Multiple Double Arithmetic on NVIDIA Tensor Cores

Howard Chen[1] and Jan Verschelde[2]

[1] University of Illinois at Chicago, Department of Mathematics, Statistics, and Computer Science, 851 S. Morgan St. (m/c 249), Chicago, IL 60607-7045, USA
hchen221@uic.edu

[2] University of Illinois at Chicago, Department of Mathematics, Statistics, and Computer Science, 851 S. Morgan St. (m/c 249), Chicago, IL 60607-7045, USA
janv@uic.edu https://www.math.uic.edu/∼jan

Abstract. A multiple double is an unevaluated sum of doubles. An NVIDIA tensor core is a specialized high-performance compute core for matrix multiplication. The Ampere A100, released in 2020, introduced tensor cores capable of 64-bit floating-point arithmetic. Every multiple double arithmetical operation requires renormalization, which involves branching, for which tensor cores are unsuited.
To solve this problem caused by renormalization, we apply a solution similar to the Ozaki scheme [Ozaki et al, Numerical Algorithms, 2012]. Our software is available under the GPU GPL license on github.

Keywords: arithmetic · multiple double · NVIDIA tensor core.

1 Introduction

An algorithm is *robust* if it does not fail for small perturbations of degenerate inputs. With multiple double arithmetic we can obtain more accuracy than what can be provided with 64-bit doubles.

Definition 1 (multiple double). *A* multiple double *is an unevaluated sum of nonoverlapping doubles.*

The algorithms for multiple double arithmetic originated in the late sixties, in efforts [2] to extend the accuracy of the then commonly used 32-bit floating-point arithmetic. In [3] and [15], algorithms are defined for quad double arithmetic, and also described in [10]. QDlib [3] and CAMPARY [5] are two software packages for quad double and general multiple double arithmetic respectively.

If the result of a computation can be represented exactly by a 64-bit double, then the next double in the sequence of multiple doubles represents the working precision in which the computation was executed. This property justifies the term error-free transformations [14] when referring to multiple double arithmetic. The drawback of working with multiple double arithmetic is its cost overhead. For instance, in order to compensate for the cost of quad double arithmetic, one needs to run computations with teraflop performance.

In order to offset the cost of multiple double algorithm, we examine the application of Graphics Processing Units (GPUs), and in particular in this paper, NVIDIA Tensor Cores.

Definition 2 (NVIDIA Tensor Core). *An* NVIDIA Tensor Core *is a specialized high-performance compute core for matrix multiplication.*

Introduced in 2020, the NVIDIA "Ampere" A100 Tensor Cores offer IEEE-compliant 64-bit floating-point (FP64) tensor core instructions [12]. An appealing aspect of tensor cores are their increased performance, relative to the regular cores. In particular, the theoretical peak performance of the FP64 (non-tensor) cores is 9.7 TFLOPS (teraflops), whereas 19.5 TFLOPS is the theoretical peak performance of the FP64 tensor cores.

Then our question becomes: *Are tensor cores useful for multiple double arithmetic?* At first sight, the answer is simply no because of the following.

Problem 1 (renormalization). Multiple double arithmetic requires a renormalization after every arithmetical operation. This renormalization involves branching and tensor cores are specialized for matrix multiplication.

Our solution is similar to the application of the Ozaki scheme [13], used in mixed precision computations [1], [4], [6], [9]. In mixed precision, accurate results are computed via lower precision arithmetic. Multiple double arithmetic is similar, using double precision arithmetic to multiple the accuracy of the results. Double precision arithmetic has the benefit of a wider range of exponents compared to what is available on lower precision processors.

New algorithms for extended precision floating-point arithmetic that avoid branching were proposed in [22], available in a registered Julia package [21]. The vectorization of multiple double arithmetic is achieved via branch-free algorithms, presented in [7] for double double, triple double, and quad double addition and multiplication, with comparisons made to MPLAPACK [8]. While related to our work, those branch-free algorithms are not suitable for execution by tensor cores.

