

Performance Evaluation of Strassen Matrix Multiplication Supporting Triple-Double Precision Floating-Point Arithmetic

Tomonori Kouya(✉)

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

Abstract. The Strassen matrix multiplication can be categorized into divide-and-conquer algorithms, and they are known as the most efficient algorithms. We previously implemented them supporting multiple precision floating-point arithmetic using MPFR and Bailey's QD libraries and have shown their effectiveness in our papers and open-source codes. In preparation for a future release, we have introduced an optimized triple-word floating-point arithmetic proposed by Fabiano et al., and we found its utility in our implementation of multiple precision matrix multiplication. In this paper, we demonstrate the effectiveness of the Strassen triple-double precision matrix multiplication through performance evaluation compared to those based on QD and MPFR libraries.

Keywords: Triple-word precision arithmetic · Matrix multiplication · Strassen algorithm

1 Introduction

All types of scientific computations require precise numerical results that satisfy users owing to their performance levels. If the targeting problems are ill-conditioned and cannot be solved with the standard IEEE 754 double precision floating-point arithmetic (binary64), then multiple precision floating-point arithmetic with a longer mantissa than binary64 must be used for their precise solutions. In this case, a reliable and stable multiple precision library, such as QD [3] by Bailey et al. or MPFR [4], should be chosen to solve the problems on the current computational environment. The QD library, which is one of the multi-component models of multiple precision floating-point arithmetic, supports only two types of floating-point arithmetic: double-double (DD, 106 bits of mantissa) and quad-double (QD, 212 bits). The QD library performs better for the same mantissa lengths than the MPFR, which supports any length of mantissa, based on the optimized natural number kernels of GNU MP (GMP) [5].

Triple-double (TD, 159 bits = 53 bits×3), which is based on triple-word arithmetic proposed by Flippo et al., is a type of multi-component model including

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12253, pp. 163–176, 2020.
https://doi.org/10.1007/978-3-030-58814-4_12

QD. They are expected to be faster than MPFR for the same length of mantissa and also offer a mid-level performance and accuracy when compared to DD and QD. Numerical experiments have shown a good performance TD computation compared with CAMPARY [2], which supports any length of mantissa based on binary32 and binary64.

We have already released multiple precision matrix multiplication library, BNCmatmul [10] using divide-and-conquer algorithms, including Strassen, and parallelized various precision matrix multiplication based on DD, QD, and MPFR libraries with OpenMP [7,8]. In comparison with Rgemm in MPLAPACK (MBLAS) [6], it produces better performance for large-sized matrices. We introduced TD arithmetic to BNCmatmul and then confirmed that TD matrix multiplication requires approximately five times larger computational time than DD and about half the computational time of QD.

In this paper, we describe the software layers of BNCmatmul and MBLAS, analyze how to parallelize the Strassen algorithm, compare the performance of BNCmatmul with Rgemm of MBLAS, and implement TD arithmetic based on triple-word arithmetic. Then, we carry out numerical experiments of TD Strassen matrix multiplication and compare their performance and accuracy with DD and QD on both Ryzen and Corei7 computational environments. Finally, we conclude our tasks and present our future works.

2 Structure of BNCmatmul and Its Performance

We have already released BNCmatmul, which features parallelized Strassen and Winograd multiple precision matrix multiplication based on OpenMP, QD-based C inline functions and MPFR. By utilizing the divide-and-conquer algorithms to reduce the number of arithmetic algorithms, high efficiency for large size matrix multiplications has been achieved over Rgemm in MBLAS and MPLAPACK.

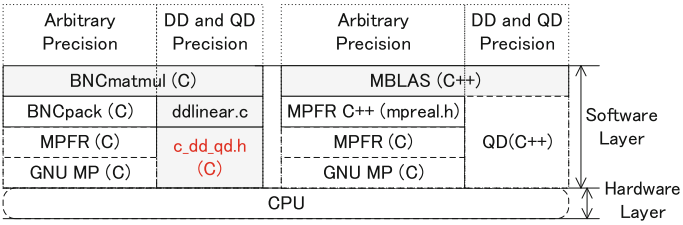

Fig. 1. Software layers of BNCmatmul and MPLAPACK(MBLAS)

In Fig. 1, we illustrate the differences and similarities of the software structure between BNCmatmul and MPLAPACK. Both these libraries are based on QD or QD-based C inline functions for DD and QD accuracy and MPFR for arbitrary accuracy. MPLAPACK is completely written in C++ and its classes (QD and MPFR C++). In contrast, BNCmatmul is written in C-language, QD-based

inline C functions (`c_dd_qd.h`), and original MPFR functions. Our implementation of the TD arithmetic is appended to `c_dd_qd.h`, reusing the same error-free transformation as incline C functions for the DD and QD arithmetic.

