Vol.2022-HPC-183 No.1
2022/3/17

尾崎スキームを用いた 6 倍精度(Triple-Double)演算行列積の高速化

打桐大雅[1]　幸谷智紀[1]

概要：IEEE754 倍精度型(Double 型, IEEE754 binary64)や単精度型（Single 型, binary32)以上の桁数を表す方式として多倍長浮動小数点数のマルチコンポーネント方式があり，この実装としては Bailey らによって開発された QD ライブラリが著名である．QD ライブラリは QD(Quadruple-Double)型と DD(Double-Double)型の 2 つの型から構成されている．その中間にあたるデータ型として，TD(Triple-Double)型があり，幸谷は CPU 上において TD 型の行列積を高速化できることを示した．一方，行列積の演算方法として行列の各要素を上位と下位に分割して高速な低精度行列積を利用する尾崎スキームが近年盛んに利用されている．本論文では TD 型行列積に対して尾崎スキーム使用して高速化を図る実装方法を提案し，ベンチマークテストによって幸谷の実装した TD 型行列積よりも高速化できることを示す．

キーワード：6 倍精度浮動小数点数 TD 型 尾崎スキーム 行列積

1. はじめに

近年では計算機の計算能力が飛躍的に向上しており，科学技術シミュレーションや深層学習など，多くの分野で向上した計算能力を活用した事例が見られる．一方で浮動小数点演算の精度の面では制限があり，ハードウェアでサポートされた IEEE754 の 2 進浮動小数点数としては単精度(binary32)と倍精度(binary64)しかないのが現状である．単精度(binary32)は 10 進数で約 7 桁，倍精度が 10 進数で約 16 桁程度しか精度が保証されない．そのため，2008 年に IEEE754 が改定され，IEEE754-2008 [1]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，GCC でサポートされている float128 型以外，一般ユーザが気軽に使える実装はない．

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演算を実装する方法の 1 つにマルチコンポーネント方式がある．代表的な実装例として Bailey らによって開発された QD ライブラリ [2]がある．現状では CPU 上でも GPU 上でも，IEEE754-2008 の binary128 よりも QD ライブラリがサポートする DD 型(Double-Double)のほう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(Triple-Double) [3] [4]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 TD 型を用いる．それを用いて近年注目されている尾崎スキームを用いた行列積と幸谷の実装した$Strassen + AVX2$を用いた行列積，単純行列積を比較してその結果を示す．

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後に 3 章において 6 倍精度浮動小数点数について紹介し，4 章において尾崎スキームの概要を紹介し，5 章において 6 倍精度浮動小数点数の尾崎スキームを用いた行列積について紹介をする．6 章では，CPU での尾崎スキームを用いた行列積の性能評価を行う．最後に 7 章にてまとめと今後の課題について述べる．

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，浮動小数点型の倍精度型(double 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 [2]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 倍浮動小数点数ライブラリである．DD 型(Double-Double)と QD 型(Quad-Double)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．その中間の 6 倍精度の研究として幸谷によって実装された TD 型(Triple-Double) [3] [4]が存在する．最適化された 6 倍精度演算方法としては Fabiano らの Triple-word 演算 [5]があるが，今回は簡易な QD 演算に基づく TD 型を実装した．

高速な行列積演算のアルゴリズムとして$Strassen$のアルゴリズム [6]が知られている．TD 型では幸谷が実装したもの [3]がある．一方，高精度な行列積計算法として，行列を各要素に分割した行列にしてから行列積を計算する手法として尾崎らが提案した尾崎スキーム [7] [8]が知られている．CPU での尾崎スキームの実装例として，椋木らによって実装されたもの [9]がある．椋木らは，4 倍精度(binary128)を倍精度(binary64)に分割し尾崎スキームを行っている．GPU での実装では，七井らによって実装されたもの [10]がある．七井らは double 型を float 型にて尾崎スキ

