

Acceleration of Multicomponent Multiple-Precision Arithmetic with Branch-Free Algorithms and SIMD Vectorization

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. Multiple-precision floating-point branch-free algorithms can significantly accelerate multicomponent arithmetic implemented by combining hardware-based binary64 and binary32, particularly for triple- and quadruple-precision computations. In this study, we achieved benchmark results on x86 and ARM CPU platforms to quantify the accelerations achieved in linear computations and polynomial evaluations by integrating these algorithms.

Keywords: branch-free · multiple-precision floating-point arithmetic · linear computation · polynomial evaluation

1 Introduction

Recent advances in artificial intelligence (AI), underpinned by deep-learning technologies, have led to the widespread adoption of large language models and generative AI systems, permeating diverse industrial sectors and everyday human activities, including software development and coding. This trend has accelerated the shortening of the mantissa lengths in floating-point formats employed in AI computations to achieve higher performance. However, in scientific and engineering computing, double-precision arithmetic is the minimum standard, and ill-conditioned problems frequently require higher precision. In such cases, the effective use of single-instruction multiple-data (SIMD) instructions and OpenMP-based parallelization techniques is crucial for harnessing the full capabilities of modern hardware. Among the existing multiple-precision arithmetic frameworks, arbitrary-precision arithmetic is dominated by MPFR [15], which relies on the multiple-precision natural-number arithmetic (MPN) kernel of GMP [5]. MPC [3], an arbitrary precision complex arithmetic library built on MPFR that achieves acceleration through the Karatsuba 3M multiplication algorithm, is widely used.

In contrast, fixed-precision multicomponent arithmetic based on error-free transformation (EFT) techniques, which employs combinations of multiple IEEE

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026 Workshops, LNCS 16760, pp. 500–516, 2027.
https://doi.org/10.1007/978-3-032-30521-3_32

binary32 (24-bit mantissa and 32-bit total length) and binary64 values supported by existing hardware, is represented by quadruple-double (QD) [1] for CPUs and GQD [12]—a CUDA port of QD, though dated—for GPUs. When a relatively short mantissa length suffices, such as in double-double (DD, 106-bit), triple-double (TD, 159-bit), and quadruple-double (QD, 212-bit) arithmetic implemented by combining two to four binary64 values, computations can be executed faster than with MPFR. Kotakemori et al.'s Lis [7] and Hishinuma et al. [6] further demonstrated that basic linear computations can be accelerated by leveraging AVX, AVX2, and AVX-512, which are x86-based SIMD instruction sets. Building on an existing AVX2 implementation, we demonstrated that basic linear computations in DD, TD, and QD precision can be accelerated on the Raspberry Pi and Snapdragon platforms by exploiting the Neon instructions [18].

Nevertheless, DD, TD, and QD arithmetic require renormalization at the conclusion of each operation. In particular, for TD and QD arithmetic, this step has been identified as a bottleneck that impedes performance gains through SIMD vectorization. Double-precision pair arithmetic, which bypasses renormalization, has been proposed as a faithfully rounded pair-arithmetic scheme [11], and practical sparse matrix solver implementations that exploit this efficiency have been reported [14]. The branch-free algorithms for multiple-precision floating-point arithmetic addressed in this work [17] eliminate all conditional branches from every algorithmic step, including renormalization, and reduce the operation count for TD and QD arithmetic relative to conventional algorithms. Consequently, this methodology can substantially accelerate triple- and quadruple-precision multicomponent arithmetic implemented by combining hardware-based binary64 and binary32 values and is immediately applicable to existing fixed-precision multiple-precision arithmetic implementations. Because the benchmarks presented in the original paper provided an insufficient comparison against existing SIMD implementations, we first quantified the speedups achievable in basic linear computations by incorporating these algorithms on the x86 and ARM CPU platforms. Furthermore, as a practical application, we analyzed the effectiveness of the proposed approach by applying it to a multiple-precision algebraic equation solver.

2 Branch-Free Algorithms for DD, TD, and QD Addition and Multiplication

The extension of fixed-precision floating-point numbers via EFT was pioneered by Dekker [2]. Multiple-precision arithmetic with an extended mantissa length was implemented by employing EFT primitives, represented by QuickTwoSum (Algorithms 1, 2, 3), which return both a rounded result and its rounding error using only existing floating-point operations (addition $\oplus$, subtraction $\ominus$, and multiplication $\otimes$). In the following, all variables are assumed to be exactly representative as normalized floating-point numbers.

QuickTwoSum (Algorithm 1) returns the floating-point addition result $s = a \oplus b$ and its rounding error e, but only under the precondition $|a| \geq |b|$; if no overflow occurs, it maintains the relation $a + b = s + e$.

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

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

TwoSum (Algorithm 2) returns the floating-point addition result $s = a \oplus b$ and its rounding error e unconditionally, regardless of the relative magnitudes of a and b, while maintaining the relationship $a + b = s + e$.

Algorithm 2 $(s, e) :=$ 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 (Algorithm 3) returns the floating-point multiplication result $p = a \otimes b$ and its rounding error e while maintaining the relationship $ab = p + e$. Because virtually all floating-point hardware currently in operation supports fused multiply–add (FMA) instructions, the FMA-based variant of TwoProd was employed throughout this work.

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

$p := a \otimes b$; $e :=$ FMA$(a, b, -p)$ $(= a \times b - p)$; **return**(p, e)

As these EFT operations are constructed entirely from existing floating-point operations and contain no conditional branches, they can be implemented directly using SIMD instructions. With AVX2, which employs 256-bit registers on x86 architectures, four binary64 or eight binary32 values can be processed simultaneously by EFT operations. In contrast, with Neon on AArch64, which employs 128-bit registers, two binary64 or four binary32 values can be processed simultaneously. Computations that contain no conditional branches and can be executed without overflowing SIMD registers in this manner are termed branch-free (BF).

By combining EFT operations, multiple-precision arithmetic with extended mantissa can be implemented by composing multiple floating-point numbers; this is referred to as the multicomponent (or multiterm) method. Because subtraction reduces to sign negation and division can be implemented via Newton's method that combines addition and multiplication, these two operations are crucial from a performance perspective. Correct handling of the rounding errors is critical in this implementation and requires a carefully derived rigorous error analysis. As

the mantissa length increases, the operation count increases, and error management becomes increasingly complex. Zhang & Aiken proposed an abstract model for automated reasoning that simplifies floating-point arithmetic [16] and introduced branch-free multicomponent multiple-precision addition and multiplication algorithms. In the following section, we present and discuss BF algorithms.