In the remaining section, the parallelized Strassen algorithms are explained and the performance of matrix multiplication provided by BNCmatmul is compared with Rgemm of MBLAS.

3 Parallelized Strassen and Winograd Algorithms

For given matrices A and B, the Strassen and Winograd algorithms for matrix multiplication $C := AB$, being categorized in the divide-and-conquer algorithms, can reduce the number of additions and multiplications compared to the normal matrix multiplication, particularly for large-sized matrices. Thus, these algorithms provide better performance when used together with large multiple precision floating-point arithmetic. We confirmed their efficiency through benchmark tests and applications of LU decomposition [7,8].

The algorithms we implemented in BNCmatmul and those in parallelization are illustrated in Fig. 2 and Fig. 3, where A, B and C are divided as

$$A := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B := \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
$$\Longrightarrow C := AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

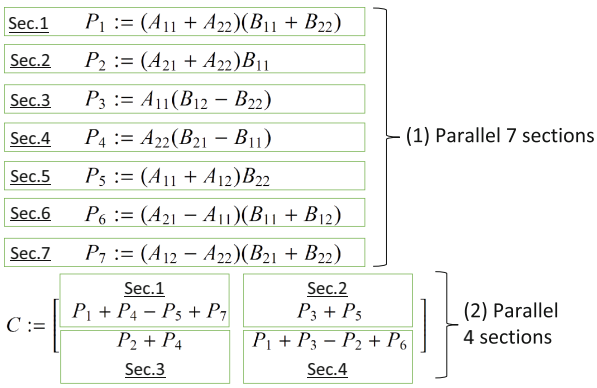

Fig. 2. Parallelized Strassen algorithm

The Strassen and Winograd matrix multiplications we implemented were parallelized in each parallelizable part of those constructed with groups of `omp section`. Although the Winograd matrix multiplication is less complex than the Strassen matrix multiplication, the part of recursive self-calls of both algorithms

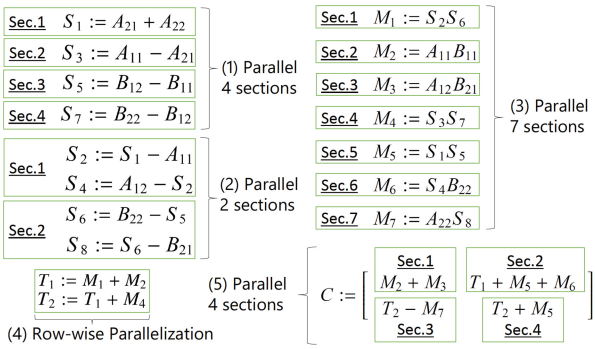

Fig. 3. Parallelized Winograd algorithms

can be divided and executed on seven threads. When the size of the matrices does not fit the Strassen and Winograd algorithms, our implementation requires parallelized dynamic padding and peeling to produce user-defined size of matrix products. Although it is well known that the number of computations of the Winograd algorithm is lesser than that of Strassen, no significant difference in computational time or effectiveness of parallelization was observed among these algorithms in DD, QD, and MPFR arbitrary precision arithmetic. Therefore, we focus on the Strassen algorithm in the rest of this paper.

Parallelization of the Strassen and Winograd matrix multiplication mentioned above can present effectiveness in DD and QD precision arithmetic, except in the case of MPFR, where our implementation produced lower performance than Rgemm.

