Optimization of Multiple-Precision LU Decomposition Using Ozaki Scheme

Tomonori Kouya(✉) and Taiga Utsugiri

Shizuoka Institute of Science and Technology, 2022-2 Toyosawa, Fukuroi, Shizuoka 437-8555, Japan
{kouya.tomonori,2121002.ut}@sist.ac.jp
https://www.sist.ac.jp/

Abstract. LU decomposition, which is typically implemented with matrix multiplication, requires more computational time than any other component of the direct method. To optimize this process, we accelerated the multiple-precision matrix multiplication stage and applied it to solve the ill-conditioned linear system. Although the SIMDized Strassen algorithm yields the best performance among of all existing algorithms, it fails to sufficiently accelerate LU decomposition. Instead, we found that within a limited precision range, the Ozaki scheme is more efficient for accelerating LU decomposition. In this study, we present the results of benchmark tests conducted to compare the Strassen algorithm with the Ozaki scheme in terms of computational time, demonstrating that the latter exhibits superior performance in LU decomposition when solving ill-conditioned linear systems of equations.

Keywords: multiple-precision floating-point arithmetic · Ozaki scheme · matrix multiplication · LU decomposition

1 Introduction

Applications requiring significant amount of floating-point computations, such as AI and scientific simulations, are ubiquitous in numerous areas of modern society. Consequently, multicore CPUs and GPUs are readily available at the consumer level, guaranteeing sufficient computation processing and results for various applications. Although IEEE754-1985 binary64 (53-bit length of mantissa) is a standard that characterizes CPU environments, it is associated with suboptimal conditions that do not satisfy the accuracy required by users, necessitating multiple-precision floating-point operations with longer mantissa lengths. Consequently, a multi-component multiple-precision arithmetic library has been developed to use multiple existing IEEE754-1985 binary32 (24 bit) and binary64 floating-point numbers, extending arithmetic precision using error-free transformation techniques. Furthermore, a fast multi-digit arbitrary precision floating-point arithmetic library has been developed based on GMP [3]'s fast multiple-precision natural number (MPN) kernel. The former is represented by the QD

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023 Workshops, LNCS 14105, pp. 529–545, 2023.
https://doi.org/10.1007/978-3-031-37108-0_34

library [1] created by Bailey et al., whereas the latter is represented by MPFR [13]. Both libraries have been in practical use for more than 20 years since their initial development, becoming the de facto standard multiple-long-precision floating-point libraries. Accordingly, MPLAPACK/MPBLAS [11] proposed by Nakata provides the basis for a standard multiple-precision numerical linear computation environment. As of March 2023, its version 2.0.1 stably provides the same functionality as binary32 and binary64 LAPACK.

However, there are currently no optimized basic linear computation libraries with multiple-precision arithmetic such as ATLAS, OpenBLAS, and Intel Math Kernel with binary32 and binary64 arithmetic. Standard optimization methods include using SIMD instructions as in the cases of AVX2, AVX-512, and parallelization with OpenMP, whereas optimization with algorithms such as divide-and-conquer is possible for multiple-precision matrix multiplication. We have constructed a fast multiple-precision basic linear algebra subprogram (BLAS) that incorporates these methods. Consequently, we successfully accelerated the matrix multiplication process using the Strassen algorithm.

Recently, progress has been made on accelerating linear computation using multiple-precision computation; namely, the Ozaki scheme was developed for conducting high-precision operations using fast binary32 and binary64 matrix multiplication (SGEMM and DGEMM). It is necessary to verify whether further algorithmic acceleration incorporating these methods is possible by conducting comparative experiments with the current optimization libraries.

A list of optimization library studies for multiple-precision matrix multiplication is presented in Table 1. A question mark indicates that there are no published studies, and the gray cells indicate that the corresponding precision calculation is not expected to be efficient. The blue "BNCmatmul" denotes our previous work [6,7,9]. The area circled in red denotes our latest work ("Utsugiri"), which is the optimization of DD (Double-Double, 106-bit length mantissa), TD (Triple-Double, 159-bit), QD (Quadruple-Double, 212-bit), and MPFR (arbitrary precision) matrix multiplication using the Ozaki scheme on CPUs, and the implementation of TS (Triple-Single, 72-bit) matrix multiplication using the Ozaki scheme on GPUs [16]. Although we could not confirm the Ozaki scheme's effectiveness on consumer GPUs, its effectiveness on CPUs was confirmed for fixed-precision matrix multiplication with a multi-component scheme, as well as for arbitrary-precision matrix multiplication of up to 768 bits as described later.

The LINPACK test, a benchmark test to determine the 500 fastest supercomputers in the world, involves solving a large set of simultaneous linear equations using a direct method, with LU decomposition being the most computationally time-consuming algorithm in the test. Currently, LU decomposition is usually implemented using fast matrix multiplication (xGEMM). If an optimized multiple-precision BLAS can accelerate this process, it can lead to developing novel fast solvers for more complex adverse nonlinear problems.

Based on the results reported in Table 1, including Lis [4], CUMP [2], and MuPAT [17], we have demonstrated that fast multiple-precision LU decomposition is possible on CPUs within a certain range of precision using fast multiple-

Table 1. List of Optimization Methods and Research for Multiple-Precision Basic Linear Computation

xGEMM	CPU				GPU	
Opt.Method	**None**	**AVX2**	**OpenMP**	**Ozaki Scheme**	**CUDA**	**Ozaki Scheme**
DS	?	?	?	?	Mukunoki	Mukunoki
TS	?	?	?	Utsugiri	Utsugiri	Utsugiri
QS	?	?	?	?	?	?
IEEE754 Binary128	MPBLAS	?	MPBLAS	Mukunoki	Mukunoki	Mukunoki
DD	MPBLAS, BNCmatmul	Lis, MuPAT BNCmatmul	Lis,MuPAT, MPBLAS, BNCmatmul	Utsugiri	Mukunoki	Mukunoki
TD	BNCmatmul	BNCmatmul	BNCmatmul	Utsugiri	Utsugiri	Utsugiri
QD	MuPAT, MPBLAS, BNCmatmul	BNCmatmul	MuPAT, MPBLAS, BNCmatmul	Utsugiri	?	?
MPFR	MPBLAS, BNCmatmul	?	MPBLAS, BNCmatmul	Utsugiri	CUMP	?

