Assessing the Performance of Mixed-Precision ILU(0)-Preconditioned Multiple-Precision Real and Complex Krylov Subspace Methods

Tomonori Kouya(✉)

Otemon Gakuin University, 2-1-15 Nishiai, Ibaraki, Osaka 567-8502, Japan
t-koya@haruka.otemon.ac.jp
https://www.otemon.ac.jp

Abstract. Krylov subspace methods are linear solvers based on matrix-vector multiplications and vector operations. While easily parallelizable, they are sensitive to rounding errors and may experience convergence issues. ILU(0), an incomplete LU factorization with zero fill-in, is a well-known preconditioning technique that enhances convergence for sparse matrices. In this paper, we implement a double-precision and multiple-precision ILU(0) preconditioner, compatible with product-type Krylov subspace methods, and evaluate its performance.

Keywords: preconditioning · Krylov subspace method · multiple-precision floating-point arithmetic

1 Introduction

With the continuous progress of digital transformation (DX), computing infrastructures capable of efficiently processing large-scale data are increasingly essential. The operation and training of AI systems, primarily based on machine learning and deep learning, are expected to drive DX forward. Consequently, faster and more efficient computers are needed, and the demand for computational power continues to grow. However, further increases in CPU and GPU clock speeds are limited, making adopting architectures optimized for parallel processing, such as those supporting single instruction, multiple data (SIMD) instructions and multi-core configurations essential. Software should likewise be optimized to run effectively on these modern architectures.

While AI applications frequently use floating-point formats with short mantissas (e.g., IEEE754 binary32 or binary16), scientific computing requires higher precision formats such as binary64 or above. Hardware advancements are driven by market demand, which often favors the former. For ill-conditioned problems requiring high precision or mixed-precision computing that combines short and long mantissa floating-point formats is a promising approach to achieve both accuracy and performance.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025 Workshops, LNCS 15890, pp. 31–41, 2026.
https://doi.org/10.1007/978-3-031-97606-3_3

To address such ill-conditioned problems, we developed an optimized basic linear algebra library, BNCmatmul, supporting multiple-precision formats beyond binary64. Libraries such as QD [1], MPFR [11], and MPC [3] offer reliable multi-component and arbitrary-precision arithmetic. MPLAPACK/MPBLAS [9] extends LAPACK/BLAS functionality to support these formats. However, current parallel architecture optimizations are limited to what is provided by C++ compilers. Similar to machine-optimized libraries like Intel MKL [8] and OpenBLAS [10], optimized multiple-precision BLAS libraries are required.

Our BNCmatmul library is optimized primarily for x86 environments. As of March 2025, we have completed the implementation of real and complex various multiple-precision floating-point operations, along with random sparse matrix-vector multiplication (SpMV). Details on SpMV are reported in [7], and although further tuning is possible, the current implementation is usable functionally.

This paper reports on applying ILU(0), factorization-free of fill-ins-as a preconditioner to product-type Krylov subspace methods: BiCG, CGS, BiCGSTAB, and GPBiCG. Unlike our AVX2-optimized SpMV, the current ILU(0) implementation is not optimized or parallelized and is, therefore, slow. Nevertheless, the implementation supports mixed-precision ILU(0) using binary64, enabling performance evaluation with test matrices from the SuiteSparse matrix collection [2]. This evaluation offers insight into the utility of the method and provides a foundation for future optimization.

The remainder of this paper is structured as follows. Section 2 describes the features of BNCmatmul. Section 3 explains the ILU(0) algorithm. Section 4 presents benchmark test results on an EPYC machine and discussions. Section 5 concludes with future work.

2 Multiple-Precision Optimized Basic Linear Algebra Library: BNCmatmul

An overview of the features of BNCmatmul [5] is shown in Fig. 1. As of March 2025, Version 0.21 has been released as open-source software, primarily supporting real-valued linear algebra routines and their optimized/parallelized versions. Version 0.22, currently under development, extends functionality to complex linear algebra based on real implementations.

