

コンシューマ向けGPUを用いた 3倍精度(Triple-Single)行列積の性能評価

打桐大雅¹ 幸谷智紀¹

概要: 標準的な浮動小数点数である倍精度型(double 型, binary64)や単精度型(single 型, binary32)より長い桁数を持つ多倍長浮動小数点数の実装方式の一つにマルチコンポーネント方式があり, 代表的な実装としては Bailey らによって開発された QD ライブラリがある. この QD ライブラリを NVIDIA GPU 上で実装したものとしては, Lu らによって開発された GQD がある. 一般にコンシューマ向け GPU では, 倍精度浮動小数点数演算は単精度浮動小数点数演算よりも多くの計算時間を要することから, 我々はマルチコンポーネント方式に基づき, single 型を 3 つ繋げた 3 倍精度浮動小数点数(Triple-Single, TS 型)と GPU で実装されているベクトル型の float3 を用いた 3 倍精度浮動小数点数(GTS 型)の実装を行った. 本論文では, それらを用いてブロック化した行列積と, 行列の各要素を上位と下位に分割して単精度行列積を組み合わせて TS 型行列積の計算を行う尾崎スキームの計算時間をコンシューマ向け GPU 上で比較し, 性能評価を行ってその結果を示す.

キーワード: 3 倍精度浮動小数点数 TS 型 GPU cuBLAS 尾崎スキーム 行列積

1. はじめに

近年 GPU の計算能力が飛躍的に向上しており, 様々な分野で GPU 計算が用いられている. ただし, 高速な IEEE754 倍精度演算(binary64, double 型)はハイエンド HPC 向けの GPU でしか使用できない. 例を挙げると, NVIDIA [1] 社の VG100 [2] があり, これは単精度演算性能が 14.8TFLOPS, 倍精度演算性能が 7.4TFLOPS で, 倍精度演算は単精度の 1/2 程度の性能差に収まっている. そのほかにも, AMD [3] 社の MI60 [4] があり, これは単精度演算性能が 14.7TFLOPS で倍精度演算性能が 7.4TFLOPS で, これも同様に倍精度演算性能は単精度の 1/2 程度の性能差である.

しかし, コンシューマ向けの GPU では倍精度演算の能力が極端に低くなっている. これは, コンシューマ向け GPU は主に映像処理を目的としており, 倍精度演算の能力は必要としていないためである. 例を挙げると, NVIDIA 社の GeForce RTX 20 シリーズ [5] があり, このシリーズの GPU では倍精度演算性能と単精度の差が 1/32 となっている. さらに GeForce RTX 30 シリーズ [6] では, 倍精度演算性能と単精度の差は 1/64 となり倍精度演算と単精度演算の性能の差は開くばかりである.

一方, 単精度や倍精度などの標準的な浮動小数点数は保証できる精度に限界がある. しかし, 科学技術の発展により倍精度程度では精度が足りなくなる問題が種々出てきている. 一方, 2008 年に IEEE754 が改定され, IEEE754-2008 [7] では binary32 と binary64 以外にも 4 倍精度(binary128)が正式に提案されたが, 4 倍精度は現状で実装されているものは少ない. さらに, 倍精度と 4 倍精度では精度や速度に大きな差がある.

倍精度型や単精度型より長い桁数を持つ多倍長浮動小数点数の実装方式の一つにマルチコンポーネント方式があり, 代表的な実装としては Bailey らによって開発された QD ライブラリ [8] があり, この QD ライブラリを NVIDIA GPU 上で実装したものとしては, Lu らによって開発された GQD [9] がある. 現状では CPU 上でも GPU 上でも, binary128 より QD や GQD がサポートする DD 型(Double-Double)の方が高速な演算を実現できており, SIMD 命令や CUDA を用いた並列化も容易である. しかしながら, これらのライブラリでは DD 型, QD 型(Quad-Double)のみ提供され, 中間の計算精度である, 椋木らが提唱する D+S 型や, 幸谷が実装した TD 型 [10] [11] に相当する実装はない.

本論文では QD ライブラリの実装方法に基づく TS 型(Triple-single)の実装をコンシューマ向け GPU 上で行い, 近年注目されている尾崎スキームに基づく実装と比較してその結果を示す. TS 型は, 科学技術計算では標準的な倍精度型より仮数部の桁数が確保されているため, 倍精度計算結果の検証用として, 他の多倍長精度型よりも高速に実行できることが期待できる. また, 単精度演算のみで実現できることから, 倍精度演算を用いる D+S 型よりも, 安価なコンシューマ向け GPU 上では高いパフォーマンスが期待できる.

本論文では以下, 2 章で本研究における関連研究を紹介したあとに, 3 章において 3 倍精度浮動小数点数について紹介し, 4 章において 3 倍精度浮動小数点における尾崎スキームを紹介する. 5 章では, GPU を用いてブロック行列積や尾崎スキームを実装し性能評価を行う. 最後に 6 章にてまとめと今後の課題について述べる.