MPLAPACK/MPBLAS [11] https://github.com/nakata/mplapack
MuPAT [17] Yagi, H et.al(2020)
Lis [4] https://www.ssisc.org/lis/
CUMP [2] https://github.com/skystar0227/CUMP
BNCmatmul [5] https://na-inet.jp/na/bnc/
Mukunoki [12] Mukunoki, D et.al.(2021)
Utsugiri [16] Utsugiri(2021-2023)

precision matrix multiplication implementing the Ozaki scheme. Although we have already reported benchmark results obtained by DD and TD LU decomposition based on the C++ class library [16], we have not obtained benchmark results for QD and MPFR LU decomposition. In this study, we present all the results obtained based on the C-based implementation of matrix multiplication and LU decomposition using DD, TD, QD, and MPFR floating-point arithmetic to avoid the overhead of C++ class libraries.

All benchmark tests conducted on the following Xeon environment show that the Ozaki scheme can also be used to accelerate multiple-precision LU decomposition on CPUs within a certain precision range.

CPU: Intel Xeon W-2295 3.0 GHz 18 cores
OS: Ubuntu 20.04.3 LTS
C/C++ compiler: Intel Compiler version 2021.5.0
MP libraries: MPLAPACK 2.0.1, MPFR 4.2.0

2 Mathematical Notation

Here, we define $\mathbb{F}_{bS}$ and $\mathbb{F}_{bL}$ as sets of the S- and L-bit mantissas of floating-point numbers, respectively. For instance, $\mathbb{F}_{b24}$ and $\mathbb{F}_{b53}$ refer to sets of IEEE754-1985 binary32 and binary64 floating-point numbers, whereas $\mathbb{F}_{b106}$, $\mathbb{F}_{b159}$, and $\mathbb{F}_{b212}$ represent examples of DD, TD, and QD precision floating-point numbers, respectively. Although any mantissa length can be selected in MPFR arithmetic,

the set of MPFR numbers is expressed as $\mathbb{F}_{bM}$, which is primarily defined as M-bit using the `mpfr_set_default_prec` function.

Moreover, we use $(\mathbf{x})_i (= x_i)$ as the i-th element of the n-dimensional vector $\mathbf{x} = [x_i]_{i=1,2,..,n} \in \mathbb{R}^n$, and $(A)_{ij} (= a_{ij})$ as the (i,j)-th element of $A = [a_{ij}]_{i=1,2,...,m,j=1,2,...,n} \in \mathbb{R}^{m\times n}$.

3 Why Do We Require a Multiple-Precision Floating-Point Arithmetic Library?

The following is a brief explanation regarding the need for multiple-precision calculations with a mantissa exceeding binary64. First, users may seek to use floating-point arithmetic to obtain a value $F(x)$ from an input value x represented by a floating-point number. Ultimately, $F(x)$ must maintain at least U significant digits. Intuitively, in numerical computations, the initial error due to rounding is assumed to be included in the input value. Suppose the number of mantissa digits for the floating-point arithmetic to be $L(> U)$. Then, the number of significant digits of $F(x)$ is U–R, where R denotes the digits lost in the process of computing $F(x)$. Furthermore, it is assumed that the algorithm cannot be modified to ensure accuracy with an L-digit computation.

Unless the calculation algorithm for $F(x)$ is modified, the error propagation minimally changes irrespective of the number of mantissa digits. Therefore, we can increase the number of mantissa digits, slightly exceeding R with some slack of α digits, to $L+R+\alpha$ digits. If the initial error can be significantly minimized, $F(x)$ with U or more significant digits can be subsequently obtained. This process illustrates how multiple-precision calculations are used to ensure accuracy.

In contrast, the multi-fold calculation method proposed by Rump and Ogita [14,15] prevents an increase in the initial error by suppressing the rounding error generated by arithmetic operations to the lower digits using error-free transformation techniques. The computational manner of the error-free transformation process is almost the same as that of multiple-precision calculations. Moreover, in this process, floating-point arithmetic with L or more digits is not used. Accordingly, there is no need to perform renormalization procedures. This offers the advantage of reducing computational complexity compared to that of a multi-component approach that uses error-free transformation techniques.

A conceptual diagram of the two aforementioned approaches is presented in Fig. 1.

Currently, not all algorithms used in numerical computation can be applied using multi-fold arithmetic, especially with nonlinear calculations. Therefore, it is necessary to employ a combination of multi-fold and multiple-precision calculations, where the basic linear calculation part, such as explicit extrapolation process solving initial value problems of ordinary differential equations, uses multi-fold calculations to reduce computational time.

