

Acceleration of Multiple Precision Matrix Multiplication Based on Multi-component Floating-Point Arithmetic Using AVX2

Tomonori Kouya(✉)

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

Abstract. In this paper, we report the results obtained from the acceleration of multi-binary64-type multiple precision block and Strassen matrix multiplications with AVX2. We target double-double (DD), triple-double (TD), and quad-double (QD) precision arithmetic designed using certain types of error-free transformation (EFT) arithmetic. Furthermore, we implement SIMDized EFT functions, which simultaneously compute with four binary64 numbers on x86_64 computing environment, and by using help of them, we also develop SIMDized DD, TD, and QD additions and multiplications. In addition, AVX2 load/store functions were adopted to efficiently speed up reading and storing matrix elements from/to memory. Owing to these combined techniques, our implemented multiple precision matrix multiplications were accelerated more than three times compared with non-accelerated ones. Our accelerated matrix multiplication modifies parallelization performance with OpenMP.

Keywords: Multiple precision floating-point arithmetic · Matrix multiplication · AVX2 · Parallelization

1 Introduction

The scale of scientific computation is becoming increasingly large, and obtaining the user-required accuracy of approximation for ill-conditioned cases requires more fractional digits in floating-point numbers than for a IEEE754 binary standard, such as binary32 (24-bits fraction) or binary64 (53-bits). To obtain a precise approximation, it is critical to adopt multiple precision floating-point (MPF) arithmetic, which can handle more fractional digits than binary64.

The implementation of MPF arithmetic is currently categorized into two groups: multi-digit type, based on integer arithmetic, and multi-component-type (or multi-term), which is constructed by several binary32 or binary64 numbers. The QD library by Bailey et al. [1] represents the latter, whereas the former is represented by GNU MPFR (Multiple Precision Floating-point Reliable

Supported by JSPS KAKENHI (Grant Number JP20K11843) and Shizuoka Institute of Science and Technology.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12953, pp. 202–217, 2021.
https://doi.org/10.1007/978-3-030-86976-2_14

library) [8], which is based on the multiple precision natural number arithmetic (MPN) kernel of GNU MP (GNU Multiple Precision arithmetic library) [12]. Both libraries are coded using C and C++ languages, and the MPN kernel is accelerated by assembler codes for various CPU architectures.

MPLAPACK based on MPBLAS [9] is an efficient MPF linear computation library that employs QD, MPFR, and bridge functions. ATLAS [11] and OpenBLAS [10] were created because the original reference BLAS, based on simple Fortran codes, has plenty of room for acceleration, and further accelerated MPF BLAS libraries are expected.

It is well known that multi-component MPF arithmetic can be accelerated using SIMD instructions, such as AVX2. Lis [6,13] is one of the most optimal AVX2-based linear and sparse linear computation libraries, and it has achieved over three times speedup using AVX2 [14].

In this paper, we report the performance of our implemented binary64-based multi-component MPF block and Strassen matrix multiplications using a technique similar to that of Lis. The targeted precisions are double-double (DD), triple-double (TD), and quadruple-double (QD) arithmetic. We adopted SIMDized error-free transformation functions on the x86_64 CPU architecture to extend the performance of these three precision MPF linear computations. QD-based addition is also utilized to obtain better performance in TD arithmetic. In our implementation, a new data structure for the MPF vector and matrix is introduced to accelerate the load and store its elements with AVX2. Owing to these efforts, we realized a higher performance for all types of MPF arithmetic than those of the Rgemm function of MPBLAS. Furthermore, we modified the speed of parallelization with OpenMP for block matrix multiplication in the AMD EPYC environment.

We determined that our Strassen matrix multiplication could not be accelerated on a CPU with more than eight cores, and that block matrix multiplication performed better than Strassen matrix multiplication on such CPUs.

2 Acceleration of DD, TD, and QD Matrix Multiplication with AVX2

Multi-component MPF arithmetic is constructed using several hard-wired binary32 or binary64 numbers and various functions of the so-called error-free transformation (EFT) [3]. The DD, TD, and QD precision MPF numbers are constructed with two binary64, three binary64, and four binary64 numbers, respectively. The most significant component (one leading binary64 number) of DD, TD, and QD precision numbers possesses the most significant digits of the entire MPF number.

Using DD precision arithmetic, it has been reported that basic linear computation can be accelerated with SIMD instructions, including AVX2, because DD arithmetic does not include complicated branches in its addition, subtraction, multiplication, and division. In contrast, TD and QD arithmetic require renormalization processes, including some conditional branches. Therefore, we

are required to implement these renormalization as simple loops, where we consider each binary64 number in the `__m256d` data type of AVX2, including three or four binary64 numbers. Because we completed all DD, TD, and QD arithmetic using the `__m256d` data type as API interfaces, our MPF matrix multiplication library has been simply implemented and accelerated with AVX2.