¹ 静岡理科大学
Shizuoka Institute of Science and Technology

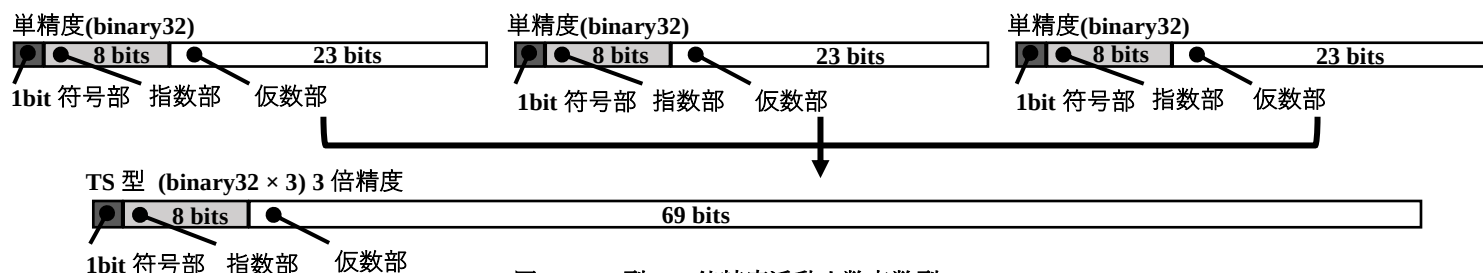


図 1 TS 型の 3 倍精度浮動小数点数型

2. 関連研究

まず浮動小数点数の仮数部を拡張するために開発された多倍長精度浮動小数点ライブラリのうち、本論文に関連するものを紹介する。有名なものとして、浮動小数点型 (binary64) を用いてマルチコンポーネント方式であらわした QD ライブラリ [8] が存在する。QD ライブラリは Bailey らによって開発・実装された CPU 用の 4 倍・8 倍精度浮動小数点ライブラリである。DD 型 (Double-Double) と QD (Quad-Double) 型のデータ型から構成される C++ クラスライブラリで、DD 型は double 型 (binary64) を 2 個用いて 4 倍精度浮動小数点数を表現し、QD 型は double 型を 4 個用いて 8 倍精度浮動小数点数を表現している。

GPU で利用できる多倍長精度浮動小数点ライブラリとしては、QD ライブラリの GPU 版として、GQD が Lu [9] らによって開発された。Lu らは NVIDIA 社が開発・提供している CUDA [12] [13] を用いて GQD を実装している。QD 型は GQD 型となり、GQD 型ではベクトル型として double4 型で表現されている。DD 型は GDD 型となり GDD 型ではベクトル型として double2 型として表現されている。単精度演算に基づく方式では椋木ら [14] によって DF (Double-Float) 型が開発されている。椋木らは DF 型を用いて疑似倍精度を再現している。

3 倍精度演算の開発としては、近年では椋木ら [15] [16] によって開発された Double+Single 型 (D+S 型) や Double+Int 型 (D+I 型) が存在する。最適化された 3 倍精度演算方法としては Fabiano らの Triple-word 演算があるが [17]、実装した結果、我々の環境下では優位性が認められなかったことから、簡易な QD 演算に基づくものを採用した。

一方、高精度な行列積の計算法として、行列を各要素で分割した行列にしてから行列積を計算する手法 [18] [19] として尾崎スキームが知られている。GPU で実装した例として七井ら [20]、椋木らによって示されたものがある。七井らは倍精度の行列を単精度の行列に分割して疑似的に倍精度演算を行い、GPU での高速化を図っている。

3. 3 倍精度 (TS 型) 浮動小数点数

今回作成する 3 倍精度 (TS 型) の概略図を図 1 に示す。TS 型は単精度 (binary32) を 3 つ使用して構成されている。

浮動小数点数の精度は仮数部の長さで決まる。単精度の仮数部は 23bit (ケチ表現で 24bit) となり、精度は十進数で約 7 桁である。今回作成した、TS 型は仮数部が $23+23+23=69$ (ケチ表現 72bit) となり精度は、約 21 桁となる。指数部は、倍精度では 11bit あるが単精度では 8bit しかない。TS 型は単精度をもとにして作成しているため指数部は 8bit の範囲でしか表現することができない。

本章では初めに、CPU における Triple-Single 型 (TS 型) 3 倍精度浮動小数点数演算の実装について示す。その後、CPU での実装を生かしつつ GPU での GTS 型 3 倍精度浮動小数点数の実装について示す。

3.1 CPU における 3 倍精度 (TS 型) 演算の実装

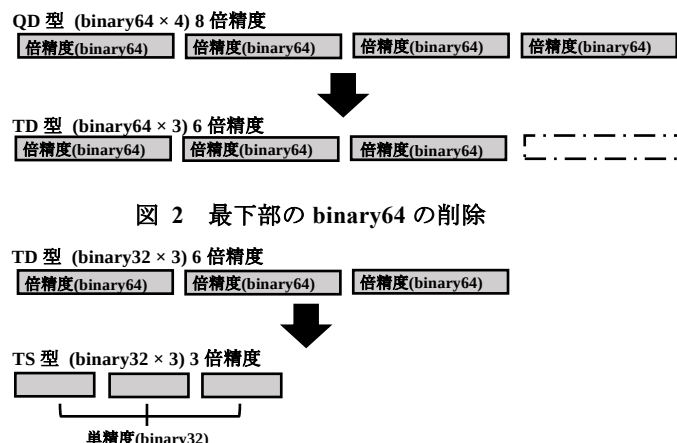


図 2 最下部の binary64 の削減

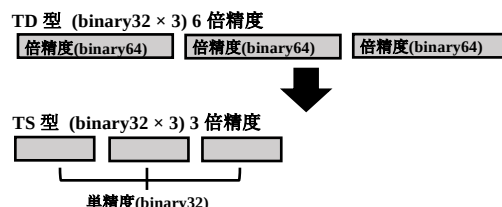


図 3 binary64 から binary32 への変換

CPU における TS 型の実装について述べる。今回の TS 型は QD ライブラリを参考にして作成を行った。まず図 2 より QD 型 (binary64 x 4) の binary64 を 1 つ削った形の TD 型を作製した。次に、図 3 より binary64 から binary32 への変換し TS 型を作成した。

以下、これらの関数について詳細を述べる。今回は行列積の実装をするため、主に TS 乗算と TS 加算の実装を行った。関数の呼び出しによるオーバーヘッドを減少させるために inline で宣言を行った。実際に実装したコードを図 4-10 に示す。

Two_Sum (図 4) は単精度の丸め誤差を考慮した加算である。 $a + b$ の演算結果を s に格納する。 err に $a + b$ で発生した丸め誤差を格納する。Quick_Two_Sum (図 5) は前述と同様に単精度の丸め誤差を考慮した加算であるが、 $|a| \geq |b|$ が成立する場合のみ使用することができる。

Two_Sum を複数組み合わせることで 3 つの浮動小数点数の加算を行う Three_Sum (図 6) を実装し a , b , c の値を入れ替えながら上位部を a , 中位部を b , 下位部を c へと再編成することができるアルゴリズムを実装した. 4 つの単精度の値を TS 型に丸めるために Renorm (図 7) を実装した. Two_Sum と Three_Sum, Renorm など組み合わせることにより TS 型同士を加算する TS_Add (図 8) を実装した.

