High-Performance Branch-Free Algorithms for Extended-Precision Floating-Point Arithmetic

David K. Zhang
dkzhang@stanford.edu
Stanford University
Stanford, CA, USA

Alex Aiken
aiken@cs.stanford.edu
Stanford University
Stanford, CA, USA

Abstract

We present new branch-free algorithms for floating-point arithmetic at double, triple, or quadruple the native machine precision. These algorithms are the fastest known by at least an order of magnitude and are conjectured to be optimal, not only in an asymptotic sense, but in their exact FLOP count and circuit depth. Unlike previous algorithms, which either use complex branching logic or are only correct on specific classes of inputs, our algorithms have computer-verified proofs of correctness for all floating-point inputs within machine overflow and underflow thresholds. Compared to state-of-the-art multiprecision libraries, our algorithms achieve up to 11.7× the peak performance of QD, 34.4× over CAMPARY, 35.6× over MPFR, and 41.4× over FLINT.

CCS Concepts

• **Mathematics of computing** → **Numerical analysis**; **Arbitrary-precision arithmetic**; • **Theory of computation** → **Numeric approximation algorithms**; **Vector / streaming algorithms**; *Simulated annealing*; *Network optimization*; • **Computing methodologies** → *Theorem proving algorithms.*

Keywords

Floating-Point Arithmetic, Floating-Point Expansions, Error-Free Transformations, Branch-Free Algorithms, Data-Parallel Algorithms

ACM Reference Format:
David K. Zhang and Alex Aiken. 2025. High-Performance Branch-Free Algorithms for Extended-Precision Floating-Point Arithmetic. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3712285.3759876

1 Introduction

Double-precision floating-point arithmetic is becoming insufficient for a growing number of applications in computational science and engineering. Modern exascale supercomputers routinely solve systems with billions of unknowns, a scale at which even mathematically well-posed problems can exhibit condition numbers on the order of $\kappa \approx 10^{10}$ to 10^{20}. In this regime, the relative error $\kappa\epsilon_{\text{double}}$ of a solution computed in double precision (with machine epsilon $\epsilon_{\text{double}} = 2^{-53} \approx 10^{-16}$) can easily reach or exceed 100%, producing unacceptably inaccurate, unstable, and non-reproducible results. These issues have already been observed in a variety of fields, including climate modeling [18], energy grid optimization [54], lattice quantum chromodynamics [1], nonlinear dynamical systems [13], and full-genome metabolic modeling [36], leading to repeated calls for the development and adoption of extended-precision arithmetic in high-performance scientific software [2, 20, 35, 43].

Despite this growing need for improved numerical accuracy, robustness, and reproducibility, existing methods for extended-precision arithmetic are rarely employed in demanding computational workloads because they are orders of magnitude slower than native machine arithmetic. Indeed, extended-precision arithmetic algorithms often involve complex branching logic and dynamic memory allocation [14, 15], causing poor performance on modern data-parallel (SIMD/SIMT) processors.

These performance issues can be addressed by a class of algorithms that we call *floating-point accumulation networks* (*FPANs*), which perform extended-precision floating-point summation without branching or dynamic allocation. The ideas underlying FPANs have been known since the 1960s [10, 31, 42], but despite their long history and excellent performance characteristics, the development and adoption of FPANs have been limited by the fact that their rounding errors are notoriously difficult to analyze.

FPANs exhibit different rounding error patterns for every permutation of the signs and magnitudes of their inputs. Proving the general correctness of an FPAN for all inputs therefore requires tedious, error-prone analysis of an exponential number of cases [28], and a mistake in even one of these cases can introduce catastrophic loss-of-precision bugs [19, 29, 33]. In fact, on multiple occasions, subtle errors in these long case analyses have led to the publication of incorrect algorithms and faulty error bounds [34, 41].

