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　66–77 (Jan. 2013)

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

椋木 大地[1,a)]　高橋 大介[2,b)]

受付日 2012年7月4日, 採録日 2012年10月14日

概要：本論文では GPU において 3 倍・4 倍精度浮動小数点演算を実現し，線形計算への適用例として Level 1–3 の代表的な BLAS（Basic Linear Algebra Subprograms）ルーチンである AXPY，GEMV，GEMM を実装して性能評価を行った結果を示す．4 倍精度演算には Double-Double 型（DD 型）の 4 倍精度演算（DD 演算）を用いた．一方で 3 倍精度演算として新たに，Double+Single 型（D+S 型）・Double+Int 型（D+I 型）の 3 倍精度フォーマットを提案し，内部の計算に DD 演算を用いることで 3 倍精度演算を行う手法を実装した．NVIDIA Tesla M2090 における性能評価では，3 倍・4 倍精度の AXPY・GEMV がメモリ律速となり，その実行時間はデータサイズに比例して，単精度ルーチンに対しておよそ 3 倍，4 倍となることを示した．我々が提案した 3 倍精度演算は，3 倍精度データに対する DD 演算がメモリ律速となるケースにおいて，4 倍精度演算に対する速度面での利点が主張できる．4 倍精度は必要ないが倍精度では精度が不足する場合では，特に PCI Express やネットワークの帯域が性能のボトルネックとなりやすい GPU クラスタ環境などで，4 倍精度に対する 3 倍精度の有効性が期待できる．

キーワード：3 倍精度，4 倍精度，GPU，BLAS

Implementation and Evaluation of Triple and Quadruple Precision Floating-point Operations on GPUs

Daichi Mukunoki[1,a)]　Daisuke Takahashi[2,b)]

Received: July 4, 2012, Accepted: October 14, 2012

Abstract: We have implemented triple and quadruple precision floating-point operations on GPUs. As an example of the application of linear algebra operations, we have implemented triple and quadruple precision subroutines of the Basic Linear Algebra Subprograms (BLAS), AXPY, GEMV and GEMM, and evaluated their performance. For quadruple precision, we used Double-Double (DD) type quadruple precision operations (DD-operations). On the other hand, in our research we are proposing Double+Single (D+S) and Double+Int (D+I) type triple precision floating-point formats and triple precision operations that use DD-operations internally. On an NVIDIA Tesla M2090, the triple and quadruple precision AXPY and GEMV are memory-bound. Therefore, the execution time of the triple and quadruple precision operations is approximately 3x and 4x that of the single precision, respectively. Our triple precision operations have the advantage of speed compared to quadruple precision, in cases where the triple precision operations are memory-bound. In cases where quadruple precision is not required, but double precision is insufficient, we predict that our triple precision operations will perform well, especially in environments such as GPU clusters where the bandwidth of the PCI Express and the network may become bottlenecks.

Keywords: triple precision, quadruple precision, GPU, BLAS

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[2] 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] mukunoki@hpcs.cs.tsukuba.ac.jp
[b)] daisuke@cs.tsukuba.ac.jp

1. はじめに

浮動小数点演算の精度は有限である．現在多くのプロセッサがIEEE 754の32 bit単精度（十進約7桁）と64 bit倍精度（十進約16桁）をサポートする．しかし悪条件の問題を扱う場合など，科学技術計算において64 bit倍精度でも精度が不足する計算が存在する．たとえば線形計算の場合，疎行列に対する反復解法では，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1985年にIEEE 754において単精度・倍精度が規格化されてから，すでに25年以上が経過した．計算機の発達で大規模な計算が可能となり，丸め誤差の蓄積が懸念されるようになったことや，より正確な計算を実現するために，浮動小数点演算の精度拡張が求められるようになった．このような背景から，2008年にはIEEE754-2008 [2] において，32 bit単精度の約4倍の精度に相当するbinary128（仮数部112 bit）が定義されている．

本論文では，GPUにおいて，32 bit単精度に対して3倍・4倍程度の精度を実現する3倍・4倍精度浮動小数点演算をソフトウェアにより実現し，その線形計算における性能を議論する．4倍精度演算には既存手法であるDouble-Double型4倍精度演算（DD演算）を用いる一方で，我々は新たにDD演算をベースとした3倍精度演算手法を提案する．3倍精度の実現や有効性に関する議論はこれまでほとんど見られなかった．しかし倍精度と4倍精度では精度に大きな差があり，4倍精度は必要ないが倍精度では精度が不足するようなケースが存在することは容易に想像できる．そのようなケースにおいて4倍精度演算より高速な3倍精度演算の実現が望まれる．また，3倍精度は4倍精度と比べてデータサイズが小さい．近年増加しているGPUクラスタにおいては，PCI Expressやノード間のネットワークにおけるデータ転送時間がアプリケーション性能のボトルネックとなることが多い．さらに，GPUのようなアクセラレータにおいては，アクセラレータ上のメモリスペースが限られていることが多く，これが原因となりPCI Expressを経由したホストとの通信が発生することもある．このような環境において4倍精度は必要ないが倍精度では精度が不足するような場合に，データサイズの小さい3倍精度が4倍精度と比べて有効となるケースが存在すると考えられる．また，x86プロセッサでは80 bitの拡張倍精度演算がサポートされているが，GPUにおいて倍精度よりも高い精度はサポートされていない．これらの背景から，我々はまず単一GPUにおける3倍精度演算の実現について検討を行うこととした．

本論文では，GPUにおいて3倍・4倍精度浮動小数点演算（乗算・加算）を実装し，GPUアクセラレーションに適した線形計算への適用例として，Level 1–3の代表的なBLAS（Basic Linear Algebra Subprograms）ルーチンであるAXPY（$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$），GEMV（$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$），GEMM（$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$）を実装して，性能を評価した．GPUとして，NVIDIA GF100アーキテクチャ（通称Fermi）のTesla M2090を対象とする．実装には同社のGPGPU開発環境であるCUDAを用いた．以下，2章において関連研究を紹介したあと，3章において4倍精度演算について，次の4章では3倍精度演算について説明する．続いて5章では線形計算への適用例としてBLASの一部の機能を実装し性能評価を行う．最後に6章で本論文のまとめとする．

2. 関連研究

ここでは拡張精度の浮動小数点演算をソフトウェアで実現した事例を紹介する．拡張精度として最も広く普及しているのがDD演算による4倍精度浮動小数点演算である．DD演算は64 bit倍精度浮動小数点数（double型）を2個用いて4倍精度浮動小数点数を表現し，倍精度演算を用いて，4倍精度演算を行う．同様の手法は1971年のDekkerの論文 [3] で見られるが，近年ではQD [4] をはじめとするBaileyらによる実装例が知られている．QDはCPU用の4倍・8倍精度演算ライブラリで，DD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倍精度演算を行う．またHidaらはDD演算を使用したCPU用のBLAS実装として，XBLAS [5] を実装している．このほかQDライブラリなどの既存の拡張精度・多倍長演算ライブラリを用いた中田によるMBLAS [6] など，DD演算を用いたライブラリやアプリケーションが複数開発されている．さらに，Powerアーキテクチャ CPU向けのIBM社製コンパイラやgccなどにおいてDD演算を用いた4倍精度演算に対応するものがある．