一方で TS 乗算では, 従来の QD ライブラリでは FMA 命令が実装されていなかったため, 単精度の丸め誤差を考慮した乗算である Two_Prod_FMA (図 9) を実装した. これは, $a \times b$ の浮動小数点の結果を p に格納し, 発生した丸め誤差を FMA 命令を用いて $(a \times b) - p$ を行い err に格納する. FMA 命令を使用しない場合は多くの計算を必要とするが, 現状では FMA 命令が多くの CPU や GPU で使用可能であることから, 今回はこの実装のみを使用することとした. これらの演算を用いて TS 同士を乗算する TS_Prod (図 10) を実装した.

```
inline float Two_Sum(float a, float b, float &err){
    float s, bb;
    s = a + b;
    bb = s - a;
    err = (a - (s - bb)) + (b - bb);
    return s;
}
```

図 4 CPU での TS 型の Two_Sum の実装

```
inline float Quick_Two_Sum(float a, float b, float &err){
    float s;
    s = a + b;
    err = b - (s - a);
    return s;
}
```

図 5 CPU での TS 型の Quick_Two_Sum の実装

```
inline void Three_Sum(float &a, float &b, float &c){
    float t1, t2, t3;
    t1 = Two_Sum(a, b, t2);
    a = Two_Sum(c, t1, t3);
    b = Two_Sum(t2, t3, c);
}
```

図 6 CPU での TS 型の Three_Sum の実装

```
inline void Renorm(float &c0, float &c1, float &c2, float &c3){
    float s0, s1, s2 = 0.0;
    s0 = Quick_Two_Sum(c2, c3, c3);
    s0 = Quick_Two_Sum(c1, s0, c2);
    c0 = Quick_Two_Sum(c0, s0, c1);
    s0 = c0; s1 = c1;
    if(s1 != 0.0){
        s1 = Quick_Two_Sum(s1, c2, s2);
        if(s2 != 0.0)
            s2 = s2 + c3;
        else
            s1 = Quick_Two_Sum(s1, c3, s2);
    }else{
        s0 = Quick_Two_Sum(s0, c2, s1);
        if(s1 != 0.0)
            s1 = Quick_Two_Sum(s1, c3, s2);
        else
            s0 = Quick_Two_Sum(s0, c3, s1);
    }
    c0 = s0; c1 = s1; c2 = s2;
}
```

図 7 CPU での TS 型の Renorm の実装

```
inline ts_real
ts_real :: TS_Add(const ts_real &a, const ts_real &b){
    float s0, s1, s2, t0, t1, t2;
    float v0, v1, v2, u0, u1, u2;
    float w0, w1, w2;
    s0 = a[0] + b[0]; s1 = a[1] + b[1]; s2 = a[2] + b[2];
    v0 = s0 - a[0]; v1 = s1 - a[1]; v2 = s2 - a[2];
    u0 = s0 - v0; u1 = s1 - v1; u2 = s2 - v2;
    w0 = a[0] - u0; w1 = a[1] - u1; w2 = a[2] - u2;
    u0 = b[0] - v0; u1 = b[1] - v1; u2 = b[2] - v2;
    t0 = w0 + u0; t1 = w1 + u1; t2 = w2 + u2;
    s1 = Two_Sum(s1, t0, t0);
    Three_Sum(s2, t0, t1);
    t0 = Two_Sum(t2, t0, t2);
    t0 = t0 + t1;
    Renorm(s0, s1, s2, t0);
    return ts_real(s0, s1, s2);
}
```

図 8 CPU での TS 型の TS_Add の実装

```
inline float
Two_Prod_FMA(float a, float b, float &err){
    s = a * b;
    err = fmaf(a, b, -p);
    return s;
}
```

図 9 CPU での TS 型の Two_Prod_FMA の実装

```
inline ts_real
ts_real :: TS_Prod(const ts_real &a, const ts_real &b){
    float p0, p1, p2, q0, q1, q2, s0;
    p0 = Two_Prod_FMA(a[0], b[0], q0);
    p1 = Two_Prod_FMA(a[0], b[1], q1);
    p2 = Two_Prod_FMA(a[1], b[0], q2);
    Three_Sum(p1, p2, q0);
    Three_Sum(p2, q1, q2);
    s0 = a[0] * b[2] + a[2] * b[0] + a[1] * b[1] + q0 + q1 + q2;
    Renorm(p0, p1, p2, s0);
    return ts_real(p0, p1, p2);
}
```

図 10 CPU での TS 型の TS_Prod の実装

3.2 GPU における 3 倍精度(GTS 型)演算の実装

GPU における GTS 型の実装について述べる. 今回の実装は CUDA を用いて行った. TS 型は, CUDA の組み込みベクトル型である float3 型を用いて格納し, typedef を用いて gts_real 型と定義した. 組み込みベクトル型である float3 型は, float 型 3 つからなる構造体で, メンバ変数の x , y , z を用いてそれぞれの要素にアクセスできる. x , y , z はそれぞれ, x は GTS 型の上位部, y は GTS 型の中位部, z は GTS 型の下位部と対応している. 今回は行列積を求めるために, 主に GTS 乗算と GTS 加算を GPU のデバイス関数として実装した. コンパイラによる意図しない最適化を防ぐために, 演算子ではなく, CUDA の組み込み関数である, 単精度加算(`_fadd_rn`), 単精度乗算(`_fmul_rn`), 単精度 FMA(`_fmaf_rn`)を使用した. そのほか, 関数の呼び出しによるオーバーヘッドを減少させるために `__forceinline__` を使用した. 実際に使用したコードを示していく. Two_Sum (図 11) は単精度の丸め誤差を考慮した加算である. $a + b$ の実行結果を s に格納しその際に発生した丸め誤差を err に格納する. Quick_Two_Sum (図 12) は前述と同様に加算であるが, $|a| \geq |b|$ が成立する場合のみ使用することができる. Two_Sum を複数用いることで Three_Sum (図