2.1 Error-Free Transformation with AVX2

As we explained earlier, we only implemented the `__m256d` data type to express and accelerate linear computation with multi-component MPF arithmetic. Therefore, C APIs such as `_mm256_[add, sub, mul, div]_pd`, and `_mm256_fmadd_pd` as fused multiply add (FMA) [2], were embedded in the required EFT functions, such as QuickTwoSum, TwoSum, and TwoProd. Subsequently, these functions were implemented as AVX2QuickTwoSum (Algorithm 1), AVX2TwoSum (Algorithm 2), and AVX2TwoProd (Algorithm 3), respectively. In these SIMDized EFT functions, a, b, s and e are `__m256d` data type, and include four binary64 data such as $a = (a_0, a_1, a_2, a_3)$, $b = (b_0, b_1, b_2, b_3)$, $s = (s_0, s_1, s_2, s_3)$, and $e = (e_0, e_1, e_2, e_3)$.

Algorithm 1. $(s, e) :=$ AVX2QuickTwoSum(a, b)

$s :=$ `_mm256_add_pd(`a, b`)`
$e :=$ `_mm256_sub_pd(`b, `_mm256_sub_pd(`s, a`))`
return (s, e)

Algorithm 2. $(s, e) :=$ AVX2TwoSum(a, b)

$s :=$ `_mm256_add_pd(`a, b`)`
$v :=$ `_mm256_sub_pd(`s, a`)`
$e :=$ `_mm256_add_pd(_mm256_sub_pd(`a, `_mm256_sub_pd(`s, v`))`, `_mm256_sub_pd(`b, v`))`
return(s, e)

Algorithm 3. $(p, e) :=$ AVX2TwoProd(a, b)

$p :=$`_mm256_mul_pd(`a`,` b`)`
$e :=$`_mm256_fmadd_pd(`a`,` b`,` $-p$`)`
return(p, e)

2.2 SIMDized DD Addition and Multiplication

Because DD arithmetic is the simplest of the multi-component MPF arithmetic, its addition and multiplication needed for matrix multiplication, which directly adopt SIMDized EFT functions, can be easily implemented, as shown in Algorithm 4 and 5. One standard DD precision number is constructed with two binary64 as $x[2] = (x[0], x[1])$. Therefore, in SIMDized DD arithmetic, each array element of $x[2]$ is expressed as one `__m256d` variable, as $x[0] = (x[0]_0, x[0]_1, x[0]_2, x[0]_3)$ and $x[1] = (x[1]_0, x[1]_1, x[1]_2, x[1]_3)$. AVX2DDadd and AVX2DDmul simultaneously execute four calculations with $x[2]$ and $y[2]$, in which they have four DD numbers: $(x[0]_i, x[1]_i)$ and $(y[0]_i, y[1]_i)$ $(i = 0, 1, 2, 3)$. In these DD arithmetic, all temporary variables are expressed as the `__m256d` data type.

Algorithm 4. $r[2] :=$ AVX2DDadd$(x[2], y[2])$

$(s, e) :=$ AVX2TwoSum$(x[0], y[0])$
$w :=$`_mm256_add_pd`$(x[1], y[1])$
$e :=$`_mm256_add_pd`(e, w)
$(r[0], r[1]) :=$ AVX2QuickTwoSum(s, e)
return $(r[0], r[1])$

Algorithm 5. $r[2] :=$ AVX2DDmul$(x[2], y[2])$

$(p_1, p_2) :=$ AVX2TwoProd$(x[0], y[0])$
$w_1 :=$`_mm256_mul_pd`$(x[0], y[1])$
$w_2 :=$`_mm256_mul_pd`$(x[1], y[0])$
$w_3 :=$`_mm256_add_pd`(w_1, w_2)
$p_2 :=$`_mm256_add_pd`(p_2, w_3)
$(r[0], r[1]) :=$ AVX2QuickTwoSum(p_1, p_2)

As previously introduced, Lis [13] and MuPaT [14] realized a DD linear computation with AVX2 that is approximately three times faster. They utilized C macros to implement SIMDized DD arithmetic. In contrast to Lis and MuPat, we adopted the C static inline functions of the SIMDized EFT for DD, TD, and QD linear computations. In the remainder of this section, we describe these algorithms using AVX2 and their performance.

2.3 SIMDized TD Addition and Multiplication

Fabiano et al. proposed an optimized triple-word arithmetic [4]. In this arithmetic set, the renormalization of triple word numbers adopts VecSum and VSEB(k) (VecSum with Error Branch). Our SIMDized TD arithmetic can be implemented with AVX2 instruction, AVX2VecSum, and AVX2VSEB(n). AVX2VecSum can be fully implemented with AVX2 functions, whereas

AVX2VSEB(n) is not completely SIMDized owing to the conditional branch embedded in it.

TD addition (TDadd) calculates $r[3] = (r[0], r[1], r[2])$ as the sum of $x[3] = (x[0], x[1], x[2])$ and $y[3] = (y[0], y[1], y[2])$. The standard approach is to merge sorting all elements of $x[3]$ and $y[3]$, normalizing them by VecSum, and expressing $r[3]$ as the TD number by VSEB(3). The following AVX2TDadd (Algorithm 6) is implemented using AVX2. This algorithm is not sufficiently accelerated with AVX2 owing to the poor performance of the merge(-sort) function and incomplete AVXVSEB(n).