[1] 静岡理工科大学

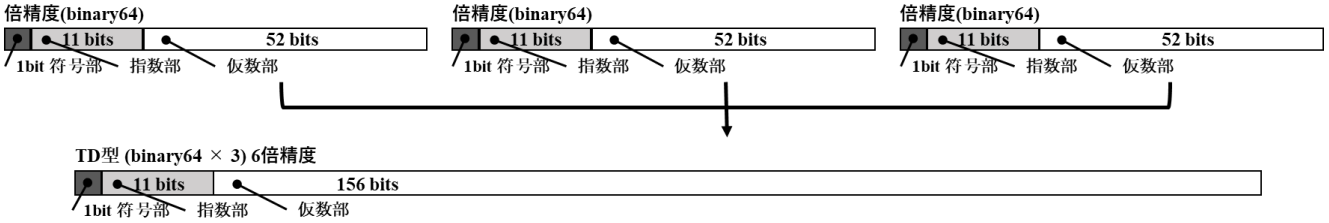

図 1　TD 型の 6 倍精度浮動小数点数型

ームを行っている．そのほかにも打桐らによって示された[11]ものがある．今回の尾崎スキームは打桐らの GPU での TS 型(Triple-Single)尾崎スキームを TD 型尾崎スキームとして CPU で実装したものを使用した．

3. 6 倍精度浮動小数点数

今回使用した 6 倍精度型(TD 型)の概略図を**図 1** に示す．TD 型は倍精度(binary64)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．倍精度の仮数部は 52bit(ケチ表現で 53bit)となり，精度は 10 進数で約 16 桁である．今回作成した TD 型は仮数部が 52+52+52=156bit(ケチ表現で 159bit)となり精度は 10 進数で約 48 桁となる．4 倍精度(binary128)は指数部が 15bit であるが倍精度は 11bit しかない．TD 型は倍精度をもとにして作成しているため指数部は 11bit の範囲でしか表現することができない．

今回使用した CPU における TD 型の実装について述べる．今回の TD 型は QD ライブラリを参考に作成を行った．今回は TD 型の行列乗算を実装するために，主に TD 乗算と TD 加算の実装を行った．実施に実装したコードを「**図 2-図 8**」に示す．

Two_Sum(**図 2**)は倍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 3**)は前述と同様に倍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 を複数組み合わせて 3 つの浮動小数点数の加算を行う Three_Sum(**図 4**)を実装した．Three_Sum は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，へと再編成するものである．$c0$-$c3$の 4 つの倍精度の値を TD 型へ丸めるために Renorm(**図 5**)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 TD 型同士を加算する TD_Add(**図 6**)を実装した．

一方で従来の QD ライブラリでは FMA 命令が実装されていなかったため，倍精度の丸め誤差を考慮した乗算である．Two_Prod_FMA(**図 7**)を実装した．これは，$a \times b$の結果をpに格納し，発生した丸め誤差をerrに格納する．FMA 命令を使用しない場合は丸め誤差を考慮した乗算に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 で使用可能であるため，今回の実装では FMA 命令を使用した．これらの演算を用いて TD 型同士を乗算する TD_Prod(**図 9**)を実装した．

```
inline double Two_sum(double a, double b, double &err){
    double s, bb;
    s = a + b;
    bb = s − a;
    err = (a − (s − bb)) + (b − bb);
    return s;
}
```

図 2　TD の Two_Sum の実装

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

図 3　TD の Quick_Two_Sum の実装

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

図 4　TD 型の Three_Sum の実装

```
inline void Renorm(double &c0, double &c1,
                   double &c2, double &c3){
    double 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 5　TD 型の Renorm の実装

```
inline td_real
td_real :: TD_Add(const td_real &a, const td_real &b){
    double s0, s1, s2, t0, t1, t2;
    double v0, v1, v2, u0, u1, u2;
    double 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 td_real(s0, s1, s2);
}
```

図 6　TD 型の TD_Add の実装

```
inline double
Two_Prod_FMA(double a, double b, double &err){
    s = a × b
    err = fma(a, b, −p);
    return s;
}
```

図 7　TD 型の Two_Prod_FMA の実装