The Ozaki scheme, a matrix multiplication algorithm that we incorporated into our library, is a technique based on the multi-fold calculation approach. By leveraging the speed of existing binary32 and binary64 xGEMM algorithms,

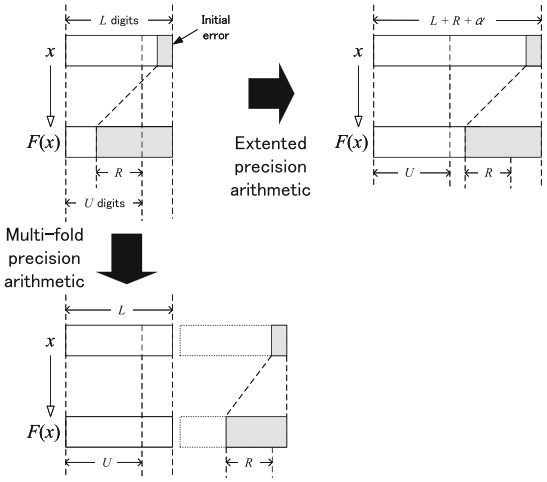

Fig. 1. Extended precision and multi-fold precision arithmetics

it significantly optimizes multiple-precision matrix multiplication under certain conditions, as revealed in our benchmark and previous studies [12,16]. We are confident that the acceleration of multiple-precision linear computation libraries will remain an important theme in guaranteeing the accuracy of broader numerical computation algorithms, following the development trend of multiple-precision numerical algorithms.

4 Algorithms and Performance of Optimized Matrix Multiplication

We focused on the optimization of multiple-precision matrix multiplication, starting with MPI support for arbitrary-precision numerical calculations [5]. Subsequently, owing to current popularity of multi-core CPUs, we accelerated multiple-precision matrix multiplication via parallelization support using OpenMP. As far as multiple-precision calculations are concerned, no other such results have been achieved by incorporating divide-and-conquer methods, such as the Strassen and Winograd algorithms. Although limited to multi-component methods, our proposed method is the only approach that supports AVX2 and has been converted to OpenMP for achieving higher speeds. However, the current code does not improve performance beyond eight threads for OpenMP acceleration, and a drastic reformulation is necessary to further improve parallelization performance. Algorithm 1 is the Strassen matrix multiplication.

The usefulness of the Ozaki scheme is also clear at multiple-precision levels owing to the success of Mukunoki et al. in accelerating the float128 precision matrix multiplication [12]. The float128 arithmetic supported by GCC features TD to QD precision performance for addition and multiplication and is likewise expected to be sufficient for this precision range.

Algorithm 1. Strassen algorithm for matrix multiplication

Input: $A = [A_{ij}]_{i,j=1,2} \in \mathbb{R}^{m\times l}$, $A_{ij} \in \mathbb{R}^{m/2\times l/2}$, $B = [A_{ij}]_{i,j=1,2} \in \mathbb{R}^{l\times n}$, $B_{ij} \in \mathbb{R}^{l/2\times n/2}$

Output: $C := \mathrm{Strassen}(A, B) = AB \in \mathbb{R}^{m\times n}$

if $m < m_0$ && $n < n_0$ **then**
  $C := AB$
end if
$P_1 := \mathrm{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$
$P_2 := \mathrm{Strassen}(A_{21} + A_{22}, B_{11})$
$P_3 := \mathrm{Strassen}(A_{11}, B_{12} - B_{22})$
$P_4 := \mathrm{Strassen}(A_{22}, B_{21} - B_{11})$
$P_5 := \mathrm{Strassen}(A_{11} + A_{12}, B_{22})$
$P_6 := \mathrm{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$
$P_7 := \mathrm{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$
$C_{11} := P_1 + P_4 - P_5 + P_7$; $C_{12} := P_3 + P_5$
$C_{21} := P_2 + P_4$; $C_{22} := P_1 + P_3 - P_2 + P_6$
$C := [C_{ij}]_{i,j=1,2}$

The Ozaki scheme is an algorithm that aims to simultaneously accelerate performance and improve accuracy by dividing matrices into more matrices with elements represented by shorter digits. Similarly to the "Split" method used in the error-free transformation technique, this approach leverages on the speed of optimized short-precision matrix multiplication (xGEMM) functions. For a given matrix $A \in \mathbb{R}^{m\times l}$ and $B \in \mathbb{R}^{l\times n}$, to obtain a matrix product $C := AB \in \mathbb{R}^{m\times n}$ of long L-bit precision, A and B are divided using the Ozaki scheme, where $D \in \mathbb{N}$ is the maximum number of divisions of short S-bit precision matrices ($S << L$), as depicted in Algorithm 2. The S-bit arithmetic is used for calculations where no particular description is given, and the L-bit arithmetic is used only where high-precision operations are required.

Fig. 2 illustrates an example of the Ozaki scheme when A and $B \in \mathbb{R}^{3\times 3}$ are divided into three short-digit matrices. The most important feature of the Ozaki scheme is that the matrices A and B are divided into A_1, A_2, A_3, B_1, B_2, and B_3 to fit within a short precision, thereby avoiding rounding errors in fast, low-precision matrix multiplication. An error-free matrix product $C_{ij} := A_i B_j (i, j = 1, 2, 3)$ can be obtained as a result, and a highly accurate matrix product C can be obtained using a multiple-precision matrix addition operation for $C := \sum_{i,j} C_{ij}$. Although the number of divisions of A and B is finite, it is difficult to determine the minimum number of divisions required to guarantee a certain accuracy threshold. As a result, benchmark tests must be performed to determine if the algorithm can be executed faster than other multiplication algorithms.