3.1 Performance Comparison of Strassen Algorithms with MPLAPACK

We depict our results by the performance comparison of the implemented parallelized Strassen multiple precision matrix multiplication with Rgemm of MBLAS. The two computational environments, Ryzen and Corei7, are prepared for those evaluations as follows:

Ryzen AMD Ryzen 1700 (2.7 GHz), 32 GB RAM, Ubuntu 16.04.5 x86_64, GCC 5.4.0, MPACK 0.8.0, MPFR 4.0.2 [4]/GMP 6.1.2 [5], QD 2.3.22 [3], BNCpack 0.8 [9]

Corei7 Intel Core i7-9700K (3.6GHz), 16 GB RAM, Ubuntu 18.04.2 x86_64, GCC 7.3.0, MPACK 0.8.0, MPFR 4.0.2 [4]/GMP 6.1.2 [5], QD 2.3.22 [3], BNCpack 0.8 [9]

C++ options `g++ -O3 -std=c++11 -fopenmp`

MPLAPACK, which was downloaded from GitHub on December 2019, was installed with QD and MPFR/GMP libraries included in it. BNCmatmul is also commonly utilized with these libraries. Strassen algorithms are recursively

called until the number of rows or columns of divided matrices is less than or equal to 32.

MBLAS of MPLAPACK provides the same type of functions and order of arguments as the original reference BLAS. The elements of matrix are supposed to be stored as column-major order. The Rgemm used in comparison, is one of the functions in MBLAS to calculate $C := \alpha AB + \beta C$, so we set $\alpha = 1$ and $\beta = 0$ to obtain the result of a simple matrix multiplication of $C := AB$. All the elements of the matrices are stored along with in column-major layout for the benefit of Rgemm.

Aside from that, our implemented matrix multiplication functions are supposed to utilize matrices along with row-major layout and written in C-language, but we used common benchmark programs written in C++ to compare them with Rgemm.

The square matrices A and B, used in our benchmark tests, are as follows:

$$A = \left[\sqrt{5}\,(i+j-1)\right]_{i,j=1}^{n},\ B = \left[\sqrt{3}\,(n-i)\right]_{i,j=1}^{n}.$$

Due to all the positive real elements of A and B, precise elements of C can be derived through our benchmark tests of $C := AB$, and the lack of one or two significant decimal digits were found in all sizes and precision arithmetic.

QD-Based Matrix Multiplication. Figure 4, illustrates the performance comparison of serial Strassen, DD and QD matrix multiplication with Rgemm. These graphs depict plots of the ratio of computational time with those of Rgemm/Strassen. Thus, the plotted number over 1 means that our Strassen matrix multiplication is faster than Rgemm. In these figures, we show the thresholds of dimension and ratio (≥ 1) such as $767, 1.04$ in DD matrix multiplication on Ryzen7, over which our Strassen matrix multiplication is faster than Rgemm. Currently, there is no existence of parallelized Rgemm with the QD library, and those figures illustrate the results of serial matrix multiplication. As per the graphs, we can understand that our Strassen matrix multiplication is faster for large-sized matrices in any cases of DD or QD.

In Fig. 5, we illustrate the efficiency of our parallelized Strassen matrix multiplication. These figures plot the performance improvement ratio of computational time with 2, 4, and 8 threads against serial computational time. According to these figures, our parallelized Strassen matrix multiplication can achieve expected efficiency for both DD and QD arithmetic.

Due to these results for both DD and QD computations, TD Strassen matrix multiplication can be expected to show similar efficiency and performance improvement.

MPFR-Based Matrix Multiplication. For comparison with MPFR-based matrix multiplication, we illustrate our MPFR-based Strassen matrix multiplication against Rgemm in Fig. 6. MPFR Rgemm is parallelized with OpenMP and provides expected improvement. The Strassen implementation and Rgemm ratio