2.1 Remarks on Double-Word (DW) Arithmetic

The simplest form of multicomponent multiple-precision floating-point arithmetic is the DW arithmetic. When binary64 (double-precision) is used as the base format, it is abbreviated as DD; when binary32 (single-precision) is used, it is abbreviated as DS. Algorithm 4 presents the default DW addition defined in QD [1], which is the sloppy or fast variant; its final rounding error is larger than that of the accurate version (Algorithm 5).

Algorithm 4 $c[2] := \mathrm{DWAdd}(a[2],\ b[2])$

$(s, e) := \mathrm{TwoSum}(a[0], b[0])$; $e := e \oplus (a[1] \oplus b[1])$; $(c[0], c[1]) := \mathrm{QuickTwoSum}(s, e)$;
return$(c[2])$

Algorithm 5 $c[2] := \mathrm{DWAccurateAdd}(a[2],\ b[2])$

$(s_1, s_2) := \mathrm{TwoSum}(a[0], b[0])$; $(t_1, t_2) := \mathrm{TwoSum}(a[1], b[1])$;
$s_2 := s_2 \oplus t_1$; $(s_1, s_2) := \mathrm{QuickTwoSum}(s1, s2)$;
$s_2 := s_2 \oplus t_2$; $(c[0], c[1]) := \mathrm{QuickTwoSum}(s1, s2)$;
return$(c[2])$

Zhang and Aiken proposed Algorithm 6 as a branch-free reformulation for accurate DW addition.

Algorithm 6 $c[2] := \mathrm{DWBFAdd}(a[2],\ b[2])$

$(g_1, g_{1e}) := \mathrm{TwoSum}(a[0], b[0])$;
$(g_2, g_{2e}) := \mathrm{TwoSum}(a[1], b[1])$;
$(g_3, g_{3e}) := \mathrm{QuickTwoSum}(g_1, g_2)$;
$g_4 := g_{1e} \oplus g_{2e}$;
$g_5 := g_4 \oplus g_{3e}$;
$(c[0], c[1]) := \mathrm{QuickTwoSum}(g_3, g_5)$;
return$(c[2])$

Algorithm 7 presents the DW multiplication defined as the default in QD.

Algorithm 7 $c[2] := \mathrm{DWMul}(a[2], b[2])$

$(p_1, p_2) := \mathrm{TwoProd}(a[0], b[0])$; $p_2 := p_2 \oplus (a[0] \otimes b[1] \oplus a[1] \otimes b[0])$;
$(c[0], c[1]) := \mathrm{QuickTwoProd}(p_1, p_2)$;
return$(c[2])$

Algorithm 8 $c[2] := \mathrm{DWBFMul}(a[2], b[2])$

$(p_{00}, p_{e00}) := \mathrm{TwoProd}(a[0], b[0])$;
$p_{01} := a[0] \otimes b[1]$;
$p_{10} = a[1] \otimes b[0]$;
$g_1 := p_{01} \oplus p_{10}$;
$g_2 := p_{e00} \oplus g_1$;
$(c[0], c[1]) := \mathrm{QuickTwoSum}(p_{00}, g_2)$;
return$(c[2])$

The corresponding BF multiplication is presented as Algorithm 8. This algorithm can be interpreted as performing all necessary multiplications, followed by arranging the error-handling steps in a branch-free manner.

Consequently, the BF variants of DW addition and multiplication involve no fewer floating-point operations than their conventional counterparts. In addition, as discussed in the subsequent sections, they exhibit no measurable performance benefit under the current default DW arithmetic. Therefore, in the following benchmark tests, DD arithmetic employs conventional (non-BF) implementations unless otherwise stated; only TD and QD arithmetic utilize the BF algorithms presented hereafter.

2.2 Remarks on TW Arithmetic

For triple-precision arithmetic (triple-word, TW), one approach is to consider the fourth component of the QD arithmetic as zero. In contrast, Fabiano et al. [4] proposed a formulation with a reduced operation count. Through SIMD implementation, we validated that the former approach outperforms the latter [9]; therefore, Algorithm 9 is used for TW addition and Algorithm 10 for TW multiplication.

Both algorithms conclude with a call to the TW renormalization function TWRenormalize [4] [1], which requires up to five QuickTwoSum operations and necessitates conditional branching via if-statements. This function has not yet been SIMD-vectorized and is considered the primary source of performance degradation.

By contrast, the proposed BF algorithms presented in Algorithms 11 and 12 introduce no conditional branches in the renormalization-equivalent steps and require fewer operations than their conventional counterparts. This renders them highly amenable to performance improvements through SIMD vectorization.

As listed in Table 1, the branch-free triple-word multiplication has the least total number of primary computations compared with our currently adopted

Algorithm 9 $c[3] := \mathrm{TWAdd}(a[3], b[3])$

$s_0 := a[0] \oplus b[0]$; $s_1 := a[1] \oplus b[1]$; $s_2 := a[2] \oplus b[2]$;
$v_0 := s_0 \ominus a[0]$;$v_1 := s_1 \ominus a[1]$;$v_2 := s_2 \ominus a[2]$;
$u_0 := s_0 \ominus v_0$; $u_1 := s_1 \ominus v_1$; $u_2 := s_2 \ominus v_2$;
$w_0 := a[0] \ominus u_0$;$w_1 := a[1] \ominus u_1$; $w_2 := a[2] \ominus u_2$;
$u_0 := b[0] \ominus v_0$; $u_1 := b[1] \ominus v_1$; $u_2 := b[2] \ominus v_2$;
$t_0 := w_0 \oplus u_0$; $t_1 := w_1 \oplus u_1$; $t_2 := w_2 \oplus u_2$;
$(s_1, t_0) := \mathrm{TwoSum}(s_1, t_0)$; $(i_1, i_2) := \mathrm{TwoSum}(s_2, t_0)$;
$(s_2, i_3) := \mathrm{TwoSum}(t_1, i_1)$; $(t_0, t_1) := \mathrm{TwoSum}(i_2, i_3)$ $t_0 := t_0 \oplus t_1 \oplus t_2$;
$c[3] := \mathrm{TWRenormalize}(s_0, s_1, s_2, t_0)$;
return$(c[3])$