Algorithm 6. $r[3] :=$ AVX2TDadd($x[3]$, $y[3]$)

$(z_0, ..., z_5) :=$ AVX2Merge$(x[0], x[1], x[2], y[0], y[1], y[2])$
$(e_0, ..., e_5) :=$ AVX2VecSum$(z_0, ..., z_5)$
$(r[0], r[1], r[2]) :=$ AVX2VSEB(3)$(e_0, ..., e_5)$
return $(r[0], r[1], r[2])$

As demonstrated in the results obtained by benchmark tests, AVX2TDadd cannot be accelerated completely with AVX2. Therefore, as an alternative, we applied QDadd (Algorithm 8) as TD addition (TDaddq) with a substitution such as $x[3] = y[3] = 0$. In our TD matrix multiplication, we adopted TDaddq and AVX2TDaddq as default TD additions.

For TD multiplication, Fabiano et al. proposed an accurate version and a fast version. We selected the latter as the default TD multiplication (TDmul) and implemented AVX2TDmul with AVX2.

Algorithm 7. $r[3] :=$ AVX2TDmul($x[3]$, $y[3]$)

$(z_{00}^{\mathrm{up}}, z_{00}^{\mathrm{lo}}) :=$ AVX2TwoProd$(x[0], y[0])$
$(z_{01}^{\mathrm{up}}, z_{01}^{\mathrm{lo}}) :=$ AVX2TwoProd$(x[0], y[1])$
$(z_{10}^{\mathrm{up}}, z_{10}^{\mathrm{lo}}) :=$ AVX2TwoProd$(x[1], y[0])$
$(b_0, b_1, b_2) :=$ AVX2VecSum$(z_{00}^{\mathrm{lo}}, z_{01}^{\mathrm{up}}, z_{10}^{\mathrm{up}})$
$c :=$ `_mm256_fmadd_pd`$(x[1], y[1], b_2)$
$z_{31} :=$ `_mm256_fmadd_pd`$(x[0], y[2], z_{10}^{\mathrm{lo}})$
$z_{32} :=$ `_mm256_fmadd_pd`$(x[2], y[0], z_{01}^{\mathrm{lo}})$
$z_3 :=$ `_mm256_add_pd`(z_{31}, z_{32})
$s_3 :=$ `_mm256_add_pd`(c, z_3)
$(e_0, e_1, e_2, e_3) :=$ AVX2VecSum$(z_{00}^{\mathrm{up}}, b_0, b_1, s_3)$
$r[0] := e_0$
$(r[1], r[2]) :=$ AVX2VSEB(2)(e_1, e_2, e_3)
return $(r[0], r[1], r[2])$

As a trial approach, we implemented TDmulq, which was converted to QDmul for the TD multiplication, although it could not realize a higher speed than TDmul. Therefore, to implement the TD matrix multiplication, we adopted the TDaddq and TDmul sets, as well as those of AVX2TDaddq and AVX2TDmul.

2.4 SIMDized QD Addition and Multiplication

For QD arithmetic, we selected a lighter addition and multiplication (sloppy addition and multiplication), and then we implemented AVX2QDadd (Algorithm 8) and AVX2QDmul (Algorithm 9) with AVX2Renorm as the renormalization of QD numbers with AVX2.

Algorithm 8. $r[4] :=$ AVX2QDadd$(x[4], y[4])$

$s_0 :=$ `_mm256_add_pd`$(x[0], y[0])$
$s_1 :=$ `_mm256_add_pd`$(x[1], y[1])$
$s_2 :=$ `_mm256_add_pd`$(x[2], y[2])$
$s_3 :=$ `_mm256_add_pd`$(x[3], y[3])$
$v_0 :=$ `_mm256_sub_pd`$(s_0, x[0])$
$v_1 :=$ `_mm256_sub_pd`$(s_1, x[1])$
$v_2 :=$ `_mm256_sub_pd`$(s_2, x[2])$
$v_3 :=$ `_mm256_sub_pd`$(s_3, x[3])$
$u_0 :=$ `_mm256_sub_pd`(s_0, v_0)
$u_1 :=$ `_mm256_sub_pd`(s_1, v_1)
$u_2 :=$ `_mm256_sub_pd`(s_2, v_2)
$u_3 :=$ `_mm256_sub_pd`(s_3, v_3)
$w_0 :=$ `_mm256_sub_pd`$(x[0], u_0)$
$w_1 :=$ `_mm256_sub_pd`$(x[1], u_1)$
$w_2 :=$ `_mm256_sub_pd`$(x[2], u_2)$
$w_3 :=$ `_mm256_sub_pd`$(x[3], u_3)$
$u_0 :=$ `_mm256_sub_pd`$(y[0], v_0)$
$u_1 :=$ `_mm256_sub_pd`$(y[1], v_1)$
$u_2 :=$ `_mm256_sub_pd`$(y[2], v_2)$
$u_3 :=$ `_mm256_sub_pd`$(y[3], v_3)$
$t_0 :=$ `_mm256_add_pd`(w_0, u_0)
$t_1 :=$ `_mm256_add_pd`(w_1, u_1)
$t_2 :=$ `_mm256_add_pd`(w_2, u_2)
$(s_1, t_0) :=$ AVX2TwoSum(s_1, t_0)
$(s_2, t_0, t_1) :=$ AVX2ThreeSum(s_2, t_0, t_1)
$(s_3, t_0) :=$ AVX2ThreeSum2(s_3, t_0, t_2)
$t_0 :=$ `_mm256_add_pd`(`_mm256_add_pd`(t_0, t_1), t_3)
$(r[0], r[1], r[2], r[3]) :=$ AVX2Renorm$(s_0, s_1, s_2, s_3, t_0)$
return $(r[0], r[1], r[2], r[3])$