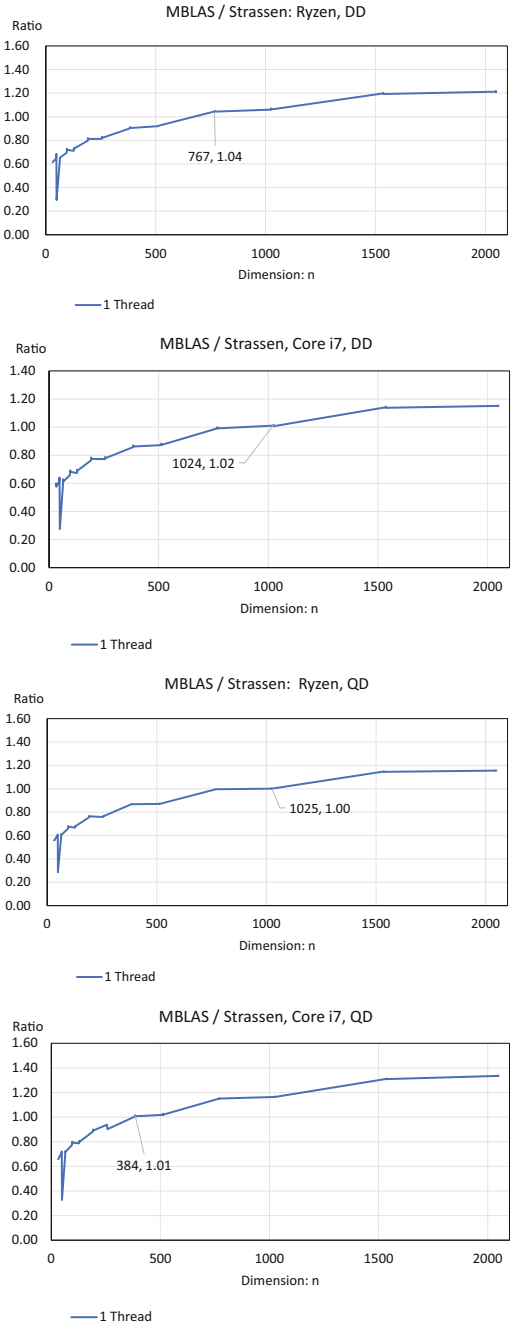

Fig. 4. Comparison of DD and QD matrix multiplication: BNCmatmul/Rgemm vs. Dimension of square matrices

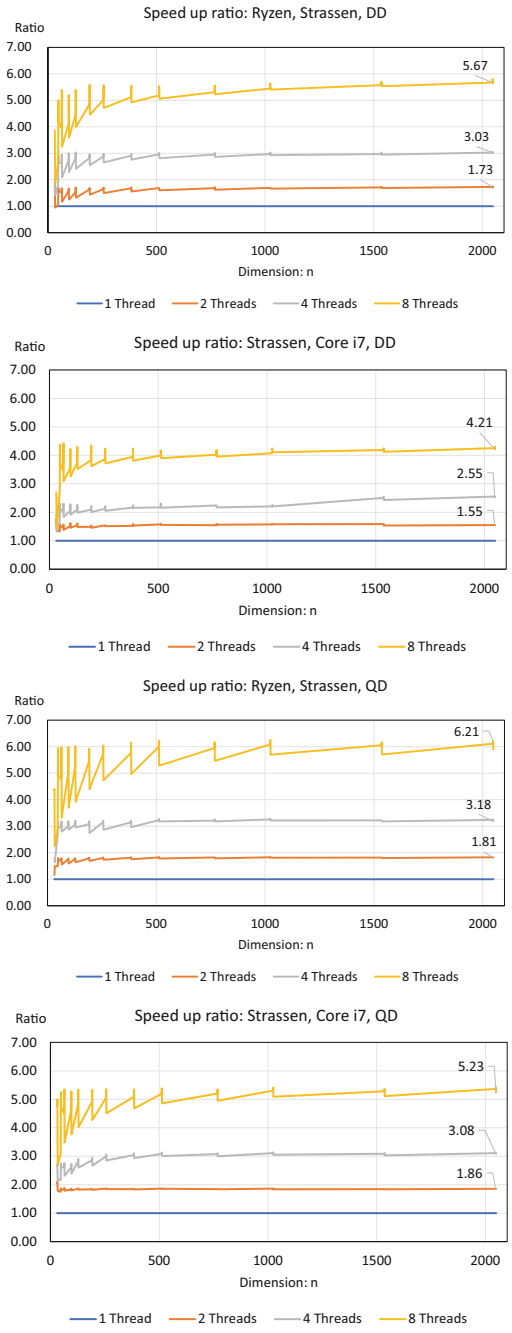

Fig. 5. Performance improvement ratio of parallelized DD an QD Strassen matrix multiplication