The next section introduces the rewriting of a product of two double double matrices as a product of two double matrices, followed by a description of balancing algorithms to ensure double double accuracy. The algorithms work for general multiple double arithmetic. Section 4 presents our software and experimental results.

2 Rewriting Products of Double Double Matrices

The fraction `f` of a double x has 52 bits, and we can distribute the bits into 4 parts `f0`, `f1`, `f2`, `f3`, each of 13 bits:

```
 f : 1001 0001 1010 0110 0110 0100 1110 1101 0010 0011 0100 1100 1100
f0 : 1001 0001 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
f1 : 0000 0000 0000 0110 0110 0100 1100 0000 0000 0000 0000 0000 0000
f2 : 0000 0000 0000 0000 0000 0000 0010 1101 0010 0010 0000 0000 0000
f3 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0100 1100 1100
```

Then we compute with 4 doubles `x0`, `x1`, `x2`, `x3`, we call them *quarters*, with exponents `0`, `-13`, `-26`, `-39`.

```
x0 : 1 0010 0011 0100 0000000000000000000000000000000000000000000000000    0
x1 : 1 1001 1001 0011 0000000000000000000000000000000000000000000000000  -13
x2 : 1 0110 1001 0001 0000000000000000000000000000000000000000000000000  -26
x3 : 1 0100 1100 1100 0000000000000000000000000000000000000000000000000  -39
```

The zero bits at the end allow for error free computations:

- Multiplying two quarters leaves at least 26 zero bits at the end.
- Long sequences of inner products can be made *without losing any bits.*

If we focus on the four quarters of the leading doubles:

$$a_{i,k} = (a_{i,k,0}, a_{i,k,1}, a_{i,k,2}, a_{i,k,3}), \quad b_{k,j} = (b_{k,j,0}, b_{k,j,1}, b_{k,j,2}, b_{k,j,3}), \tag{1}$$

then we can map the elements of matrices $A_{i,k}$ and $B_{j,k}$ as

$$A_{i,k} = \begin{bmatrix} a_{i,k,0} \\ a_{i,k,1} \\ a_{i,k,2} \\ a_{i,k,3} \end{bmatrix}^T, \quad B_{k,j} = \begin{bmatrix} b_{k,j,0} & b_{k,j,1} & b_{k,j,2} & b_{k,j,3} \\ 0 & b_{k,j,0} & b_{k,j,1} & b_{k,j,2} \\ 0 & 0 & b_{k,j,0} & b_{k,j,1} \\ 0 & 0 & 0 & b_{k,j,0} \end{bmatrix}, \tag{2}$$

contributing to the product

$$C_{i,j} = \begin{bmatrix} a_{i,k,0}b_{k,j,0} \\ a_{i,k,1}b_{k,j,0} + a_{i,k,0}b_{k,j,1} \\ a_{i,k,2}b_{k,j,0} + a_{i,k,1}b_{k,j,1} + a_{i,k,0}b_{k,j,2} \\ a_{i,k,3}b_{k,j,0} + a_{i,k,2}b_{k,j,1} + a_{i,k,1}b_{k,j,2} + a_{i,k,0}b_{k,j,3} \end{bmatrix}. \tag{3}$$

In (3), observe the convolutions with the sum of the last two indices in the product of two numbers equal to the index of the quarter.

A double double matrix, represented by two matrices $(A_{\rm hi}, A_{\rm lo})$ of high and low doubles, is mapped into a sequence of 8 matrices (A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7), where the first four matrices are the quarters of the high doubles, and the last four matrices collects the quarters of the low doubles. Similarly for the second double double matrix $(B_{\rm hi}, B_{\rm lo})$, the quarters are mapped into a sequence of 8 matrices (B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7). Then the product of two double double matrices is rewritten as the product of the matrices

$$\overline{A} := \begin{bmatrix} A_0 & A_1 & \cdots & A_7 \end{bmatrix} \quad \text{and} \quad \overline{B} := \begin{bmatrix} B_0 & B_1 & \cdots & B_7 \\ & B_0 & \cdots & B_6 \\ & & \ddots & \vdots \\ & & & B_0 \end{bmatrix} \tag{4}$$

which then leads to the product $\overline{C} := \begin{bmatrix} C_1 & C_2 & \ldots & C_8 \end{bmatrix}$, computed as $\overline{A} \cdot \overline{B}$ by the tensor cores in double precision.