The following are C-based multi-component multiple-precision floating-point operations: DD (106 bits), TD (159 bits), QD (212 bits), and MPFR-based multi-digit arbitrary precision operations with 256 bits, 512 bits, and 768 bits. We summarize the results of benchmark tests conducted on Xeon to compare

Algorithm 2. Ozaki scheme for multiple-precision matrix multiplication

Input: $A \in \mathbb{F}_{bL}^{m\times l}, B \in \mathbb{F}_{bL}^{l\times n}$
Output: $C \in \mathbb{F}_{bL}^{m\times n}$
$A^{(S)} := A$, $B^{(S)} := B : A^{(S)} \in \mathbb{F}_{bS}^{m\times l}$, $B^{(S)} \in \mathbb{F}_{bS}^{l\times n}$
$\mathbf{e} := [1\ 1\ ...\ 1]^T \in \mathbb{F}_{bS}^{l}$
$\alpha := 1$
while $\alpha < D$ **do**
  $\boldsymbol{\mu}_A := [\max_{1\le p\le l} |(A^{(S)})_{ip}|]_{i=1,2,...,m} \in \mathbb{F}_{bS}^{m}$
  $\boldsymbol{\mu}_B := [\max_{1\le q\le l} |(B^{(S)})_{qj}|]_{j=1,2,...,n} \in \mathbb{F}_{bS}^{n}$
  $\boldsymbol{\tau}_A := [2^{\lceil \log_2((\boldsymbol{\mu}_A)_i)\rceil + \lceil (S+\log_2(l))/2\rceil}]_{i=1,2,...,m} \in \mathbb{F}_{bS}^{m}$
  $\boldsymbol{\tau}_B := [2^{\lceil \log_2((\boldsymbol{\mu}_B)_j)\rceil + \lceil (S+\log_2(l))/2\rceil}]_{j=1,2,...,n} \in \mathbb{F}_{bS}^{n}$
  $S_A := \boldsymbol{\tau}_A \mathbf{e}^T$
  $S_B := \mathbf{e}\boldsymbol{\tau}_B^T$
  $A_\alpha := (A^{(S)} + S_A) - S_A$: $A_\alpha \in \mathbb{F}_{bS}^{m\times l}$
  $B_\alpha := (B^{(S)} + S_B) - S_B$: $B_\alpha \in \mathbb{F}_{bS}^{l\times n}$
  $A := A - A_\alpha$, $B := B - B_\alpha$: L-bit FP arithmetic
  $A^{(S)} := A$, $B^{(S)} := B$
  $\alpha := \alpha + 1$
end while
$A_D := A^{(S)}$, $B_D := B^{(S)}$
$C := O$
for $\alpha = 1, 2, ..., D$ **do**
  for $\beta = 1, 2, ..., D - \alpha + 1$ **do**
    $C_{\alpha\beta} := A_\alpha B_\beta$
  end for
  $C := C + \sum_{\beta=1}^{D-\alpha+1} C_{\alpha\beta}$: L-bit FP arithmetic
end for

the computational time (seconds), maximum element-wise relative error of C for MPBLAS (Rgemm), and our Strassen matrix multiplication with 256 bits, 512 bits, and 768 bits as precision operations. Although our previous benchmark test conducted on a Core i7 [16] CPU was based on a C++ implementation, the Intel Math Kernel's DGEMM is common with the C-based codes, so there is approximately a 1% difference in computational time.

The square matrices $A, B \in \mathbb{R}^{n\times n}$ used are set as

$$(A)_{ij} := \sqrt{5}(i + j - 1), (B)_{ij} := \sqrt{3}(n - j + 1). \tag{1}$$

Figure 3 shows three results for DD(upper)-, TD(middle)-, and QD(lower)-precision matrix multiplication on Xeon. The left figures show computational time, whereas the right figures display the maximum of element-wise relative errors in computed matrix C. "OZ" denotes results of the Ozaki scheme. For instance, "OZ 5" denotes the Ozaki scheme with a maximum number of matrix partitions set to 5 ($D = 5$). As mentioned earlier, the Ozaki scheme's performance depends on the distribution of the elements in matrices A and B, number of precision digits of the matrix element calculation required by C, and performance of the xGEMM used. From the righthand side figure in Fig. 3, observe

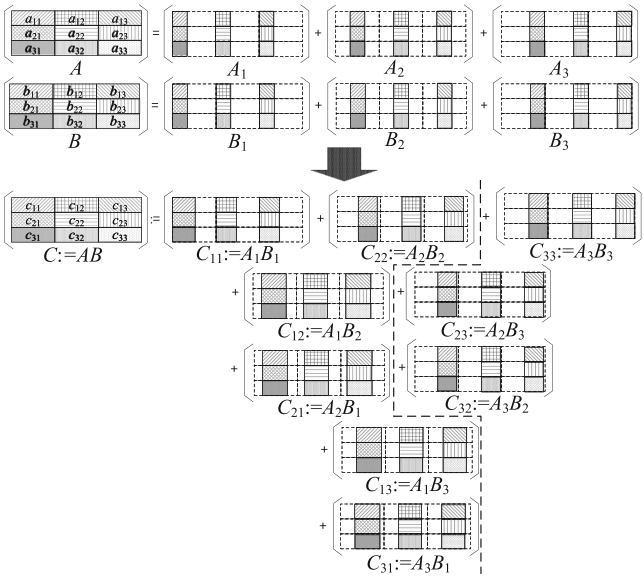

Fig. 2. Matrix multiplication based on Ozaki scheme when the matrices are divided into three components

that the relative error is at least smaller than that of the Strassen matrix multiplication at more than five divisions for DD, more than eight divisions for TD, and more than 12 divisions for QD. In the QD case, however, the accuracy drops by 1–2 decimal places. The Ozaki scheme also yielded faster performance for the three precision calculations. The difference is small for DD but large for TD and QD, which is reasonable as the performance of float128 used by Mukunoki et al. is between TD and QD.

We subsequently increased calculation precision further. Calculation results for MPFR 256 bits, 512 bits, and 768 bits are presented in Fig. 4.

We have already demonstrated that Strassen matrix multiplication is the fastest MPFR matrix multiplication algorithm by a considerable margin [6]; the present results reaffirmed that it is faster than MPBLAS. Furthermore, the Ozaki scheme is faster than the MPBLAS scheme by 256 bits, in addition to yielding the same or lower relative error compared to the Strassen matrix multiplication with more than 14 divisions for 256 bits, more than 25 divisions for 512 bits, and more than 38 divisions for 768 bits. Accordingly, the computational time at 768 bits is equal to or greater than that of the Strassen matrix multiplication. For 38 or more divisions, the Strassen matrix multiplication is expected to be faster in many cases.

