-bit numbers, *Commun. ACM*, Vol.8, pp.175–177 (1965).
5) 村上弘: HPC 用に欲しい数値演算ハードウェア機構, 情報処理学会研究報告, Vol.2010–HPC–128, pp.1–3 (2010).
6) Granlund, T. and the GMP development team: GMP: GNU Multiple Precision Arithmetic Library, http://gmplib.org/.
7) Li, X.S., Demmel, J.W., Bailey, D.H., Hida, Y., Iskandar, J., Kapur, A., Martin, M.C., Thompson, B., Tung, T. and Yoo, D.J.: XBLAS – Extra Precise Basic Linear Algebra Subroutines, http://www.netlib.org/xblas/.
8) Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), http://mplapack.sourceforge.net/.
9) Göddeke, D., Strzodka, R. and Turek, S.: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, *International Journal of Parallel, Emergent and Distributed Systems*, Vol.22 (2007).
10) Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation, *ACM SIGGRAPH 2006 Research Posters* (2006).
11) Lu, M., He, B. and Luo, Q.: Supporting Extended Precision on Graphics Processors, *Proc. Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)* (2010).
12) Nakasato, N.: A fast GEMM implementation on the cypress GPU, *SIGMETRICS Perform. Eval. Rev.*, Vol.38, pp.50–55 (2011).
13) 中田真秀, 高雄保嘉, 野田茂穂, 姫野龍太郎: GPU による倍々精度行列–行列積の高速化, 計算工学講演会論文集, Vol.16 (2011).
14) Dekker, T.J.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol.18, pp.224–242 (1971).
15) Knuth, D.E.: *The Art of Computer Programming Vol.2 Seminumerical Algorithms*, Addison-Wesley, 3rd edition (1998).
16) Shewchuk, J. R.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, *Discrete and Computational Geometry 18*, pp.305–363 (1997).
17) NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit).
18) Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y. and Sun, N.: Fast Implementation of DGEMM on Fermi GPU, http://asg.ict.ac.cn/projects/dgemm/sc11_dgemm.pdf (2011).