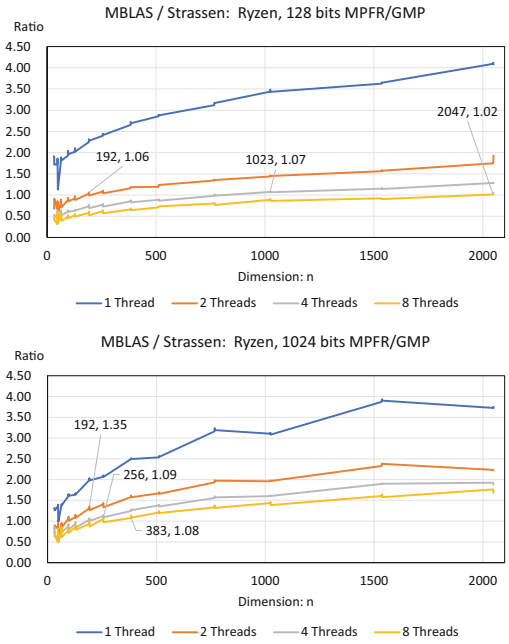

Fig. 6. Ratio of computational time of MPFR matrix multiplication: Rgemm/Strassen

will become smaller when the number of threads increases for both 128- and 1024-bits MPFR arithmetic.

Primarily, MPFR-based 106-bits matrix multiplication is approximately 18 times slower than the DD-based one, and MPFR-based 212-bits one about two times slower than the QD-based one. For the 159-bits TD computation, we observed that MPFR-based matrix multiplication is seven times slower than our TD-implemented one. In the rest of this paper, the benchmark tests of the TD arithmetic compare only the evaluation of DD and QD computations.

4 Triple-Double Matrix Multiplication

A well-optimized triple-word arithmetic was proposed by Fabiano et al. [1], which is one of the multi-component models of the multiple precision floating-point arithmetic, and is based on the error-free transformation techniques commonly used in DD and QD arithmetic. The triple-word arithmetic can be implemented by using the existing floating-point arithmetic such as IEEE single precision (binary32) or binary64. We choose binary64 to implement TD arithmetic as the middle precision between DD and QD computations. The TD matrix multiplication we implemented utilizes TD addition (TDadd) and TD multiplication (TDmul) in which they proposed a set of TD arithmetic. In the remaining section, we describe the algorithms for TD and its related arithmetic.

Error-free transformation algorithms: QuickTwoSum, TwoSum and TwoProd-FMA (Algorithm 1–3) are used as pieces to construct the TD arithmetic. In this paper, the entire floating-point variables are binary64, and $\oplus$, $\ominus$, $\otimes$ and FMA mean standard binary64 arithmetic.

Algorithm 1 $(s, e) := \mathrm{QuickTwoSum}(a, b)$

$s := a \oplus b$
$e := b \ominus (s \ominus a)$
return (s, e)

Algorithm 2 $(s, e) := \mathrm{TwoSum}(a, b)$

$s := a \oplus b$
$v := s \ominus a$
$e := (a \ominus (s \ominus v)) \oplus (b \ominus v)$
return (s, e)

TwoProd functions are implemented in two ways: one with utilizing Split function of binary64 value and another with utilizing fused multiply-add (FMA) arithmetic. In our implementation of TD arithmetic, we choose TwoProd-FMA because it is the simplest and native FMA arithmetic that is available on both Ryzen and Core i7 computational environments.

Algorithm 3 $(p, e) := \mathrm{TwoProd}(a, b)$

$p := a \otimes b$
$e := \mathrm{FMA}(a, b, -p)$ $(= a \times b - p)$
return (p, e)

4.1 Algorithms of TD Arithmetic

Triple-word precision floating-point numbers are constructed by three pieces of existing floating-point numbers. We express each element of triple-word floating-point number as $x := (x_0, x_1, x_2)$, in which all the pieces x_i $(i = 0, 1, 2)$ are expressed with binary64.

Renormalization: VecSum and VSEB(k). All types of triple-word arithmetic require renormalization just before returning the results as correct triple-word floating-point number. For this process, Fabiano et al. proposed the combination of VecSum (Algorithm 4) and VSEB(k) (VecSumErrBlanch, Algorithm 5).