```
inline ts_real
td_real::TD_Prod(const td_real &a, const td_real &b) {
    double 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 td_real(p0, p1, p2);
}
```

図 8　TD 型の TD_Prod の実装

4. 尾崎スキームの概要

この章では尾崎スキームの概要を紹介する．

尾崎スキームは，行列を各要素の上位ビットと下位ビットの行列に分割してから，それぞれの要素を行列積にて計算する手法である．行列A, Bを上位ビットの行列を$A1$，$B1$として下位ビットの行列を$A2$，$B2$とすると行列積$A \times B$は**式 1**-**式 3** のようにあらわすことができる．

$$A = A1 + A2 \qquad \textbf{式 1}$$

$$B = B1 + B2 \qquad \textbf{式 2}$$

$$C = A1 \times B1 + A1 \times B2 + A2 \times B \qquad \textbf{式 3}$$

ここでは，$A1 \times B1$は桁あふれが起こらないような$A1$，$B1$となるように分割を行う．

$A1 \times B1$が誤差なしで計算できるための条件は下記のとおりである．行列A, Bのサイズを$n \times n$とし単位相対丸めを倍精度で$u = 2^{-53}$とすると ，仮数部の有効桁数は$-\log_2 u$に当てはめると有効桁数は 53 となる．上位ビットを A1，B1 としてその要素を$A1_{i,k}$，$B1_{k,j}$とする． 行列積$C1$ は$C1 = A1 \times B1$と表すことができる．$C1$の要素を$C1_{i,j}$とすると$C1_{i,j}$は**式 4** であらわすことができる．

$$C1_{i,j} = \sum_{k=1}^{n} A1_{i,k} \times B1_{k,j} \qquad \textbf{式 4}$$

Aの各行(Bの各列)の要素をその行(列)の最大値の上位Xビットで分割する．Aの行，Bの列はそれぞれn個の要素があるため，Xビットで分割したn個の浮動小数点数の要素を足し合わせた値が有効桁数以内であれば精度を保ったままの演算が可能である．n個の浮動小数点数を足し合わせた場合の桁上りは最大で$\lceil \log_2 n \rceil$である．一方で，$A1$，$B1$の各要素の仮数部の$X+1$ビット以降がすべて 0 とすると$A1_{i,k}$，$B1_{k,j}$の仮数部の先頭からの非 0 ビットの数は最大で$2X$ビットとなるので，**式 5**, **式 6** よりXを求めることができる．よって**式 6** を満たすように行列の各要素を上位ビットと残りの下位ビットに分割を行う．

$$-\log_2 u \geq 2 \times X + \lceil \log_2 n \rceil \qquad \textbf{式 5}$$

$$X = \left\lfloor \frac{-\log_2 u - \lceil \log_2 n \rceil}{2} \right\rfloor \qquad \textbf{式 6}$$

次にA, Bを上位ビットの行列と残りの下位ビットの行列に分割する方法を示す．行列積を求めるときに，行列Aは各行に注目して計算する．S_Aという各行の要素が行列Aの各行に比べて大きな絶対値を持つ行列を用意する．行列AよりS_Aを用いて分割を行う式を**式 7** にて示す．行列Aに行列S_Aを加算することにより行列Aの下位ビットが桁落ちを起こす．そして桁落ちした値からS_Aを減算することにより乗算を行っても桁落ちが発生しない$A1$を求めることができる．

$$A1 = ((A + S_A)) - S_A) \qquad \textbf{式 7}$$

行列S_Aは**式 8** を用いて求めることができる．E^tは$(1,1,1,\ldots,1)^t$の n 次ベクトルである．行列Aのi行目の最大値$M_A(i)$は**式 9** を用いて求める．行列Aの各行の絶対値の最大値M_Aを求め，その値の上位Xビットまで取得する．それらの値と**式 6** などを用い**式 10** にてT_Aを求める．

$$S_A = T_A \times E^t \qquad \textbf{式 8}$$