Algorithm 10 $c[3] := \mathrm{TWMul}(a[3], b[3])$

$(p_0, q_0) := \mathrm{TwoProd}(a[0], b[0])$; $(p_1, q_1) := \mathrm{TwoProd}(a[0], b[1])$;
$(p_2, q_2) := \mathrm{TwoProd}(a[1], b[0])$;
$(p_3, q_3) := \mathrm{TwoProd}(a[0], b[2])$; $(p_4, q_4) := \mathrm{TwoProd}(a[1], b[1])$;
$(p_5, q_5) := \mathrm{TwoProd}(a[2], b[0])$;
$(i_1, i_2) := \mathrm{TwoSum}(p_1, p_2)$; $(p_1, i_3) := \mathrm{TwoSum}(q_0, i_1)$; $(p_2, q_0) := \mathrm{TwoSum}(i_2, i_3)$;
$(i_1, i_2) := \mathrm{TwoSum}(p_2, q_1)$; $(p_2, i_3) := \mathrm{TwoSum}(q_2, i_1)$; $(q_1, q_2) := \mathrm{TwoSum}(i_2, i_3)$;
$(i_1, i_2) := \mathrm{TwoSum}(p_3, p_4)$; $(p_3, i_3) := \mathrm{TwoSum}(p_5, i_1)$; $(p_4, p_5) := \mathrm{TwoSum}(i_2, i_3)$;
$(s_0, t_0) := \mathrm{TwoSum}(p_2, p_3)$; $(s_1, t_1) := \mathrm{TwoSum}(q_1, p_4)$; $s_2 := q_2 \oplus p_5$;
$(s_1, t_0) := \mathrm{TwoSum}(s_1, t_0)$; $s_2 := s_2 \oplus (t_0 \oplus t_1)$; $(q_0, q_3) := \mathrm{TwoSum}(q_0, q_3)$;
$(q_4, q_5) := \mathrm{TwoSum}(q_4, q_5)$; $(t_0, t_1) := \mathrm{TwoSum}(q_0, q_4)$; $t_1 := t_1 \oplus (q_3 \oplus q_5)$;
$(t_0, t_1) := \mathrm{TwoSum}(q_3, s_1)$;
$c[3] := \mathrm{TWRenormalize}(p_0, p_1, s_0, t_0)$;
return$(c[3])$

and Fabiano's and triple-word multiplications. However, we could not find a clear difference among the three types of triple-word additions.

2.3 Remarks on QW Arithmetic

Quadruple-precision arithmetic (quadruple-word, QW) is based on the implementation provided in QD. As noted previously, TD arithmetic is an abbreviated form of QD arithmetic with the fourth component zeroed. Both the QD addition and multiplication conclude with a call to the QW renormalization function (QWRenormalize [1]), which contains conditional branches and requires up to seven QuickTwoSum function calls.

The BF algorithms are presented in Algorithms 13 and 14. Although the time required to discover these BF algorithms is unknown, the resulting algorithms are remarkably concise and involve substantially reduced operation counts. Consequently, the benefit of SIMD vectorization is expected to be more pronounced for QW arithmetic than for TW arithmetic.

We present Table 2 for comparison with our adopted and branch-free quadruple-word algorithms. Similarly, as listed in Table 1, branch-free multi-

Algorithm 11 $c[3] := \mathrm{TWBFAdd}(a[3], b[3])$

$(a_1, b_1) := \mathrm{TwoSum}(a[0], b[0])$; $(c_1, d_1) := \mathrm{TwoSum}(a[1], b[1])$;
$(e_1, f_1) := \mathrm{TwoSum}(a[2], b[2])$;
$(a_2, c_2) := \mathrm{QuickTwoSum}(a_1, c_1)$; $b_2 := b_1 \oplus f_1$;
$(d_2, e_2) := \mathrm{TwoSum}(d_1, e_1)$; $(a_3, d_3) := \mathrm{QuickTwoSum}(a_2, d_2)$;
$(b_3, c_3) := \mathrm{TwoSum}(b_2, c_2)$; $c_4 := c_3 \oplus e_2$;
$(c_5, d_5) := \mathrm{TwoSum}(c_4, d_3)$; $(b_6, c_6) := \mathrm{TwoSum}(b_3, c_5)$;
$(c[0], b_7) = \mathrm{QuickTwoSum}(a_3, b_6)$; $c_7 := c_6 \oplus d_5$;
$(c[1], c[2]) := \mathrm{QuickTwoSum}(b_7, c_7)$;
return$(c[3])$

Algorithm 12 $c[3] := \mathrm{TWBFMul}(a[3], b[3])$

$(a_0, b_0) := \mathrm{TwoProd}(a[0], b[0])$; $(c_0, e_0) := \mathrm{TwoProd}(a[0], b[1])$;
$(d_0, f_0) := \mathrm{TwoProd}(a[1], b[0])$;
$g_0 := a[0] \otimes b[2]$; $h_0 := a[1] \otimes b[1]$; $i_0 := a[2] \otimes b[0]$;
$(c_1, d_1) := \mathrm{TwoSum}(c_0, d_0)$; $e_1 := e_0 \oplus f_0$; $g_1 := g_0 \oplus i_0$;
$(b_2, c_2) := \mathrm{TwoSum}(b_0, c_1)$; $g_2 := g_1 \oplus h_0$; $(a_3, b_3) := \mathrm{QuickTwoSum}(a_0, b_2)$;
$c_3 := c_2 \oplus d_1$; $e_3 := e_1 \oplus g_2$; $c_4 := c_3 \oplus e_3$;
$(b_5, c_5) := \mathrm{QuickTwoSum}(b_3, c_4)$; $(c[0], b_6) := \mathrm{QuickTwoSum}(a_3, b_5)$;
$(c[1], c[2]) := \mathrm{QuickTwoSum}(b_6, c_5)$;
return$(c[3])$

plication is the most lightweight, and no significant difference exists between the two types of quadruple-word additions.