13)を実装し a , b , c の値を入れ替えながら上位部を a , 中位部を b , 下位部を c へと再編成することができるアルゴリズムを実装した. 4つの単精度の値をTS型へまとめるためにRenorm(図14)を実装した. Two_SumとThree_Sum, Renormなどを組み合わせることによりTS型同士を加算するGTS_Add(図15)を実装した. 一方でTS乗算では, 従来のGQDライブラリではFMA命令が実装されていなかったため, 単精度の丸め誤差を考慮した乗算であるTwo_Prod_FMA(図16)を実装した. これは, $a \times b$ の浮動小数点の結果を p に格納し, 発生した丸め誤差をFMA命令を用いて $(a \times b) - p$ を行い err に格納する. これらの演算を用いてGTS同士を乗算するGTS_Prod(図17)を実装した.

```
_device_ _forceinline_
float Two_Sum(float a, float b, float &err){
    float s, bb;
    s = _fadd_rn(a, b);
    bb = _fadd_rn(s, -a);
    err = _fadd_rn(_fadd_rn
        (a, -_fadd_rn(s, -bb)), _fadd_rn(b, -bb));
    return s;
}
```

図11 GPUでのGTS型のTwo_Sumの実装

```
_device_ _forceinline_
float Quick_Two_Sum(float a, float b, float &err){
    float s;
    s = _fadd_rn(a, b);
    err = _fadd_rn(b, -_fadd_rn(s, -a));
    return s;
}
```

図12 GPUでのGTS型のQuick_Two_Sumの実装

```
_device_ _forceinline_
void Three_Sum(float &a, float &b, float &c){
    float t1, t2, t3;
    t1 = Two_Sum(a, b, t2);
    a = Two_Sum(c, t1, t3);
    b = Two_Sum(t2, t3, c);
}
```

図13 GPUにおけるGTS型のThree_Sumの実装

```
_device_ _forceinline_
void Renorm(float &c0, float &c1, float &c2, float &c3){
    float s0, s1, s2 = 0.0;
    s0 = Quick_Two_Sum(c2, c3, c3);
    s0 = Quick_Two_Sum(c1, s0, c2);
    c0 = Quick_Two_Sum(c0, s0, c1);
    s0 = c0; s1 = c1;
    if(s1 != 0.0){
        s1 = Quick_Two_Sum(s1, c2, s2);
        if(s2 != 0.0)
            s2 = _fadd_rn(s2, c3);
        else
            s1 = Quick_Two_Sum(s1, c3, s2);
    }else{
        s0 = Quick_Two_Sum(s0, c2, s1);
        if(s1 != 0.0)
            s1 = Quick_Two_Sum(s1, c3, s2);
        else
            s0 = Quick_Two_Sum(s0, c3, s1);
    }
    c0 = s0; c1 = s1; c2 = s2;
}
```

図14 GPUでのGTS型のRenormの実装

```
_device_ _forceinline_
gts_real GTS_Add(const gts_real &a, const gts_real &b){
    float s0, s1, s2, t0, t1, t2, v0, v1, v2;
    float u0, u1, u2, w0, w1, w2;
    s0 = _fadd_rn(a.x, b.x); s1 = _fadd_rn(a.y, b.y);
    s2 = _fadd_rn(a.z, b.z);
    v0 = _fadd_rn(s0, -a.x); v1 = _fadd_rn(s1, -a.y);
    v2 = _fadd_rn(s2, -a.z);
    u0 = _fadd_rn(s0, -v0); u1 = _fadd_rn(s1, -v1);
    u2 = _fadd_rn(s2, -v2);
    w0 = _fadd_rn(a.x, -u0); w1 = _fadd_rn(a.y, -u1);
    w2 = _fadd_rn(a.z, -u2);
    u0 = _fadd_rn(b.x, -v0); u1 = _fadd_rn(b.y, -v1);
    u2 = _fadd_rn(b.z, -v2);
    t0 = _fadd_rn(w0, u0); t1 = _fadd_rn(w1, u1);
    t2 = _fadd_rn(w2, u2);
    s1 = Two_Sum(s1, t0, t0);
    Three_Sum(s2, t0, t1);
    t0 = _fadd_rn(t0, _fadd_rn(t1, t2));
    Renorm(s0, s1, s2, t0);
    return make_ts(s0, s1, s2);
}
```

図15 GPUでのGTS型のGTS_Addの実装

```
_device_ _forceinline_
float Two_Prod_FMA(float a, float b, float &p, float &e){
    p = _fmul_rn(a, b);
    e = _fmaf_rn(a, b, -p);
    return s;
}
```

図16 GPUでのTS型のTwo_Prod_FMAの実装

```
_device_ _forceinline_
gts_real GTS_Prod(const gts_real &a, const gts_real &b){
    float p0, p1, p2;
    float q0, q1, q2;
    float s0;
    Two_Prod_FMA(a.x, b.z, p0, q0);
    Two_Prod_FMA(a.x, b.y, p1, q1);
    Two_Prod_FMA(a.y, b.x, p2, q2);
    Three_Sum(p1, p2, q0);
    Three_Sum(p2, q1, q2);
    s0 = _fmaf_rn(a.x, b.z, _fmaf_rn(a.z, b.x,
        _fmaf_rn(a.y, b.y, _fadd_rn(q0, _fadd_rn(q1, q2)))));
    Renorm(p0, p1, p2, s0);
    return make_ts(p0, p1, p2);
}
```

図17 GPUでのGTS型のGTS_Prodの実装

4. 3倍精度浮動小数点数における尾崎スキーム

この章では今回作成した尾崎スキームについて紹介する. 今回作成した尾崎スキームの概要図を図18に示す. 説明のため, 2分割の場合を説明する. まず, TS型行列 A とTS型行列 B がある. それを上位のビットから乗算を行った際に桁落ちが起こらないような分割を行う. 今回のTS型からfloat型への分割は七井ら[20]のものを参考にして実装した. float型の行列の $A1$, $A2$, $B1$, $B2$ を作成する. float型のそれぞれの行列にて $A1 \times B1$, $A1 \times B2$, $A2 \times B1$, $A2 \times B2$ の行列積を行い $C1$, $C2$, $C3$, $C4$ を作成する. 最後に, $C1$ - $C4$ をTS加算で加算を行いTS型行列 C を作成する. これが今回作成した尾崎スキームの概要である. 次の節では今回作成したアルゴリズムについて詳しく述べていく.