Note that upon quartering, the size of the data increases by 4 times. For example, a quartered double double $n \times n$ matrix has 8 times the size of a regular $n \times n$ matrix of doubles, and a quartered quad double matrix has 16 times the size. Instead of stacking quartered matrices into $\overline{A}$ and convoluting quartered matrices into $\overline{B}$, the first matrix could be obtained from convoluting quartered matrices and the second matrix could be a stacking of quartered matrices.

In the overview of the algorithm to multiply two double double matrices on tensor cores, we list the following steps:

1. Split all double doubles in 8 parts.
2. Rewrite into one single matrix product of double matrices.
3. Run the matrix multiplication on the tensor cores.
4. Extract the parts from the product.
5. Add the parts of the product into a double double matrix.

The last step must be done in double double arithmetic, summing 8 doubles into one double double for each element of the product. This double double summation is massively parallel and can be executed on regular cores.

The main question then remains: *Is the result accurate?*

We call a sequence of positive doubles *balanced* if their exponents lie on an equally spaced grid, for example: 0, -13, -26, -29, etc. The inner product of two balanced sequences then reduces to a sum of doubles with fractions that have only their first 26 bits nonzero, or equivalently, with 26 trailing zero bits. Sums of doubles with 26 nonzero bit fractions are computed exactly as long as there is no overflow, that is: if the sum fits in the 52-bit size fraction of a double. If N is the size of such sum, then how large can N be before overflow occurs? Consider:

$$\underbrace{N \left(\sum_{i=0}^{25} 2^i \right)}_{B} \leq 2^{52}, \tag{5}$$

where B is the largest number with 26 leading nonzero bits. Evaluating the inequality (5) yields 67,108,865 as the upper bound for N.

The next section addresses the case when the doubles are not balanced.

3 Balancing Algorithms

We have to group the numbers by sign and size, keeping in mind the loss of accuracy in floating-point arithmetic.

Problem 2. Subtracting floating-point numbers of the same magnitude may lead to losing many significant bits in the fraction.

Therefore, group elements by sign as follows:

- let A_+ contain all positive numbers of A, and
- let A_- contain all negative numbers of A,

so $A = A_+ + A_-$, and likewise $B = B_+ + B_-$, then

$$AB = (A_+ + A_-)(B_+ + B_-) = A_+B_+ + A_+B_- + A_-B_+ + A_-B_-. \tag{6}$$

Problem 3. Adding numbers with exponents of different magnitude may lead to losing many significant bits in the fraction.

To solve this problem, write A as $A_0 + A_1 + \cdots + A_{12}$, where the index is the exponent of the leading quarter of the double, corresponding to the 13 possible exponents of the leading quarters in the distribution of the doubles of A. Do likewise for B and execute many matrix multiplications.

Problem 4. A double double x is represented by a high double x_h, and a low double x_ℓ. The problem is then that x_h and x_ℓ may have opposite signs.

To solve this problem, assuming $x_h > 0$ and the exponent of x_h is zero, do:

$$b := 1 \times 2^{-52}; \quad x_h := x_h - b; \quad x_\ell := x_\ell + b. \tag{7}$$

Obviously, the value $x_h + x_\ell$ of the double double does not change by these steps. Observe the following. As the original $x_\ell < 0$, and in particular, as $x_\ell < -1 \times 2^{-52}$, we have $x_\ell + b > 0$, and thus the new $x_\ell > 0$. As x_ℓ grows in size, we may loose the last bit of x_ℓ. Therefore, in a multiple double, the last bit should then be added to the next double in the sequence of doubles representing the multiple double, using multiple double arithmetic.

Problem 5. If we quarter a double x in four parts: x_0, x_1, x_2, and x_3, we expect the exponents to be 0, -13, -26, and -39, respectively. The problem is then that x_1, x_2, and x_3 may have many leading zeros in their fractions. Therefore, their exponents may be much less than -13, -26, and -39, in the worst case: -25, -38, and -51, causing great loss of accuracy.

