

Acceleration of Matrix Multiplication Based on Triple-Double (TD), and Triple-Single (TS) Precision Arithmetic

Taiga Utsugiri and Tomonori Kouya(✉)

Shizuoka Institute of Science and Technology, 2200-2 Toyosawa, Fukuroi 437-8555, Japan
2121002.ut@sist.ac.jp

Abstract. In this study, we present the results obtained from the acceleration of multi-component-type multiple precision matrix multiplication using the Ozaki Scheme and OpenMP. We aim for triple-double (TD) and triple-single (TS) precision matrix multiplication on CPU and GPU using TD and TS arithmetic based on error-free transformation (EFT) arithmetic. Owing to these combined techniques, on CPU, our implemented multiple precision matrix multiplications were more than seven times faster than Strassen matrix multiplication with AVX2. Similar to using Ozaki scheme, on GPU, our implemented multiple precision matrix multiplications could not be accelerated. Furthermore, we report that our accelerated matrix multiplication can modify parallelization performance with OpenMP on CPU.

Keywords: Triple-double (TD) precision arithmetic · Triple-single (TS) precision arithmetic · Matrix multiplication · Ozaki scheme

1 Introduction

It is well-known that the accuracy of numerical computation with floating-point arithmetic is limited owing to the effect of round-off errors. The hardware-supported IEEE754 floating-point numbers are binary32 (24-bit fraction) and binary64 (53-bit fraction). Although binary128 was officially proposed by IEEE754-2008, there is no implementation that normal users can use other than the float128 type supported by GCC.

The implementation of multi-precision floating-point (MPF) arithmetic is divided into two categories: multi-digit-type based on integer arithmetic and multi-component-type (or multi-term), which is constructed by several binary32 or binary64 numbers using error-free transformation (EFT) [1]. EFT is a method to calculate floating point arithmetic without loss of errors. The QD library proposed by Bailey et al. [2] is a de-facto standard, one of the multi-component-type MPF libraries, which consists of double-double (DD) and quadruple-double (QD) data types for the CPU. Lu et al. implemented the GQD library on GPU, which is based on QD library [3].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13378, pp. 406–423, 2022.
https://doi.org/10.1007/978-3-031-10562-3_29

While DD and QD precision arithmetic have been recently prepared, triple-double (TD) or triple-single (TS) arithmetic between DD and QD have been proposed in some studies. TD and TS calculations are expected to be comparable in performance and accuracy between DD and QD; thus, they can achieve the best results not quiet with DD but in shorter computational time than QD.

As an optimized three-word operation method, Fabiano et al. [4] proposed optimized triple-word operation. Although we cannot confirm its good performance, we selected it to implement TD by omitting the forth and least element of QD value.

The Strassen algorithm [5] is known as a high-speed matrix multiplication algorithm that uses divide-and-conquer approach to reduce the number of operations. Kouya [6] implemented and accelerated TD Strassen matrix multiplication with AVX2. On the contrary, the Ozaki scheme developed by Ozaki et al. [7, 8] is a highly accurate and well-performed matrix multiplication method that takes advantage of highly performed hard-wired precision matrix multiplication, such as DGEMM or SGEMM, provided by Intel Math Kernel and cuBLAS. Mukunoki et al. [9] used the Ozaki scheme to implement MPF matrix multiplication. Utsugiri et al. [10] reported some implementations on GPU.

As mentioned above, various optimization techniques for multi-component-type multiple precision matrix products have already been exploited, however, which technique is the most efficient is unclear. In particular, we should know the effectiveness of the Ozaki scheme compared with other optimized matrix products in TD and TS arithmetic. The main purpose of our study is to show the performance of the Ozaki scheme on CPU and GPU environments.

In this paper, we describe the implementation of the Ozaki scheme for TD and TS matrix multiplication on CPU and GPU and discuss the performance and accuracy. The programming languages used are C++ with an Intel compiler in the CPU and C++ with CUDA in the GPU.

Furthermore, we parallelize the inner process of the Ozaki scheme using OpenMP on CPU. As a result of our benchmark test, we can reveal that the Ozaki scheme implemented on CPU is faster than Strassen + AVX2 matrix multiplication. However, on the GPU, the Ozaki scheme is slower than the simple matrix multiplication. In addition, we can observe that the parallelized TD matrix multiplication on CPU is faster in a small matrix size than without OpenMP, but not in a large matrix size.