Algorithm 9. $r[4] := \mathrm{AVX2QDmul}(x[4], y[4])$

$s_0 :=$ `_mm256_add_pd`$(x[0],\ y[0])$
$(p_0, q_0) := \mathrm{AVX2TwoProd}(x[0], y[0])$
$(p_1, q_1) := \mathrm{AVX2TwoProd}(x[0], y[1])$
$(p_2, q_2) := \mathrm{AVX2TwoProd}(x[1], y[0])$
$(p_3, q_3) := \mathrm{AVX2TwoProd}(x[0], y[2])$
$(p_4, q_4) := \mathrm{AVX2TwoProd}(x[1], y[1])$
$(p_5, q_5) := \mathrm{AVX2TwoProd}(x[2], y[0])$
$(p_1, p_2, q_0) := \mathrm{AVX2ThreeSum}(p_1, p_2, q_0)$
$(p_2, q_1, q_2) := \mathrm{AVX2ThreeSum}(p_2, q_1, q_2)$
$(p_3, p_4, p_5) := \mathrm{AVX2ThreeSum}(p_3, p_4, p_5)$
$(s_0, t_0) := \mathrm{AVX2TwoSum}(p_2, p_3)$
$(s_1, t_1) := \mathrm{AVX2TwoSum}(q_1, p_4)$
$s_2 :=$ `_mm256_add_pd`(q_2, p_5)
$(s_1, t_0) := \mathrm{AVX2TwoSum}(s_1, t_0)$
$s_2 :=$ `_mm256_add_pd`$(s_2,$ `_mm256_add_pd`$(t_0, t_1))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$(x[0], y[3]))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$(x[1], y[2]))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$(x[2], y[1]))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$(x[3], y[0]))$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_0)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_3)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_4)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_5)$
$(r[0], r[1], r[2], r[3]) := \mathrm{AVX2Renorm}(p_0, p_1, s_0, s_1, s_2)$
return $(r[0], r[1], r[2], r[3])$

Excluding AVX2Renorm, all parts of AVX2QDadd and AVX2QDmul were sufficient for full implementation with AVX2; hence, a maximum increase of speed can be achieved with AVX2 rather than DD and TD. Additionally, they are completely faster than MPFR 212-bits matrix multiplication. Therefore, the results obtained from the SIMDized QD arithmetic verify that the over-quad-double MPF arithmetic with AVX2 is potentially more efficient than the MPFR arithmetic.

2.5 DD, TD, and QD MFLOPS with Element-Wise Addition and Multiplication of Vectors

The vector data structure that we implemented for multi-component MPF matrix multiplication is presented in this section. Although it is standard to use each MPF variable as one element of the array that expresses vector and matrix, we implemented one set of several arrays with each component of the vector or matrix element, as it is necessary for us to derive the maximum performance using the `_m256d` data-type of AVX2. Owing to these methods of expressing the MPF vector, highly performed continuous load and store instructions, such as `__mm256_load_pd` and `__mm256_store_pd` functions, can be utilized to read and store elements of a vector or matrix by four elements at once.

Therefore, our MPF vector and matrix data structure have two binary64, three binary64, and four binary64 arrays as DD, TD, and QD linear computations, respectively. By adopting these data structures, the TD linear computation case, as shown in Fig. 1, two sets of three calls of load functions can settle down two operands of AVX2TD[add, mul] (`bncavx2_rtd_[add, mul]`). Our adopted vector or matrix data structure for AVX2 can reduce the process time required to read and write, because it eliminates the need to read each binary64 number.

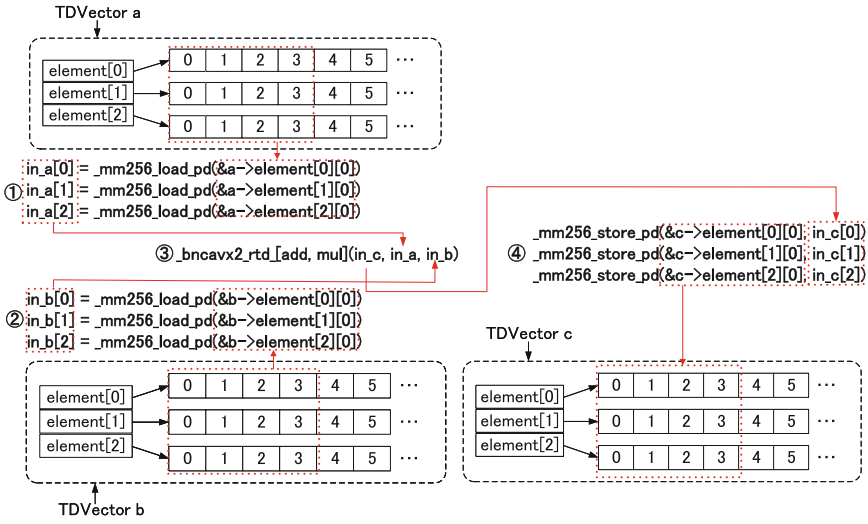

Fig. 1. TD vector data type that can be treated with load/store instructions

In the remainder of this paper, we adopted two types of computational environments to evaluate the implemented codes.