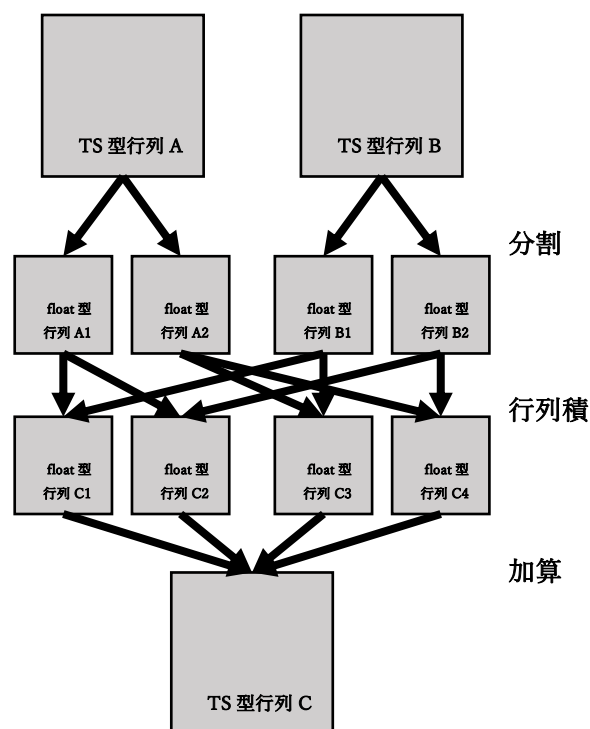


図 18 TS 型の尾崎スキームの概要図

4.1 TS 型における尾崎スキームの実装

アルゴリズム 1 作成した TS 型行列積に用いた尾崎スキームのアルゴリズム(3 分割, $n \times n$ 行列の場合)

Input: A, B : A, B は TS 型の正方行列
Output: C : C は TS 型の正方行列

- 1: $A^{(s)} = A, B^{(s)} = B$
: $A^{(s)}, B^{(s)}$ は float 型の $n \times n$ の正方行列
- 2: $\alpha = 1$
- 3: While($\alpha < 3$)
- 4: $M_A(i)_{1 \leq i \leq n} = \max_{1 \leq k \leq n} |A^{(s)}_{i,k}|$: M_A は n 次ベクトル
- 5: $M_B(j)_{1 \leq j \leq n} = \max_{1 \leq p \leq n} |B^{(s)}_{p,j}|$: M_B は n 次ベクトル
- 6: $T_A(i)_{1 \leq i \leq n} = 2^{\wedge(\text{ceil}(\log_2(M_A(i))) + \text{ceil}(\frac{24 + \log_2(n)}{2}))}$
: T_A は n 次ベクトル
- 7: $T_B(j)_{1 \leq j \leq n} = 2^{\wedge(\text{ceil}(\log_2(M_B(j))) + \text{ceil}(\frac{24 + \log_2(n)}{2}))}$
: T_B は n 次ベクトル
- 8: $S_A = T_A \cdot E^t$: $E = (1, 1, 1, \dots, 1)^t$ の n 次ベクトル
- 9: $S_B = E \cdot T_B^t$
- 10: $A_\alpha = (A^{(s)} + S_A) - S_A$
- 11: $B_\alpha = (B^{(s)} + S_B) - S_B$
- 12: $A = A - A_\alpha, B = B - B_\alpha$
- 13: $A^{(s)} = A, B^{(s)} = B$
- 14: $\alpha = \alpha + 1$
- 15: End While
- 16: $A_\alpha = A^{(s)} - A_{\alpha-1}$
- 17: $B_\alpha = B^{(s)} - B_{\alpha-1}$
- 18: For($\alpha = 1; \alpha < 4; \alpha++$)
- 19: For($\beta = 1; \beta < 4; \beta++$)
- 20: $C_\beta = A_\beta \cdot B_\beta$: Sgemm にて計算
- 21: End For
- 22: $C = C_1 + C_2 + C_3$: TS 加算にて計算
- 23: End For

ここでは, TS 型の尾崎スキームの具体的な実装について述べていく. 今回使用したアルゴリズムをアルゴリズム 1 へ示す. 1 行目で A, B の上位 float を $A^{(s)}, B^{(s)}$ へコピーする. 3, 4 行目で $A^{(s)}$ の各行の最大値と $B^{(s)}$ の各列の最大値を求めて M_A, M_B へそれぞれ格納する. 6~9 行目で M_A, M_B と float 型の仮数部である 24bit を用いて行列積を行った際に桁落ちしないような行列にするために S_A, S_B を求めている. 10, 11 行目では行列積を行っても桁落ちしない A_1, B_1 を求めている. 12 行目で TS 型行列 A, B から A_1, B_1 を減算し A, B の値を更新して 13 行目で $A^{(s)}, B^{(s)}$ の値を更新している.

今回は 3 分割であるためこの行為を 2 回繰り返して最後に A_3, B_3 を $A^{(s)} - A_2, B^{(s)} - B_2$ を行い求めている. これにより TS 型行列 A, B を上位から 3 分割行うことができた. 18 行目から 23 行目までそれぞれ要素ごとに乗算を行い, TS 型行列 C に TS 加算を行い, 行列 C に値を入れていく. なお, 乗算を行う際は cuBLAS の Sgemm を用いて行う.

5. GPU におけるブロック行列積と尾崎スキームによる性能評価

ブロック化した行列積と 4 章で示した尾崎スキームの手法を GPU 上で実施して評価した. 評価に用いた GPU は NVIDIA 社の GeForce GTX 1080 と Quadro RTX 6000 にてベンチマークを実施した. それぞれの GPU の性能と今回使用した PC のスペックやソフトウェアのバージョンを表 1, 表 2 に示す. 今回の実験で使用した N 次正方行列要素として $(ru - 0.5) \times \exp(1.0 \times rn)$ で計算される乱数とした. ここで ru は $[0, 1)$ の一様乱数であり, rn は標準正規分布にともなう乱数とする. 今回は, 下記の 2 つの評価を行った.