2 Overview of Triple-Double (TD), Triple-Single (TS)

Figures 1 and 2 show the schematics of the TD and TS precisions, respectively. TD precision consists of three double precisions (binary64) and TS precision consists of three single precisions (binary32). Our implementation uses the “td_real” data type as the TD floating-point number, and “ts_real” as the TS number; therefore, both “td_real a” and “ts_real a” consist of a[0] as the most significant component, a[1] in the middle, and a[2] as the least significant component. The precision of floating-point numbers is determined by the length of the mantissa part. The double-precision mantissa part has 53 bits (including one hidden bit), with a precision of approximately 16 decimal digits, and the single-precision mantissa part has 24 bits, with a precision of approximately seven decimal digits. The mantissa part of the TD precision is 159 bits, with a precision of

approximately 48 decimal digits, and the mantissa part of the TS precision is 72 bits, with a precision of approximately 21 decimal digits. On the contrary, IEEE754 binary128 has a 15-bit exponent portion; TD precision has 11 bits and TS precision has only 8 bits. Therefore, the valid range of these TD and TS values are smaller than binary128.

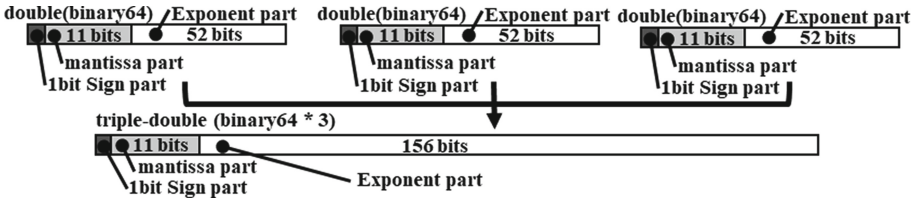

Fig. 1. Schematic of triple-double (TD) precision

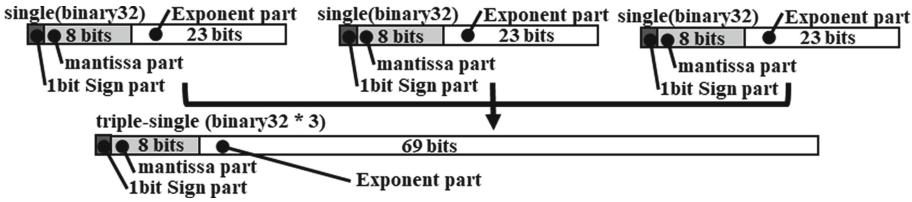

Fig. 2. Schematic of triple-single (TS) precision

2.1 Implementation of Triple-Double (TD) Precision on the CPU

Here, we describe that implementation of TD precision arithmetic. EFT, TD multiplication and TD addition are necessary to implement TD precision matrix multiplication. They have been implemented as inline functions to approve fast call of TD arithmetic functions. The implemented codes are shown in Algorithm 1–Algorithm 7.

Two_Sum (Algorithm 1) is addition that considers double precision rounding error. In *Two_Sum*, the result of $a+b$ operation with binary64 is stored in s, and the correspoind rounding error is stored in *err*. *Quick_Two_Sum* (Algorithm 2) is also addition that considers double precision rounding error as Two_Sum, however it can be used only when $|a| \geq |b|$. *Three_Sum* (Algorithm 3) is implemented by combining multiple *Two_Sum* and adding three floating-point numbers. *Three_Sum* reconstructs the upper part as a, the middle part as b, and the lower part as c, while exchanging the values of a, b, and c. *Two_Sum* and *Three_Sum* can be used to calculate the number of floating-point numbers. *TD_Add* (Algorithm 5) is implemented by combining *Two_Sum* and *Three_Sum* for TD precision addition. Renorm (Algorithm 4) is a function to normalize TD number.

The conventional QD library as default, does not use the FMA operation; therefore, in our TD arithmetic, *Two_Prod _FMA* (Algorithm 6) is prepared as a default Two_Prod function with "fma" utilized in the standard C library, which is a multiplication that considers double precision rounding errors. *Two_Prod _FMA* stores the $a \times b$ result in p and the rounding errors that occur in err. Because of these EFT functions, *TD_Prod* (Algorithm 7) performs TD precision multiplication.