Corei9. Intel Core i9-10900X (3.6 GHz, 10 cores), 16 GB RAM Ubuntu 18.04.2, GCC 7.3.0

EPYC. AMD EPYC 7402P (1.5 GHz, 24 cores), 64 GB RAM, Ubuntu 18.04.5, GCC 7.5.0

AVX2 instruction has also a set of `__m256` data-type that holds eight binary32 numbers. For confirmation, we also implemented binary32-based multi-component SIMDized MPF arithmetic and ran benchmark tests on both Corei9 and EPYC environments, but their performance did not improve over those of binary64-based SIMDized MPF arithmetic.

We demonstrated DD mega floating-point operations per second (MFLOPS), TD MFLOPS, and QD MFLOPS on a Corei9 environment to compare with three types of element-wise addition and multiplication of vectors in Fig. 2, Fig. 3, and Fig. 4. In these FLOPS tests, we adopted two n-dimensional real vectors

$\mathbf{a} = [a_1\ a_2\ \ldots\ a_n]^T$ and $\mathbf{b} = [b_1\ b_2\ \ldots\ b_n]^T$ as random numbers. Thereafter, we evaluated the number of additions or multiplications per second by computing each element, such as $a_i + b_i$ or $a_i \cdot b_i$. "AVX2 L/S" depicts MFLOPS with SIMDized calculation and AVX2 Load/Store instructions, "AVX2 Set" SIMDized calculation and AVX2 Set function (`_mm256_set_pd`) and normal substitution to arrays, and complete "Normal" standard calculation without using any AVX2 instructions. The DD, TD, and QD MFLOPS values are presented on the vertical axis, and n is the length of the vectors in the horizontal axis.

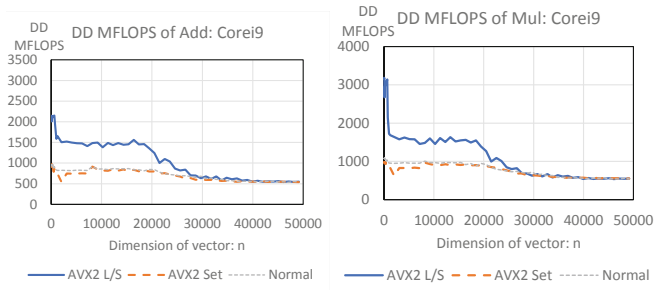

Fig. 2. DD MFLOPS of DD element-wise addition and multiplication of vectors

Figure 2 illustrates three types of DD MFLOPS. DD arithmetic requires less computation than TD or QD; therefore, the utilization of AVX2 load/store instructions is largely effective for the cache memory of a CPU. In fact, when all elements of the vectors are stored in cache memory, the DD MFLOPS is 1.5 times larger than the other two types of DD computations. For vector sizes beyond cache memory, both DD addition and multiplication for all types of calculations have the same performance.

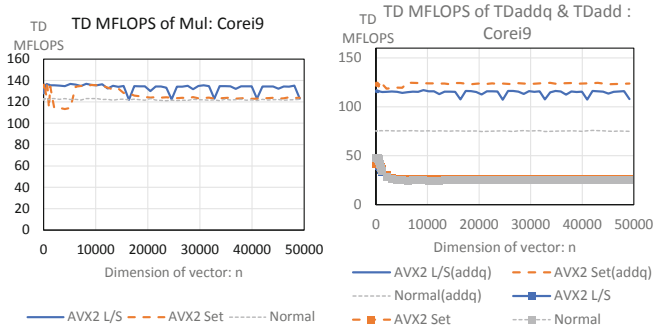

Fig. 3. TD MFLOPS of TD element-wise addition and multiplication of vectors

Figure 3 presents the TD MFLOPS for different processes. As can be observed in the MFLOPS of the original triple-double addition (TDadd and AVX2TDadd) presented in the left figure, they realized approximately 26 TD MFLOPS owing to the poor performance of the merge function in Algorithm 6; hence, SIMDization is ineffective for them. In contrast, TDaddq performed well at 75 TD MFLOPS, and AVX2TDaddq with load/store instructions also achieved approximately 115 TD MFLOPS, with AVX2 set at 123 TD MFLOPS. The TD addition to the Corei9 environment is an exceptional case in which the AVX2 load/store performance is less than that of the AVX2 set. Otherwise, TDmul cannot achieve sufficient efficiency with AVX2, as more than approximately 20 TD MFLOPS is achieved. Through these performance tests, we can infer that cache memory is not effective for TD arithmetic.