$$M_A(i) = max_{1 \leq k \leq n} |A_{i,k}| \qquad \textbf{式 9}$$

$$T_A(i) = 2^{\left(\lceil log2(M_A(i)) \rceil + \frac{\lceil -\log_2 u + \lceil \log_2 n \rceil \rceil}{2}\right)} \qquad \textbf{式 10}$$

また，同様に行列Bを分割する行列S_Bは**式 11** を用いて求めることができる．しかし，行列積を求めるときに，行列Bは各列に注目して計算するため，各列の要素が行列Bの各列に比べて大きな絶対値を持つ行列S_Bを用意する．行列B

のj列の最大値$M_B(j)$は**式 12** を用いて求める．その後，**式 13** にて T_Bを求める

$$S_B = E \cdot {T_B}^t \quad \text{式 11}$$
$$M_B(j) = max_{1 \le p \le n} |B_{p,j}| \quad \text{式 12}$$
$$T_B(j) = 2^{\left(\lceil log2(M_B(j)\rceil + \frac{\lceil -\log_2 u + \lceil \log_2 n \rceil \rceil}{2}\right)} \quad \text{式 13}$$

下位ビットとして$A2$，$B2$を求める際には，**式 14**，**式 15**$B2 = B - B1$ **式 15**
のように求める．

$$A2 = A - A1 \quad \text{式 14}$$
$$B2 = B - B1 \quad \text{式 15}$$

$A2$，$B2$も同様に，桁あふれが起こらないような分割を行う際には同様のアルゴリズムを適応すると求めることができる．この時の上位ビットは$A2$，$B2$，下位ビットを$A3$，$B3$とすると**式 16**，**式 17** のようにあらわすことができる．この時の$A2$, $B2$は桁あふれが起こらないような処理がなされたものであり，**式 14, 式 15** で求めた$A2$，$B2$とは異なるものである．これにより，3 分割以降の分割が可能である．

$$A3 = (A - A1) - A2 \quad \text{式 16}$$
$$B3 = (B - B1) - B3 \quad \text{式 17}$$

6 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を紹介する．

今回作成した尾崎スキームは TD 型を Double 型に分割するものを作成した．Double 型の行列乗算の部分は $Intel\ Math\ Kernel\ Library(intel\ MKL)$の中に実装されている BLAS ライブラリを用いて実装を行った．ここでは，3 分割を行った際の実装例を示す．後述するが 3 分割では精度が出ないためベンチマークでは 11 分割までのベンチマークを実施した．3 分割のアルゴリズムを**アルゴリズム 1** に示す．1 行目で A, B の上位 double を$A^{(D)}$，$B^{(D)}$へコピーを行う．3，4 行目で$A^{(D)}$の各行の最大値と$B^{(D)}$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 double 型の仮数部である 53bit を用いて，S_A，S_Bを求める．S_A，S_Bを求める理由として，A，Bをそれぞれ分割したA_1，B_1が行列積を行った際に桁あふれが起きない行列にするためである．10，11 行目では桁あふれが起こらないようなA_1，B_1を求める．12，13 行目でA，BからA_1，B_1を減算しA，Bと$A^{(D)}$，$B^{(D)}$の値を更新する．今回は，3 分割であるためこの操作を 2 回繰り返した後に 16，17 行目でA_3，B_3を求める．この操作をすることによる TD 型の上位から桁あふれが起こらないようなA_1～A_3，B_1～B_3を求めることができる．18～23 行目まででそれぞれの要素ごとに乗算を行い TD 型行列 C に TD 加算を行い行列 C に値を入れていく．TD 型の精度に影響のある範囲で乗算を行っているため，$A_3 \times B_3$のような下位同士の演算を省くことにより高速化を図っている．乗算を行う際は，intel MKL の Dgemm を用いて計算を行う．