Algorithm 1. inline double Two_sum(double a, double b, double &err)

```
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;
}
```

Algorithm 2. inline double Quick_Two_Sum(double a, double b, double &err)

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

Algorithm 3. inline void Three_sum(double &a, double &b, double &c)

```
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);
}
```

Algorithm 4. inline void Renorm(double &$c0$, double &$c1$, double &$c2$, double &3)

```
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;
}
```

Algorithm 5. inline td_real td_real::TD_Add(const td_real &a, const td_real &b)

```
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);
}
```

Algorithm 6. inline double Two_Prod_FMA(double a, double b, double &err)

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

Algorithm 7. inline td_real td_real:: TD_Prod(const td_real &a, const td_real &b)

```
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);
}
```

2.2 Implementation of Triple-Single Precision (TS) on GPU

This section describes the implementation of TS precision on GPU. The GPUs for consumers, such as NVIDIA GTX and RTX series, are designed to accelerate the processing graphics of games and deep learning using binary16 and binary32, rather than to outperform binary64. We confirmed performance of binary64 and TD arithmetic by comparing it to binary32 and TS arithmetic, and as the result, we conclude that TS matrix multiplication should be made available to users of consumer GPUs.

Algorithm 8–Algorithm 14, which contain parts of TS matrix multiplication, are almost similar to TD and related arithmetic as shown in Sect. 2.1. We can observe from these algorithms, that all data types used are “float” (binary32) rather than “double” (binary64).

Algorithm 8. __device__ __forceinline__ float Two_Sum(float a, float b, float &err)

```
__device__ __forceinline__ 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;
}
```

Algorithm 9. __device__ __forceinline__ float Quick_Two_Sum(float a, float b, float &err)

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

Algorithm 10. __device__ __forceinline__ void Three_Sum(float &a, float &b, float &c)

```
__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);
                                    }
```

Algorithm 11. __device__ __forceinline__ void Renorm(float &$c0$, float &$c1$, float &$c2$, float &$c3$)

```
__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 = 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;
}
```

Algorithm 12. __device__ __forceinline__ ts_real ts_real::TS_Add(const ts_real &a, const ts_real &b)

```
__device__ __forceinline__ 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, 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);
}
```

Algorithm 13. __device__ __forceinline__ float Two_Prod_FMA(float a, float b, float &err)

```
__device__ __forceinline__ float Two_Prod_FMA(float a, float b, float &err){
  s = a × b
  err = fmaf(a, b, −p);
  return s;
}
```

Algorithm 14. __device__ __forceinline__ ts_real ts_real::TS_Prod(const ts_real &a, const ts_real &b)

```
__device__ __forceinline__ 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);
}
```

3 Ozaki Scheme for Triple-Double (TD) Precision on CPU

The Ozaki scheme [7, 8] has been developed to implement accurate and fast matrix multiplication by accumulating binary64 or binary32 matrix products using highly optimized DGEMM or SGEMM. Here, we describe its usage for TD and TS matrix multiplications.

First, we explain the Ozaki scheme for TD matrix multiplication. To begin, we split the original TD matrices into binary64 matrices. The divided binary64 matrices were multiplied using the DGEMM of the BLAS library implemented in the Intel Math Kernel Library (intel MKL) [11]. Algorithm 15 shows how to split three binary64 matrices from the original matrices A and B. However, as described later, the number of splits should be limited to a maximum of 10.

In the 1st line, the first binary64 value of TD elements of A and B are copied to $A^{(D)}$ and $B^{(D)}$. In the 4th and 5th lines, the maximum value of each row of $A^{(D)}$ and each column of $B^{(D)}$ are obtained and stored in M_A and M_B, respectively. In the 6th to 7th lines, T_A and T_B are obtained using M_A and M_B, respectively and 53 bits which is a double precision mantissa. In the 8th to 9th lines, T_A and T_B, and the translocation vector E are used to find S_A and S_B. The reason for finding S_A and S_B is to make a matrix that does not cause overflow when A_1 and B_1, which include elements near the $\pm$Inf of binary64 value, perform matrix multiplication. In the 10th and 11th lines, A_1 and B_1 are obtained so that overflow does not occur. In the 12th and 13th lines, A_1 and B_1 are subtracted from A and B, and the elements of A and B are updated. At this time, because it is divided into three, A_3 and B_3 are obtained on the 16th and 17th lines after repeating this operation twice. By performing this operation from producing A_α and B_α even with binary64 arithmetic, round-off errors owing to digit overflow does not create and then can gain precise numerical results with TD arithmetic. From the 18th to 23rd lines, after DGEMM run for obtaining each C_α, TD addition is performed and then each element is stored in the TD matrix C.