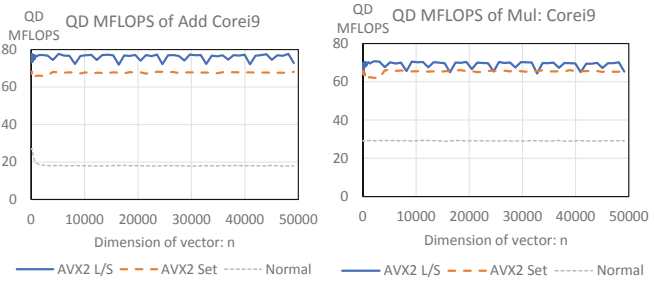

Fig. 4. QD MFLOPS of QD element-wise addition and multiplication of vectors

Figure 4 illustrates QD MFLOPS on the Corei9 environment. Similar to TD arithmetic, cache memory is never effective for QD arithmetic, and AVX2 can provide maximum performance for both QD addition and multiplication. AVX2QDadd is four times faster than normal QDadd, whereas AVX2QDmul is two times faster than normal QDmul. We can expect better performance for longer multi-component MPF arithmetic with AVX2.

As demonstrated above, MPF MFLOPS and SIMDized arithmetic with AVX2 load/store instructions can achieve the best performance for almost all cases. These results are similar to those of EPYC. Therefore, we can expect high-performance matrix multiplication with AVX2 for DD, TD, and QD arithmetic.

3 Benchmark Tests of DD, TD, and QD Matrix Multiplication

To evaluate the performance of the multi-component MPF matrix multiplication, we consider a dense real matrix multiplication to obtain $C := AB \in \mathbb{R}^{n\times n}$, where $A = [a_{ij}] \in \mathbb{R}^{n\times n}$, $B = [b_{ij}] \in \mathbb{R}^{n\times n}$ and c_{ij}, and the elements of C are obtained as

$$c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj}, \tag{1}$$

To draw the power of cache memories in the majority of current CPUs, "block" algorithms are adopted in dense matrix multiplications, which normally divide A and B into small block matrices with fewer rows and columns than the block size $n_{\min}$. We fixed $n_{\min} = 32$ for the benchmark tests in this study.

The "Strassen" algorithm is a divide-and-conquer method [5]. For even dimensional matrices A and B, four divided matrices are prepared and $C := AB$ is obtained, as shown in Fig. 5. We have already implemented BLAS level 1 MPF functions such as array substitution, vector addition, and scalar multiplication, as well as BLAS level 2 MPF functions such as matrix-vector multiplication, which are necessary for the Strassen algorithm.

Among all types of matrix multiplication, we adopted the row-major method to access the elements of matrices and parallelized them using OpenMP [7].

We fixed matrices A and B as follows:

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

Because all elements were positive, there was no significant loss of significant digits in matrix multiplication. We determined one or two decimal digits of loss for all types of matrix multiplication.

All the codes for matrix multiplication were written in C or C++ languages, and the major programs adopted for the benchmark, including Rgemm of MPBLAS, were compiled with the following options:

```
g++ -O3 -std=c++11 -mavx2 -mfma -fopenmp
```

MPBLAS 0.8.0, which was downloaded in June 2019 from Github, was installed and used in our C++ benchmark programs.

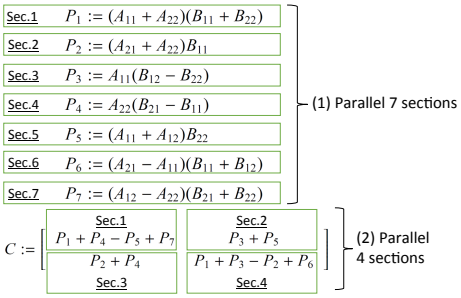

Fig. 5. Strassen matrix multiplication and its parallelization

3.1 Computational Time of Serial Matrix Multiplication and Comparison with Rgemm (MPBLAS)

First, we present the results obtained by serial matrix multiplication on both the Corei9 and EPYC environments in Table 1. In the tables, the "B" fields, "B + A," "S" fields, and "S + A" represent the computational time (s) of block matrix multiplication, block matrix multiplication with AVX2, Strassen matrix multiplication, and Strassen matrix multiplication with AVX2, respectively. The computational time ("M" fields) provided by the Rgemm of MPBLAS are presented in the rightmost columns of the tables. The underlined values in the tables below indicate the minimal computational time for the number of dimensions n in the environment.

We have previously reported that serial Strassen matrix multiplication and parallelized versions for large n are more efficient than the Rgemm of MPBLAS [7]. Our accelerated versions of block and Strassen matrix multiplication with AVX2 can be achieved more than twice as fast as the MPBLAS for any size of matrices. Although TD arithmetic is currently not supported by MPBLAS, we can confirm that the computational time of TD matrix multiplication is located between DD and QD arithmetic for any size and any type of algorithm.