3 Benchmark Tests for Real and Complex Square Matrix Multiplication

As discussed previously, the adoption of BF arithmetic is expected to yield significant performance gains through SIMD vectorization for triple- and quadruple-precision computations. Here, we report performance evaluation results for real and complex square matrix multiplication obtained by integrating BF arithmetic

Table 1. Operation counts for Triple-Word arithmetic

Triple-Word (TW)	Add	Sub	Mul	Div	FMA
TW+TW	19	34	0	0	0
Fabiano TW+TW	20	40	0	0	0
BF TW+TW	21	36	0	0	0
TW×TW	42	74	6	0	6
Fast Fabiano TW×TW	24	38	0	0	3
BF TW×TW	14	16	6	0	3

Algorithm 13 $c[4] := \mathrm{QWBFAdd}(a[4], b[4])$

$(a_1, b_1) := \mathrm{TwoSum}(a[0], b[0])$; $(c_1, d_1) := \mathrm{TwoSum}(a[1], b[1])$;
$(e_1, f_1) := \mathrm{TwoSum}(a[2], b[2])$;
$(g_1, h_1) := \mathrm{TwoSum}(a[3], b[3])$; $(a_2, c_2) := \mathrm{QuickTwoSum}(a_1, c_1)$;
$b_2 := b_1 \oplus h_1$; $(d_2, e_2) := \mathrm{TwoSum}(d_1, e_1)$;
$(f_2, g_2) := \mathrm{TwoSum}(f_1, g_1)$; $(b_3, g_3) := \mathrm{TwoSum}(b_2, g_2)$;
$(c_3, d_3) := \mathrm{QuickTwoSum}(c_2, d_2)$; $(e_3, f_3) := \mathrm{TwoSum}(e_2, f_2)$;
$(a_4, c_4) := \mathrm{QuickTwoSum}(a_2, c_3)$; $(d_4, e_4) := \mathrm{QuickTwoSum}(d_3, e_3)$;
$(b_5, d_5) := \mathrm{TwoSum}(b_3, d_4)$; $e_5 := e_4 \oplus f_3$;
$(b_6, c_6) := \mathrm{TwoSum}(b_5, c_4)$; $(d_6, e_6) := \mathrm{TwoSum}(d_5, e_5)$;
$(a_7, b_7) := \mathrm{QuickTwoSum}(a_4, b_6)$; $(c_7, d_7) := \mathrm{QuickTwoSum}(c_6, d_6)$;
$e_8 := e_6 \oplus g_3$; $(b_8, c_8) := \mathrm{QuickTwoSum}(b_7, c_7)$;
$d_9 := d_7 \oplus e_8$; $(c[0], b_{10}) := \mathrm{QuickTwoSum}(a_7, b_8)$;
$(c_{10}, d_{10}) := \mathrm{QuickTwoSum}(c_8, d_9)$; $(c[1], c_{11}) := \mathrm{QuickTwoSum}(b_{10}, c_{10})$;
$(c[2], c[3]) := \mathrm{QuickTwoSum}(c_{11}, d_{10})$;
return$(c[4])$

Algorithm 14 $c[4] := \mathrm{QDBFMul}(a[4], b[4])$

$(a_0, b_0) := \mathrm{TwoProd}(a[0], b[0])$; $(c_0, e_0) := \mathrm{TwoProd}(a[0], b[1])$;
$(d_0, f_0) := \mathrm{TwoProd}(a[1], b[0])$; $(g_0, j_0) := \mathrm{TwoProd}(a[0], b[2])$;
$(h_0, k_0) := \mathrm{TwoProd}(a[1], b[1])$ $(i_0, l_0) := \mathrm{TwoProd}(a[2], b[0])$;
$m_0 := a[0] \otimes b[3]$; $n_0 := a[1] \otimes b[2]$; $o_0 := a[2] \otimes b[1]$;
$p_0 := a[3] \otimes b[0]$; $(c_1, d_1) := \mathrm{TwoSum}(c_0, d_0)$;
$(e_1, f_1) := \mathrm{TwoSum}(e_0, f_0)$; $(g_1, i_1) := \mathrm{TwoSum}(g_0, i_0)$;
$j_1 := j_0 \oplus l_0$; $m_1 := m_0 \oplus p_0$; $n_1 := n_0 \oplus o_0$;
$(b_2, c_2) := \mathrm{TwoSum}(b_0, c_1)$; $(e_2, h_2) := \mathrm{TwoSum}(e_1, h_0)$;
$f_2 := f_1 \oplus j_1$; $i_2 := i_1 \oplus k_0$; $m_2 := m_1 \oplus n_1$;
$(a_3, b_3) := \mathrm{QuickTwoSum}(a_0, b_2)$; $(c_3, d_3) := \mathrm{QuickTwoSum}(c_2, d_1)$;
$(e_3, g_3) := \mathrm{TwoSum}(e_2, g_1)$; $f_3 := f_2 \oplus m_2$; $h_3 := h_2 \oplus i_2$;
$(c_4, e_4) := \mathrm{TwoSum}(c_3, e_3)$; $d_4 := d_3 \oplus h_3$; $f_4 := f_3 \oplus g_3$;
$d_5 := d_4 \oplus e_4$; $(c_6, d_6) := \mathrm{TwoSum}(c_4, d_5)$; $(b_7, c_7) := \mathrm{TwoSum}(b_3, c_6)$;
$d_7 := d_6 \oplus f_4$; $(c[0], b_8) := \mathrm{QuickTwoSum}(a_3, b_7)$;
$(c_8, d_8) := \mathrm{TwoSum}(c_7, d_7)$; $(c[1], c_9) := \mathrm{TwoSum}(b_8, c_8)$;
$(c[2], c[3]) := \mathrm{QuickTwoSum}(c_9, d_8)$;
return$(c[4])$

into our multiple-precision basic linear computation library BNCmatmul [8], executed on the following x86 AVX2 and Arm Neon environments.

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

Snapdragon Lenovo IdeaPad Slim 5 (Snapdragon X Plus X1P-42-100, 3.4 GHz, 34 MB L3 cache, 8 cores), Windows 11, WSL2 + Ubuntu 24.04.2 LTS, GCC 13.3.0

Standard intrinsic functions, provided as C functions, were used for SIMD implementation in both environments. Arm Neon vectorization and OpenMP

Table 2. Operation counts for quad-word arithmetic

Quad-Word (QW)	Add	Sub	Mul	Div	FMA
QW+QW	26	58	0	0	0
BF QW+QW	37	66	0	0	0
QW×QW	67	94	12	0	9
BF QW×QW	38	52	10	0	6

parallelization were implemented primarily with assistance from the Anthropic Claude by converting the existing AVX2 code [18].