Algorithm 15. Ozaki scheme algorithm used for the created TD precision matrix multiplication (in the case of three splits, n × n matrix)

$Input: A, B$: A and B are TD precision square matrices
$Output: C$: C is a TD type square matrix
$A^{(D)} = A,\ B^{(D)} = B$
: $A^{(D)}$ and $B^{(D)}$ are double precision $n \times n$ square matrices.
$\alpha = 1$
$While(\ \alpha < 3)$
$M_A(i)_{1\le i\le n} = max_{1\le k\le n} |A^{(D)}{}_{i,k}|$: M_A is an nth − order vector
$M_B(j)_{1\le j\le n} = max_{1\le p\le n} |B^{(D)}{}_{p,j}|$: M_B is an nth − order vector
$T_A(i)_{1\le i\le n} = 2\text{^}(ceil\left(log2\left(M_A(i)\right)\right) + ceil\left(\frac{53 + log2(n)}{2}\right))$
: T_A is an nth − order vector
$T_B(j)_{1\le j\le n} = 2\text{^}(ceil\left(log2\left(M_B(j)\right)\right) + ceil\left(\frac{53 + log2(n)}{2}\right))$
: T_B is an nth − order vector
$S_A = T_A \cdot E^t$: $E = (1, 1, 1, \ldots, 1)^t$ is an nth − order vector
$S_B = E \cdot {T_B}^t$
$A_\alpha = \left(A^{(D)} + S_A\right) - S_A$
$B_\alpha = \left(B^{(D)} + S_B\right) - S_B$
$A = A - A_\alpha,\ B = B - B_\alpha$: Calculated by TD calculation
$A^{(D)} = A,\ B^{(D)} = B$
$\alpha = \alpha + 1$
$End\ While$
$A_\alpha = A^{(D)} - A_{\alpha-1}$
$B_\alpha = B^{(D)} - B_{\alpha-1}$
$For(\alpha = 1;\ \alpha < 4;\ \alpha++)$
$For(\beta = 1;\ \beta < 5 - \alpha;\ \beta++)$
$C_\beta = A_\alpha \times B_\beta$: Calculated by Dgemm of intel MKL
$End\ For$
$C\ += \sum_{k=1}^{4-\alpha} C_k$: Calculated by TD addition
$End\ For$

4 Ozaki Scheme for Triple-Single (TS) Precision on GPU

The concrete implementation of the TS precision Ozaki scheme is described as shown in Algorithm 16.

Primarily, it is similar to Algorithm 15 for TD Ozaki scheme, however there are different points caused by change of base precision from binary64 to binary32. For example, A and B are TS matrices constructed from elements of three binary32 numbers, and $A^{(S)}$, $B^{(S)}$, S_A, S_B M_A, M_B, A_α, and B_α are also binary32 matrices. In addition, SGEMM for $A_\alpha \times B_\beta$ is performed using CUDA's cuBLAS Sgemm [12].

Algorithm 16. Ozaki scheme algorithm used for the created TS precision matrix multiplication (in the case of three splits, n × n matrix)