3.2 Speedup Ratio of Parallelization and Computational Time

Next, we present the results of the parallelization using OpenMP. As a typical case of the speedup ratio of parallelization, we presented Fig. 6 on Corei9. As shown in this figure, we can infer that stable speedup has been achieved for block matrix multiplication by approximately 10 threads and that the speedup ratio of the Strassen matrix multiplication was limited by approximately six times using more than eight threads. This limitation occurs owing to Strassen's parallelization method, which produces new threads in recursive calls. We assume that the scheduling of many threads in calls is not executed smoothly.

In contrast to the Corei9 environment, although the EPYC environment always suspends the parallelization performance with over 16 threads for DD, TD, and QD matrix multiplications (the left figure of Fig. 7), AVX2 can eliminate the limitation of parallelization, as demonstrated on the right side of Fig. 7. The DD and QD matrix multiplication on the EPYC can be described as TD.

The limitation of the speedup ratio of the parallelized Strassen matrix multiplication is demonstrated to be similar to that in the Corei9 environment, and AVX2 cannot eliminate the failure of parallelization. We infer that to improve the performance of parallelized block matrix multiplication for over eight threads on CPUs with many cores, it is necessary to completely rewrite the Strassen routine.

Table 1. Serial computational time of matrix multiplication (unit: s): Corei9 (left) and EPYC (right)

DD : Corei9						DD : EPYC					
n	B	B+A	S	S+A	M	n	B	B+A	S	S+A	M
1023	7.84	2.46	4.35	1.57	5.98	1023	9.27	2.37	5.00	1.43	7.55
1024	7.86	2.46	4.34	1.55	6.00	1024	9.31	2.37	4.98	1.41	7.54
1025	8.61	2.68	4.40	1.59	6.01	1025	10.16	2.57	5.03	1.44	7.59
4095	507.8	162.7	212.3	74.47	390.7	4095	595.3	163.6	243.6	68.04	482.8
4096	509.1	161.8	212.5	74.07	390.7	4096	609.1	163.6	243.6	67.75	484.0
4097	518.0	161.7	213.3	74.94	391.0	4097	611.4	152.6	244.7	71.03	483.8
TD : Corei9						TD : EPYC					
1023	50.06	20.54	26.75	12.64	N/A	1023	61.12	21.19	32.18	12.97	N/A
1024	50.10	20.54	26.96	12.48	N/A	1024	61.15	21.16	31.98	12.78	N/A
1025	54.28	22.56	26.95	12.61	N/A	1025	66.46	23.17	32.20	12.89	N/A
4095	3202	1316	1276	619.5	N/A	4095	3918	1379	1555	632.3	N/A
4096	3205	1317	1276	618.6	N/A	4096	3934	1378	1554	631.5	N/A
4097	3272	1345	1287	620.5	N/A	4097	4002	1387	1561	634.2	N/A
QD : Corei9						QD : EPYC					
1023	102.8	31.41	54.90	19.55	73.76	1023	127.8	42.82	71.23	25.17	83.95
1024	102.9	31.40	54.71	19.41	76.04	1024	127.8	42.78	71.10	25.07	84.18
1025	111.8	34.41	55.23	19.63	74.12	1025	139.6	46.88	71.55	25.32	84.47
4095	6491	2013	2720	970.5	4729	4095	8177	2753	3440	1245	5384
4096	6493	2013	2721	968.6	4730	4096	8188	2754	3438	1243	5390
4097	6624	2059	2725	972.6	4727	4097	8371	2808	3449	1247	5394

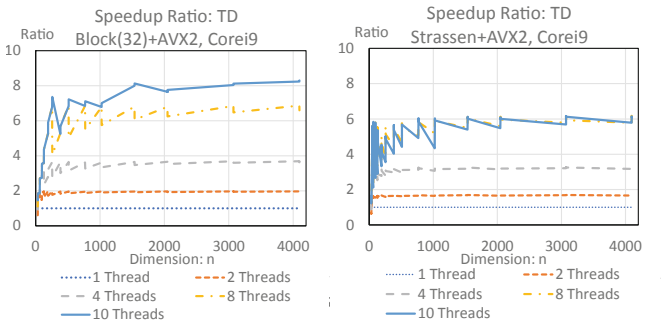

Fig. 6. Speedup ratio of parallelized TD block matrix multiplication on Corei9

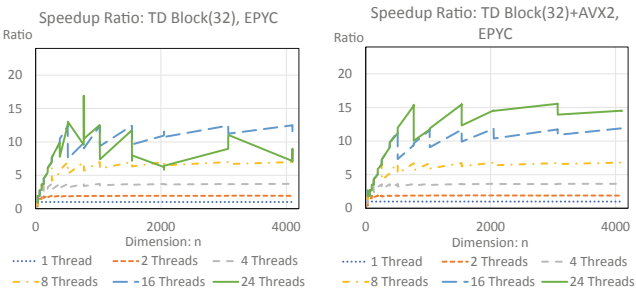

Fig. 7. Speedup ratio of parallelized TD block matrix multiplication on EPYC

Table 2. Seconds of parallelized matrix multiplication: 10 threads on Corei9 (left) and 24 threads on EPYC (right)