GPU上では，Göddekeら [7] が倍精度演算に対応していないGPUに対して，DD演算と同様の手法で単精度浮動小数点数を2個連結する方式を用いて倍精度浮動小数点数を表現し，単精度浮動小数点演算を用いて倍精度演算をエミュレートしている．Thall [8] も同様の方式による倍精度演算と，単精度浮動小数点数を4個用いて4倍精度浮動小数点数を表現する手法で，4倍精度演算をGPU上に実装している．またLuら [9] はQDのGPU版であるGQDを実装している．さらに，DD演算を用いたGPUにおける4倍精度行列積の実装例として，後述する我々の研究のほかに，Nakasato [10]，中田ら [11]，中村ら [12] の研究がある．

一方で3倍精度演算として，1960年代に48 bitワードを3個利用して3倍精度演算を実現した事例 [13] があるほか，村上の文献 [14] 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．しかし近年のx86プロセッサやGPUにおいて，3倍精度演算をソフトウェアで実現した事例は見当たらない．

なお，我々はこれまでにDD演算を用いた拡張精度

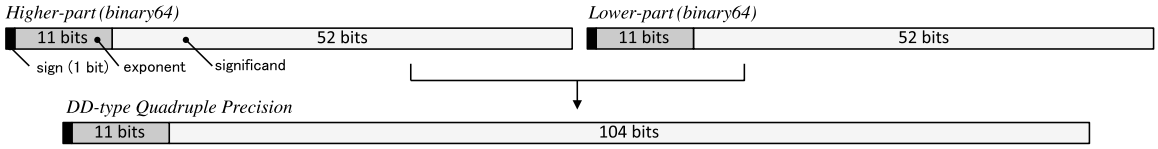


図 1　DD 型 4 倍精度浮動小数点型
Fig. 1　DD-type quadruple precision floating-point format.

AXPY・GEMV・GEMM の実装と評価を GPU 上で行ってきた．これまでの報告として，GT200 アーキテクチャの GPU における 4 倍精度 BLAS ルーチンの実装と評価 [15]，GF100 アーキテクチャ GPU における 4 倍・8 倍精度 BLAS ルーチンの性能評価 [16]，および GF100 アーキテクチャ GPU における 3 倍精度演算の検討 [17] も参照されたい．

3. 4 倍精度浮動小数点演算

はじめに GPU における Double-Double 型 4 倍精度演算（DD 演算）を用いた 4 倍精度浮動小数点演算の実装について示す．GPU における DD 演算の実装例は複数報告されているが，本論文では DD 演算をベースとした 3 倍精度演算を提案し，4 倍精度演算との比較を行うため，本章で改めて GPU における DD 演算の概要と実装を説明する．

3.1 DD 型 4 倍精度演算

DD 演算は double 型の浮動小数点型フォーマットにおける指数部・仮数部を 4 倍精度演算にそのまま利用し，分岐のない倍精度演算の組合せで実現できる．そのため GMP [18] などの整数演算を用いた高精度浮動小数点演算手法のように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，高い倍精度演算を持つ GPU に適している．

図 1 に DD 演算における 4 倍精度浮動小数点型（DD 型）の構造を示す．DD 型では 4 倍精度浮動小数点数 a を 2 つの倍精度浮動小数点数（IEEE 754-1985 binary64）a_{hi} と a_{lo} によって $a = a_{hi} + a_{lo}$（$a_{lo} <= 0.5\,\mathrm{ulp}(a_{hi})$）と表す．binary64 は仮数部が 52 bit（ケチ表現により実際には 53 bit 相当）であり，DD 型の仮数部は 104 bit（同様にケチ表現で 106 bit 相当）で，十進では約 32 桁の精度となる．一方で，指数部は binary64 と同様の 11 bit となる．

DD 演算では 4 倍精度浮動小数点数 a（$a = a_{hi} + a_{lo}$），b（$b = b_{hi} + b_{lo}$）どうしの計算を，2 桁の筆算の原理で計算する（**図 2**）．演算はすべて倍精度演算を用いて行われる．細部が異なるアルゴリズムが複数提案されているが基本原理は同じであり，本論文では QD ライブラリ [4] のアルゴリズムに従った．図 2 において $e(a_{hi} + b_{hi})$，$e(a_{hi} \times b_{hi})$ はそれぞれ $a_{hi} + b_{hi}$，$a_{hi} \times b_{hi}$ の倍精度演算で生じた丸め誤差を表している．また，最後に行われる Normalization（正規化）は，演算結果が $a_{lo} <= 0.5\,\mathrm{ulp}(a_{hi})$ を満たすようにするために行われる．なお QD ライブラリでは DD 加算において，106 bit の精度を保証するアルゴリズムと，

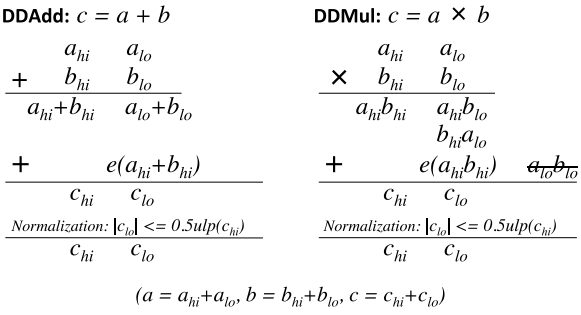


図 2　DD 演算による加算・乗算
Fig. 2　DD-operations (Addition and multiplication).

106 bit 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} と b_{lo}）どうしの加算時にキャリが発生した場合，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．本論文では速度を優先するため Sloppy アルゴリズムを用いる．

3.2 GPU における DD 演算の実装

CUDA における DD 演算の実装について述べる．DD 型は CUDA の組み込みベクトル型である double2 型に格納し，我々はこれを typedef によるエイリアスで cuddreal 型と定義した．double2 型は double 型 2 つからなる構造体であり，メンバ変数 x, y で各要素にアクセスできる．x に DD 型の上位部，y に下位部を格納した．CUDA では 128 bit のメモリアクセス命令がサポートされており，double2 型は 1 命令（中間言語 PTX の ld.global.v2.f64 など）でロード・ストアを行うことができる．

BLAS ルーチンの実装を行うために，DD 乗算と DD 加算を GPU のデバイス関数として実装した．関数呼び出しのオーバヘッドを抑制するためにデバイス関数を `__forceinline__` で宣言し，コンパイル時のインライン展開を抑制した．また，コンパイラによる乗算・加算の FMA 化で DD 演算アルゴリズムが意図せず最適化されることを抑制するために，すべての倍精度演算命令は組み込み関数である `__dadd_rn`（倍精度加算），`__dmul_rn`（倍精度乗算），`__fma_rn`（倍精度 FMA）を用いて記述した．

実際に実装したデバイス関数を示す．**図 3**（TwoSum [19]）は丸め誤差を考慮した倍精度加算，すなわち $a+b$ の浮動小数点演算結果 s と，発生した丸め誤差 e を計算する．**図 4**

```
__device__ __forceinline__ void TwoSum
    (double a, double b, double &s, double &e){
    double v;
    s = __dadd_rn(a, b);
    v = __dadd_rn(s, -a);
    e = __dadd_rn(__dadd_rn
        (a, -__dadd_rn(s, -v)), __dadd_rn(b, -v));
}
```

図 3　TwoSum の実装
Fig. 3　Implementation of TwoSum.

```
__device__ __forceinline__ void QuickTwoSum
    (double a, double b, double &s, double &e){
    s = __dadd_rn(a, b);
    e = __dadd_rn(b, -__dadd_rn(s, -a));
}
```

図 4　QuickTwoSum の実装
Fig. 4　Implementation of QuickTwoSum.

```
__device__ __forceinline__ void DDAdd
    (cuddreal a, cuddreal b, cuddreal &c){
    cuddreal t;
    TwoSum(a.x, b.x, t.x, t.y);
    t.y = __dadd_rn(t.y, __dadd_rn(a.y, b.y));
    QuickTwoSum(t.x, t.y, c.x, c.y);
}
```

図 5　DDAdd の実装
Fig. 5　Implementation of DDAdd.

```
__device__ __forceinline__ void TwoProdFMA
    (double a, double b, double &p, double &e){
    p = __dmul_rn (a, b);
    e = __fma_rn (a, b, -p);
}
```

図 6　TwoProdFMA の実装
Fig. 6　Implementation of TwoProdFMA.