To address these difficulties, the present authors recently introduced a technique to automate the rounding error analysis of FPANs using SMT solvers [53]. By reducing the analysis of thousands of rounding error patterns to an efficient computer-checkable form, our technique enables the development of FPANs with rigorous correctness guarantees and error bounds that provably hold for all inputs. In fact, the availability of an efficient verification procedure allows us to perform systematic computer search over the space of all FPANs to identify the most efficient and precise algorithms possible within this class.

In this paper, we present novel FPAN-based algorithms, discovered by our search and verification procedure, that are the fastest known for extended-precision floating-point arithmetic in software.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759876

Our algorithms extend FPANs beyond addition to implement subtraction, multiplication, division, and square root in a branch-free data-parallel fashion. They operate on *floating-point expansions* that represent high-precision numbers as sequences of two, three, or four machine-precision numbers, extending the effective precision to double, triple, or quadruple the native machine precision. Thus, on a processor that natively supports double precision, our algorithms provide fast, branch-free arithmetic operations at quadruple, sextuple, or octuple precision.

In addition to having formally-verified precision guarantees, our algorithms outperform state-of-the-art multiprecision libraries by a factor of 11.7×–69.3×. In fact, we conjecture our algorithms to be optimal, not up to an arbitrary constant factor, but in their exact operation count and circuit depth.

In summary, this paper makes the following contributions:

- We state high-performance branch-free algorithms that use FPANs to perform extended-precision floating-point addition, subtraction, multiplication, division, and square root.
- We present six particular FPANs with formally-verified error bounds for addition and multiplication of floating-point expansions with two, three, and four terms.
- We demonstrate that our algorithms significantly outperform all existing multiprecision arithmetic libraries in benchmarks of extended-precision BLAS kernels.

2 Background

2.1 Floating-Point Preliminaries

We assume familiarity with the basic notions of binary floating-point arithmetic, referring to IEEE 754 [21] and the Handbook of Floating-Point Arithmetic [40] for details. Given a processor that implements binary floating-point arithmetic at a fixed *machine precision* p, we develop algorithms for floating-point arithmetic at *extended precisions* roughly equal to $2p$, $3p$, and $4p$. Although we use $p = 53$ (IEEE double precision) in our benchmarks, all algorithms presented in this paper also work for other values of p.

Following Knuth's convention [31], we distinguish exact infinite-precision mathematical operations from rounded floating-point operations using the circled operators $\oplus, \ominus, \otimes, \oslash$, and $\sqrt[\circ]{}$. As is usual in studies of extended-precision arithmetic [4, 5, 28], we assume that all floating-point operations are executed with the roundTiesToEven rounding-direction attribute (i.e., rounded to nearest with even tiebreaking), which we denote by $\mathsf{RNE}_p(x)$. Thus, we can write:

$$x \oplus y \coloneqq \mathsf{RNE}_p(x + y) \tag{1}$$

$$x \ominus y \coloneqq \mathsf{RNE}_p(x - y) \tag{2}$$

$$x \otimes y \coloneqq \mathsf{RNE}_p(xy) \tag{3}$$

$$x \oslash y \coloneqq \mathsf{RNE}_p(x/y) \tag{4}$$

$$\sqrt[\circ]{x} \coloneqq \mathsf{RNE}_p(\sqrt{x}) \tag{5}$$

whenever the argument of $\mathsf{RNE}_p(z)$ lies within the normalized underflow and overflow thresholds $2^{e_{\min}} \le |z| \le 2^{e_{\max}}$.

For simplicity of exposition, we do not distinguish between positive and negative zero in this paper and assume that all floating-point numbers are finite and normalized or zero. No loss of generality is incurred by ignoring subnormal numbers because our underlying formal verification technique transparently handles subnormal values [53]. See Section 4.4 for further discussion of signed zero, infinity, and NaN semantics.

2.2 Approaches to Extended Precision

Several approaches to computer arithmetic beyond double precision have been considered in prior work.