$Input: A, B$: A and B are TS precision square matrices
$Output: C$: C is a TS precision square matrix
$A^{(S)} = A,\ B^{(S)} = B$
: $A^{(S)}$ and $B^{(S)}$ are single precision $n \times n$ square matrices.
$\alpha = 1$
$While(\alpha < 3)$
$M_A(i)_{1\le i\le n} = max_{1\le k\le n} |A^{(S)}{}_{i,k}|$: M_A is an nth − order vector
$M_B(j)_{1\le j\le n} = max_{1\le p\le n} |B^{(S)}{}_{p,j}|$: M_B is an nth − order vector
$T_A(i)_{1\le i\le n} = 2\hat{}(ceil\left(log2\left(M_A(i)\right)\right) + ceil\left(\frac{24 + log2(n)}{2}\right))$
: T_A is an nth − order vector
$T_B(j)_{1\le j\le n} = 2\hat{}(ceil\left(log2\left(M_B(j)\right)\right) + ceil\left(\frac{24 + log2(n)}{2}\right))$
: T_B is an nth − order vector
$S_A = T_A \cdot E^t$: $E = (1, 1, 1, \ldots, 1)^t$ is an nth − order vector
$S_B = E \cdot {T_B}^t$
$A_\alpha = \left(A^{(S)} + S_A\right) - S_A$
$B_\alpha = \left(B^{(S)} + S_B\right) - S_B$
$A = A - A_\alpha,\ B = B - B_\alpha$: Calculated by TS calculation
$A^{(S)} = A,\ B^{(S)} = B$
$\alpha = \alpha + 1$
$End\ While$
$A_\alpha = A^{(S)} - A_{\alpha-1}$
$B_\alpha = B^{(S)} - B_{\alpha-1}$
$For(\alpha = 1;\ \alpha < 4;\ \alpha++)$
$For(\beta = 1;\ \beta < 5 - \alpha;\ \beta++)$
$C_\beta = A_\alpha \times B_\beta$: Calculated by Sgemm of cuBLAS
$End\ For$
$C \mathrel{+}= \sum_{k=1}^{4-\alpha} C_k$: Calculated by TS addition
$End\ For$

5 Performance Evaluation by TD and TS Matrix Multiplications

We show the results of the benchmark tests performed on both CPUs and GPUs, whose execution environments are show in Table 1. TD matrix multiplication is performed on CPUs, and TS matrix multiplication on GPUs. We use intel Core i7 6700k and intel Core i7 11700 as CPUs, and NVIDIA GeForce GTX 1080 and NVIDIA GeForce RTX 3070 as GPUs. Intel MKLs with multi-thread facilities are also utilized to achieve the best performance of the Ozaki scheme.

The elements of matrices used for the benchmark test are random numbers calculated by $(ru - 0.5) \times exp(1.0 \times rn)$, where ru is a uniform random number of [0, 1], and rn is a random number associated with the standard normal distribution. Our benchmark tests are categorized into the following four types:

1. Execution time and relative error comparison of matrix multiplication using TD precision 12-split Ozaki scheme and matrix multiplication using Strassen + AVX2 on CPU.
2. Comparison of execution time using TD precision 12-split Ozaki scheme between the serial matrix multiplication and parallelized one using OpenMP on CPU.
3. Profiling the execution time among matrix size of matrix multiplication split, multiplication, and addition using TD 12-split Ozaki scheme + OpenMP on CPU.
4. Execution time comparison of matrix multiplication using TS precision 7 to 12-split Ozaki scheme and matrix multiplication using shared memory on GPU.

The maximum relative error is calculated by (1), where $\boldsymbol{E^*_{i,j}}$ represents the operation result of the QD type matrix multiplication, and $\boldsymbol{E_{i,j}}$ represent the elements of the result of the matrix multiplication of the Ozaki scheme.

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

Table 1. Execution environment of i7 6700K (left) and i7 11700 (right)

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**
CUDA	**10.1**

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**
CUDA	**10.1**

5.1 Comparison with Matrix Multiplication Execution Time of TD 12-Split Ozaki Scheme and TD Strassen + AVX2 on CPU

The comparison of the execution time of the matrix multiplication using the 12-division Ozaki scheme of TD precision and matrix multiplication using Strassen + AVX2 of TD precision is shown in Fig. 3. The matrix size (N) was 500 to 5000, and the performance was evaluated at 500 pitch of matrix size. For performance comparison, we used the matrix multiplication using Strassen + AVX2 as counterpart.

The left part of Fig. 4. uses the i7 6700k, and the right part uses the i7 11700 for performance evaluation. On i7 6700K, we can observe that the matrix product using the TD Ozaki scheme is up to approximately 4.5 times faster than the matrix multiplication using Strassen + AVX2. On the right of Fig. 3, the i7 11700 shows that the matrix product using the TD Ozaki scheme is approximately 4.5 times faster than the matrix product using Strassen + AVX2. At N = 4500, Strassen + AVX2 is much slower and the speed difference with the Ozaki scheme is significant.

Figure 4 shows the comparison with their relative error of up to 5000 as matrix size (N). The number of split of matrices in the Ozaki scheme has trade-off relation with the accuracy of the final TD matrix C. While we can gain accurate C by increasing the number of splits, the performance becomes slower owing to more number of DGEMM and TD additions.