(QuickTwoSum [3]) も同様に $a+b$ 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アルゴリズムであり，演算結果の正規化 ($a_{lo} <= 0.5\,\mathrm{ulp}(a_{hi})$) に用いる．これらを用いた DD 加算 (DDAdd [4]) を**図 5** に示す．

一方，DD 乗算では，**図 6** に示す丸め誤差を考慮した倍精度乗算 (TwoProdFMA [20]) を用いる．この関数は $a \times b$ の浮動小数点演算結果 p と，発生した丸め誤差 e 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 bit で保持する，倍精度 Fused-Multiply Add (FMA) 命令を用いる．FMA 命令を使用しない場合，より複雑な計算が必要となるが，本論文で用いる Tesla M2090 をはじめとする多く GPU は，この FMA 命令をサポートしている．最後に DD 乗算 (DDMul [4]) を**図 7** に示す．

3.3　GPU における DD 演算の理論ピーク演算性能

Tesla M2090 における DD 演算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，1.30 [GHz] × 16 [SM] × 32 [CUDACore] × (2 [Flop]/2 [Cycle]) = 665.6 [GFlops] と計算できる．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．したがって，この理論ピーク演算性能はすべての計算が積和演算である場合の値である．

```
__device__ __forceinline__ void DDMul
    (cuddreal a, cuddreal b, cuddreal &c){
    cuddreal t;
    TwoProdFMA (a.x, b.x, t.x, t.y);
    t.y = __dadd_rn(t.y, __dadd_rn
        (__dmul_rn(a.x, b.y), __dmul_rn(a.y, b.x)));
    QuickTwoSum (t.x, t.y, c.x, c.y);
}
```

図 7　DDMul の実装
Fig. 7　Implementation of DDMul.

表 1　GPU における DD 演算に必要な倍精度演算命令数
Table 1　Double precision floating-point instruction counts for DD-type operations on GPUs.

	DDAdd	DDMul
Double Precision Add: `__dadd_rn`	11	5
Double Precision Mul: `__dmul_rn`	0	3
Double Precision FMA: `__fma_rn`	0	1
Sum	11	9

同様に，積和演算における DD 演算の理論ピーク演算性能を算出する．GPU における DD 加算・DD 乗算に必要な倍精度演算命令数を**表 1** に示す．DD 乗算＋ DD 加算の場合，必要な倍精度演算命令は 20 命令である．Tesla M2090 では倍精度演算は 2 サイクルで実行できる．したがって，実行サイクル数は 40 サイクルである．1 回の DD 演算を 1DDFlop と定義すると，積和演算における DD 演算の理論ピーク演算性能は，1.30 [GHz] × 16 [SM] × 32 [CUDACore] × (2 [DDFlop]/40 [Cycle]) = 33.28 [GDDFlops] となる．なお，1DDFlop の演算には平均 (11+10)/2 = 10.5Flop の倍精度演算が行われているため，33.28GDDFlops は倍精度演算に換算すると 33.28 [GQuadFlops] × 10.5 [Flop/DDFlop] = 349.44 [GFlops] であり，倍精度の理論ピーク演算性能の約半分である．これは DD 積和演算を構成する 20 命令中，FMA 命令が 1 回しか使用されないためである．

4.　3 倍精度浮動小数点演算

次に GPU における 3 倍精度浮動小数点演算の実現手法について説明する．我々は 2 種類の 3 倍精度浮動小数点数フォーマットと，前章で示した DD 演算を用いた 3 倍精度演算手法を提案する．

4.1　D+S 型 3 倍精度浮動小数点演算

我々は DD 型と同様の原理で，3 倍精度数を倍精度型

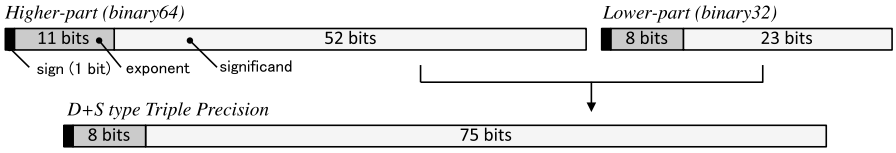


図 8　D+S 型 3 倍精度浮動小数点型
Fig. 8　D+S-type triple precision floating-point format.

と単精度型に格納する Double+Single（D+S）型 3 倍精度フォーマットを提案した [17]．この D+S 型では図 8 に示すように，3 倍精度数 a は倍精度数 a_{hi} と単精度数 a_{lo} によって $a = a_{hi} + a_{lo}$（$a_{lo} <= 0.5\,\mathrm{ulp}(a_{hi})$）と表す．仮数部は $52 + 23 = 75$ bit（ケチ表現で 77 bit 相当），十進で約 23 桁の精度である．一方で，指数部は a_{hi} において倍精度（binary64）と同様の 11 bit を格納することが可能であるが，$a = a_{hi} + a_{lo}$ と表現するため，a_{lo} の単精度（binary32）の指数部サイズである 8 bit の範囲でしか表現することができない．

D+S 型の演算手法として，我々は DD 演算と同様の原理で D+S 演算を検討した [17]．DD 演算はすべての演算に倍精度演算を用いるが，D+S 演算では単精度数で表現される D+S 型の下位部の計算において，一部に単精度演算を用いることができる．しかし次章で説明するように，D+S 演算は DD 演算より低速となることが分かっている．

4.2　GPU における 3 倍精度演算の実装

本論文でターゲットとする Tesla M2090 では，倍精度演算に比べて単精度演算を高速に実行できるため，我々は当初，D+S 演算が DD 演算より高速に実現できることを期待した．しかし，Tesla M2090 と同一アーキテクチャである Tesla C2050 上では，D+S 演算は DD 演算よりも最大約 1.26 倍低速となることが確認された [17]．これは D+S 演算では倍精度型と単精度型の型変換（キャスト）が多発し，このキャストに倍精度演算と同等の実行サイクル数を要するからである．

そこで我々は D+S 型のデータを DD 型に変換し，DD 演算を用いて計算する手法をとった．GPU のグローバルメモリ上にある D+S 型データを DD 型に変換してレジスタ・シェアードメモリに置き，レジスタ上の計算をすべて DD 演算で計算して，最終結果を D+S 型に変換してグローバルメモリ上に戻す（図 9）．この手法では DD 演算を用いるため，理論ピーク演算性能は DD 型を用いた場合と同等である．しかし我々は GPU の Byte/Flop 比に対して演算の Byte/Flop 比が大きい場合であれば，DD 演算においてもメモリ律速となるケースが存在することを示した [16]．DD 演算の実行レイテンシが D+S 型データへのアクセスレイテンシによって隠蔽される場合，DD 型の代わりに D+S 型を用いることで，データサイズの削減により，4 倍精度よりも高速な 3 倍精度演算が実現できると考えられる．

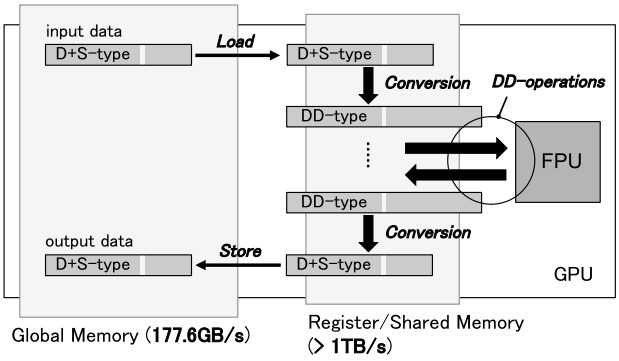


図 9　GPU における DD 演算を用いた 3 倍精度演算
Fig. 9　Triple precision operations using DD-operations on GPUs.