DD : Corei9 10 Threads					DD : EPYC 24 Threads				
n	B	B+A	S	S+A	n	B	B+A	S	S+A
1023	1.15	0.35	0.86	0.39	1023	0.78	0.20	0.80	0.59
1024	1.15	0.35	0.69	0.33	1024	0.73	0.20	0.77	0.54
1025	1.23	0.37	0.79	0.38	1025	0.79	0.22	0.84	0.57
4095	61.51	19.68	33.28	16.63	4095	160.68	10.74	35.67	25.43
4096	61.97	19.67	32.60	16.19	4096	191.24	10.72	35.24	24.90
4097	62.65	19.42	34.20	16.44	4097	169.09	10.28	36.26	25.23
TD : Corei9 10 Threads					TD : EPYC 24 Threads				
1023	7.65	3.03	4.57	2.91	1023	4.88	1.81	5.16	2.80
1024	7.67	3.02	4.11	2.07	1024	6.18	1.81	4.68	2.30
1025	8.12	3.22	4.30	2.13	1025	9.01	1.95	4.85	2.36
4095	409.94	159.68	202.34	107.08	4095	549.21	95.02	229.31	117.44
4096	410.48	159.66	196.19	100.55	4096	437.79	94.76	223.25	109.44
4097	416.29	162.04	198.99	101.92	4097	570.95	95.98	225.54	110.36
QD : Corei9 10 Threads					QD : EPYC 24 Threads				
1023	15.23	4.65	9.12	3.43	1023	26.95	3.35	8.91	4.07
1024	15.22	4.64	8.73	3.13	1024	28.98	3.37	8.40	3.50
1025	16.18	4.95	8.88	3.20	1025	27.35	3.60	8.53	3.56
4095	824.23	247.43	437.68	158.57	4095	863.46	171.27	415.35	179.00
4096	822.68	247.48	431.15	153.63	4096	865.12	171.13	407.80	170.27
4097	834.92	251.47	434.29	155.50	4097	882.51	172.95	410.26	171.72

Table 2 presents all computational time (s) with the same number of threads similar to those of the cores on Corei9 (10 cores) and EPYC (24 cores).

As shown in Table 2, block matrix multiplication tends to be increasingly faster than the Strassen on a CPU with many cores, owing to the limitations posed by parallelization of Strassen matrix multiplication. In fact, Strassen matrix multiplication is the fastest on Corei9, and more cases of shorter block matrix multiplication are observed on EPYC. We expect parallelized block matrix multiplication with DD, TD, and QD precision to be the fastest on a CPU with over 32 cores.

4 Conclusion and Future Work

Based on the accelerated DD, TD, and QD MPF matrix multiplication implementations in AVX2, we can confirm that block matrix and Strassen matrix multiplication show much higher speedup than the previous implementations. Furthermore, we can confirm that the speedup ratio is not limited by paralleling it with OpenMP, and it is stabilized in the EPYC environment.

Although the current implementation of parallelized Strassen matrix multiplication has exhibited a limitation of acceleration in CPUs with more than eight cores, BLAS level 1 and 2 MPF functions have been implemented owing to their required implementations in Strassen matrix multiplication. We will apply and develop linear and nonlinear algorithms with our accelerated MPF linear computation library in our future work.

References

1. Bailey, D.: QD. https://www.davidhbailey.com/dhbsoftware/
2. Intel Corp.: The intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/
3. Dekker, T.J.: A floating-point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (1971) https://doi.org/10.1007/BF01397083
4. Fabiano, N., Muller, J.M., Picot, J.: Algorithms for triple words arithmetic. IEEE Trans. Comput. **68**, 1573–1583 (2019)
5. Golub, G.H., Loan, C.: Matrix Computations (4th ed.). Johns Hopkins University Press (2013)
6. Hishinuma, T., Fujii, A., Tanaka, T., Hasegawa, H.: AVX acceleration of DD. Arithmetic between a sparse matrix and vector. Parallel Process. Appl. Math., 622–631 (2014)
7. Kouya, T.: Performance evaluation of multiple precision matrix multiplications using parallelized Strassen and Winograd algorithms. JSIAM Lett. **8**, 21–24 (2015). https://doi.org/10.14495/jsiaml.8.21
8. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2), 13 (2007). http://doi.acm.org/10.1145/1236463.1236468
9. MPLAPACK/MPBLAS: Multiple precision arithmetic LAPACK and BLAS. http://mplapack.sourceforge.net/
10. OpenBLAS. http://www.openblas.net/

11. ATLAS. http://math-atlas.sourceforge.net/
12. Granlaud, T., GMP development team: the GNU multiple precision arithmetic library. https://gmplib.org/
13. Kotakemori, T., Fujii, S., Hasegawa, H., Nishida, A.: Lis: Library of iterative solvers for linear systems. https://www.ssisc.org/lis/
14. Yagi, H., Ishiwata, E., Hasegawa, H.: Acceleration of interactive multiple precision arithmetic toolbox MuPAT using FMA, SIMD, and OpenMP. Adv. Parallel Comput. **36**, 431–440 (2020)