The library supports the following five floating-point types:

Double (D). IEEE754 binary64 format with a 53-bit mantissa (52-bit + 1 hidden bit).

Multi-component-type. Formats based on binary64 that extend mantissa length using multiple components:
Double-double (DD). 106-bit mantissa using two binary64 values.
Triple-double (TD). 159-bit mantissa using three binary64 values.
Quadruple-double (QD). 212-bit mantissa using four binary64 values.

Multi-digit-type. Arbitrary precision types provided by MPFR (real) and MPC (complex), built on GMP's multiple-precision natural number (MPN) kernel.

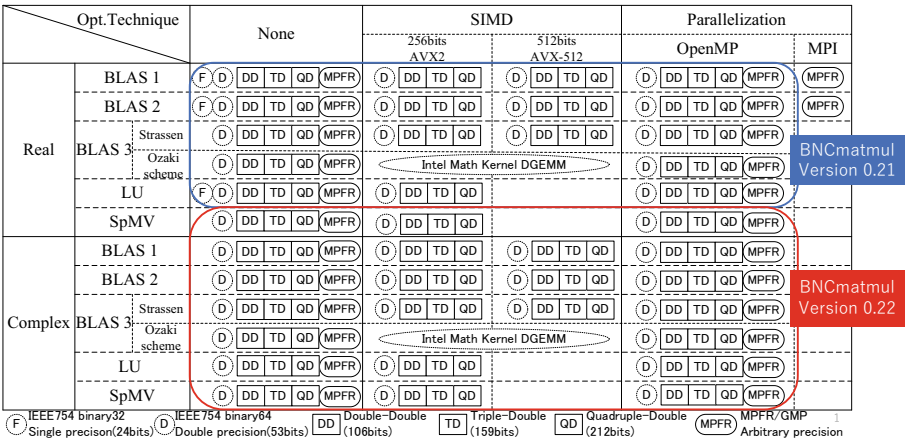

Fig. 1. Current development status of BNCmatmul

The SpMV implemented in this work builds on our prior implementation of MPFR-based arbitrary-precision compressed row storage (CRS) multiplication [6], extended to support all five floating-point types. This feature is included in Version 0.22, and detailed performance evaluation results can be found in [7].

Version 0.22 is scheduled for release soon. Looking ahead, Version 0.23 will expand the capabilities of the library for nonlinear computations based on the current linear algebra core. It will also introduce SIMD-optimized "pair arithmetic" for multi-component types. Additionally, we plan to adapt the library for consumer-grade computing platforms, including environments with 128-bit SIMD support such as Arm NEON, WebAssembly (WASM), and x86 SSE.

3 Product-Type Krylov Subspace Methods and Mixed-Precision ILU(0) Preconditioner

We consider solving the linear system:

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

where $A \in \mathbb{C}^{n\times n}$ is a complex-valued square coefficient matrix, $\mathbf{b} \in \mathbb{C}^n$ is a known n-dimensional vector, and $\mathbf{x} \in \mathbb{C}^n$ is the unknown n-dimensional vector to be solved. In this study, we assume A is non-singular. For comparison purposes, we also consider real-valued cases where $A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$, and $\mathbf{x} \in \mathbb{R}^n$.

When A is sparse, iterative methods such as Krylov subspace methods and GMRES are effective due to their reliance on matrix-vector multiplication, which is easy to parallelize. However, these algorithms are highly sensitive to rounding errors and may fail to converge within the theoretical limit of n iterations. To improve convergence, preconditioning is commonly applied. One such technique is the incomplete LU factorization (ILU) that approximates $A \approx LU$ while

controlling fill-ins. We use ILU(0), which introduces no fill-in, as defined in the Algorithm 1.