アルゴリズム 1　作成した TD 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×n 行列の場合)

Input: A, B　　:A, B は TD 型の正方行列
Output: C　　:C は TD 型の正方行列
1: $A^{(D)} = A$，$B^{(D)} = B$
　　:$A^{(D)}$，$B^{(D)}$は double 型の $n \times n$ の正方行列
2: $\alpha = 1$
3: $While(\alpha < 3)$
4: 　$M_A(i)_{1 \le i \le n} = max_{1 \le k \le n} |{A^{(D)}}_{i,k}|$　: M_Aは n 次ベクトル
5: 　$M_B(j)_{1 \le j \le n} = max_{1 \le p \le n} |{B^{(D)}}_{p,j}|$: M_Bは n 次ベクトル
6: 　$T_A(i)_{1 \le i \le n} = 2\text{^}(ceil\left(log2\left(M_A(i)\right)\right) + ceil\left(\frac{53+log2(n)}{2}\right))$
　　: T_Aは n 次ベクトル
7: 　$T_B(j)_{1 \le j \le n} = 2\text{^}(ceil\left(log2\left(M_B(j)\right)\right) + ceil\left(\frac{53+log2(n)}{2}\right))$
　　: 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: 　$A_\alpha = \left(A^{(s)} + S_A\right) - S_A$
11: 　$B_\alpha = \left(B^{(s)} + S_B\right) - S_B$
12: 　$A = A - A_\alpha$，$B = B - B_\alpha$ TD 減算にて計算
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: 　$For(\beta = 1; \beta < 5 - \alpha; \beta++)$
20: 　　$C_\beta = A_\alpha \cdot B_\beta$　: intel MKL の Dgemm にて計算
21: 　*End For*
22: 　$C \mathrel{+}= \sum_{k=1}^{4-\alpha} C_k$　: TD 加算にて計算
23: *End For*

5. CPU における尾崎スキームを用いた行列積による性能評価

この章では 4 章で示した手法を CPU にて実装したものを用いてベンチマークを実施した．評価に用いた CPU は intel Core i7 6700k と intel Core i7 11700 である．そのほかのマシンスペックを**表 1**，**表 2** に示す．

今回の実験で使用した行列の要素として$(ru - 0.5) \times exp(1.0 \times rn)$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 2 つの比較を行った．

《1》TD 型の 12 分割の尾崎スキームを用いた行列積と$Strassen + AVX2$を用いた行列積，単純行列積の速度比較
《2》TD 型の 6～12 分割の尾崎スキームを用いた行列積と$Strassen + AVX2$を用いた行列積の相対誤差比較

今回の最大相対誤差は以下の式の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 11 分割の尾崎スキームの行列積の結果の要素である．

$$\max_{1 \le i,\ j \le n} \frac{\left| E^*_{i,j} - E_{i,j} \right|}{\left| E^*_{i,j} \right|}$$

表 1　Core i7 6700k の実行環境

CPU	Intel Core i7 6700K
Memory	16GB
GPU	NVIDIA GeForce GTX 1080
OS	Ubuntu20.04.2 LTS