Tesla M2090 のレジスタのバンド幅は明らかにされていないが，オンチップメモリであるシェアードメモリのバンド幅は Tan らの論文 [21] を参考に計算すると 1331.2 GB/s であり，レジスタは少なくともこれ以上の速度があると考えられる．一方でオフチップメモリであるグローバルメモリのバンド幅は公称の理論ピーク値で 177.6 GB/s である．したがって，演算器からグローバルメモリ上のデータへのアクセス時間において支配的となるのはグローバルメモリへのアクセスレイテンシである．この方式では，レジスタ・シェアードメモリ上では DD 型を用いるため，これらの容量を圧迫する欠点はあるが，グローバルメモリ上で D+S 型を用いることにより，グローバルメモリへのアクセス時間とメモリスペースを節約することができる．その結果，3 倍精度演算に要する時間を最小限にとどめることができる．

4.3　D+I 型 3 倍精度浮動小数点演算

D+S 型では指数部の大きさが，下位桁を格納する 32 bit 単精度型（binary32）の 8 bit に制約される．表現できる数の範囲が単精度と同等であるため，たとえば倍精度を必要としていた問題を高精度化したいケースでは，指数部不足によるオーバフロー・アンダフローが発生する可能性がある．仮数部を倍精度よりも大きくする以上，少なくとも倍精度と同等の指数部サイズを必要とするケースが存在すると考えられる．そこで，前述の D+S 型の計算のように演算には DD 演算を利用することを前提として，64 bit 倍精度型（binary64）および DD 型と同等の 11 bit の指数部を確保する手法を検討した．

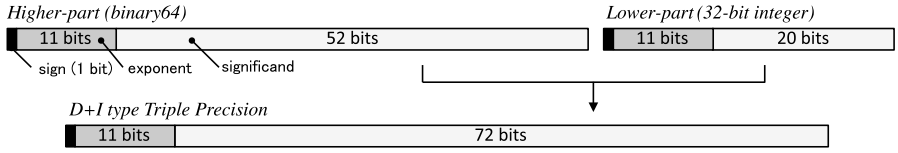


図 10　D+I 型 3 倍精度浮動小数点型
Fig. 10　D+I-type triple precision floating-point format.

我々は DD 型の下位データが格納されている 64 bit 倍精度型の符号部（1 bit）＋指数部（11 bit）を 32 bit 整数型（int 型）に格納し，余りの 20 bit に下位データが格納されている 64 bit 倍精度型の仮数部 20 bit 分を格納することにした（**図 10**）．本論文ではこの 3 倍精度型フォーマットを Double+Int（D+I）型と呼ぶ．この D+I 型において，int 型部に格納された値には整数としての意味は存在せず，単に double 型の符号部，指数部と仮数部上位 20 bit 分のビット列を格納しただけである．D+I 型ではトータルの仮数部ビット数が 72 bit（ケチ表現で 74 bit 相当）となり，D+S 型と比べて 3 bit 減少する．しかし，指数部は 64 bit 倍精度（binary64）や DD 型と同じ 11 bit が確保される．

D+I 型の演算には D+S 型と同様の方法で DD 演算を用いるが，D+I 型と DD 型の変換が必要である．変換は C 言語の共用体機能を用いて，double 型と 64 bit 整数型を関連づけ，シフト操作で行う．すなわち，DD 型から D+I 型への変換は，まず下位の double 型を 64 bit 整数型として参照し，右シフト操作で元の double 型の仮数部 32 bit 分を捨て，32 bit 整数型に格納する（**図 11**）．また D+I 型から DD 型への変換では，まず下位の 32 bit 整数型を 64 bit 整数型に代入し，32 bit 分左シフト操作を行い，共用体で double 型として格納する（**図 12**）．この変換操作は双方向とも 1 回の 64 bit 整数シフトと 1 回の 32 bit–64 bit 整数型の型変換で済む．

なお，この DD 型から D+I 型への変換では，単純に仮数部下位桁を切り捨てており，これは 0 への丸めに相当する．本論文が対象とする GPU は IEEE 準拠の 4 つの丸めモードをサポートしており，標準では最近接偶数丸めが使用されるため，DD 演算および DD 型から D+S 型への型変換には最近接偶数丸めが用いられている．最近接偶数丸めで許容される誤差量は 0.5 ulp 以内であるが，0 への丸めでは 1 ulp 以内であるため，D+I 型では実際の仮数部長で表現できる値に対して許容される誤差量が D+S 型と比べて大きいといえる．丸め誤差を減らすために最近接偶数丸めを行う処理を追加すべきであるが，変換アルゴリズムが複雑になるため，環境によっては丸め処理が性能上のボトルネックとなることも考えられる．最近接偶数丸めを行う手法を検討した結果，64 bit 整数型の論理積 1–2 回，比較 1–3 回，および 32 bit 整数の加算 0–1 回で実現できる手法があるが，本論文では D+I 型の最もシンプルな実装方法として，0 への丸めを行った場合を取り上げる．

```
union double_int64{
    int64_t int64_;
    double double_;
};

__host__ __device__ __forceinline__ void dd_to_di
    (cuddreal dd, double &d, int32_t &i){
    int64_t l;
    union double_int64 u;
    u.double_ = dd.y;
    l = u.int64_ >> 32;
    d = dd.x;
    i = (int32_t)l;
}
```

図 11　共用体と DD 型から D+I 型への変換
Fig. 11　Union and conversion from DD-type to D+I-type.

```
__host__ __device__ __forceinline__ void di_to_dd
    (double d, int32_t i, cuddreal &dd){
    union double_int64 u;
    int64_t l;
    l = (int64_t)i;
    u.int64_ = l << 32;
    dd.x = d;
    dd.y = u.double_;
}
```

図 12　D+I 型から DD 型への変換
Fig. 12　Conversion from D+I-type to DD-type.

4.4　グローバルメモリにおける 3 倍精度ベクトルデータの格納

科学技術計算ではベクトルデータを扱うことが多く，特に GPU はベクトルデータの計算に特化したアーキテクチャを持つ．一方で，3 倍精度データをグローバルメモリに格納する際には，効率的なメモリアクセスを実現するために注意すべき点がある．

GPU 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 の場合，128 Byte 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．4 倍精度の場合，double2 型に格納されている DD 型は 16 Byte データであるため，double2 型の配列は 128 Byte のメモリアライ

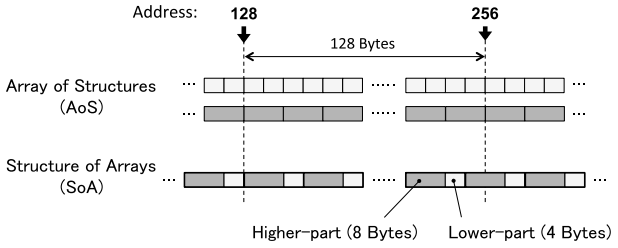


図 13　AoS レイアウトと SoA レイアウト
Fig. 13　Array of structures (AoS) layout and Structure of Arrays (SoA) layout.