To solve this problem, if the exponents of x_0 and x_1 differ by more than 13, assuming $x_0 > 0$, and assuming x_0 belongs to A_0, i.e.: its exponent is zero, do:

$$b := 1 \times 2^{-14}; \quad x_0 := x_0 - b; \quad x_1 := x_1 + b. \tag{8}$$

Again obviously, the value of $x_0 + x_1$ remains invariant by these steps. Observe the following. Subtracting b does not alter the exponent of x_0. As the exponent of x_1 is less than -13, $x_1 + b$ has exponent -13, and thus the exponents of x_0 and x_1 are 0 and -13 respectively. To ensure that x_1 still has many trailing zero bits, the bits in the fraction after the 13-th position have to be added to the next quarter x_2, which may be beneficial if the exponent of x_2 is less than -26. Similar steps and arguments apply then to adjust x_2 and x_3 so they have the desired exponents of -26 and -39 respectively.

A special case occurs when one double in the sequence representing a multiple double is equal to zero. In that case, we assume the unnormalized representation of zero with the desired exponent, and just do nothing. In any case, when the exponents of the doubles lie on a grid, the argument explained at the end of the previous section applies and very large inner products are computed accurately.

4 Multiple Double Matrix Multiplication

In our software, we applied a top down and a bottom up methodology. The bottom up method does not rely on any available code for matrix multiplication on tensor cores, whereas the top down method started from adjusting the `dmmaTensorCoreGemm` from the CUDA samples [11].

4.1 Hardware and Software

We used the CUDA Software Development Kit, `nvcc` version 12.4, on two different NVIDIA GPUs, in computers with specifications listed below:

1. Microway GPU Server with NVIDIA Ampere A100 80GB GPU, with two Intel Xeon 5318Y 2.1 GHz 24-core CPUs and 256 GB internal memory, running Rocky Linux 9.3. The host code is compiled with `g++` 11.4.1. Using shared memory, our modified `dmmaTensorCoreGemm` reached 13.75 FP64 TFLOPS.
2. NVIDIA RTX 4080 housed in 2023 Alienware gaming laptop, with a 24-core Intel i9-13900 HX at 2.2GHz and 32 GB internal memory, running Windows 11 and the 2022 Community Visual Studio version. The FP64 Tensor Cores on RTX 4080 gave a performance of 0.50 TFLOPS when running the modified `dmmaTensorCoreGemm`.

Our software is available under the GPU GPL license, at `https://github.com/`

- `janverschelde/PHCpack/src/GPU/TensorCores`
- `hchen221/multiple-double-arithmetic-matrix-matrix-multiplication`

The PHCpack source code [16] contains the codes for the multiple double arithmetic, taken from QDlib [3] for double doubles and quad doubles and generated with CAMPARY [5] for octo doubles and hexa doubles, applied in [17], [18], and [19]. The first experiments with vectorized multiple double arithmetic in PHCpack were reported in [20]. The `TensorCores` folder implements the top down method, where as the other software applies the bottom up method.

In all experiments, random numbers were generated.

4.2 Bottom Up Method

Matrix products were computed using 2 methods. For the first method, using `dmmaTensorCoreGemm` as a reference, we defined a WMMA kernel[3] for a matrix matrix multiplication which was executed on $\overline{C} = \overline{A} \cdot \overline{B}$, as defined in (4), then extracted the parts from $\overline{C}$ and merged them into a matrix of double doubles C. For the second method, we computed the matrix product $A \cdot B$ of double doubles, where a single thread executes an inner product computation. Addition and multiplication operations within the inner product computations were done using double double arithmetic.