Algorithm 1. ILU(0) Factorization

Require: Sparse matrix $A \in \mathbb{C}^{n\times n}$ (non-singular)
Ensure: Matrices $\tilde{L}, \tilde{U} \in \mathbb{C}^{n\times n}$ such that $A \approx \tilde{L}\tilde{U}$
1: Initialize $\tilde{L} := I$, $\tilde{U} := 0$
2: **for** $k = 1$ to n **do**
3: **if** $U_{kk} := A_{kk} \neq 0$ **then**
4: **for** each $i > k$ such that $A_{ik} \neq 0$ **do**
5: $\tilde{L}_{ik} := A_{ik}/\tilde{U}_{kk}$
6: **for** each $j > k$ such that $A_{kj} \neq 0$ **do**
7: **if** $A_{ij} \neq 0$ **then**
8: $A_{ij} := A_{ij} - \tilde{L}_{ik} \cdot \tilde{U}_{kj}$
9: **end if**
10: **end for**
11: **end for**
12: **for** each $j > k$ such that $A_{kj} \neq 0$ **do**
13: $\tilde{U}_{kj} := A_{kj}$
14: **end for**
15: **end if**
16: **end for**

Once the ILU(0) factorization is complete, we apply the preconditioner $M = \tilde{L}\tilde{U}$ before starting the iterations. During each iteration, we solve $M\mathbf{z} = \mathbf{r}$ using forward and backward substitutions, as described in Algorithm 2.

Among the product-type Krylov methods, applying ILU(0) preconditioning to the BiCG method results in Algorithm 3:

In BiCG, both $M\mathbf{z} = \mathbf{r}$ and $M^H\mathbf{z} = \mathbf{r}$ must be solved at each iteration. In contrast, other product-type methods such as CGS, BiCGSTAB, and GPBiCG require only one preconditioning solve per iteration.

Our ILU(0) implementation supports both real and complex CRS sparse matrices and handles Hermitian systems. In line with previous research by Hishinuma et al. [4], the SuiteSparse Matrix Collection [2] provides only binary64 matrices, therefore, we implemented mixed-precision SpMV and forward/backward substitutions with binary64 and higher precisions to assess the impact of mixed-precision ILU(0) on convergence and performance. In the BiCG method described in Algorithm 3, all computations are performed entirely in high-precision arithmetic. Only when binary64 ILU(0) is used, Lines 5, 6, 19, and 20 in Algorithm 3 are executed using binary64 $\tilde{L}_{ij}$ and $\tilde{U}_{ij}$, along with high-precision y_i and z_i in Algorithm 2.

4 Benchmark Test

This section presents and discusses the results of benchmark tests conducted on the following 32-core EPYC computing environment:

Algorithm 2. Preconditioning via ILU(0): Solving $M^{-1}\mathbf{r} = \mathbf{z}$

Require: Residual vector $\mathbf{r} \in \mathbb{C}^n$, ILU(0) factors $\tilde{L}$, $\tilde{U}$
Ensure: Preconditioned vector $\mathbf{z} \in \mathbb{C}^n$
Forward substitution to solve $\tilde{L}\mathbf{y} = \mathbf{r}$
for $i = 1$ to n **do**
  $y_i := r_i$
  for $j = 1$ to $i - 1$ **do**
    if $\tilde{L}_{ij} \neq 0$ **then**
      $y_i := y_i - \tilde{L}_{ij} \cdot y_j$
    end if
  end for
end for
Backward substitution to solve $\tilde{U}\mathbf{z} = \mathbf{y}$
for $i = n$ downto 1 **do**
  $z_i := y_i$
  for $j = i + 1$ to n **do**
    if $\tilde{U}_{ij} \neq 0$ **then**
      $z_i := z_i - \tilde{U}_{ij} \cdot z_j$
    end if
  end for
  $z_i := z_i / \tilde{U}_{ii}$
end for

Hardware: AMD EPYC 9354P, 3.7 GHz, 32 cores, Ubuntu 20.04.6 LTS
Software: Intel Compiler version 2021.10.0, GNU MP 6.2.1, MPFR 4.1.0, MPC 1.2.1

For the linear system defined in Eq. 1, the exact solution vector $\mathbf{x}$ is set as follows:

- For real matrices: $\mathbf{x} := \sqrt{2}\ [1\ 2\ \ldots\ n]^T$
- For complex matrices: $\mathbf{x} := \sqrt{2 + 3\mathrm{i}}\ [1\ 2\ \ldots\ n]^T$

The right-hand side vector is computed as $\mathbf{b} := A\mathbf{x}$.