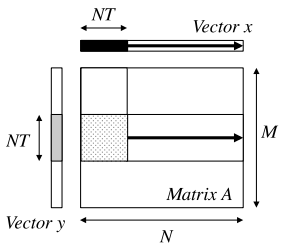


図 14　GEMV カーネル
Fig. 14　GEMV kernel.

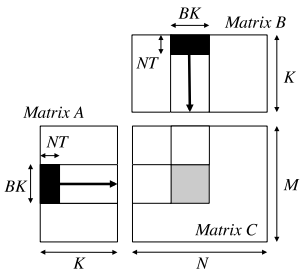


図 15　GEMM カーネル
Fig. 15　GEMM kernel.

ンメントを満たすことができる．一方，3 倍精度では D+S 型，D+I 型の 3 倍精度データが 12 Byte であるため，D+S 型または D+I 型の構造体を用いて配列を確保した場合，128 Byte 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 3 倍精度ベクトルデータをグローバルメモリ上に確保する際に，8 Byte 型で定義される上位部の配列と，4 Byte 型で定義される下位部の配列に分けて配置すべきである．この方式を，Structure of Arrays（SoA：配列の構造体）レイアウトと呼ぶ．一方で，D+S 型・D+I 型の構造体の配列を用いて 3 倍精度ベクトルデータを配置する方式を Array of Structures（AoS：構造体の配列）レイアウトと呼ぶ．**図 13** に AoS レイアウトと SoA レイアウトの概要を示す．

我々は Tesla M2090 と同一アーキテクチャである Tesla C2050 上の D+S 型において，メモリ律速となるケースで，SoA レイアウトが AoS レイアウトに対して最大約 1.3 倍高速であることを確認した [17]．このため，次章で示す BLAS の実装ではグローバルメモリにおける D+S 型および D+I 型の格納に，SoA レイアウトを用いた．

5. GPU における BLAS への適用と性能評価

これまでに示した 3 倍・4 倍精度演算の性能を示すために，BLAS ルーチンを実装して性能評価を行う．ベクトルどうしの演算（Level 1 BLAS），行列–ベクトル演算（Level 2 BLAS），行列–行列演算（Level 3 BLAS）の代表的なルーチンとして，以下の 3 ルーチンを実装した．

- Level 1 BLAS：AXPY（$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$）
- Level 2 BLAS：GEMV（$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$）
- Level 3 BLAS：GEMM（$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$）

本章では BLAS の実装と Byte/Flop 比を用いた性能予測，そして性能評価結果について述べる．性能評価では 3 倍・4 倍精度の各ルーチンの絶対性能を示すとともに，単精度・倍精度ルーチンに対する相対性能についても言及する．

5.1　3 倍・4 倍精度 BLAS の実装

実装した BLAS ルーチンは行列の転置や，ベクトルのストライドアクセスの特殊ケースは実装しておらず，条件分岐でスキップするようにした．それ以外は BLAS の仕様に準拠した実装を行った．BLAS カーネルの実装手法は単精度・倍精度の場合と同様の一般的な手法である．倍精度のカーネルをベースに，型を DD 型，D+S 型，D+I 型に置き換えるとともに，乗算と加算を DD 演算に置き換えた．また，D+S 型，D+I 型では前述の SoA レイアウトを用いたが，グローバルメモリからレジスタへのロード時に DD 型への変換を行い，レジスタ・シェアードメモリ上では AoS 形式の DD 型で保持して，DD 演算を行うようにした．

表 2　評価環境
Table 2　Evaluation environment.

CPU	Intel Xeon E5-2670 (2.6 GHz, 8-core) ×2
GPU	NVIDIA Tesla M2090 (6 GB, GDDR5, ECC-enabled)
GPU Bus	PCI-Express 2.0 x16
OS	Red Hat Enterprise Linux Server 6.1 (2.6.32-131.0.15.el6.x86_64)
CUDA	CUDA 4.1.28
Compiler	gcc 4.4.5 (-O3), nvcc 4.1 (-O3 -arch sm_20)

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 14**，GEMM のカーネルを**図 15** 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=8 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．また，GEMM ではグローバルメモリからの読み込み時にテクスチャキャッシュを使用した．

5.2　評価手法

評価環境を**表 2** に示す．入力に用いた行列およびベク

表 3　Tesla M2090 の理論ピーク演算性能と Byte/Flop・Byte/DDFlop 比

Table 3　Theoretical peak performance and Bytes/Flop and Bytes/DDFlop of the Tesla M2090.

	理論ピーク演算性能	Byte/Flop 比
単精度	1331.2 GFlops	0.088 Byte/Flop
倍精度	665.6 GFlops	0.176 Byte/Flop
DD 演算	33.28 GDDFlops	3.52 Byte/DDFlop

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．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・D+I 型については DD 型から型変換を行った．

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．実行時間はカーネル実行時間のみを測定し，PCI Express による GPU–CPU 間のデータ転送時間は含まれていない，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 DD 演算回数である DDFlops 値を算出した．4.2 節で述べたように，3 倍精度演算にも DD 演算を用いているため，3 倍・4 倍精度ルーチンともに DDFlops を用いて性能を示す．また，単精度・倍精度演算に対する相対性能を確認するために，CUDA Toolkit に含まれる CUBLAS 4.1 [22] の単精度・倍精度ルーチンの性能も測定した．

なお，3 倍・4 倍精度 BLAS の演算結果は MBLAS [6] による 8 倍精度演算の結果と比較を行い，計算結果に誤りがないことを確認している．

5.3　Byte/Flop 比による性能予測

我々の提案した 3 倍精度演算はメモリ律速となる場合に 4 倍精度演算より高速となる．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の性能予測を示す．BLAS ルーチンの Byte/Flop 比が GPU の Byte/Flop 比を上回る場合，性能はメモリ律速となる．メモリ律速である場合，性能はグローバルメモリへのロード・ストア時間に依存するため，3 倍・4 倍精度型のデータサイズに比例して，3 倍・4 倍精度演算の実行時間は単精度比で 3 倍，4 倍となる．

Tesla M2090 における各精度の理論ピーク演算性能と Byte/Flop 比を**表 3** に示す．DD 演算に対しては，1 秒間の DD 演算回数を DDFlops と表し，Byte/Flop 比に対して Byte/DDFlop で表した．Tesla M2090 のグローバルメモリの公称理論ピークメモリバンド幅は 177.6 GB/s（ECC 無効時）であるが，ECC を有効にした環境で AXPY においてメモリバンド幅を測定したところ，実測値の最大は約 112–117 GB/s であった．そこでメモリのバンド幅を 117 GB/s と仮定し，Tesla M2090 の理論ピーク演算性能を用いて Byte/Flop 比を計算した．

表 4　BLAS ルーチンの Byte/Flop・Byte/DDFlop 比（ベクトル：N，行列：$N \times N$）

Table 4　Bytes/Flop and Bytes/DDFlop of BLAS subroutines (Vector size: N, Matrix size: $N \times N$).