To summarize these results, a two-dimensional map of valid matrix multiplication algorithms is presented in Fig. 5.

Strassen matrix multiplication is generally efficient when using long precision and large matrices, as it significantly reduces the amount of arithmetic opera-

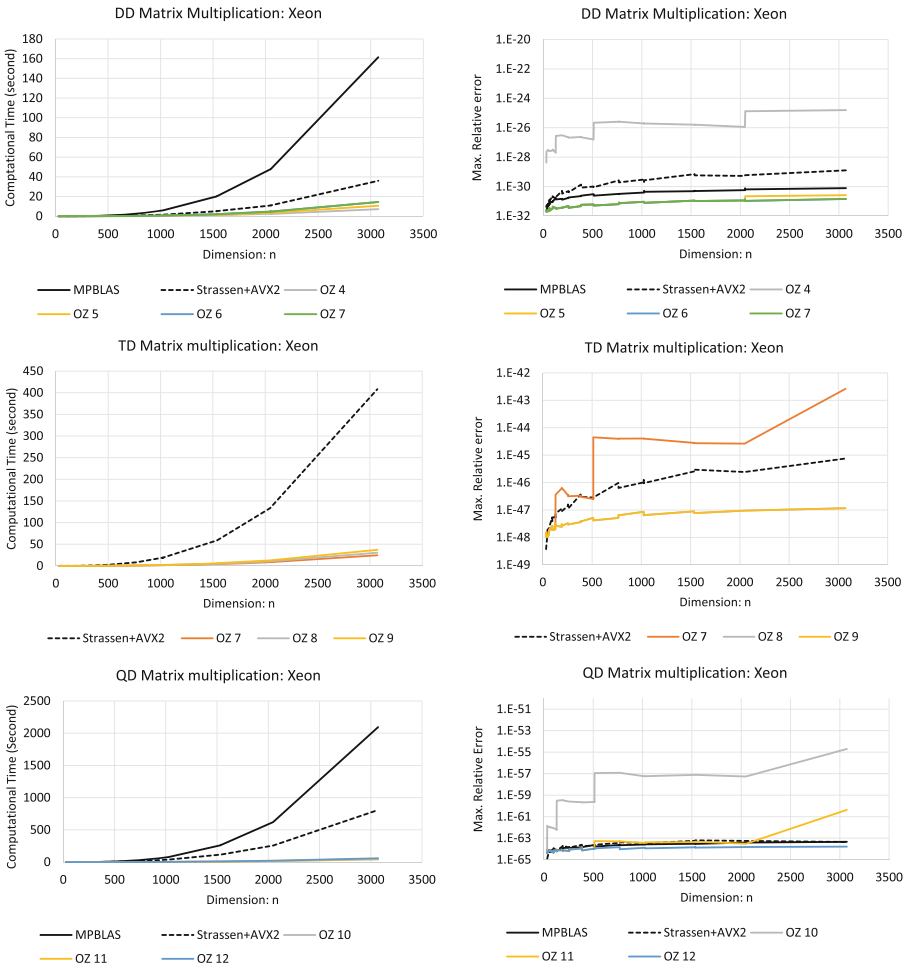

Fig. 3. Computational time (left) and relative error (right) of matrix multiplication using Ozaki scheme: DD prec.(upper), TD prec.(middle), QD prec.(lower)

tions. Blocking is effective when using short precision and mid-sized matrices owing to the presence of cache memory in CPUs. The simple triple-loop method is efficient for small matrices.

The Ozaki scheme is more powerful within a specific range of partitions. When more precision and numerous divisions are necessary to guarantee the output accuracy, the Ozaki scheme tends to be slower than the Strassen algorithm. However, we can confirm that the Ozaki scheme yields high performance within a practical range centered on float128 arithmetic [12].