We can observe that the Ozaki scheme has better accuracy than Strassen + AVX2 when the number of splits exceeds 10. As the matrix size increases, the accuracy of the Ozaki scheme gradually deteriorates until the matrix size (N) is up to 5000, however if

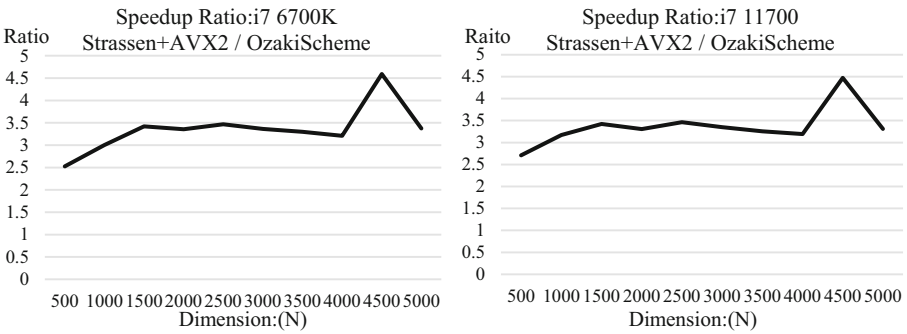

Fig. 3. Speedup ratio of matrix multiplication using TD precision 12-split Ozaki scheme and Strassen + AVX2 in Core i7 6700K (left) and Core i7 11700 (right)

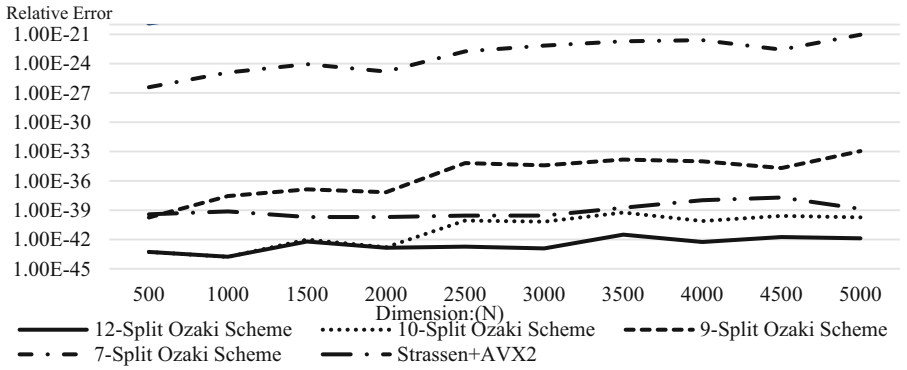

Fig. 4. TD type Ozaki scheme 7–12 division and relative error comparison of matrix product using Strassen + AVX2

the matrix size is larger than 5000, it is possible that more splits are required. Up to N = 5000, the maximum accuracy is achieved with 12 splits.

5.2 Comparison of Execution Time Between Matrix Multiplication Using the TD 12-Split Ozaki Scheme and Parallelized One with OpenMP on CPUs

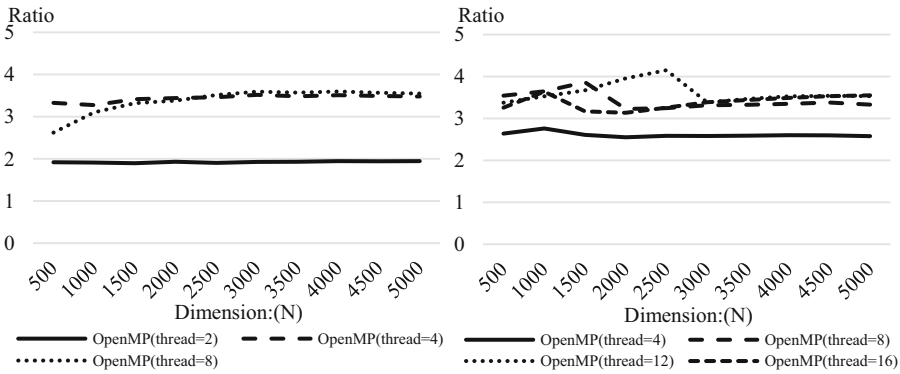

Fig. 5. It shows how fast it is for each number of threads used compared to the execution time when OpenMP is not used in Core i7 6700K (left) and Core i7 11700 (right).