	AXPY	GEMV	GEMM
単精度 [Byte/Flop]	6	2	$8/N$
倍精度 [Byte/Flop]	12	4	$16/N$
D+S/D+I 型 3 倍精度 +DD 演算 [Byte/DDFlop]	18	6	$24/N$
DD 型 4 倍精度 +DD 演算 [Byte/DDFlop]	24	8	$32/N$

続いて，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，BLAS ルーチンの Byte/Flop・Byte/DDFlop 比を**表 4** に示す．これらの値は簡単のためにオーダの低い項を無視して計算しているほか，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る．例として 4 倍精度 GEMM の場合の算出法を示す．DD 型 1 要素は 16 Byte であるため，グローバルメモリに対するロード・ストア量を $4N^2 \times 16$ [Byte] と見なす．一方，GEMM では $2N^3 + 3N^2$ [DDFlop] の演算が行われる．簡単のために演算回数の $O(N^2)$ の項を無視すると，$(4N^2 \times 16)$ [Byte]$/2N^3$ [DDFlop] $= 32/N$ [Byte/DDFlop] となる．

表 4 において，Tesla M2090 の Byte/Flop 比より数字が大きい Byte/Flop 比を持つ BLAS ルーチンは，Tesla M2090 においてメモリ律速となる可能性が高い．すなわち，3 倍・4 倍精度ルーチンの実行時間が単精度比で 3 倍，4 倍となり，4 倍精度の代わりに 3 倍精度を用いる速度面での利点が生まれる可能性がある．

5.4　性能評価結果

5.4.1　AXPY（$y = \alpha x + y$）

AXPY の DDFlops 性能を**図 16** に示す．AXPY は 5.3 節に示した Byte/Flop 比による性能予測からすべての精度においてメモリ律速となると考えられる．3 倍精度ルーチンでは，D+S 型と D+I 型のグラフは重なり，ほぼ同一の性能を示した．AXPY は他の 2 ルーチンと比較して Byte/Flop 比が大きく，メモリのロード・ストア時間に対する BLAS の演算時間が小さいものの，DD 型–D+I 型の変換はボトルネックとならないことが示された．

次に，CUBLAS による単精度 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**図 17** に示す．グラフの右側では，単精度 AXPY に対して倍精度，3 倍精度，4 倍精度がそれぞれ 2 倍，3 倍，4 倍の実行時間に近づ

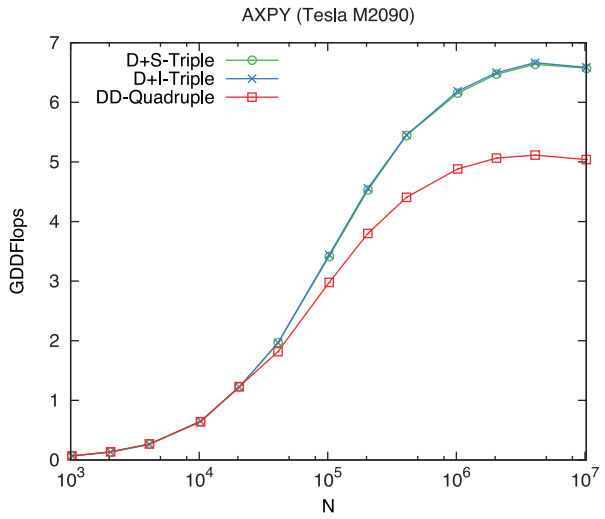


図 16　3 倍・4 倍精度 AXPY（$\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$）の性能（縦軸の DDFlops は 1 秒あたりの DD 演算回数を表す）

Fig. 16　Performance of triple and quadruple precision AXPY ($\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$) (DDFlops: DD-type floating-point operations per second).

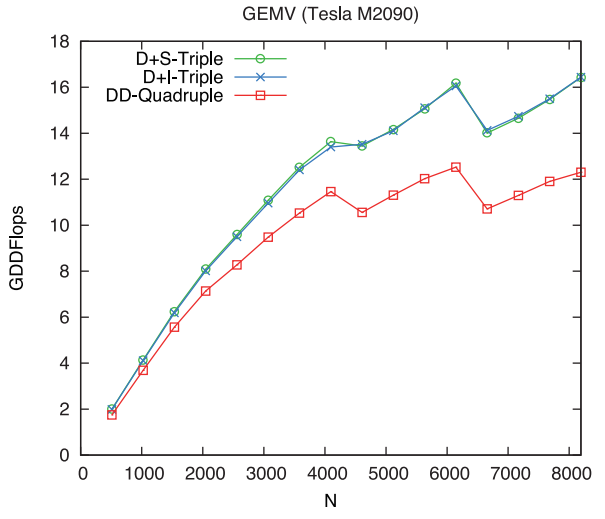


図 18　3 倍・4 倍精度 GEMV（$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$）の性能（縦軸の DDFlops は 1 秒あたりの DD 演算回数を表す）

Fig. 18　Performance of triple and quadruple precision GEMV ($\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$) (DDFlops: DD-type floating-point operations per second).

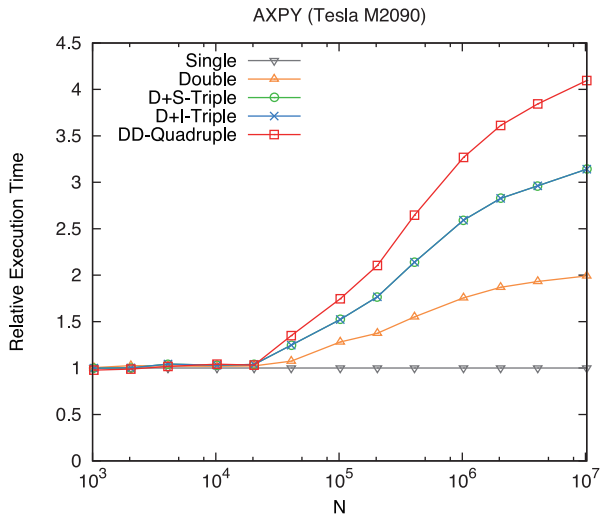


図 17　AXPY（$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$）の単精度ルーチンを基準とした相対実行時間

Fig. 17　Relative execution time of AXPY ($\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$) (the y-axis represents the relative execution time compared to the single precision subroutines).

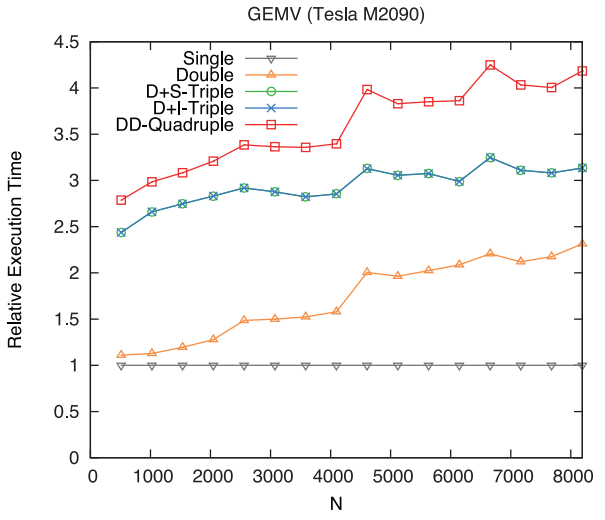


図 19　GEMV（$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$）の単精度ルーチンを基準とした相対実行時間

Fig. 19　Relative execution time of GEMV ($\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$) (the y-axis represents the relative execution time compared to the single precision subroutines).