3.1 Real Matrix Multiplication

The square matrices employed in the tests are defined as follows:

$$\begin{aligned} &C := AB, \\ &A = [a_{ij}]_{i,j=1,2,\ldots,n}, B = [b_{ij}]_{i,j=1,2,\ldots,n} \in \mathbb{R}^{n\times n} \\ &a_{ij} := \sqrt{5}(i+j-1), b_{ij} := \sqrt{3}(n-i+1) \end{aligned} \tag{1}$$

as defined in (1).

Our library supports four matrix multiplication schemes: the naive triple-loop method, a block-based algorithm, Strassen's algorithm, and the Ozaki scheme. In this study, Strassen's algorithm was employed to isolate the performance contribution of the BF arithmetic. The recursion in Strassen's algorithm terminates when the submatrix size decreases below 32×32. The performance benefit of BF arithmetic was measured for both nonvectorized and SIMD-vectorized (AVX2 and Neon) implementations. Execution times with OpenMP parallelization using 8 and 32 threads were also reported. Because our Strassen routine did not achieve further speedup beyond eight threads, the results for the block-based algorithm (B32 threads) were included only for the 32-thread case. The numerical accuracy of all computed results falls within the ranges of 29.0–30.4 significant decimal digits for DD, 45.5–47.5 for TD, and 61.7–63.5 for QD precision.

Figure 1 shows the computation times for real square matrix multiplication at TD and QD precision. The horizontal axis represents the matrix size $n = 32$–2049, and the vertical axis represents the computation time in seconds.

Notably, BF arithmetic provides no benefit and degrades performance for DD precision, whereas it yields clear speedups for TD and QD precisions. The resulting speedups over the MPBLAS [13] DD- and QD-precision Rgemm for $n = 2049$ are summarized in Table 3.

Table 4 lists the average speedup ratios owing to BF arithmetic for DD precision, where BF arithmetic offers no benefit compared to nonSIMD (NS) and SIMD-vectorized implementations.

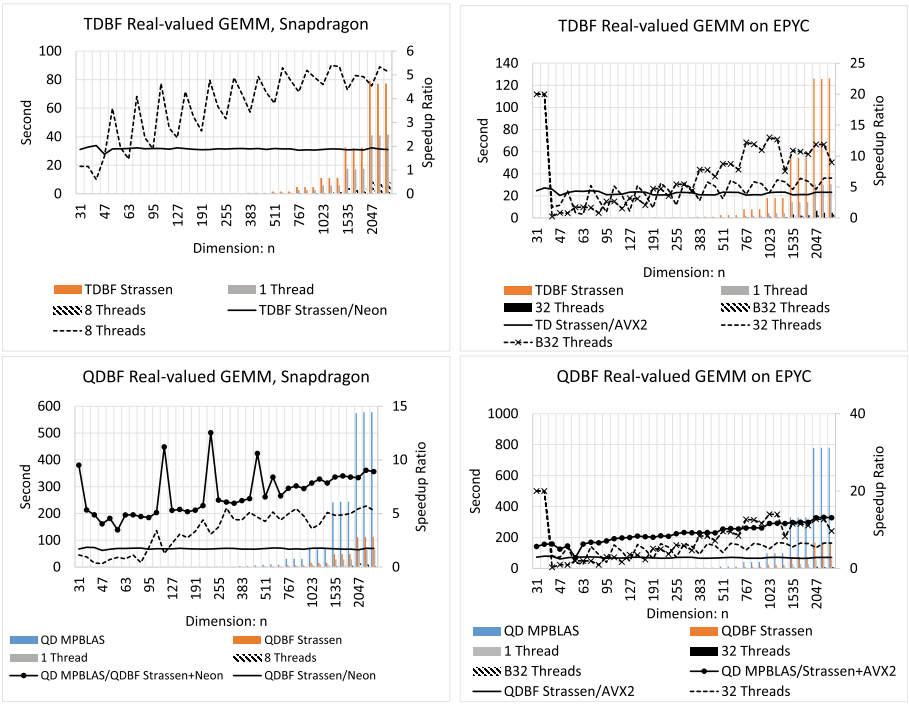

Fig. 1. Computation time (s) of real-valued Strassen matrix multiplication: Snapdragon (left), EPYC (right).

Table 3. Real-valued matrix multiplication time (s) for $n = 2049$

	MPBLAS	S+SIMD	BF S+SIMD	8 Threads	32 Threads	B32 Threads
DD Snapdragon	13.6	9.7	15.4	2.5	–	–
EPYC	13.9	6.7	10.3	1.5	1.4	0.9
TD Snapdragon	–	61.7	39.9	8.1	–	–
EPYC	–	46.1	30.4	4.8	4.7	3.4
QD Snapdragon	577.5	101.4	65.6	12.3	–	–
EPYC	781.0	117.2	59.9	9.2	9.1	6.2

Table 4. Average speedup with branch-free (BF) arithmetic: standard / BF

Speedup	DD		TD		QD	
ratio	NS	SIMD	NS	SIMD	NS	SIMD
Snapdragon	0.6	0.7	1.2	1.4	2.0	1.4
EPYC	1.0	0.7	1.4	1.5	1.5	1.9

As expected, DD precision exhibited a slowdown of 0.6–1.0×; however, TD precision achieved speedups of 1.2–1.5× and QD precision achieved speedups of 1.4–2.0×. Although the true benefit of BF arithmetic is difficult to discern from the benchmarks in the original study, the aforementioned results demonstrate that BF arithmetic effectively accelerates TD and QD precision computations. Furthermore, speedups were achieved through Arm Neon vectorization and OpenMP-based parallelization.