Hardware implementation. Hardware support for floating-point formats beyond double precision is extremely limited among current high-performance computer architectures. To our knowledge, only IBM POWER9 CPUs include hardware support for quaduple precision, and no processors have ever featured hardware support[1] beyond quadruple precision. Intel and AMD x86 CPUs support an 80-bit extended floating-point format, but this is retained only for backward compatibility with the x87 FPU and is not intended for use in modern high-performance applications. Execution of x87 instructions is not pipelined and carries a significant performance penalty for switching in and out of legacy mode.

Notably, extended-precision floating-point formats are completely absent from GPUs, which provide the bulk of computational horsepower in modern heterogeneous supercomputers.

FPGAs. FPGA implementation of quadruple-precision arithmetic has been studied in prior work [7, 23–25], but supercomputer adoption of FPGAs remains low. None of the 100 fastest supercomputers on the November 2024 TOP500 list use FPGAs to provide any significant fraction of their computational throughput.

Software FPU emulation. In the absence of hardware support, the conventional approach to extended-precision arithmetic is to first implement big integer operations in software, then build floating-point operations using big integer operations. The most common technique, implemented in the GMP [15], MPFR [14], FLINT [17], and Boost.Multiprecision [37] libraries, is to represent big integers in base 2^{32} or 2^{64} using arrays of machine words as digits.

An alternative technique, implemented in the MPRES-BLAS library [22], is to store a big integer N as a sequence of remainders $r_i \coloneqq N \bmod m_i$ modulo pairwise coprime divisors $m_1, m_2, \ldots, m_n$. Certain arithmetic operations, including addition and multiplication, can be performed directly on this sequence of remainders, and the Chinese Remainder Theorem allows N to be uniquely reconstructed from $r_1, r_2, \ldots, r_n$ and $m_1, m_2, \ldots, m_n$.

Regardless of which big integer representation is used, implementing floating-point arithmetic on top of an integer abstraction requires sophisticated conditional logic to handle mantissa alignment, normalization, and rounding after each floating-point operation. Libraries that adopt this approach unavoidably include complex branching code that substantially degrades performance compared to native machine arithmetic.

Floating-Point Expansions. Another approach that sidesteps big integers entirely is to implement extended-precision floating-point operations directly in terms of machine-precision floating-point arithmetic. In this framework, a high-precision constant $C \in \mathbb{R}$ is

[1] Quadruple-precision floating-point arithmetic is specified as an optional extension in the PA-RISC, SPARC, and RISC-V ISAs, but to our knowledge, no hardware implementation of these architectures has ever implemented such an extension.

represented as a *floating-point expansion*, i.e., a sequence of successive machine-precision approximations of the form:

$$\begin{aligned} x_0 &\coloneqq \mathsf{RNE}_p(C) \\ x_1 &\coloneqq \mathsf{RNE}_p(C - x_0) \\ x_2 &\coloneqq \mathsf{RNE}_p(C - x_0 - x_1) \\ &\vdots \\ x_{n-1} &\coloneqq \mathsf{RNE}_p(C - x_0 - x_1 - \cdots - x_{n-2}) \\ C &\approx x_0 + x_1 + x_2 + \cdots + x_{n-1} \end{aligned} \tag{6}$$

Provided that no overflow or underflow occurs in this process, the final n-term expansion $(x_0, x_1, \ldots, x_{n-1})$ approximates C with precision $np + n - 1$.

$$|C - (x_0 + x_1 + \cdots + x_{n-1})| \leq 2^{-(np+n-1)}|C| \tag{7}$$

To achieve the full precision of $np + n - 1$ bits, a floating-point expansion must be *nonoverlapping*, i.e.,

$$|x_i| \leq \frac{1}{2}\mathsf{ulp}(x_{i-1}) \tag{8}$$

for each $i = 1, \ldots, n-1$. As illustrated in Figure 1, this property ensures that no bits of C are redundantly represented in more than one component of the expansion and allows the sign bit to provide an extra implicit bit of precision.

Floating-point expansions enable the construction of branch-free arithmetic algorithms that leverage the mantissa alignment, normalization, and rounding logic built into a processor's native floating-point hardware. However, these algorithms face a different set of challenges arising from the propagation of rounding errors and maintenance of the nonoverlapping invariant, discussed below.