《1》TS 型と D+S 型, DD 型のブロック化した行列積の計算能力の比較

《2》TS 型のブロック化した行列積, 9 分割, 10 分割, 11 分割, 12 分割の尾崎スキームの速度と最大相対誤差

今回の最大相対誤差は以下の式(1)ように示されるものとする. ここで $E_{i,j}^*$ は QD 型の行列積の演算結果であり, $E_{i,j}$ は TS 型のブロック化した行列積, 9 分割, 10 分割, 11 分割, 12 分割の尾崎スキームの行列積の結果の要素である.

$$\max_{1 \leq i, j \leq n} \frac{|E_{i,j}^* - E_{i,j}|}{|E_{i,j}^*|} \quad (1)$$

表 1 GTX1080 の実行環境

CPU	Intel i7 6700K
Memory	16GB
GPU	NVIDIA GeForce GTX 1080
	ベースクロック 1607 MHz
	CUDA コア 2560 コア
	VRAM GDDR5X 8GB
OS	Ubuntu 20.04.2 LTS
CUDA	11.2

表 2 RTX6000 の実行環境

CPU	AMD EPYC 7402P
Memory	128GB
GPU	NVIDIA Quadro RTX 6000
	ベースクロック 1440 MHz
	CUDA コア 4,608 コア
	VRAM GDDR6 24GB
OS	Ubuntu 18.04.5 LTS
CUDA	11.0

5.1 TS 型と D+S 型・DD 型のブロック化した行列積の計算能力の比較

ブロック化した行列積の(DD・TS)GFlops を図 19, 図 20 に示す. 行列サイズは 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 の正方行列の行列積にて性能評価を行った. 行列積は 32×32 にてブロック化を行い, ブロック化した行列はシェアードメモリ内に保存し高速化を図った. 性能比較のために 4 倍精度である DD 型, 棕木らが提案した D+S 型との比較を行っている. GPU は図 19 では GTX1080 を使用し, 図 20 では RTX6000 を使用して評価を行った. 図 19 では今回作成した TS 型は D+S 型, DD 型よりも最大で 5.9 倍高速であることが示された. 図 20 でも同様に, TS 型は D+S 型, DD 型よりも最大で 9.5 倍高速であることが示された. 参考に, double 型と DF 型の演算速度比較を図 21, 図 22 を示す. GTX1080 と RTX6000 のそれぞれで DS 型のほうが double 型よりも高速であるという結果が出ている. GTX1080 では, DF 型と double 型との差は最大で 2.1 倍となり, RTX6000 では, DF 型と double 型との差は最大で 1.4 倍となることが示された.

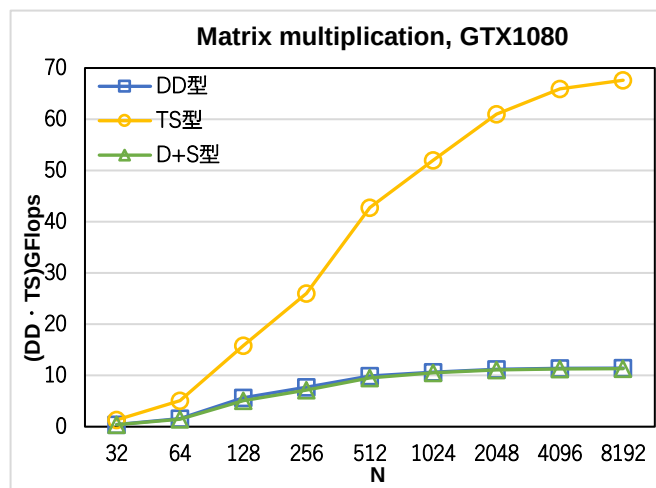


図 19 GTX1080 を用いた TS 型, DD 型, D+S 型のブロック化した行列積演算(縦軸の(DD・TS)GFlops は 1 秒当たりの DD・TS 演算回数を表す)

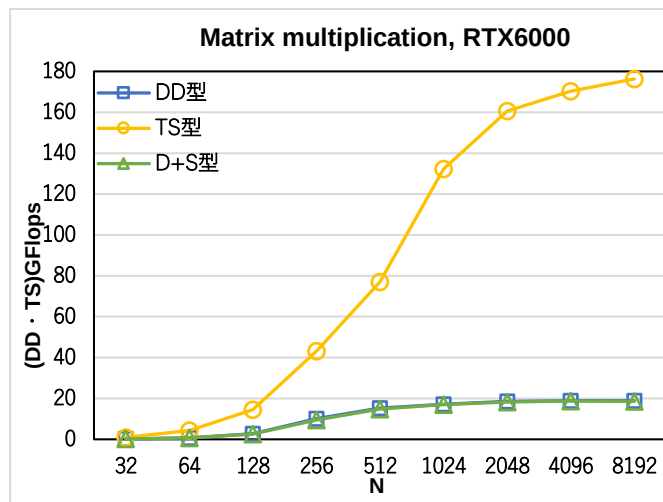


図 20 RTX6000 を用いた TS 型, DD 型, D+S 型のブロック化した行列積演算(縦軸の(DD・TS)GFlops は 1 秒当たりの DD・TS 演算回数を表す)

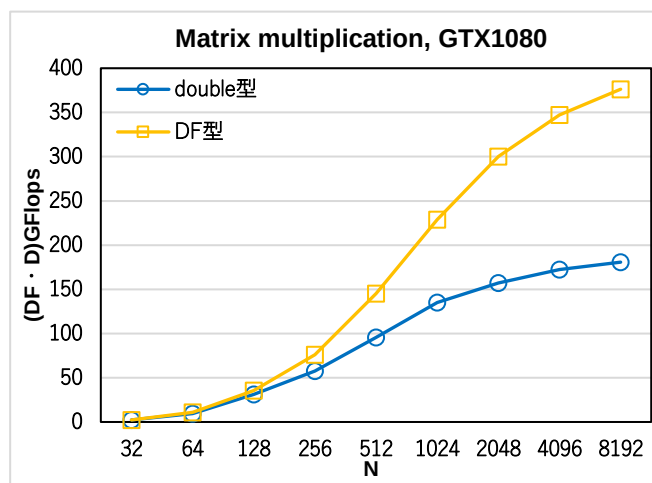


図 21 GTX1080 を用いた DF 型, double 型のブロック化した行列積演算(縦軸の(DF・D)GFlops は 1 秒当たりの DF・double 演算回数を表す)