The performance of sparse matrix preconditioning strongly depends on the structure of the matrix and the target computing environment. In this test, we selected the following two example matrices from the SuiteSparse Matrix Collection, shown in Fig. 2:

- Real sparse matrix: `mcfe` ($n = 765$)
- Complex sparse matrix: `dwg961b` ($n = 961$)

As demonstrated subsequently, neither matrix converges without ILU(0) preconditioning under floating-point precision below 256 bits. Since both are originally stored in binary64 precision, we evaluated the performance using:

- Mixed-precision SpMV implementations using binary64 and DD/TD/QD (BiCG_d, etc.)

Algorithm 3. Preconditioned BiCG Method for Complex Systems

Require: Complex matrix $A \in \mathbb{C}^{n\times n}$, RHS vector $\mathbf{b} \in \mathbb{C}^n$, initial guess $\mathbf{x}_0 := 0$
Require: Preconditioner $M \approx A$, tolerance ϵ
Ensure: Approximate solution $\mathbf{x}$

$\mathbf{x} := \mathbf{x}_0$
$\mathbf{r}_0 := \mathbf{b} - A\mathbf{x}$
$\mathbf{r} := \mathbf{r}_0$
$\tilde{\mathbf{r}} := \mathbf{r}$
Solve $M\mathbf{z} := \mathbf{r}$
Solve $M^H\tilde{\mathbf{z}} := \tilde{\mathbf{r}}$
$\mathbf{p} := \mathbf{z}$, $\tilde{\mathbf{p}} := \tilde{\mathbf{z}}$
$\rho := \langle \tilde{\mathbf{z}}, \mathbf{r}\rangle$
for $k = 1$ to max_iter **do**
  $\mathbf{q} := A\mathbf{p}$
  $\tilde{\mathbf{q}} := A^H\tilde{\mathbf{p}}$
  $\alpha := \rho/\langle \tilde{\mathbf{p}}, \mathbf{q}\rangle$
  $\mathbf{x} := \mathbf{x} + \alpha\mathbf{p}$
  $\mathbf{r} := \mathbf{r} - \alpha\mathbf{q}$
  $\tilde{\mathbf{r}} := \tilde{\mathbf{r}} - \overline{\alpha}\tilde{\mathbf{q}}$
  if $\|\mathbf{r}\| < \varepsilon_r\|\mathbf{r}_0\| + \varepsilon_a$ **then**
    break
  end if
  Solve $M\mathbf{z} := \mathbf{r}$
  Solve $M^H\tilde{\mathbf{z}} := \tilde{\mathbf{r}}$
  $\rho_{\mathrm{new}} := \langle \tilde{\mathbf{z}}, \mathbf{r}\rangle$
  $\beta := \rho_{\mathrm{new}}/\rho$
  $\mathbf{p} := \mathbf{z} + \beta\mathbf{p}$
  $\tilde{\mathbf{p}} := \tilde{\mathbf{z}} + \overline{\beta}\tilde{\mathbf{p}}$
  $\rho := \rho_{\mathrm{new}}$
end for