gcc/g++	9.3.0
icc/icpc	2021.5.0

表 2　Core i7 11700 の実行環境

CPU	Intel Core i7 11700
Memory	32GB
GPU	NVIDIA GeForce RTX 3070
OS	Ubuntu20.04.2 LTS
gcc/g++	9.3.0
icc/icpc	2021.5.0

5.1 TD 型の 12 分割の尾崎スキームと Strassen+AVX2, 単純行列積の速度比較

TD 型の 12 分割の尾崎スキームを用いた行列積と$Strassen + AVX2$を用いた行列積, 単純行列積の実行時間比較を図 9，図 10 に示す．行列サイズ(N)は 20～2000 で 20 ずつの間隔で性能評価を行った．性能比較のために，幸谷が実装した$Strassen + AVX2$を用いた行列積と単純行列積との比較を行った．CPU は図 9 が i7 6700k を使用し，図 10 が i7 11700 を使用して性能評価をしている．図 9 では今回作成した TD 型の尾崎スキームを用いた行列積は幸谷の$Strassen + AVX2$を用いた行列との比較で，最大で約 10 倍，単純行列積との比較では，最大で約 33 倍高速であることが示された．図 10 でも同様に，TD 型の尾崎スキームを用いた行列積は$Strassen + AVX2$を用いた行列積よりも約 8 倍，単純行列積との比較では，最大で約 38 倍高速であることが示された．一方でNが 10～100 で 10 ずつの間隔で切り取った場合の図を図 11, 図 12 に示す. 図 11 が i7 6700K を図 12 が i7 11700 で行ったときを示したグラフである．図 11 では今回作成した TD 型の尾崎スキームを用いた行列積よりも単純行列積のほうが$N = 60$まで，$Strassen + AVX2$を用いた行列積では$N = 90$まで尾崎スキームを用いた行列積よりも高速であった．図 12 でも同様に単純行列積では$N = 60$まで$Strassen + AVX2$を用いた行列積は$N = 80$まで尾崎スキームを用いた行列積より高速であった．今回開発した尾崎スキームは低次元での行列積では, 単純行列積や$Strassen + AVX2$を用いた行列積に及ばないという結果になった．

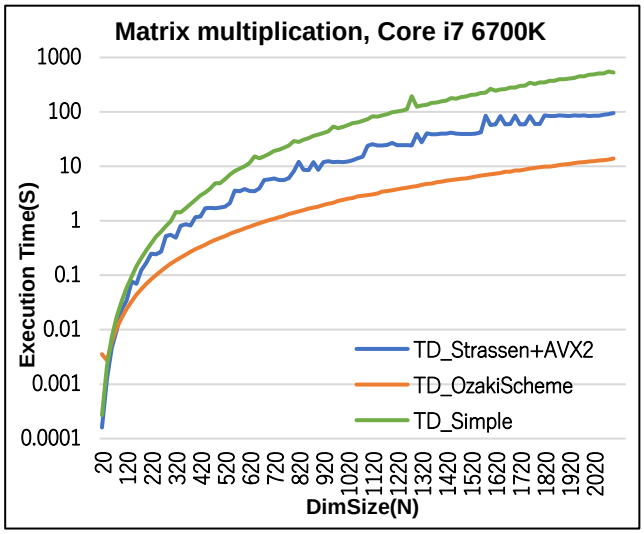

図 9　Core i7 6700K での TD 型の 12 分割尾崎スキームを用いた行列積，Strassen+AVX2 を用いた行列積，単純行列積の実行時間比較

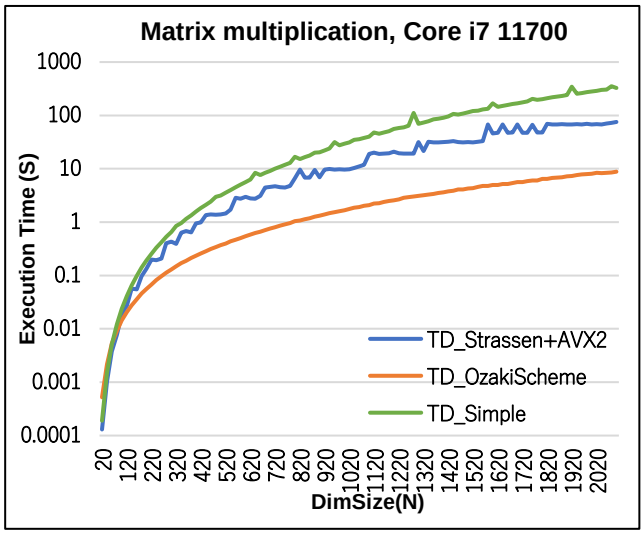

図 10　Core i7 11700 での TD 型の 12 分割尾崎スキームを用いた行列積，Strassen+AVX2 を用いた行列積，単純行列積の実行時間比較

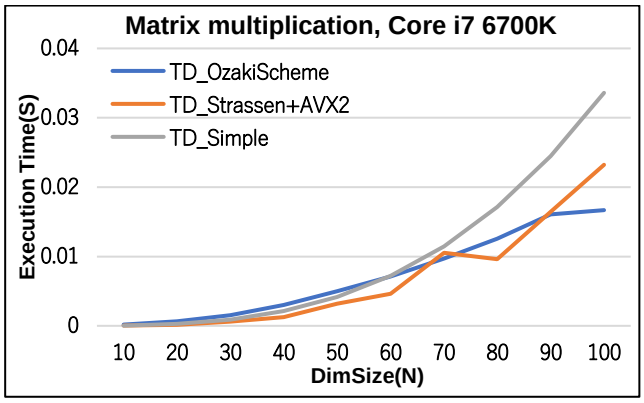

図 11　Core i7 6700K での行列サイズが 10～100 での行列積のベンチマーク

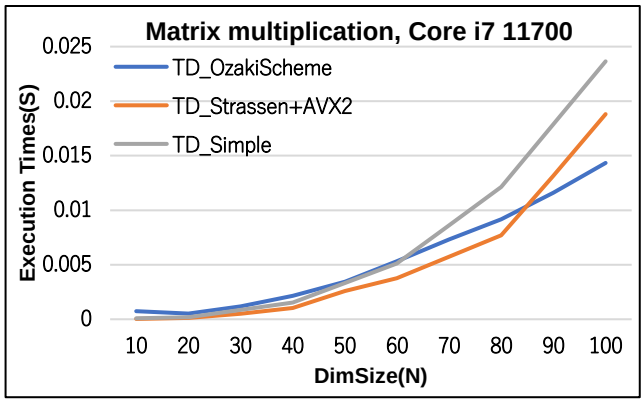

図 12 Core i7 11700 での行列サイズが 10～100 での行列積のベンチマーク

5.2 TD 型の 6～12 分割の尾崎スキームを用いた行列積と Strassen+AVX2 を用いた行列積の相対誤差比較

TD 型の 6～12 分割の尾崎スキームを用いた行列積と $Strassen + AVX2$ を用いた行列積の相対誤差の比較を**図 13** に示す．行列サイズ(N)は 100～2100 で 100 ずつの間隔で性能評価を実施した．**図 13** より尾崎スキームの 9 分割まで は$Strassen + AVX2$に精度の面で負ける結果になったが尾崎スキーム 10 分割以降では$N = 2100$の時点では精度の面で優位性を持っていることが分かった．しかし，Nが増加すると尾崎スキームの分割数を増やさなければ精度を保てなくなるためより多くの桁数での検証が必要である．

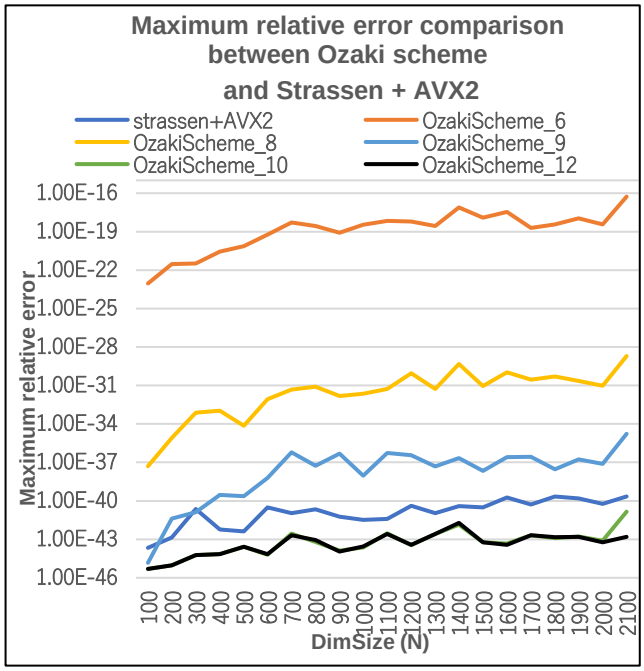

図 13 TD 型の 6～12 分割の尾崎スキームを用いた行列積と Strassen+AVX2 を用いた行列積の相対誤差比較

6. まとめ