3.2 Benchmark Tests for Complex Matrix Multiplication

Owing to the previously demonstrated superiority of BF arithmetic for TD and QD computations, complex matrix multiplication benchmarks employed BF-based implementations for comparison. Complex square matrix multiplication is defined using a uniform random variable $r_u \in [0, 1]$ and a standard normal random variable $r_n \in \mathbb{R}$ as

$$\begin{aligned} C &:= AB, \\ A = [a_{ij}]_{i,j=1,2,\ldots,n}, B &= [b_{ij}]_{i,j=1,2,\ldots,n} \in \mathbb{C}^{n\times n} \\ a_{ij} &:= \exp(r_n)(r_u - 1/2) + \exp(r_n)(r_u - 1/2)\sqrt{-1} \\ b_{ij} &:= \exp(r_n)(r_u - 1/2) + \exp(r_n)(r_u - 1/2)\sqrt{-1} \end{aligned} \tag{2}$$

Our complex matrix multiplication was implemented using three real matrix multiplications via the 3M method, a variant of the Karatsuba algorithm [10] as follows:

$$\begin{aligned} C_1 := \mathrm{Re}(A)\mathrm{Re}(B),\ C_2 &:= \mathrm{Im}(A)\mathrm{Im}(B) \\ \mathrm{Re}(C) &:= C_1 - C_2 \\ \mathrm{Im}(C) &:= (\mathrm{Re}(A) + \mathrm{Im}(A))(\mathrm{Re}(B) + \mathrm{Im}(B)) - C_1 - C_2. \end{aligned} \tag{3}$$

Therefore, both the Strassen- and block-based algorithms perform three real matrix multiplications. The numerical accuracy, measured as the minimum of the real and imaginary parts, falls within the ranges of 23.2–24.4 significant decimal digits for DD, 39.2–40.8 for TD, and 55.8–56.5 for QD precision.

Figure 2 presents the benchmark results for complex matrix multiplication on the EPYC and Snapdragon platforms.

The results demonstrate performance characteristics consistent with those observed for real matrix multiplication. In particular, extracting the computation times for $n = 2049$ reveals that our implementation achieved speedups that significantly exceeded those for real matrix multiplication relative to MPBLAS, as listed in Table 5.

Consequently, our complex matrix multiplication requires only 2.8–3.5× the computation time of real matrix multiplication across all precision levels. In contrast, MPBLAS, which employs the standard 4M method for each operation, requires 3.5–18.4× the computation time. These results indicate that the 3M-based implementation becomes increasingly advantageous as precision and matrix size increase.

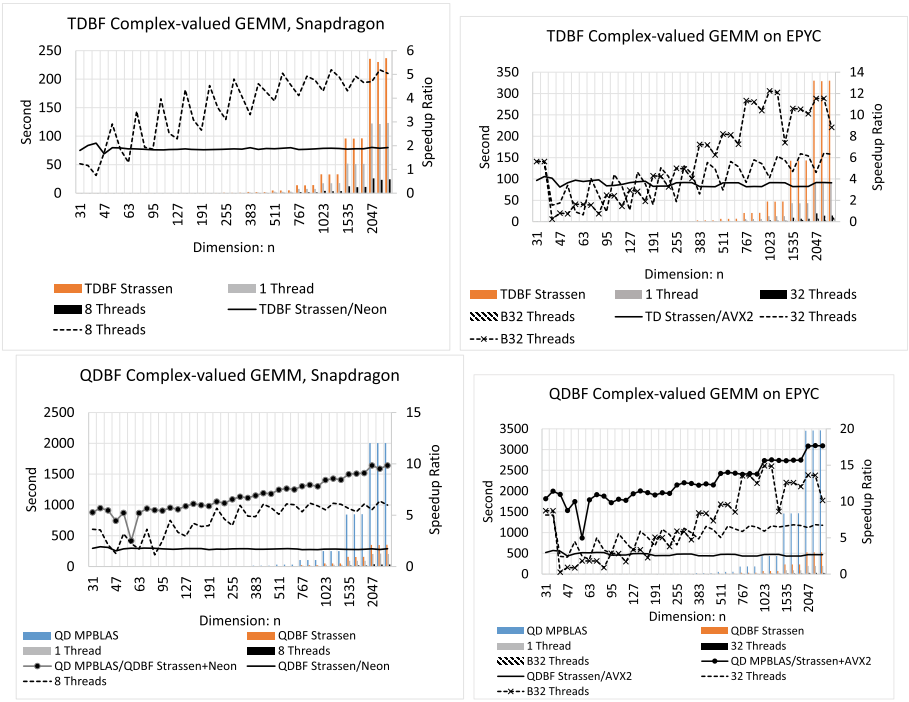

Fig. 2. Computation time (s) of complex-valued Strassen matrix multiplication: Snapdragon (left), EPYC (right).

4 Benchmark Tests for Polynomial Evaluation and Algebraic Equation Solvers

As demonstrated by the matrix multiplication benchmarks, BF arithmetic offers no speedup in DD precision when conventional sloppy addition and multiplication are employed. In this section, we present the benchmark results for a polynomial evaluation and an algebraic equation solver based on the Durand–Kerner (DK) method, both of which utilize TD and QD precision arithmetic, for which performance improvements are anticipated. As described previously, we did not use the BF algorithm for the DD evaluation of a polynomial, but used the BF polynomial evaluation for TD and QD arithmetic.

4.1 Evaluation of Real-Coefficient Polynomial Functions

For the polynomial evaluation, the coefficients of the degree-n real-coefficient polynomial $p_n(x) = \sum_{i=0}^{n} a_i x^i$ were generated randomly, and the results were compared among the standard Horner's method, Estrin's method, and SIMD-vectorized Estrin's method.