Denote C_{TC} as the result from the Tensor Core computation and C_{CUDA} as the result from the regular CUDA core computation. Let the maximum error be

$$\epsilon_{\max} := \max_{i,j \in [n]^2} \left| C_{\mathrm{TC}}[i,j] - C_{\mathrm{CUDA}}[i,j] \right| \tag{9}$$

We also tracked the amount of time it takes each method for different values of n. The time for the Tensor Cores t_{TC} was assessed as the amount of time it takes to construct $\overline{A}$ and $\overline{B}$, execute the Tensor Core kernel (a.k.a. WMMA kernel), and to assemble the result into multiple doubles. The time for the CUDA kernel t_{CUDA} was assessed as the amount of time it takes to execute a the CUDA kernel alone. For the sake of a direct comparison of matrix multiplication execution, we also tracked the time to execute the Tensor Core kernel alone t_{WMMA}. Results from sample runs for double doubles and quad doubles are listed in Table 1.

[3] WMMA abbreviates Warp Matrix Multiply Accumulate.

Table 1. Units of all times t_{WMMA}, t_{TC}, t_{CUDA} are seconds: t_{WMMA} measures the execution of a single WMMA kernel, while t_{TC} is the time to make $\overline{A}$ and $\overline{B}$, execute the WMMA kernel, and combine the result into a multiple double; t_{CUDA} is the time of a multiple double matrix multiplication with a regular CUDA kernel. Corresponding to the times are the flops f_{WMMA} and f_{CUDA}, in teraflops units.

	double double				quad double	
n	1024	2048	3072	4096	1024	2048
t_{WMMA}	0.000108	0.000106	0.000106	0.000257	0.000118	0.00011
t_{TC}	0.379792	1.19952	3.16265	7.93621	0.752966	4.23463
t_{CUDA}	0.096794	0.450174	1.19599	2.33072	0.285909	1.57585
f_{WMMA}	4.6	2.7	2.4	1.7	3.6	1.8
f_{CUDA}	2.8	2.9	2.8	2.9	3.7	3.9
$\epsilon_{\max}$	3.5E−28	8.7E−28	1.6E−27	2.4E−27	2.0E−59	4.1E−59

The execution of the WMMA kernel alone (t_{WMMA}) is significantly shorter than that of a regular CUDA core (t_{CUDA}), but the full process involving the usage of Tensor Cores (t_{TC}) takes significantly longer than the CUDA cores. The extent to which t_{TC} increases with respect to n is greater than that of t_{CUDA}, due to the size of $\overline{A}, \overline{B}$. The arithmetic intensity (#operations per data unit) of multiple double arithmetic benefits regular CUDA kernels, whereas matrix multiplication in double precision is memory bound, rather than compute bound.

4.3 Top Down Method

The top down method applies the `dmmaTensorCoreGemm` of the CUDA software development kit on the product of double double matrices, rewritten as a product of double matrices which contain the parts of the doubles of the input matrices. In the top down method, the dimensions of the `dmmaTensorCoreGemm` are 8192-by-4096 for the first matrix and 4096-by-8192 for the second matrix. Table 2 lists the dimensions adapted for higher precisions.

Table 2. Dimensions of the matrix A of `dmmaTensorCoreGemm` adapted for double double (dd), quad double (qd), octo double (od), and hexa double (hd).

	d	dd	qd	od	hd
#rows of A	8192	1024	512	256	128
#columns of A	4096	512	256	128	64

A 1024-by-512 double double matrix is rewritten into the first 8192-by-4096 matrix of double numbers in the input of `dmmaTensorCoreGemm`. The second 4096-by-8192 input matrix is obtained by rewriting a 512-by-8192 double double matrix. Each time the precision level doubles, the dimensions of the multiple double matrices are cut in half, as illustrated by Table 2.

With the version of `dmmaTensorCoreGemm` which uses shared memory, a performance of 13.75 TFLOPS is achieved, accomplishing double double accuracy. For a comparison: $13.75 > 9.7$; the theoretical peak performance of the regular CUDA cores on the NVIDIA A100 GPU is 9.7 TFLOPS with FP64 arithmetic.

5 Conclusions

To apply FP64 tensor cores to multiply multiple double matrices, the doubles in the matrices are partitioned introducing trailing zeros in the fractions of the doubles. Grouping matrices by signs and exponents, balancing algorithms redistribute bits in the parts of the fractions to ensure

the exponents of the parts remain on grid. Our free and open source software demonstrates the acceleration of multiple double arithmetic with FP64 tensor cores.