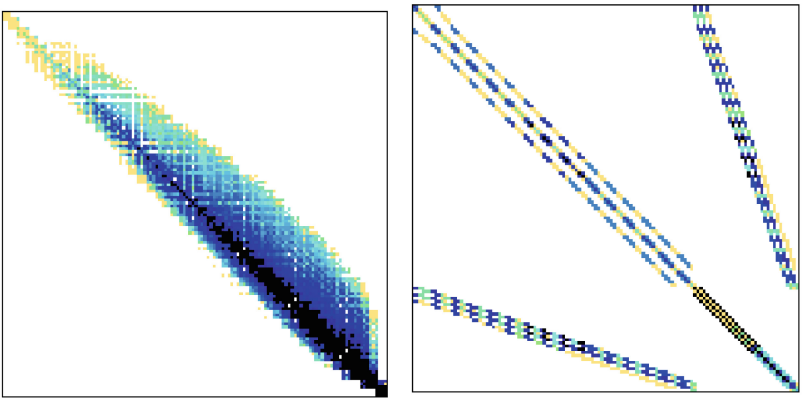

Fig. 2. Structures of non-zero elements in the example matrices: `mcfe` (left) and `dwg961b` (right) adapted from [2].

Table 1. Real linear equation: mcfe

mcfe, $n = 765$	DD			TD			QD		
Algorithm	[1]	[2]	[3]	[1]	[2]	[3]	[1]	[2]	[3]
BiCG	2295	0.26	0.11	2295	3.00	1.31	2295	3.64	1.58
BiCG_p	2295	0.23	0.10	2295	1.14	0.50	2295	1.70	0.74
BiCG_ILU(0)	13	0.67	51.31	13	0.74	56.85	13	0.78	59.77
BiCG_d	2295	0.25	0.11	2295	2.88	1.25	2295	3.47	1.51
BiCG_dp	2295	0.27	0.12	2295	1.13	0.49	2295	1.61	0.70
BiCG_d_ILU(0)	13	0.67	51.31	13	0.69	53.23	13	0.73	56.23
CGS	2295	0.25	0.11	2295	3.10	1.35	2295	4.01	1.75
CGS_p	2295	0.09	0.04	2295	0.70	0.30	2295	0.93	0.41
CGS_ILU(0)	9	0.52	57.67	9	0.55	60.67	9	0.59	66.00
CGS_d	2295	0.23	0.10	2295	2.98	1.30	2295	3.84	1.67
CGS_dp	2295	0.22	0.10	2295	0.69	0.30	2295	0.91	0.40
CGS_d_ILU(0)	9	0.47	52.56	9	0.51	56.56	9	0.54	59.78
BiCGSTAB	2295	0.25	0.11	2295	3.19	1.39	2295	4.23	1.84
BiCGSTAB_p	2295	0.08	0.04	2295	0.81	0.35	2295	1.12	0.49
BiCGSTAB_ILU(0)	9	0.51	56.89	9	0.52	58.11	9	0.56	62.56
BiCGSTAB_d	2295	0.24	0.10	2295	3.08	1.34	2295	4.06	1.77
BiCGSTAB_dp	2295	0.24	0.10	2295	0.83	0.36	2295	1.11	0.48
BiCGSTAB_d_ILU(0)	9	0.47	52.44	9	0.49	54.11	9	0.51	57.00
GPBiCG	2295	0.26	0.12	2295	3.53	1.54	2295	4.57	1.99
GPBiCG_p	2295	0.12	0.05	2295	1.20	0.52	2295	1.49	0.65
GPBiCG_ILU(0)	8	0.46	57.63	8	0.50	62.38	8	0.54	67.38
GPBiCG_d	2295	0.26	0.11	2295	3.42	1.49	2295	4.40	1.92
GPBiCG_dp	2295	0.23	0.10	2295	1.13	0.49	2295	1.47	0.64
GPBiCG_d_ILU(0)	8	0.43	53.50	9	0.51	56.56	8	0.49	61.13

- Mixed-precision ILU(0) forward/backward substitution using binary64 (BiCG_d_ILU(0), etc.)
- Parallel implementations using 32-thread OpenMP without preconditioning (BiCG_p, etc.)

Although our 32-threaded parallelized SpMVs are not optimized for any particular type of sparse matrix, they were used for this benchmarking.