Table 5. Complex-valued matrix multiplication time (s) for $n = 2049$

Precision	Platform	MPBLAS	S+SIMD	8 Threads	32 Threads	B32 Threads
DD	Snapdragon	72.1	29.9	7.8	–	–
	EPYC	254.3	20.4	5.0	4.4	3.0
TDBF	Snapdragon	–	123.2	24.4	–	–
	EPYC	–	90.5	14.6	14.4	10.3
QDBF	Snapdragon	2004.2	203.4	34.0	–	–
	EPYC	3463.1	196.1	29.2	29.2	19.4

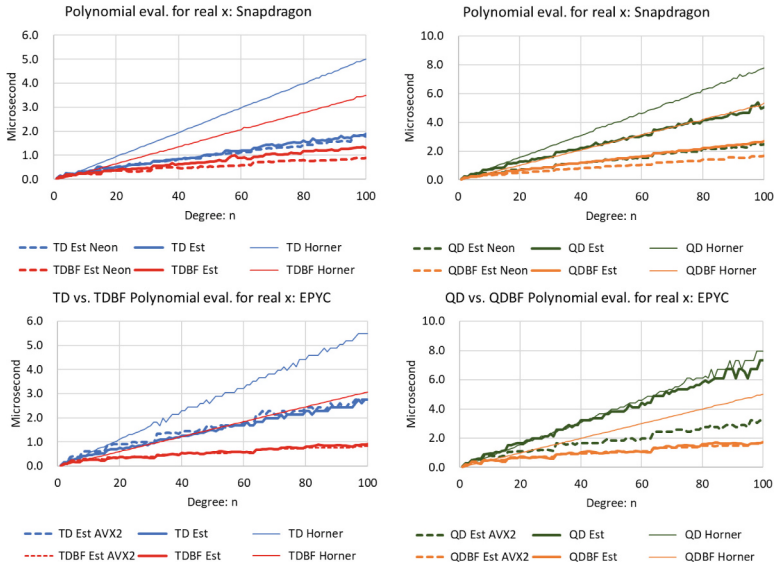

Fig. 3. Computation time (μs) for evaluating a real-coefficient polynomial $p_n(x)$ at real arguments: Snapdragon (top), EPYC (bottom).

Figure 3 shows the computation time (in microseconds) required to evaluate the real-coefficient polynomial $p_n(x)$ in real arguments. The upper panel corresponds to the Snapdragon platform and the lower panel to the EPYC platform; the standard arithmetic (NoBF) is compared with branch-reduction BF arithmetic (BF). As the degree (or equivalently, the evaluation count corresponding to the problem size) increases, the computation time increases, and the effects of branch elimination and vectorization become more pronounced, particularly for high-precision variants such as TD and QD, where the operation count and cost are dominant. Across both architectures, a consistent trend is observed: BF reduces the computation time relative to NoBF for TD and QD precisions.

Figure 4 shows the computation time (in microseconds) required to evaluate the real-coefficient polynomial $p_n(x)$ for the complex arguments. Because each

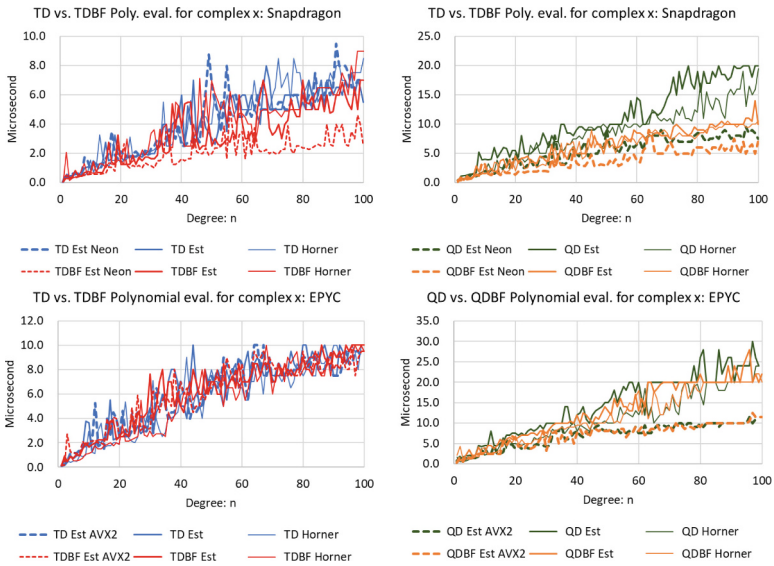

Fig. 4. Computation time (μs) for evaluating a real-coefficient polynomial $p_n(x)$ at complex arguments: Snapdragon (top) and EPYC (bottom).

complex multiplication and addition decomposes into multiple real operations, the computation times are higher than those for real-argument evaluation, as shown in Fig. 3. In contrast, as the number of operations increases, the impact of the conditional branching and normalization steps also increases; therefore, BF arithmetic should, in principle, remain effective for complex-argument evaluation. Although the Snapdragon results largely conform to this expectation, the EPYC results show a slight discernible difference. In particular, the performance difference for TD arithmetic appears to be minimal, even when one exists.

4.2 Algebraic Equation Solver Based on the DK Method

The method being compared is the quadratically convergent DK method, which is a classical simultaneous iterative solver. The approximate roots after k iterations are expressed as follows:

$$\mathbf{z}_k = [z_1^{(k)} \; z_2^{(k)} \; \ldots \; z_n^{(k)}]^T \in \mathbb{C}^n,$$

The recurrence relationship, represented using the monic polynomial $q_n(x) = p_n(x)/a_n$, is expressed by

$$z_i^{(k+1)} := z_i^{(k)} - \frac{q_n(z_i^{(k)})}{\prod_{j=1, j\neq i}^{n}(z_i^{(k)} - z_j^{(k)})} \tag{4}$$

as the quadratically convergent DK method.

The Aberth initial values are expressed as

$$z_i^{(0)} := -\frac{c_{n-1}}{n} + r \exp\left\{\left(\frac{2(i-1)\pi}{n} + \frac{3}{2n}\right)\mathrm{i}\right\} \tag{5}$$

In the experiments, the radius r was defined using the number of nonzero coefficients $n_{\mathrm{nz}} \le n$ as

$$r := \max_{0 \le i \le (n-1)} |n_{\mathrm{nz}}\, c_i|^{1/(n-i)}.$$

The test problem was the Chebyshev integration problem, in which the polynomial coefficients with $a_n = 1$ were computed as follows:

$$\begin{cases} a_{n-(2k-1)} := 0 \\ a_{n-2k} := -\sum_{2j+1}^{k} a_{n-2(k-j)} \end{cases} \tag{6}$$

where $k = 1, 2, \ldots, \lfloor n/2 \rfloor$.

Table 6 presents the benchmark results obtained by applying the DK method with DD, TD, and QD precision arithmetic to the Chebyshev integration problem, as described previously.