Future work will apply this multiple double tensor core matrix multiplication to accelerate the blocked Householder QR, extending [18], with the application to compute Taylor series [19].

References

1. Abdelfattah, A., Anzt, H., Boman, E.G., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., Higham, N.J., Li, X.S., Loe, J., Luszczek, P., Pranesh, S., Rajamanickam, S., Ribizel, T., Smith, B.F., Swirydowicz, K., Thomas, S., Tomov, S., Tsai, Y.M., Yang, U.M.: A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. International Journal of High Performance Computing Applications **35**(4), 344–369 (2021)
2. Dekker, T.J.: A floating-point technique for extending the available precision. Numerische Mathematik **18**(3), 224–242 (1971)
3. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: 15th IEEE Symposium on Computer Arithmetic (Arith-15 2001). pp. 155–162. IEEE Computer Society (2001)
4. Higham, N.J., Mary, T.: Mixed precision algorithms in numerical linear algebra. Acta Numerica pp. 347–414 (2022)
5. Joldes, M., Muller, J.M., Popescu, V., W., T.: CAMPARY: Cuda Multiple precision arithmetic library and applications. In: Mathematical Software – ICMS 2016, the 5th International Conference on Mathematical Software. pp. 232–240. Springer-Verlag (2016)
6. Kashi, A., Lu, H., Brewer, W., Rogers, D., Matheson, M., Shankar, M., Wang, F.: Mixed-precision numerics in scientific applications: survey and perspectives. The Journal of Supercomputing **82**(287), 1–67 (2026)
7. Kouya, T.: Acceleration of multicomponent multiple-precision arithmetic with branch-free algorithms and SIMD vectorization, `arXiv:2603.14926v2 [cs.MS] 7 May 2026`
8. Maho, N.: MPLAPACK version 2.0.1. user manual, `arXiv:2109.13406v2 [cs.MS] 12 Sep 2022`
9. Mukunoki, D., Ozaki, K., Ogita, T., Imamura, T.: DGEMM using tensor cores, and its accurate and reproducible versions. In: Adayappan, P., Chamberlain, B.L., Juckeland, G., Ltaief, H. (eds.) High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science, vol. 12151, pp. 230–248. Springer-Verlag (2020)
10. Muller, J.M., Brunie, N., de Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic. Springer-Verlag, second edn. (2018)
11. NVIDIA: CUDA Samples, at `https://github.com/NVIDIA/cuda-samples`.
12. NVIDIA: NVIDIA A100 Tensor Core GPU Architecture (2020), whitepaper available via `https://www.nvidia.com`.
13. Ozaki, K., Ogita, T., Oishi, S., Rump, S.M.: Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. Numerical Algorithms **59**, 95–118 (2012)
14. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica **19**, 287–449 (2010)
15. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. **18**(3), 305–363 (1997)
16. Verschelde, J.: Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. ACM Trans. Math. Softw. **25**(2), 251–276 (1999), available at `https://github.com/janverschelde/PHCpack`
17. Verschelde, J.: Accelerated polynomial evaluation and differentiation at power series in multiple double precision. In: The 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 740–749. IEEE (2021)
18. Verschelde, J.: Least squares on GPUs in multiple double precision. In: The 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 828–837. IEEE (2022)
19. Verschelde, J.: GPU accelerated Newton for Taylor series solutions of polynomial homotopies in multiple double precision. In: Boulier, F., Mou, C., Sadykov, T.M., Vorozhtsov, E.V. (eds.) Proceedings of the 26th International Workshop on Computer Algebra in Scientific Computing (CASC 2024). Lecture Notes in Computer Science, vol. 14938, pp. 328–348. Springer-Verlag (2024)
20. Verschelde, J.: Multiword arithmetic and parallel computing. ACM SIGAda Ada Letters **45**(2), 67–68 (2025)

21. Zhang, D.K.: MultiFloats.jl, `https://github.com/dzhang314/MultiFloats.jl`
22. 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. pp. 695–710. ACM (2025)