Subsequently, we evaluated parallelization of the process of the Ozaki scheme with OpenMP. At this time, the parts of split and addition from original matrices were parallelized because of the shortening development time.

The comparison of the execution time with matrix multiplication using the TD type 12-split Ozaki scheme and parallelized one with OpenMP is shown in Fig. 5. The matrix size (N) was 500 to 5000, and the performance was evaluated in 500 pitch as matrix size. The left part of Fig. 5 is the performance on i7 6700K, and right part on i7 11700. According to these figures, we can observe that parallelization with OpenMP is not more effective than expected and that the speedup ratio by parallelization in small matrix size is better than in larger matrix size. The analysis of parallelization will be described later.

5.3 Profiling Execution Time by Matrix Size of Matrix Multiplication Split, Multiplication, and Addition Using TD 12-Split Ozaki scheme + OpenMP on CPU

We show the result of profiling the execution time constructed by split, multiplication, and addition in the matrix multiplication using the TD type 12-split Ozaki scheme + OpenMP. The matrix size (N) was 500 to 5000, and the performance was evaluated at 500 as matrix size. The left part of Fig. 6 shows the profile on the i7 6700k and the right part of Fig. 6 on the i7 1700. As the matrix size of i7 6700K and i7 11700 increased, the multiplication ratio in total execution time increased.

As shown in Fig. 6, we understand that the parallelization performance causes the multiplication ratio in the Ozaki scheme to occupy the major part of the computation time.

Because multiplication is calculated using the DGEMM of intel MKL, the parallelization of OpenMP is considered inapplicable to the entire process of our implementation. As previously described, the parallelization with OpenMP is performed only in the part of the split and addition from the original matrices; therefore we believe that acceleration can be achieved by parallelizing the usage of multiple MKL DGEMM processes with OpenMP.

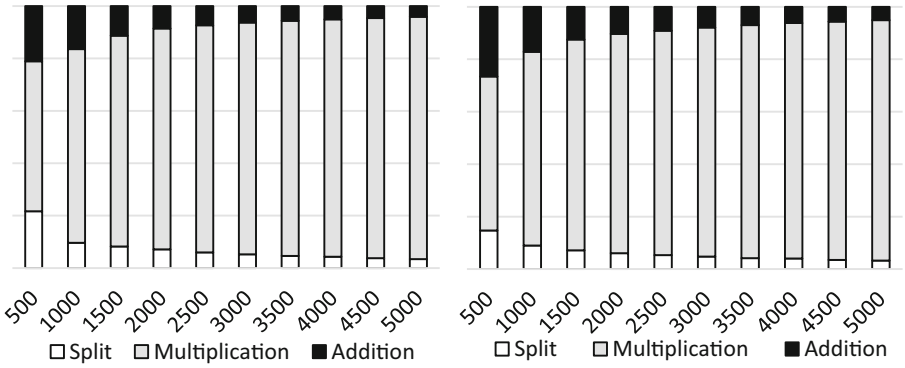

Fig.6. Ratio of execution time by matrix size of matrix multiplication split, multiplication, addition using TD 12-split Ozaki scheme in Core i7 6700K (left) and Core i7 11700 (right)

5.4 Speed Comparison of Matrix Multiplication Using TS Precision 7 to 12-Split Ozaki Scheme and Blocked Using Shared Memory on GPU

The execution times of the 7 to 12-split TS Ozaki scheme and the TS blocked matrix product are shown in Fig. 7. The matrix size (N) was 512 to 4096, and the performance was evaluated in 512 as matrix size pitch. The matrix multiplication was blocked by 32 $\times$ 32. "tsmatmul" represents the execution time of a TS simple matrix multiplication. As environment of GPU, Fig. 7 (left) uses GeForce GTX 1080 and Fig. 7 (right) uses GeForce RTX 3070.

It is observed that the TD precision Ozaki scheme is inefficient for GPUs without support of special hardware, such as TensorCore, and that the TS simple matrix multiplication using shared memory is faster than using the TS Ozaki scheme. Furthermore, the matrix multiplication using the TS Ozaki scheme is faster on GTX 1080 than on RTX 3070. TS simple matrix multiplication on RTX3070 is faster than on GTX1080, because memory access of Ozaki scheme and unused Tensor core may be the cause of delay.