Table 6. Benchmark test (unit: second) for the Chebyshev integration problem ($n = 64$)

Snapdragon	DD	TD	QD	EPYC	DD	TD	QD
NoBF	0.0528	0.174	0.59	NoBF	0.0627	0.306	0.989
NoBF Neon	0.0381	0.172	0.47	NoBF AVX2	0.0431	0.251	0.608
BF	—	0.14	0.383	BF	—	0.214	0.628
BF Neon	—	0.111	0.336	BF AVX2	—	0.156	0.415

Table 6 lists the benchmark results for the Chebyshev integration problem ($n = 64$). The results with and without SIMD vectorization—Neon on the Snapdragon platform and AVX2 on the EPYC platform—are listed side-by-side (NoBF Neon/AVX2 and BF Neon/AVX2), and the effects of BF conversion (BF) are compared for TD and QD precisions. The effect of BF conversion is pronounced for TD and QD; on the EPYC, BF achieves an approximately 1.43× speedup for TD and 1.57× for QD relative to NoBF, and even when combined with SIMD (NoBF AVX2→BF AVX2), speedups of approximately 1.61× for TD and 1.47× for QD are obtained. We can observe similar trends for the Snapdragon, where the combination of BF conversion and Neon vectorization is effective for TD and QD; in particular, TD achieves an approximately 1.55× speedup in the NoBF Neon→BF Neon transition. These results confirm that the combined use of BF conversion and SIMD vectorization is effective for multi-component multiple-precision arithmetic with TD and QD precisions.

5 Conclusion and Future Work

In this study, we applied BF algorithms that suppress conditional branching to multicomponent multiple-precision arithmetic (DD/TD/QD) and evaluated their performance on CPU SIMD (AVX2 on EPYC) and Arm Neon. Benchmark tests were conducted for real and complex square matrix multiplications, real- and complex-valued evaluations of real-coefficient polynomials, and the Chebyshev integration problem. The results confirm that, for high precision (particularly TD and QD), the combined use of BF conversion and SIMD vectorization is effective and reduces the computation time compared with the standard implementation (NoBF). In addition, the results validate that BF conversion offers no benefit to DD precision arithmetic. These findings indicate that achieving performance improvements in multicomponent arithmetic requires (i) branch elimination (BF conversion) and (ii) the exploitation of instruction-level parallelism (SIMD vectorization) to be combined with an implementation strategy selected according to the target problem, precision level, and architecture.

The following directions are identified for future research:

1. **Adaptive implementation selection (autotuning)**: As the optimal execution path (NoBF/BF, SIMD width, and loop partitioning) varies with the precision level (DD/TD/QD) and problem size, we will introduce a mechanism for selecting the optimal implementation at either runtime or build time.
2. **Support for wider SIMD extensions and instruction sets**: Extensions to AVX-512 and Arm SVE/SVE2 will be pursued, along with establishing design guidelines for data layout and reduction operations as register width increases.
3. **Memory access optimization and enhanced parallelization**: In addition to improving cache efficiency in matrix operations and polynomial evaluation, we will optimize the coordination of thread-level parallelism via OpenMP with SIMD vectorization, including NUMA placement and scheduling.
4. **Extension to application problems**: The scope of the application will be expanded to include high-precision linear algebra problems (QR decomposition, eigenvalue problems, and iterative methods), special function evaluation, numerical integration, and optimization, and the effectiveness in practical settings (convergence behavior, total runtime, and energy efficiency) will be evaluated.

Through these efforts, we aim to realize accelerated multicomponent high-precision arithmetic in a more general and portable form by establishing a high-precision numerical computing infrastructure applicable to a wide range of computational environments, ranging from mobile devices to supercomputers.

Acknowledgment. This research was supported by JSPS KAKENHI (grant number 23K11127).

References

1. Bailey, D.H.: QD. https://www.davidhbailey.com/dhbsoftware/
2. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**(3), 224–242 (1971)
3. Enge, A., Théveny, P., Zimmermann, P.: MPC. http://www.multiprecision.org/mpc/
4. Fabiano, N., Muller, J.M., Picot, J.: Algorithms for triple-word arithmetic. IEEE Trans. Comput. **68**, 1573–1583 (2019)
5. Granlaud, T., GMP development team. The GNU Multiple Precision arithmetic library. https://gmplib.org/
6. 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., Waś Niewski, J., eds., Parallel Processing and Applied Mathematics, pp. 622–631, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg (2014)
7. Kotakemori, T., Fujii, S., Hasegawa, H., Nishida, A.: Lis: Library of iterative solvers for linear systems. https://www.ssisc.org/lis/
8. Kouya, T.: BNCmatmul. https://github.com/tkouya/bncmatmul
9. Kouya, T.: Acceleration of multiple precision matrix multiplication based on multi-component floating-point arithmetic using avx2. In: Gervasi, O., et al., eds., Computational Science and Its Applications – ICCSA 2021, pp. 202–217, Cham (2021). Springer International Publishing
10. Kouya, T.: Performance evaluation ofÂ accelerated complex multiple-precision lu decomposition. In: Gervasi, O., et al., eds., Computational Science and Its Applications, ICCSA 2024 Workshops, pp. 3–19, Cham (2024). Springer Nature, Switzerland
11. Lange, M., Rump, S.M.: Faithfully rounded floating-point computations. ACM Trans. Math. Softw. **46**, No. 3 (July 2020)
12. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN '10, pp. 19–26, New York, NY, USA, (2010). ACM
13. MPLAPACK/MPBLAS. Multiple precision arithmetic LAPACK and BLAS. https://github.com/nakatamaho/mplapack
14. Mukunoki, D., Ozaki, K.: Sparse iterative solvers using high-precision arithmetic with quasi multi-word algorithms. In: 2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 33–40. (2025)
15. MPFR Project. The MPFR library. https://www.mpfr.org/
16. Zhang, D.K., Aiken, A.: Automatic verification ofÂ floating-point accumulation networks. In: Piskac, R., Rakamarić, Z., eds. Computer Aided Verification, pp. 215–237, Cham 2025. Springer Nature, Switzerland (2025)
17. Zhang, D.K., Aiken, A.: High-performance branch-free algorithms for extended-precision floating-point arithmetic. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '25, p. 695–710, New York, NY, USA, 2025. Association for Computing Machinery (2025)
18. Kouya, T.: Trial Approach to Accelerate Multi-component-type Multiple-precision Basic Linear Computation with Arm Neon Intrinsics (in Japanese). Technical report, HPC (2025)