Algorithm 4 $(e_0, ..., e_{n-1}) := \mathrm{VecSum}(x_0, ..., x_{n-1})$

$s_{n-1} := x_{n-1}$
for $i = n-2$ to 0 **do**
  $(s_i, e_{i+1}) := \mathrm{TwoSum}(x_i, s_{i+1})$
end for
$e_0 := s_0$
return $(e_0, ..., e_{n-1})$

Algorithm 5 $(y_0, ..., y_{k-1}) := \mathrm{VSEB}(k)(e_0, ..., e_{n-1})$

$j := 0$
$\epsilon_0 := e_0$
for $i = 0$ to $k-3$ **do**
  $(r_i, \epsilon_{i+1}^t) := \mathrm{TwoSum}(\epsilon_i, e_{i+1})$
  if $\epsilon_{i+1}^t \neq 0$ **then**
    $y_j := r_i$
    $\epsilon_{i+1} := \epsilon_{i+1}^t$
    $j := j + 1$
  else
    $\epsilon_{i+1} := r_i$
  end if
end for
$(y_j, y_{j+1}) := \mathrm{TwoSum}(\epsilon_{k-2}, e_{k-1})$
$y_{j+2} := 0, ..., y_{k-1} := 0$
return $(y_0, ..., y_{k-1})$

TDadd and TDmul. TDadd (Algorithm 6), addition of two triple-word numbers, add $x := (x_0, x_1, x_2)$ and $y := (y_0, y_1, y_2)$, and then produce the sum $r = (r_0, r_1, r_2) := x + y$. At first, it merges x and y and then executes VecSum and VSEB(3) to renormalize and return r.

Algorithm 6 $r := \mathrm{TDadd}(x, y)$

$(z_0, ..., z_5) := \mathrm{Merge}(x_0, x_1, x_2, y_0, y_1, y_2)$
$(e_0, ..., e_5) := \mathrm{VecSum}(z_0, ..., z_5)$
$(r_0, r_1, r_2) := \mathrm{VSEB}(3)(e_0, ..., e_5)$
return (r_0, r_1, r_2)

Fabiano et al. proposed two types of triple-word multiplications, one is more accurate and the other one is faster. We choose the faster version of multiplication (Algorithm 7) due to the lesser number of computations.

TDadd and TDmul are only required for matrix multiplication. For this, we implement the arithmetic with error-free transformation functions, which have been prepared in `c_dd_qd.h`. Therefore, we can compare the performance and accuracy of DD, QD, and TD matrix multiplications with common the single inline C library.

Algorithm 7 $r := \mathrm{TDmul}(x, y)$

$(z_{00}^{\mathrm{up}}, z_{00}^{\mathrm{lo}}) := \mathrm{TwoProd}(x_0, y_0)$
$(z_{01}^{\mathrm{up}}, z_{01}^{\mathrm{lo}}) := \mathrm{TwoProd}(x_0, y_1)$
$(z_{10}^{\mathrm{up}}, z_{10}^{\mathrm{lo}}) := \mathrm{TwoProd}(x_1, y_0)$
$(b_0, b_1, b_2) := \mathrm{VecSum}(z_{00}^{\mathrm{lo}}, z_{01}^{\mathrm{up}}, z_{10}^{\mathrm{up}})$
$c := \mathrm{FMA}(x_1, y_1, b_2)$
$z_{31} := \mathrm{FMA}(x_0, y_2, z_{10}^{\mathrm{lo}})$
$z_{32} := \mathrm{FMA}(x_2, y_0, z_{01}^{\mathrm{lo}})$
$z_3 := z_{31} \oplus z_{32}$
$s_3 := c \oplus z_3$
$(e_0, e_1, e_2, e_3) := \mathrm{VecSum}(z_{00}^{\mathrm{up}}, b_0, b_1, s_3)$
$r_0 := e_0$
$(r_1, r_2) := \mathrm{VSEB}(2)(e_1, e_2, e_3)$
return (r_0, r_1, r_2)

The complexity of DD, TD, and QD arithmetic are shown in Table 1. The addition and multiplication of DD and QD arithmetic are fast (sloppy) versions in the QD library.

Table 1. Number of elementary arithmetic

	$\oplus, \ominus$	$\otimes$	FMA	if	Machine epsilon
DDadd and DDmul	21	3	1	0	1.2e–32
TDadd and TDmul	80	3	5	8	1.4e–48
QDadd and QDmul	294	13	10	10	1.5e–64

The total amount of computations shows that TDadd and TDmul require about 3.8 times more complexity compared to that of DDadd and DDmul, and they require about 0.3 times compared with QDadd and QDmul. We can expect a mid-level performance and accuracy between DD and QD arithmetic.