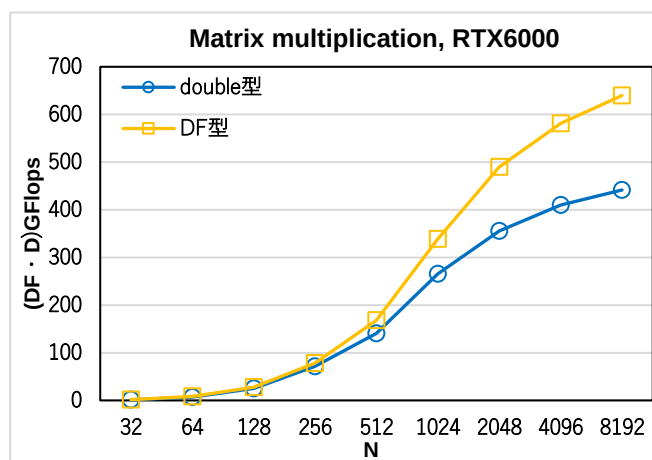


図 22 RTX6000 を用いた DF 型, double 型のブロック化した行列積演算(縦軸の(DF・D)GFlops は 1 秒当たりの DF・double 演算回数を表す)

5.2 TS 型の尾崎スキームの実行結果比較

この節では、尾崎スキームとブロック化した行列積の実行時間の比較と相対誤差の比較を行った結果を示す。

5.2.1 TS 型尾崎スキーム・ブロック化した行列積の実行時間の比較

TS 型尾崎スキームの実行時結果を図 23、図 24 に示す。行列サイズは 256, 512, 1024, 2048, 4096 の正方行列の行列積にて性能評価を行った。尾崎スキームの分割数は 9, 10, 11, 12 分割の 4 種類とブロック化した行列積の合わせて 5 種類で実験を行った。GPU は図 23 では GTX1080 を使用し、図 24 では RTX6000 を使用した。図 23 より、9～12 分割の尾崎スキームは今回実験したすべての行列サイズにてブロック化した行列積に実行時間としては勝ることはなかった。図 24 も同様にすべての行列サイズでブロック化した行列積に実行時間として勝ることはなかった。これは、尾崎スキームの分割数の増加により全体の計算時間がブロック化したものより多くなっているせいであると考えられる。しかし、 $N = 4096$ の付近ではかなり近接しているためより大きい行列では逆転することがあるかもしれない。尾崎スキームはメモリをブロック化した行列積よりも大量に消費するためこれ以上の桁数を調べるができなかった。今後はもっと大きな行列サイズにて検証を行う必要がある。

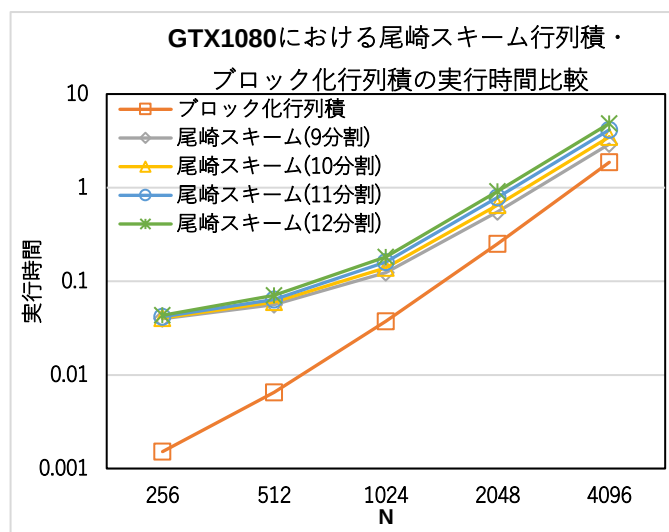


図 23 GTX1080 を用いた TS 型の尾崎スキーム・ブロック化した行列積の実行時間比較

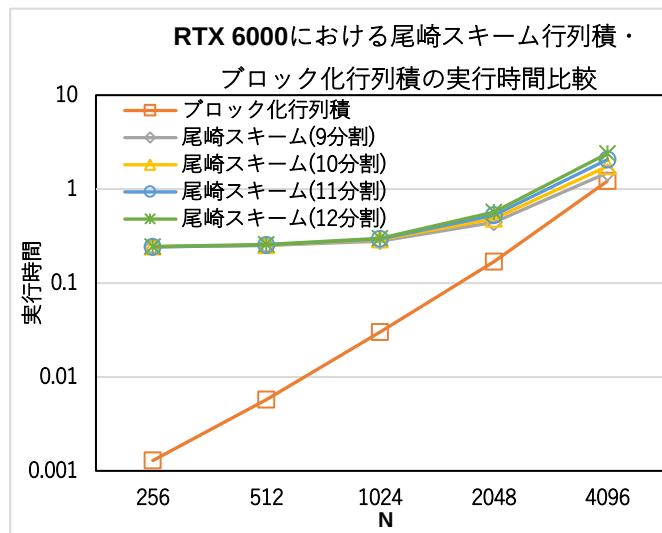


図 24 RTX6000 を用いた TS 型の尾崎スキーム・ブロック化した行列積の実行時間比較

5.2.2 TS 型尾崎スキーム・ブロック化した行列積の相対誤差の比較

TS 型の尾崎スキームで得られた行列積の相対誤差を図 25 に示す。行列サイズは 256, 512, 1024, 2048, 4096 の正方行列の行列積にて性能評価を行った。尾崎スキームの分割数は 9, 10, 11, 12 分割の 4 種類とブロック化した行列積の合わせて 5 種類で実験を行った。図 25 より尾崎スキーム(9 分割)を例にとると $N = 512$ までは $10^{-(21)}$ を保っていたが $N = 1024$ になると $10^{-(17)}$ まで相対誤差が増大した。尾崎スキーム(12 分割)は $N = 4096$ の場合でも精度は $10^{-(21)}$ を保っていた。この結果より分割数を増加させると相対誤差は小さくなった。ブロック化した行列積は $N = 256$ では $10^{-(17)}$, $n = 4096$ では $10^{-(15)}$ の相対誤差であった。