We first present results using DD, TD, and QD multi-component precision formats. Table 1 lists the results for the real matrix `mcfe` ($n = 765$) under a maximum iteration count of $3n = 2295$, with convergence thresholds $\varepsilon_r = 10^{-13}$ and $\varepsilon_a = 10^{-100}$. The metrics in each column are as follows:

[1] Number of iterations
[2] Total computation time (in seconds)
[3] Average time per iteration (in milliseconds), calculated as [2] divided by [1]

As shown, without ILU(0) preconditioning, all methods hit the iteration limit without converging, regardless of the precision format. Mixed-precision SpMV does not negatively impact convergence, nor does it reduce the effectiveness of ILU(0). Mixed-precision forward/backward substitution maintains the same number of iterations.

Next, Table 2 lists the results for the complex sparse matrix `dwg961b` ($n = 961$), again with DD, TD, and QD precision formats. As in the previous case, none of the Krylov subspace methods converge without ILU(0) preconditioning.

Table 2. Complex linear equation: dwg961b

dwg961b, $n = 961$	DD			TD			QD		
Algorithm	[1]	[2]	[3]	[1]	[2]	[3]	[1]	[2]	[3]
BiCG	2883	0.79	0.27	2883	8.96	3.11	2883	13.10	4.55
BiCG_p	2883	1.61	0.56	2883	6.73	2.33	2883	10.77	3.73
BiCG_ILU(0)	758	199.67	263.41	516	170.33	330.09	442	206.52	467.24
BiCG_d	2883	0.76	0.26	2883	8.64	3.00	2883	12.66	4.39
BiCG_dp	2883	1.83	0.63	2883	6.71	2.33	2883	10.36	3.59
BiCG_d_ILU(0)	758	62.74	82.77	512	45.34	88.55	438	82.33	187.96
CGS	2883	0.85	0.29	2883	9.89	3.43	2883	15.49	5.37
CGS_p	2883	0.70	0.24	2883	3.67	1.27	2883	6.04	2.10
CGS_ILU(0)	1057	268.15	253.69	666	215.51	323.59	529	244.34	461.88
CGS_d	2883	0.81	0.28	2883	9.60	3.33	2883	14.99	5.20
CGS_dp	2883	1.67	0.58	2883	3.72	1.29	2883	6.07	2.10
CGS_d_ILU(0)	1064	84.79	79.69	664	57.05	85.92	526	48.36	91.93
BiCGSTAB	2883	0.93	0.32	2883	10.51	3.65	2883	17.56	6.09
BiCGSTAB_p	2883	0.69	0.24	2883	4.27	1.48	2883	8.19	2.84
BiCGSTAB_ILU(0)	2209	551.94	249.86	1242	401.85	323.55	778	354.55	455.72
BiCGSTAB_d	2883	0.89	0.31	2883	10.21	3.54	2883	17.10	5.93
BiCGSTAB_dp	2883	1.50	0.52	2883	4.23	1.47	2883	8.17	2.83
BiCGSTAB_d_ILU(0)	1998	159.13	79.65	1117	96.01	85.96	831	76.92	92.56
GPBiCG	2883	1.12	0.39	2883	12.53	4.35	2883	21.92	7.60
GPBiCG_p	2883	0.92	0.32	2883	6.30	2.18	2883	12.64	4.38
GPBiCG_ILU(0)	1954	489.50	250.51	1143	370.68	324.31	869	396.26	456.00
GPBiCG_d	2883	1.08	0.37	2883	12.23	4.24	2883	21.42	7.43
GPBiCG_dp	2883	2.09	0.72	2883	6.33	2.20	2883	12.67	4.39
GPBiCG_d_ILU(0)	1871	149.19	79.74	1162	100.73	86.68	896	84.30	94.09

Compared to the real-valued case, solving complex linear systems takes more time, consistent with our previous benchmark study [7]. Furthermore, the cost per iteration when using ILU(0) preconditioning is significantly higher.

We also benchmarked arbitrary-precision arithmetic using MPFR and MPC with a 256-bit mantissa to observe the impact on convergence. Results indicate that like the multi-component case, no convergence is achieved without ILU(0). The effect of mixed-precision in these settings is summarized in Table 3.