4.2 Benchmark Tests for TD-Based Matrix Multiplication

The computational time (unit: second) obtained through our benchmark tests with 1023–1025 and 2047–2049 dimensional square matrix multiplications are shown in Fig. 7. In these figures, we can see each seconds of DD, TD, and QD Strassen matrix multiplication. The Ryzen and Corei7 processors have 8 cores, and, thus, we depict serial and parallelized computational time with OpenMP with 8 threads. Table 2 shows maximum relative errors in all elements of C generated from computations of Fig. 7, so we can confirm that our matrix multiplications obtained expected accuracy with DD, TD and QD arithmetic.

According to these figures, any dimensional TD matrix multiplication requires about five times of DD and approximately half of QD for serial and

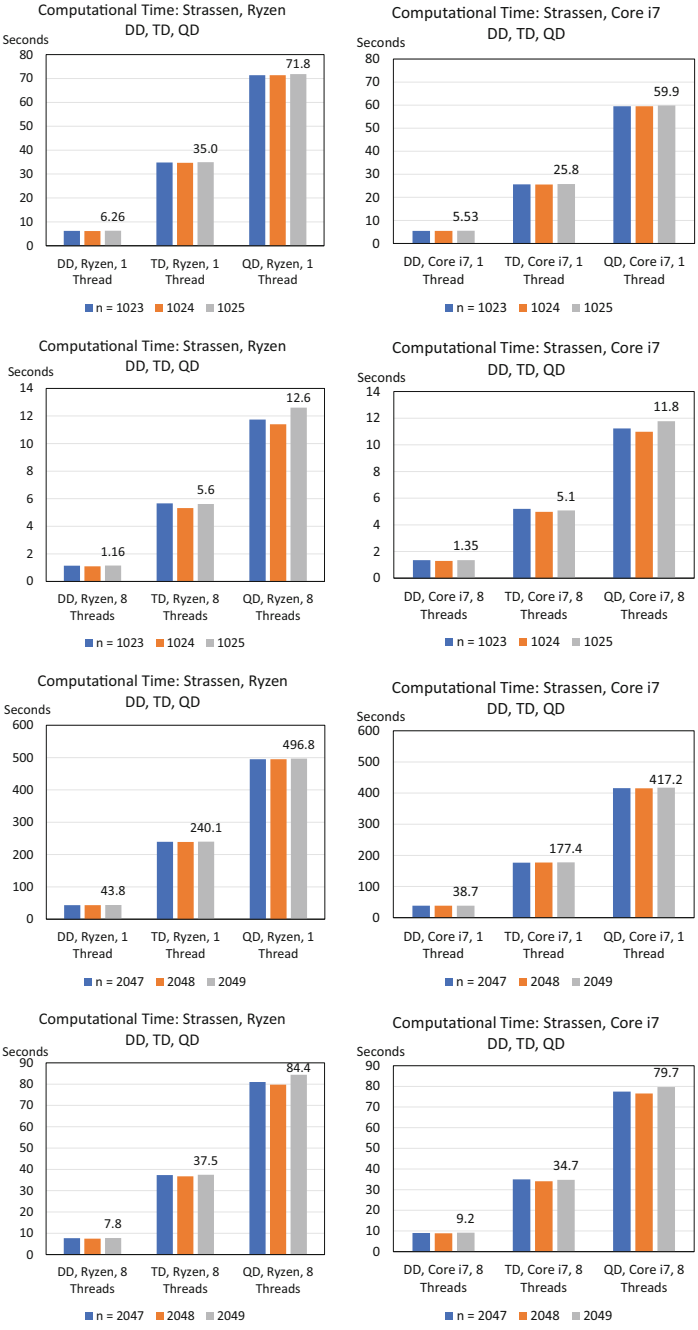

Fig. 7. Computational time (unit: second) of TD Strassen matrix multiplication

Table 2. Maximum relative error of elements of C arisen by Strassen algorithm

Dimension	1023	1024	1025
DD	4.8e–30	4.3e–30	3.4e–30
TD	1.3e–46	1.3e–46	1.4e–46
QD	4.3e–63	4.7e–63	5.0e–63
Dimension	2047	2048	2049
DD	9.3e–30	9.9e–30	1.1e–29
TD	3.2e–46	4.0e–46	3.4e–46
QD	1.2e–62	1.1e–62	9.8e–63

parallel cases. These ratios against DD and QD matrix multiplications are reasonable, as shown in Table 1.