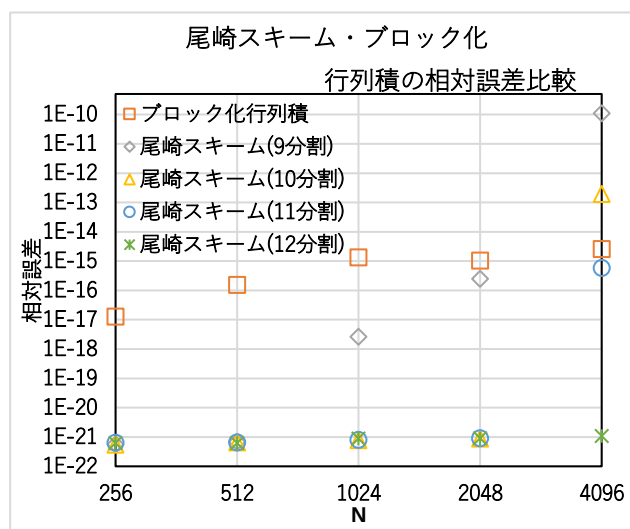


図 25 TS 型の尾崎スキーム・ブロック化した行列積の相対誤差比較

6. まとめ

本論文では、CPU・GPUでの3倍精度浮動小数点数演算を実装しGPUにおいてブロック化した行列積及び尾崎スキームにて性能評価し結果を示した。その結果、ブロック化した行列積の性能評価では、TS型の3倍精度演算は既存のD+S型よりも高速であることが示された。これはコンシューマ向けGPU上では倍精度演算性能より単精度演算性能が著しく高いことに起因している。

TS型の尾崎スキームでは、分割数の増加に伴う単精度行列積計算の増加によって、ブロック化した行列積より計算時間を要するということが分かった。しかし、相対誤差を見てみるとブロック化した行列積は $10^{(-17)} \sim 10^{(-14)}$ の相対誤差があるが、尾崎スキームは12分割で $N=4096$ の場合でも $10^{(-21)}$ の精度があることが分かった。現状では、速度の面をとるとブロック化した行列積のほうを利用したほうが良い。しかし、精度の面では尾崎スキームを行ったほうが良いといえる。そのため、使い分けが必要である。

今後の課題としては、まずTS型の尾崎スキームの改良である。今回は、TS型をfloat型で分割したため分割数が増加し計算速度が出なかった。そのため、椋木らが開発したDF(Double-Float)型を用いることで分割数を減らすことができる。それと同時に行列積だけでなくdouble型では桁落ちしてしまうような悪条件の連立一次方程式などにTS型を適応し高速化を図っていきたい。

7. 参考文献

- [1] NVIDIA ホームページ, NVIDIA. Available: <https://www.nvidia.com/ja-jp/>. [アクセス日: 27 10 2021].
- [2] NVIDIA: NVIDIA QUADRO VG100, NVIDIA. Available: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-volta-gv100-data-sheet-us-nvidia-704619-r3-web.pdf>. [アクセス日: 27 10 2021].
- [3] AMD ホームページ, AMD. Available: <https://www.amd.com/>. [アクセス日: 27 10 2021].
- [4] AMD: MI60, AMD. Available: <https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf>. [アクセス日: 27 10 2021].
- [5] GeForce RTX 20 シリーズと 20 SUPER グラフィックスカード, NVIDIA. Available: <https://www.nvidia.com/ja-jp/geforce/20-series/>. [アクセス日: 27 10 2021].
- [6] GeForce RTX 30 シリーズ グラフィックス カード概要, NVIDIA. Available: <https://www.nvidia.com/ja-jp/geforce/graphics-cards/30-series/>. [アクセス日: 27 10 2021].
- [7] IEEE: IEEE Standard for Floating-Point, IEEE Std 754-2008, 2008.
- [8] D. Bailey, (C++/Fortran-90 double-double and quad-double package). Available: <https://www.davidhbailey.com/dhbsoftware/>. [アクセス日: 27 10 2021].
- [9] Lu, M., He, B. and Luo, Q.: Supoporting Extended Precision on Graphics Processors, 6th International Workshop on Data Management on New Hardware, 2010.
- [10] T. Kouya, Performance Evaluation of Strassen Matrix Multiplication Supporting Triple-Double Precision Floating-Point Arithmetic, ICCSA2020, 2020.
- [11] T. Kouya, Acceleration of Multiple Precision Matrix Multiplication Based on Multi-component Floating-Point Arithmetic Using AVX2, ICCSA2021, 2021.
- [12] NVIDIA, CUDA Math API. Available: <https://docs.nvidia.com/cuda/cuda-math-api/>. [アクセス日: 27 10 2021].
- [13] NVIDIA, cuBLAS 仕様書, Available: <https://docs.nvidia.com/cuda/cublas/index.html>. [アクセス日: 27 10 2021].
- [14] 椋木, 今村, 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, 2014/12/10.
- [15] 椋木, 高橋, 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, 2013, Jan.
- [16] 椋木, 高橋, GPU による 3 倍精度浮動小数点演算の検討, 情報処理学会研究報告, 2011.
- [17] N. Fabiano, J. Muller and J. Picot, Algorithms for Triple-Word Arithmetic, IEEE Transactions on Computers, 2019.
- [18] Ozaki, Ogita, Oishi and Rump, S.M., Error Free Transformations of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and Its Applications, Numerical Algorithms, 2012.
- [19] Ichimura, Ogita, Katagiri, Nagai and Ogita, Threaded Accurate Matrix-Matrix Multiplications with Sparse Matrix-Vector Multiplications, 2018 IRRR international Parallel and Distributed Processing Symposium Workshop, 2018.
- [20] 七井, 藤本, 小さい定数個の単精度行列積への分割を用いた尾崎スキームによる倍精度乗算のゲーミング GPU 上での評価, 情報処理学会論文誌, 2021, Jan.