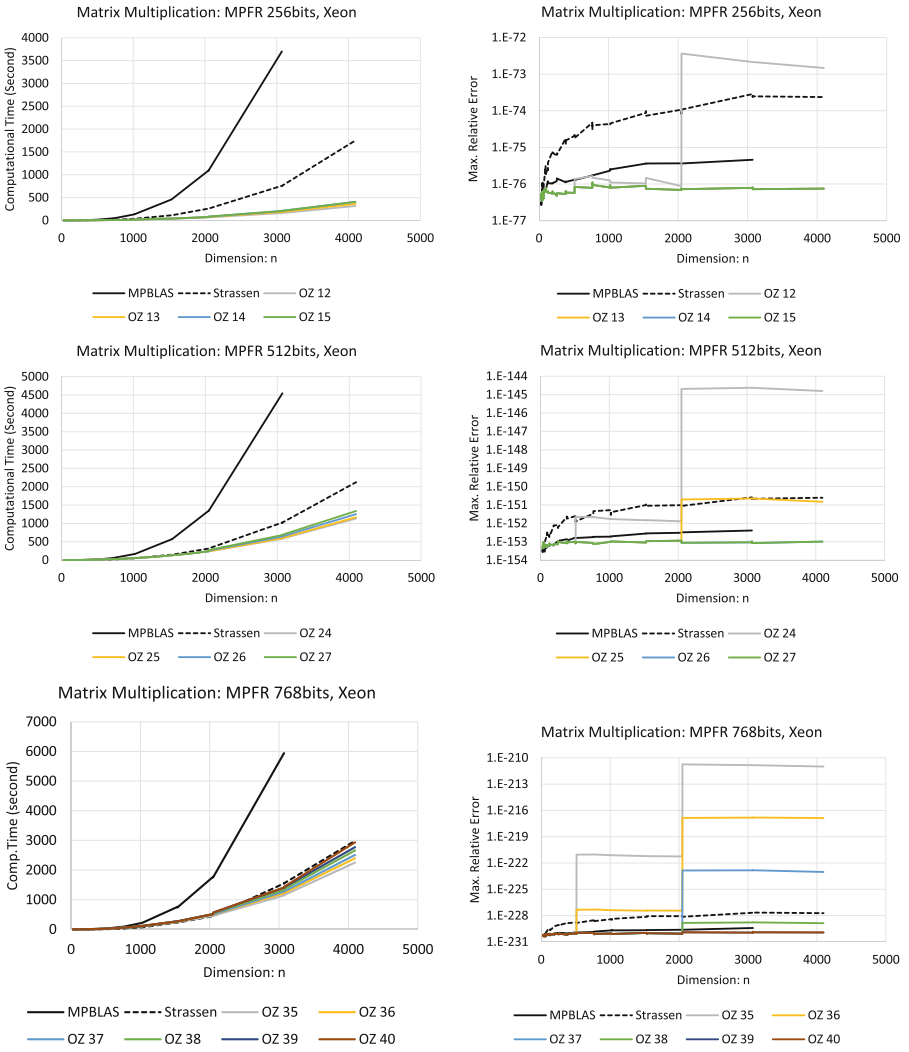

Fig. 4. Computational time (left) and relative error (right) of matrix multiplication using Ozaki scheme: MPFR 256 bits prec.(upper), MPFR 512 bits prec.(middle), and MPFR 768 bits prec.(lower)

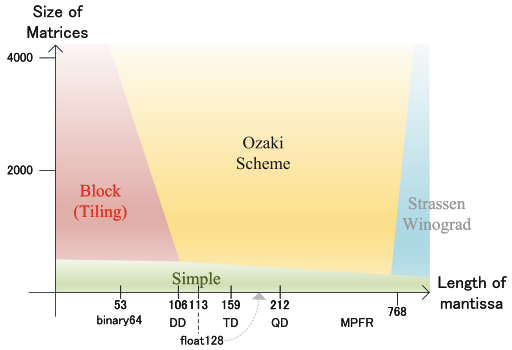

Fig. 5. Map of efficient algorithms for long precision matrix multiplication on Xeon

5 Performance of LU Decomposition Applied with Optimized Matrix Multiplication

As an application of optimized matrix multiplication, we implemented LU decomposition in the direct method for various benchmark tests, including the Top500, and measured its utility on Xeon. The corresponding n-dimensional linear system of equation is:

$$A\mathbf{x} = \mathbf{b}, \tag{2}$$

where $A \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^n$ become ill-conditional as follows:

A: The diagonal matrix is $D = \mathrm{diag}[d_1 \cdots d_n]$, where $d_i := 10^{-26(i-1)/n}$. The random matrix is $R \in \mathbb{R}^{n\times n}$; then, A is calculated as $A := RDR^{-1}$ using the mpmath Python library. Therefore, the condition number of A is $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = 10^{26(n-1)/n}$, which requires super-DD precision arithmetic to gain accuracy.

$\mathbf{x}$, $\mathbf{b}$: After setting $\mathbf{x} = [0\ 1\ \cdots\ n-1]^T$, $\mathbf{b} := A\mathbf{x}$ is calculated using mpmath.

Assuming that LU decomposition in the current LAPACK standard allows using fast matrix multiplications, a constant width $K(=$ MIN_DIM$)$ is predefined as stated in Fig. 6, and the rectangular component $A - L_{12}U_{21}$ is updated for each K column. Therefore, the complexity of matrix multiplication changes with respect to K; MPLAPACK's LU decomposition (Rgetrf) is also implemented using Rgemm.

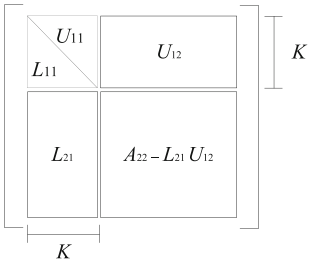

Fig. 6. LU decomposition using matrix multiplication

Various benchmark tests [8,10] have revealed that as long as multiple-precision floating-point operations are used, the typical LU decomposition implementation, wherein calculations are performed column-wise, yields the fastest performance. Furthermore, using the Strassen matrix multiplication does not necessarily improve raw performance. However, we demonstrated that matrix multiplication using the Ozaki scheme was faster than the Strassen matrix multiplication for TD, QD, and MPFR 256 bits, indicating that faster LU decomposition may be achievable.

Results of the multi-component fixed-precision and multi-digit arbitrary-precision calculations are stated below.

5.1 DD, TD, and QD Precision LU Decomposition

LU decompositions were performed for DD, TD, and QD accuracy, with corresponding graphs in Fig. 7 showing the computational time for LU decomposition (left) and maximum element-wise relative error (right) of the numerical solution $\mathbf{x}$ obtained by performing forward and backward substitutions.

The DD-precision problem is a bad condition problem that can only achieve one order of magnitude accuracy at best. This accuracy drops by 2–3 orders of magnitude further when the Strassen algorithm or Ozaki scheme is used. The computational time required to maximize accuracy also exceeds that required for the usual LU decomposition; for TD- and QD-precision calculations, the difference in accuracy is suboptimal. Moreover, the computational time is sufficiently smaller than that required for the usual LU decomposition.

A table of the minimum computational times and maximum relative errors for each precision calculation, including the MPLAPACK(Rgetrf) results, is provided in Table 2.