Table 3. MPFR and MPC 256-bit benchmark results

MPFR and MPC 256bits	mcfe			dwg961b		
Algorithm	[1]	[2]	[3]	[1]	[2]	[3]
BiCG	2295	11.93	5.20	2883	7.60	2.64
BiCG_p	2295	7.31	3.19	2883	1.43	0.50
BiCG_ILU(0)	13	1.10	84.46	401	0.90	2.25
BiCG_d	2295	17.70	7.71	2883	13.37	4.64
BiCG_dp	2295	7.33	3.20	2883	1.68	0.58
BiCG_d_ILU(0)	13	0.86	66.31	2883	72.52	25.15
CGS	2295	11.84	5.16	2883	9.78	3.39
CGS_p	2295	5.47	2.38	2883	1.34	0.46
CGS_ILU(0)	9	0.82	91.44	468	0.59	1.26
CGS_d	2295	17.60	7.67	2883	17.80	6.17
CGS_dp	2295	5.33	2.32	2883	1.68	0.58
CGS_d_ILU(0)	9	0.63	70.00	2087	0.47	0.22
BiCGSTAB	2295	12.12	5.28	2883	13.40	4.65
BiCGSTAB_p	2295	5.74	2.50	2883	2.00	0.69
BiCGSTAB_ILU(0)	9	0.79	87.22	716	0.59	0.83
BiCGSTAB_d	2295	17.88	7.79	2883	24.65	8.55
BiCGSTAB_dp	2295	5.53	2.41	2883	2.49	0.86
BiCGSTAB_d_ILU(0)	9	0.60	66.89	2883	0.47	0.16
GPBiCG	2295	13.82	6.02	2883	17.60	6.10
GPBiCG_p	2295	7.30	3.18	2883	3.40	1.18
GPBiCG_ILU(0)	8	0.76	95.00	674	0.76	1.13
GPBiCG_d	2295	19.97	8.70	2883	31.06	10.77
GPBiCG_dp	2295	7.50	3.27	2883	3.96	1.37
GPBiCG_d_ILU(0)	8	0.58	72.50	2883	0.49	0.17

Finally, Table 4 summarizes the reduction ratios in computation time when using ILU(0) and mixed-precision ILU(0) substitutions.

The results indicate the following for real-valued systems:

Table 4. Computation time ratio: real vs. complex systems

Ratio	mcfe				dwg961b			
Algorithm	DD	TD	QD	MPFR	DD	TD	QD	MPC
ILU(0)/BiCG	456.4	43.5	37.7	16.2	960.1	106.2	102.8	15.1
ILU(0)_d/BiCG_d	476.7	42.4	37.2	8.6	313.6	29.5	42.8	8.2
ILU(0)/ILU(0)_d	1.0	1.1	1.1	1.3	3.2	3.7	2.5	1.2
ILU(0)/CGS	538.0	45.0	37.8	17.7	863.5	94.3	86.0	14.5
ILU(0)_d/CGS_d	515.4	43.5	35.7	9.1	284.3	25.8	17.7	7.8
ILU(0)/ILU(0)_d	1.1	1.1	1.1	1.3	3.2	3.8	5.0	1.2
ILU(0)/BiCGSTAB	526.5	41.8	34.0	16.5	773.7	88.8	74.8	13.5
ILU(0)_d/BiCGSTAB_d	505.7	40.4	32.2	8.6	258.0	24.3	15.6	7.5
ILU(0)/ILU(0)_d	1.1	1.1	1.0	1.3	3.1	3.8	4.9	1.2
ILU(0)/GPBiCG	500.9	40.6	33.8	15.8	644.8	74.6	74.8	10.5
ILU(0)_d/GPBiCG_d	481.5	40.4	32.2	8.3	212.9	20.4	12.7	6.4
ILU(0)/ILU(0)_d	1.1	1.1	1.1	1.3	3.1	3.7	4.8	1.2

- ILU(0) preconditioning increases total time by factors of 456.4 – 538.0 (DD), 40.6 – 45.0 (TD), and 33.8 – 37.8 (QD).
- Mixed-precision substitutions reduce time by -4.5 – 5.3% compared to standard ILU(0).
- For MPFR 256-bit, mixed-precision yields up to 47.1 – 48.5% reduction in ILU(0) cost.