2.3 Error-Free Transformations

Consider the problem of adding two floating-point expansions, $(x_0, x_1, \ldots, x_{n-1})$ and $(y_0, y_1, \ldots, y_{n-1})$. One naïve approach is to add the expansions term-by-term:

$$(x_0 \oplus y_0,\ x_1 \oplus y_1,\ \ldots,\ x_{n-1} \oplus y_{n-1}) \tag{9}$$

This strategy is completely incorrect and degrades the accuracy of result to machine precision. There are two issues at play:

- Each of the floating-point sums $x_i \oplus y_i$ is rounded, and the rounding error $(x_i + y_i) - (x_i \oplus y_i)$ must be accounted for in the subsequent term $x_{i+1} \oplus y_{i+1}$.
- If the result of $x_i \oplus y_i$ is smaller in magnitude than x_i or y_i, then it may overlap the result of $x_{i+1} \oplus y_{i+1}$. Mantissa bits must then be redistributed between these two terms in order to maintain the nonoverlapping invariant.

Both of these issues can be resolved using an ingenious class of floating-point algorithms called *error-free transformations*.

An error-free transformation simultaneously computes both a rounded floating-point operation, such as $x \oplus y$, and the exact rounding error $(x + y) - (x \oplus y)$ incurred in that operation. Miraculously, it accomplishes this task using only rounded machine-precision operations. In this paper, we will use error-free transformations for addition (TwoSum, Algorithm 1) [31, 42] and multiplication (TwoProd, Algorithm 2) [10, 49, 50] in addition to a more efficient variant of TwoSum (FastTwoSum, Algorithm 3) used in cases where the relative magnitudes of the inputs are known in advance.

Algorithm 1: $(s, e) \coloneqq \mathsf{TwoSum}(x, y)$

Input: Two floating-point numbers (x, y).
Output: Two floating-point numbers (s, e) such that $s = x \oplus y$ and $e = (x + y) - (x \oplus y)$ exactly.

1 $s \coloneqq x \oplus y$
2 $x_{\text{eff}} \coloneqq s \ominus y$
3 $y_{\text{eff}} \coloneqq s \ominus x_{\text{eff}}$
4 $\delta_x \coloneqq x \ominus x_{\text{eff}}$
5 $\delta_y \coloneqq y \ominus y_{\text{eff}}$
6 $e \coloneqq \delta_x \oplus \delta_y$
7 **return** (s, e)

Algorithm 2: $(p, e) \coloneqq \mathsf{TwoProd}(x, y)$

Input: Two floating-point numbers (x, y).
Output: Two floating-point numbers (p, e) such that $p = x \otimes y$ and $e = xy - (x \otimes y)$ exactly.

1 $p \coloneqq x \otimes y$
2 $e \coloneqq \mathsf{FMA}(x, y, -p)$
3 **return** (p, e)

Algorithm 3: $(s, e) \coloneqq \mathsf{FastTwoSum}(x, y)$

Input: Two floating-point numbers (x, y) satisfying $x = \pm 0.0$, $y = \pm 0.0$, or $\text{exponent}(x) \geq \text{exponent}(y)$.
Output: Two floating-point numbers (s, e) such that $s = x \oplus y$ and $e = (x + y) - (x \oplus y)$ exactly.

1 $s \coloneqq x \oplus y$
2 $y_{\text{eff}} \coloneqq s \ominus x$
3 $e \coloneqq y \ominus y_{\text{eff}}$
4 **return** (s, e)

Error-free transformations help to resolve the issues previously identified with addition of floating-point expansions. By computing the exact rounding error incurred by each operation, they enable rounding errors to be propagated between terms. Moreover, the correctly-rounded floating-point sum $x \oplus y$ is guaranteed not to overlap the rounding error term $(x + y) - (x \oplus y)$; otherwise, by definition, the sum would not have been correctly rounded. This property enables the use of TwoSum and FastTwoSum to maintain the nonoverlapping invariant by redistributing mantissa bits between adjacent terms.