Table 2. Computational time (seconds) of DD, TD, and QD LU decomposition, with corresponding maximum relative errors of **x**

Prec	Method	K	Second	Rel.Err
DD 106bits	Rgetrf	N/A	15.8	2.4E − 01
	Normal LU	1	5.1	1.9E − 01
	Strassen+AVX2	32	7.4	1.8E + 01
	OZ 6	128	4.5	9.4E + 01
TD 159bits	Normal LU	1	118.6	3.9E − 17
	Strassen+AVX2	32	95.6	5.9E − 17
	OZ 7	96	19.8	1.7E − 17
QD 212bits	Rgetrf	N/A	207.4	3.8E − 34
	Normal LU	1	155.3	1.4E − 33
	Strassen+AVX2	64	180.8	1.5E − 32
	OZ 12	160	83.2	5.8E − 33

Table 3. Minimum computational time (second) of MPFR LU decomposition and maximum relative errors of **x**

Prec	Method	K	Second	Rel.Err
MPFR 256 bits	Rgetrf	N/A	398.6	2.1E − 50
	Normal LU	1	250.0	1.5E − 49
	Strassen	96	273.7	8.3E − 50
	OZ 13	320	167.6	3.2E − 50
MPFR 512 bits	Rgetrf	N/A	492.3	2.5E − 127
	Normal LU	1	341.1	2.6E − 127
	Strassen	32	390.4	3.1E − 126
	OZ 25	576	333.4	4.1E − 126
MPFR 768 bits	Rgetrf	N/A	627.8	5.1E − 205
	Normal LU	1	481.4	1.9E − 203
	Strassen	32	491.1	3.6E − 203
	OZ 38	736	536.6	3.7E − 203

Generally, observe that our LU decomposition implementation is comparable to Rgetrf in terms of speed. Furthermore, compared to the usual LU decomposition, our LU decomposition is 1.1 times faster with the Ozaki scheme ($D = 6$)



Fig. 7. Computational time (left) and relative error (right) of LU decomposition using the Ozaki scheme: DD prec.(upper), TD prec.(middle), QD prec.(lower)

regarding DD accuracy, 6.0 times faster with the Ozaki scheme ($D = 7$) regarding TD accuracy, and 1.9 times faster with the Ozaki scheme ($D = 12$) regarding QD accuracy. The square matrix multiplication scheme shown above is 6.0 times faster than the Ozaki scheme. These improvements in performance are in line with the results of the square matrix multiplication benchmark test presented earlier.

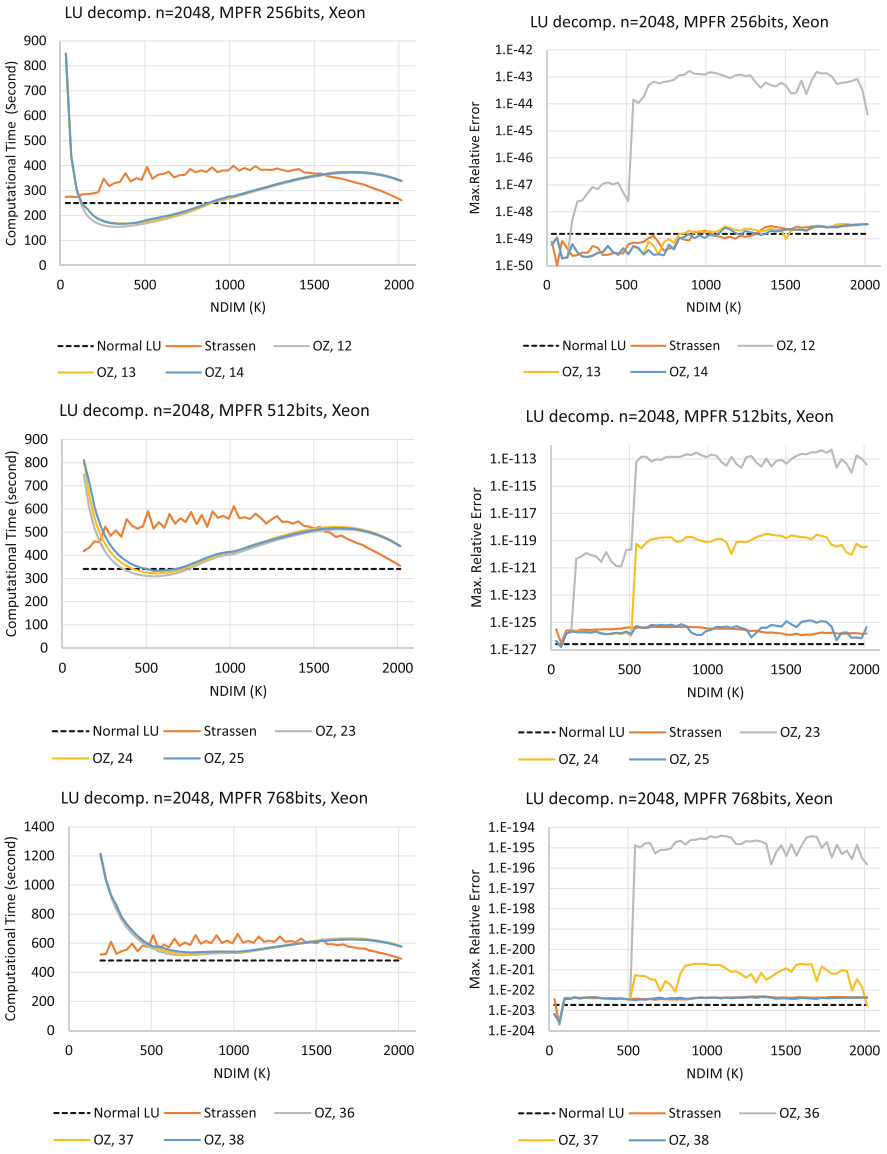

Fig. 8. Computational time (left) and relative error (right) of LU decomposition using Ozaki scheme: MPFR 256 bits(upper), MPFR 512 bits(middle), and MPFR 768 bits(lower)

5.2 MPFR 256-Bit, 512-Bit, and 768-Bit LU Decomposition

We now explain the arbitrary precision calculation in detail. Results for the MPFR 256 bits, 512 bits, and 768 bits calculations are shown in Fig. 8.