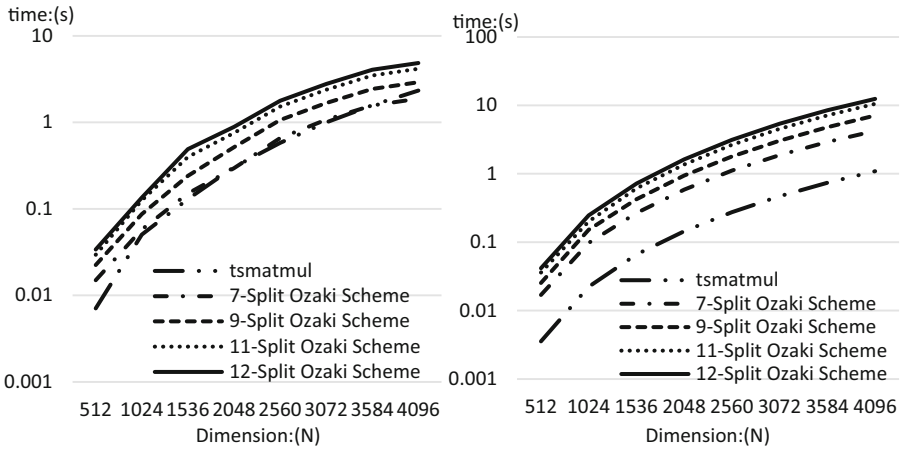

Fig. 7. Speed comparison of matrix multiplication using TS type 7 to 12 split Ozaki scheme and matrix multiplication using shared memory in GTX 1080 (left) and RTX 3070 (right)

6 Conclusion and Future Work

In this study, we explain the implementation of the matrix multiplication using the TD and TS Ozaki scheme and then evaluate their performances on CPUs and GPUs, respectively. We can conclude that on CPU, the TD Ozaki scheme is faster than the matrix multiplication using TD Strassen + AVX2. However, on GPU, the TS simple matrix multiplication using shared memory is faster than using the TS Ozaki scheme. We believe that lack of special hardware causes slow TS matrix multiplication with Ozaki scheme.

For our future work, we will solve the following issues:

1. Confirm how the usage the Tensor core on the GPUs are effective.
2. Review, on CPUs, the multiplication program to accelerate our current TD matrix multiplication that uses Intel MKL.
3. Implement and evaluate quadruple-double (QD) and quadruple-single (QS) precision matrix multiplication with the Ozaki scheme, OpenMP on CPUs and GPUs, respectively.

Acknowledgement. This study was supported by JSPS KAKENHI, Grant Number JP20K11843, and the Shizuoka Institute of Science and Technology. We acknowledge all organizations that continuously encourage our study and appreciate unknown referees who provide sharp and fruitful advice for polishing this paper.

References

1. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**, 224–242 (1971). https://doi.org/10.1007/BF01397083

2. Bailey, D.: QD. https://www.davidhbailey.com/dhbsoftware/
3. Lu, M., et al: supporting extended precision on graphics processors. In: Proceedings of the Sixth International Workshop on Data Management on New Hardware, pp. 19–26, June 2010
4. Fabiano, N., Muller, J.M., Picot, J.: Algorithms for triple words arithmetic. IEEE Trans. Comput. **68**, 1573–1583 (2019)
5. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**, 354–356 (1969)
6. Kouya, T.: Performance evaluation of multiple precision matrix multiplications using parallelized Strassen and Winograd algorithms. JSIAM Lett. **8**, 21–24 (2016). https://doi.org/10.14495/jsiaml.8.21
7. Ozaki, K., Ogita, T., Oishi, S.I., Rump, S.M.: Error free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. Numer. Algorithms **59**, 95–118 (2012)
8. Ichimura, S., Katagiri, T., Ozaki, K., Ogita, T., Nagai, T.: Threaded accurate matrix-matrix multiplications with sparse matrix-vector multiplications. In: IRRR International Parallel and Distributed processing Symposium Workshop (2018)
9. Mukunoki, D., Ozaki, K., Ogita, T., Imamura, T.: Accurate matrix multiplication on Binary128 format accelerated by Ozaki scheme (2021)
10. Utsugiri, Kouya: konsyumamuke GPU wo motiita 3baiseido (Triple Single) gyouretsuseki no seinouhyouka, HPC-182, pp. 1–8 (2021). (in Japanese)
11. Intel Corp.: Intel® oneAPI Math Kernel Library https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
12. Nvidia Corp.: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cublas/index.html