いている．これらの性能差は単精度型に対して倍精度，3 倍精度，4 倍精度型がそれぞれ 2 倍，3 倍，4 倍のデータサイズであることに起因する．メモリ律速であるため，グローバルメモリへのロード・ストア時間に性能が依存した結果である．一方，$N < 10{,}000$ では各精度の性能差がほとんど見られない．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．

5.4.2　GEMV（$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$）

GEMV の DDFlops 性能を**図 18** に示す．また，単精度 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**図 19** に示す．AXPY と同様にメモリ律速であり，グラフの右側では単精度に対して倍精度，3 倍精度，4 倍精度がそれぞれ 2 倍，3 倍，4 倍の実行時間に近づき，3 倍精度演算の 4 倍精度演算に対する速度面での優位性が確認された．しかし，グラフの左側，$N < 4{,}000$ では，3 倍・4 倍精度の性能差が小さくなっている．このとき，4 倍精度の実行時間は単精度比で 4 倍未満であることや，単精度・倍精度の性能差も小さくなっていることから，AXPY と同様にカーネル生成コストが実行時間において支配的になっていると推測できる．一方で，N が小さくなるに従って，

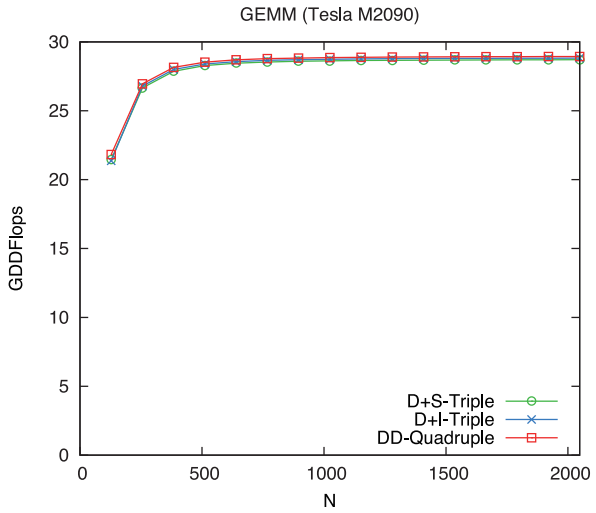


図 20　3 倍・4 倍精度 GEMM（$\boldsymbol{C} = \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$）の性能（縦軸の DDFlops は 1 秒あたりの DD 演算回数を表す）

Fig. 20　Performance of triple and quadruple precision GEMM ($\boldsymbol{C} = \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$) (DDFlops: DD-type floating-point operations per second).

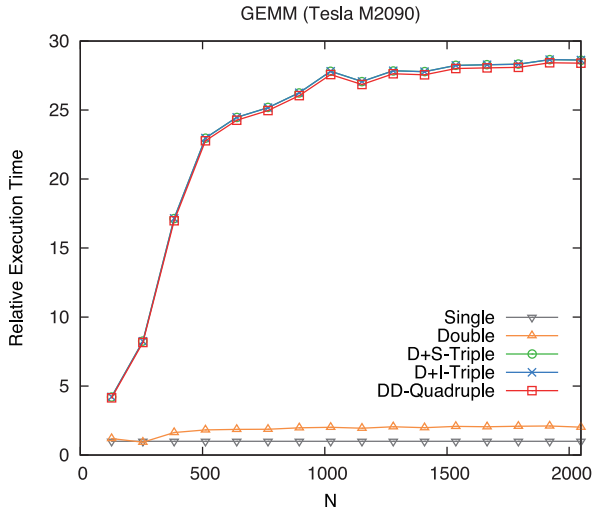


図 21　GEMM（$\boldsymbol{C} = \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$）の単精度ルーチンを基準とした相対実行時間

Fig. 21　Relative execution time of GEMM ($\boldsymbol{C} = \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$) (the y-axis represents the relative execution time compared to the single precision subroutines).

単精度・倍精度と 3 倍・4 倍精度の性能差が大きくなっている．$N = 1{,}024$ における理論ピーク演算性能に対する実行効率は，倍精度ルーチンが $N = 8{,}192$ に対して約 1/2.3 であるのに対して，4 倍精度ルーチンは約 1/3.3 に落ち込んでおり，我々の実装した GEMV カーネルが CUBLAS の実装と比べて，問題サイズが小さい場合に十分最適化できていないことが原因であると考えられる．

5.4.3　GEMM（$\boldsymbol{C} = \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$）

GEMM の DDFlops 性能を**図 20** に示す．GEMM はすべての精度において，5.3 節での Byte/Flop 比による性能予測から演算律速であると考えられる．演算律速であるため，DD 演算を使用した 3 倍精度 GEMM は 4 倍精度 GEMM と同じ性能になり，3 倍精度型を使用する速度面での優位性は得られない結果となった．

次に，単精度 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**図 21** に示す．GEMM は演算律速であるため，相対実行時間として，演算に要するサイクル数，すなわち理論ピーク演算性能の差が表れるはずである．表 3 に示すように，Tesla M2090 における単精度演算，倍精度演算，DD 演算の理論ピーク演算性能は 1 : 2 : 40 の比にあり，理論上，DD 演算は倍精度演算の 20 倍の計算コストを要することになる．しかし実際には 3 倍・4 倍精度 GEMM が倍精度 GEMM の約 14 倍の実行時間となった．これは我々の実装した 3 倍・4 倍精度 GEMM が，DD 演算の理論ピーク演算性能に対して 85%以上の実行効率を達成しているのに対して，CUBLAS の倍精度 GEMM の実行効率が約 61%（単精度 GEMM で約 62%）と低いためである．Tan らの論文 [21] では CUBLAS の倍精度 GEMM の実行効率が約 58%である理由を考察するとともに，より高速な倍精度 GEMM を実現しているが，そのチューニング手法はきわめて難易度が高いことが分かる．このように，理論的には倍精度の 20 倍の計算コストを要する DD 演算であるが，実際には倍精度のプログラムにおいて高い実行効率を得ることが難しいため，倍精度演算と比較した DD 演算の実際の計算コストは，理論的なコストよりも実際には小さくなる場合が多いと考えられる．

5.5　演算精度について

最後に，実際の計算における演算結果の一例として，GEMV と GEMM における各精度の演算結果の比較を示す．$N \times N$ の行列に対する MBLAS による 8 倍精度演算の結果と各精度の演算結果の相対誤差を求めた．入力ベクトルおよび行列は 0–1 の範囲の倍精度の一様乱数で初期化した．3 倍・4 倍・8 倍精度演算を行う際の入力データは仮数部 52 bit（倍精度）より下位をゼロで埋めたものであり，すべての精度の計算において同じ入力データを用いている．また本論文では D+I 型において，DD 型から D+I 型への変換時に 0 への丸めを採用してきたが，最近接偶数丸めを採用した場合の誤差量も示す．なお，倍精度，DD 型 4 倍精度では演算に最近接偶数丸めが用いられているほか，D+I 型・D+S 型ともに演算は最近接偶数丸めによる DD 演算で行われている．

結果を**表 5** に示す．DD 型からの変換時に 0 への丸めを採用した D+I 型は，D+S 型に対して約 16–17 倍の誤差量となった．一方で最近接偶数丸めを採用した場合，D+S 型に対する誤差量は約 8–9 倍となり，0 への丸めを行った場合と比べて誤差量は約半分となった．これは最近接偶数丸めでは許容される誤差量が 0.5 ulp 以内であるのに対して，

表 5　8 倍精度演算の結果に対する相対誤差（入力は 0–1 の倍精度一様乱数）
Table 5　Error relatives to the octuple precision results (input: double precision uniform random numbers between 0 and 1).