Although the Ozaki scheme shows a performance improvement over the Strassen matrix multiplication, there are few areas where the Ozaki scheme can be used effectively beyond 512 bits in the context of this problem.

Table 3 reports the minimal computational times and corresponding maximum relative errors using MPFR calculation.

With fixed precision, observe that our LU decomposition implementation is faster than Rgetrf. Thus, compared to normal LU decomposition, the MPFR 256 bits precision is 1.5 times faster when the Ozaki scheme ($D = 13$) is used, MPFR 512 bits precision is 1.0 times faster when the Ozaki scheme ($D = 25$) is used, and MPFR 768 bits precision is 0.9 times faster when the Ozaki scheme ($D = 38$) is used. Again, observe that the performance is equivalent at 512 bits, and conversely slower at 768 bits.

6 Conclusion and Future Research Directions

As a result of the abovementioned implementation and benchmark tests, we conclude that our implementation of the Ozaki scheme is faster than the Strassen matrix multiplication in the ranges of TD, QD, and MPFR 256 bits and 512 bits accuracy using Intel Math Kernel's DGEMM, specifically for the LU decomposition process.

Future directions of research include the further optimization of our library for multiple-precision linear computation, implementations of the following functions that are not yet supported, and additional performance evaluations via benchmark tests.

1. Optimization of complex linear basic computation and application to the direct method.
2. Implementation of standard-precision basic linear computation using multi-fold precision arithmetic.
3. Development of multiple-precision sparse matrix-vector multiplication (SpMV).

Regarding complex matrix multiplication, it is necessary to evaluate the performance of the Strassen and Ozaki schemes by comparing their accuracies and computational times.

In parallel, it is important to demonstrate the usefulness of the Ozaki scheme for numerical solutions of ordinary and partial differential equations, which are especially important in scientific and engineering simulations, through examples of adverse conditions that require multiple-long-precision linear calculations.

Acknowledgement. This study was financially supported by Kakenhi 20K11843, and also partially supported by the proposed-study fund of Shizuoka Institute of Science and Technology. We appreciate these concerned people and unknown referees who provided helpful comments to improve this paper.

References

1. Bailey, D.: QD. https://www.davidhbailey.com/dhbsoftware/
2. CUMP: Library for arbitrary precision arithmetic on CUDA
3. Granlaud, T., development team, G.: The GNU Multiple Precision arithmetic library. https://gmplib.org/
4. Kotakemori, T., Fujii, S., Hasegawa, H., Nishida, A.: Lis: Library of iterative solvers for linear systems. https://www.ssisc.org/lis/
5. Kouya, T: On BNCpack, multiple-precision numerical computation library with MPFR/GMP. OYO-SURI **21**(3), 197–206 (2011). http://ci.nii.ac.jp/naid/110009426223/
6. Kouya, T.: Accelerated multiple precision matrix multiplication using Strassen's algorithm and Winograd's variant. JSIAM Lett. **6**, 81–84 (2014). https://doi.org/10.14495/jsiaml.6.81
7. Kouya, Tomonori: Performance evaluation of strassen matrix multiplication supporting triple-double precision floating-point arithmetic. In: Gervasi, O. (ed.) ICCSA 2020. LNCS, vol. 12253, pp. 163–176. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58814-4_12
8. Kouya, T.: Acceleration of LU decomposition supporting double-double, triple-double, and quadruple-double precision floating-point arithmetic with avx2. In: 2021 IEEE 28th Symposium on Computer Arithmetic (ARITH), pp. 54–61 (2021). https://doi.org/10.1109/ARITH51176.2021.00021
9. Kouya, T.: Acceleration of multiple precision matrix multiplication based on multi-component floating-point arithmetic using AVX2. In: Gervasi, O., Murgante, B., Misra, S., Garau, C., Blečić, I., Taniar, D., Apduhan, B.O., Rocha, A.M.A.C., Tarantino, E., Torre, C.M. (eds.) ICCSA 2021. LNCS, vol. 12953, pp. 202–217. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86976-2_14
10. Kouya, T.: Optimization of mixed-precision iterative refinement using parallelized direct methods. In: 2022 International Conference on Engineering and Emerging Technologies (ICEET), pp. 1–6 (2022). https://doi.org/10.1109/ICEET56468.2022.10007230
11. MPLAPACK/MPBLAS: Multiple precision arithmetic LAPACK and BLAS. https://github.com/nakatamaho/mplapack
12. Mukunoki, D., Ozaki, K., Ogita, T., Imamura, T.: Accurate matrix multiplication on binary128 format accelerated by ozaki scheme. In: 50th International Conference on Parallel Processing. ICPP 2021, Association for Computing Machinery, New York, NY, USA, pp. 1–11 (2021). https://doi.org/10.1145/3472456.3472493
13. Project, M.: The MPFR library. https://www.mpfr.org/
14. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part I: faithful rounding. SIAM J. Sci. Comput. **31**(1), 189–224 (2008)
15. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part II: sign, k-fold faithful and rounding to nearest. SIAM J. Sci. Comput. **31**(2), 1269–1302 (2008)
16. Utsugiri, T., Kouya, T.: Acceleration of multiple precision matrix multiplication using ozaki scheme (2023). https://doi.org/10.48550/ARXIV.2301.09960, https://arxiv.org/abs/2301.09960
17. 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)