本論文では，6 倍精度尾崎スキームを用いた行列積を実装し CPU において幸谷の実装した$Strassen + AVX2$を用いた行列積性能評価し結果を示した．その結果，尾崎スキームを用いた行列積の性能評価では幸谷の作成した $Strassen + AVX2$ よりも高速であることが示された．しかし，低次元の行列サイズでは$Strassen + AVX2$や単純行列積に及ばないという結果も示された．これは，尾崎スキームはメモリアクセスを多く行うため，メモリアクセス時間が増加してしまったのではないかと考える．尾崎スキームの精度の面では行列サイズが 2100 までの結果として 10 分割あれば精度が足りていることが分かった．

今後の課題としては，TD 型の尾崎スキームの改良である．今回のプログラムでは行列積は intel MKL を用いているもののそのほかは愚直にプログラムを書いただけである．そのため，並列化やそのほかの最適化を行っていない．その部分の並列化や最適化を行うことで高速化を図っていきたい．

謝辞 本研究の一部は科学技術研究費 20K11843 の女性を利用して行われた．また静岡理工科大学研究支援費の支援も受けて行われた．関係各位に感謝する．

参照文献

[1] “IEEE: IEEE Standard for Floating-Point,” IEEE Std 754-2008, 2008.
[2] D. Bailey ,S. Li , Y Hida .:,“(C++/Fortran-90 double-double and quad-double package),Available: https://www.davidhbailey.com/dhbsoftware/.
[3] T. Kouya, “Performance Evaluation of StrassenMatrix Multiplication Supporting Triple-Double Precision Floating-Point Arithmetic,” ICCSA2020, 2020.
[4] T. Kouya, “Acceleration of Multiple Precision Matrix Multiplication Based on Multi-component Floating-Point Arithmetic Using AVX2,” ICCSA2021, 2021.
[5] N.Fabiano, J.Muller and j.Picot, “Algorithms for Triple-Word Arithmetic,” IEEE Transactions on Computers, 2019.
[6] V.Strassen, “Gaussian elimination is not optimal,” Numerische Mathmatik, 1969.
[7] 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.
[8] Ichimura, Ozaki, Katagiri, Nagai and Ogita, “Threaded Accurate Matrix-Matrix Multiplications with Sparse Matrix-Vector Multiplications,” 2018 IRRR international Parallel and Distributed Processing Symposium Workshop, 2018.
[9] mukunoki, Ozaki, Ogita and Imamura, “Accurate Matrix Multiplication on Binary128 Format Accelerated by Ozaki Scheme,” 2021.
[10] 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.
[11] 打桐，幸谷，“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,” 2021.
[12] Lu M., He B. and Luo Q.:, “Supoprting Extended Precision on Graphics Processors,” 6th International Workshop on Data Management on New Hardware, 2010.