The performance improvement ratio of TD Strassen matrix multiplication is illustrated in Fig. 8.

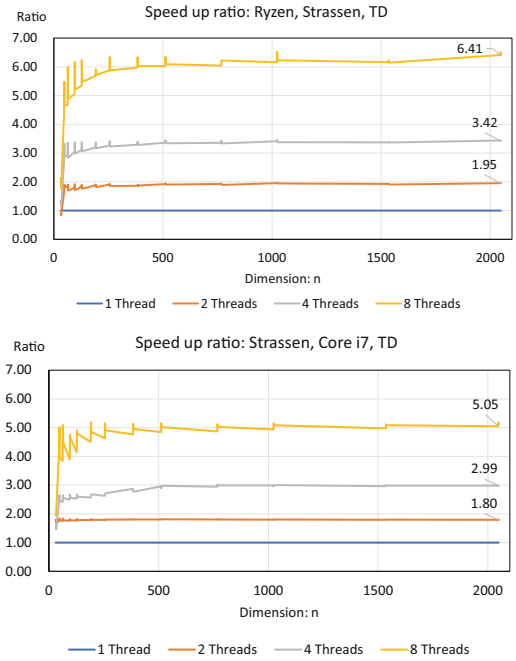

Fig. 8. Performance improvement ration of parallelized TD Strassen matrix multiplication

For any dimensional square matrices, parallelized TD Strassen shows similar level of speedup ratio against the number of OpenMP threads compared with Fig. 5.

5 Conclusion and Future Works

We implemented TD arithmetic based on Fabiano et al.'s proposed triple-word floating-point arithmetic and demonstrated that we can obtain the expected performance and accuracy for TD Strassen matrix multiplication. They require approximately five times more computational time compared to DD matrix multiplication and about half computational time against QD. The parallelization improvement ratio with OpenMP can be achieved at the expected levels as compared with DD and QD computations. Currently, the QD library does not contain a TD class; thus, we cannot compare our TD matrix multiplication with MPLAPACK, but our TD matrix multiplication would be faster than Rgemm of MPLAPACK for larger sizes of matrices if TD class existed.

For future works, we will release a new BNCmatmul, including TD implementation, and verify the efficiency for TD of LU decomposition. We can also customize DD, TD, and QD arithmetic efficiently according to users' requirements and achieve accuracy under 212-bits mantissa.

References

1. Fabiano, N., Muller, J.-M., Picot, J.: Algorithms for triple-word arithmetic. IEEE Trans. Comput. **68**(11), 1573–1583 (2019)
2. Joldes, M., Muller, J.-M., Popescu, V., Tucker, W.: CAMPARY: cuda multiple precision arithmetic library and applications. In: Greuel, G.-M., Koch, T., Paule, P., Sommese, A. (eds.) ICMS 2016. LNCS, vol. 9725, pp. 232–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42432-3_29
3. Bailey, D.H.: QD. https://www.davidhbailey.com/dhbsoftware/
4. MPFR Project, The MPFR library. https://www.mpfr.org/
5. Granlaud, T.: GMP development team: The GNU Multiple Precision arithmetic library. https://gmplib.org/
6. Nakata, M.: MPLAPACK(MBLAS). https://github.com/nakatamaho/mplapack
7. Kouya, T.: Accelerated multiple precision matrix multiplication using Strassen's algorithm and Winograd's variant. JSIAM Lett. **6**, 81–84 (2014)
8. Kouya, T.: Performance evaluation of multiple precision matrix multiplications using parallelized Strassen and Winograd algorithms. JSIAM Lett. **8**, 21–24 (2016)
9. Kouya, T.: BNCpack. https://na-inet.jp/na/bnc/
10. Kouya, T.: BNCmatmul. https://na-inet.jp/na/bnc/bncmatmul-0.2.tar.bz2