In complex systems, mixed-precision preconditioning significantly improves performance, especially for multi-component formats. However, in MPC-based arbitrary precision, the effect is less pronounced.

- ILU(0) preconditioning increases total time by factors of 644.8 – 960.1 (DD), 74.6 – 106.2 (TD), and 74.8 – 102.8 (QD).
- Mixed-precision substitutions reduce time by 58.4 – 83.1% compared to standard ILU(0).
- For MPFR 256-bit, mixed-precision yields up to 39.2 – 46.0% reduction in ILU(0) cost.

5 Conclusion and Future Work

The benchmark results presented above indicate that, for cases where preconditioning is essential, the computational cost of ILU(0) increases with higher arithmetic precision. Specifically, DD precision incurs a 456x to 960x overhead, TD incurs 41x to 106x, and QD incurs 34x to 103x compared to un-preconditioned runs. In such cases, the use of mixed-precision SpMV and mixed-precision forward/backward substitution provides a modest reduction in computational cost,

typically around 58.4 – 83.1% for multi-component types in complex Krylov subspace methods.

In the MPFR 256-bit precision tests, the benefit of using mixed-precision substitution was even more evident, reducing ILU(0) computation time by approximately 39.2 – 48.5%. This shows that even a naive mixed-precision implementation can yield significant performance improvements, especially in high-precision environments.

Furthermore, for complex-valued systems, the benefit of mixed-precision is more pronounced when using multi-component types (DD, TD, QD). In contrast, the MPC-based arbitrary-precision results demonstrated minimal variation, likely due to the already high cost of complex arithmetic in those settings.

Currently, our ILU(0) implementation is not optimized or parallelized. The ILU factorization itself and the forward/backward substitutions are implemented sequentially, leading to significant computational overhead. To further improve performance, especially for practical use in high-precision environments, future work should include:

- SIMD optimization and OpenMP parallelization of ILU(0) routines
- Improved memory access patterns and cache-aware implementations
- Enhanced support for mixed precision preconditioning techniques
- Benchmarking on wider hardware platforms including Arm NEON, WASM, and SSE

Acknowledgment. This research was supported by JSPS KAKENHI Grant Number 23K11127. We thank all anonymous referees for their helpful comments, which greatly contributed to the revision of this paper.

References

1. D.H. Bailey. QD. https://www.davidhbailey.com/dhbsoftware/
2. The SuiteSparse Matrix Collection. https://sparse.tamu.edu/
3. Andreas Enge, Philippe Théveny, and Paul Zimmermann. MPC. http://www.multiprecision.org/mpc/
4. Hishinuma, T., Fujii, A., Tanaka, T., Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J., (eds.) Parallel Processing and Applied Mathematics, pp. 622–631, Berlin, Heidelberg. Springer (2014)
5. Tomonori Kouya. BNCmatmul. https://github.com/tkouya/bncmatmul
6. Kouya, T.: A highly efficient implementation of multiple precision sparse matrix-vector multiplication and its application to product-type krylov subspace methods. Int. J. Num. Meth. Appl. **7**(2), 107–119 (2012)
7. Kouya, T.: Performance evaluation of accelerated real and complex multiple-precision sparse matrix-vector multiplication. In: 2024 International Conference on Engineering and Emerging Technologies (ICEET), pp. 1–6 (2024)
8. Intel Math Kernel Library. http://www.intel.com/software/products/mkl/
9. MPLAPACK/MPBLAS. Multiple precision arithmetic LAPACK and BLAS. https://github.com/nakatamaho/mplapack
10. OpenBLAS. http://www.openblas.net/
11. MPFR Project. The MPFR library. https://www.mpfr.org/