	GEMV		GEMM	
	$N = 100$	$N = 1{,}000$	$N = 100$	$N = 1{,}000$
倍精度	2.77e-16	4.60e-16	2.70e-16	7.83e-16
D+S 型 3 倍精度（DD 型からの変換：最近接偶数丸め）	8.20e-25	1.36e-24	8.75e-25	1.34e-24
D+I 型 3 倍精度（DD 型からの変換：0 への丸め）	1.41e-23	2.24e-23	1.40e-23	2.15e-23
D+I 型 3 倍精度（DD 型からの変換：最近接偶数丸め）	6.36e-24	1.16e-23	6.93e-24	1.07e-23
DD 型 4 倍精度	1.92e-32	6.57e-32	2.14e-32	6.45e-32

0 への丸めは 1 ulp 以内となるからである．したがって，本論文で示した D+I 型は D+S 型と比べて指数部ビット長が 3 bit 少ないほか，0 への丸めを用いた場合には最近接偶数丸めと比べて丸めモードの違いによる誤差の増大を考慮する必要がある．

6. まとめ

本論文では，GPU（NVIDIA Tesla M2090）において 3 倍・4 倍精度浮動小数点演算を実装し，線形計算への適用例として Level 1–3 の代表的な BLAS ルーチンを実装し性能評価を行った結果を示した．4 倍精度演算には既存手法である DD 演算を用いた．一方で 3 倍精度演算として，D+S 型・D+I 型の 2 種類の 3 倍精度データフォーマットを提案し，DD 演算を用いて 3 倍精度演算を行う手法を提案した．D+S 型は原理上，指数部ビット長が 32 bit 単精度と同じ 8 bit に制限される．D+I 型は指数部が 64 bit 倍精度（binary64）や DD 型と同じ 11 bit である一方で，その分 D+S 型と比べて仮数部が 3 bit 少ない．また，本論文で示した実装では丸めモードに 0 への丸めを用いたため，最近接偶数丸めと比べて丸めモードに起因する誤差量の増大が生じた．D+S 型と D+I 型は BLAS ルーチンにおける性能評価において性能差は見られないため，指数部と仮数部のトレードオフを考慮して，アプリケーションによって使い分けを検討すべきである．

Tesla M2090 における性能評価では，3 倍・4 倍精度の AXPY・GEMV がメモリ律速となり，その実行時間はデータサイズに比例して，単精度ルーチン比でおよそ 3 倍，4 倍（倍精度比でそれぞれ 1.5 倍，2 倍）となることを示した．4 倍精度は必要ないが倍精度では精度が不足する場合であれば，我々が提案した 3 倍精度演算は，3 倍精度データに対する DD 演算がメモリ律速となるケースにおいて，4 倍精度演算に対する速度面での利点が主張できる．一方で，我々の 3 倍精度演算は DD 演算を用いて計算を行っているため，理論ピーク演算性能は DD 型 4 倍精度演算と同等であり，演算律速の計算では速度面で 3 倍精度を用いるメリットはない．しかし GPU クラスタ環境では，GPU が高い浮動小数点演算性能を持つ一方で，グローバルメモリよりはるかに低速な PCI Express やネットワークの帯域が，アプリケーションにおいて性能のボトルネックとなることが多い．また GPU 上のメモリスペースが限られていることや，これが原因となって PCI Express を経由したホストとの通信が発生することもある．このような環境ではデータサイズの小さい 3 倍精度が 4 倍精度と比べて有効となるケースが存在すると考えられる．

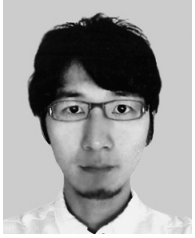
我々は今後，実アプリケーションへの応用例として，精度に敏感な疎行列に対する反復解法への適用例を示したいと考えている．疎行列計算はメモリ律速となりやすい処理であり，特に GPU クラスタ環境では，3 倍精度が有効となるケースが存在すると考えられる．

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] 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] IEEE: IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, pp.1–58 (2008).
[3] Dekker, T.J.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol.18, pp.224–242 (1971).
[4] Bailey, D.H.: QD (C++/Fortran-90 double-double and quad-double package), available from ⟨http://crd.lbl.gov/~dhbailey/mpdist/⟩.
[5] 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, available from ⟨http://www.netlib.org/xblas/⟩.
[6] Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), available from ⟨http://mplapack.sourceforge.net/⟩.
[7] 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).
[8] Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation, *ACM SIGGRAPH 2006 Research Posters* (2006).
[9] Lu, M., He, B. and Luo, Q.: Supporting Extended

Precision on Graphics Processors, *Proc. 6th International Workshop on Data Management on New Hardware* (*DaMoN 2010*) (2010).
[10] Nakasato, N.: A Fast GEMM Implementation On the Cypress GPU, *SIGMETRICS Perform. Eval. Rev.*, Vol.38, pp.50–55 (2011).
[11] 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).
[12] 中村光典，中里直人：OpenCL による四倍精度行列積の高速化，情報処理学会研究報告，Vol.2012–HPC–133, No.27, pp.1–8 (2012).
[13] Ikebe, Y.: Note on Triple-Precision Floating-Point Arithmetic with 132-bit Numbers, *Comm. ACM*, Vol.8, pp.175–177 (1965).
[14] 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, No.17, pp.1–3 (2010).
[15] Mukunoki, D. and Takahashi, D.: Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs, *Proc. 10th International Conference on Applied Parallel and Scientific Computing* (*PARA 2010*), Part I, Lecture Notes in Computer Science, No.7133, pp.249–259, Springer-Verlag (2012).
[16] 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).
[17] Mukunoki, D. and Takahashi, D.: Implementation and Evaluation of Triple Precision BLAS Subroutines on GPUs, *Proc. 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing* (*PDSEC-12*) (2012).
[18] Granlund, T. and the GMP development team: GMP: GNU Multiple Precision Arithmetic Library, available from ⟨http://gmplib.org/⟩.
[19] Knuth, D.E.: *The Art of Computer Programming Vol.2 Seminumerical Algorithms*, *3rd edition*, Addison-Wesley (1998).
[20] Karp, A.H. and Markstein, P.: High-Precision Division and Square Root, *ACM Trans. Math. Softw.*, Vol.23, pp.561–589 (1997).
[21] 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).
[22] NVIDIA Corporation: CUBLAS Library (included in CUDA Toolkit).



椋木 大地　（学生会員）

1985 年生．2006 年岐阜工業高等専門学校電子制御工学科卒業．2009 年筑波大学図書館情報専門学群卒業．2011 年同大学大学院システム情報工学研究科博士前期課程修了．修士（工学）．現在，同研究科博士後期課程在学中．GPU における拡張精度演算に関する研究に従事．

高橋 大介　（正会員）

1970 年生．1991 年呉工業高等専門学校電気工学科卒業．1993 年豊橋技術科学大学工学部情報工学課程卒業．1995 年同大学大学院工学研究科情報工学専攻修士課程修了．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退．同年同大学大型計算機センター助手．1999 年同大学情報基盤センター助手．2000 年埼玉大学大学院理工学研究科助手．2001 年筑波大学電子・情報工学系講師．2004 年同大学大学院システム情報工学研究科講師，2006 年同助教授，2007 年同准教授．2011 年同大学システム情報系准教授，2012 年同教授．博士（理学）．並列数値計算アルゴリズムに関する研究に従事．1998 年度情報処理学会山下記念研究賞，1998 年度，2003 年度情報処理学会論文賞各受賞．日本応用数理学会，ACM，IEEE，SIAM 各会員．