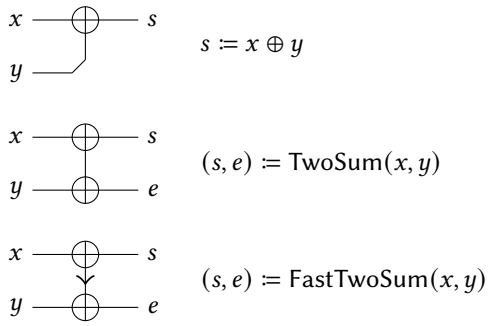
These capabilities make error-free transformations fundamental building blocks for computing and manipulating floating-point expansions. However, even with these powerful tools in hand, the development of branch-free arithmetic algorithms remains challenging. To construct such an algorithm, we must devise a single, fixed sequence of error-free transformations that correctly propagates all rounding errors while removing overlapping bits between all adjacent terms. It is straightforward to construct such a sequence for a single particular input, but it is very difficult to ensure that the same sequence works for all possible inputs. This is the challenge that motivates the study of floating-point accumulation networks.

$$
\begin{array}{rl}
\text{high-precision constant} & C = 1\,0\,1\,.\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,1\,1\,.\,.\,. \\
\text{expansion with } |x_1| > \mathsf{ulp}(x_0) & \left\{ \begin{array}{rl} x_0 = & 1\,0\,1\,.\,0\,0\,0 \\ x_1 = & \quad 0\,.\,0\,1\,1\,1\,0\,1\,1 \end{array} \right\} \text{ precision } < 2p \text{ (10 bits)} \\
\text{expansion with } |x_1| \le \mathsf{ulp}(x_0) & \left\{ \begin{array}{rl} x_0 = & 1\,0\,1\,.\,0\,1\,1 \\ x_1 = & \quad 0\,.\,0\,0\,0\,1\,0\,1\,1\,0\,1 \end{array} \right\} \text{ precision } \ge 2p \text{ (12 bits)} \\
\text{expansion with } |x_1| \le \frac{1}{2}\mathsf{ulp}(x_0) & \left\{ \begin{array}{rl} x_0 = & 1\,0\,1\,.\,1\,0\,0 \\ x_1 = & -0\,.\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1 \end{array} \right\} \text{ precision } \ge 2p+1 \text{ (13 bits)}
\end{array}
$$

Figure 1: Decomposition of a high-precision constant C into overlapping and nonoverlapping floating-point expansions with $p = 6$ mantissa bits per term. Light blue digits represent a shift stored in the exponent and are not explicitly represented in the mantissa. The final expansion rounds x_0 up instead of down, causing x_1 to be negative and the mantissa of x_1 to contains the one's complement of the corresponding bits in C. This allows the sign bit of x_1 to provide an extra implicit bit of precision.

3 Floating-Point Accumulation Networks

Floating-point accumulation networks (FPANs) are branch-free algorithms that apply a fixed sequence of floating-point sum, TwoSum, and FastTwoSum operations to a fixed number of inputs. We present FPANs using graphical diagrams consisting of horizontal *wires* and three types of vertical *gates*, representing addition, TwoSum, and FastTwoSum, respectively.



The downward-pointing arrowhead on the FastTwoSum gate serves as a mnemonic reminder that the top input, if nonzero, must be larger in magnitude than the bottom input.

A complete example of an FPAN is shown in Figure 2. To execute the algorithm specified by an FPAN diagram, the floating-point input values are placed on the wires as specified by the labels on its left-hand side. Then, the values flow left-to-right across the wires, and whenever two values (x_i, x_j) encounter a gate, their values are updated according to corresponding operation $x_i \leftarrow x_i \oplus x_j$, $(x_i, x_j) \leftarrow \mathsf{TwoSum}(x_i, x_j)$, or $(x_i, x_j) \leftarrow \mathsf{FastTwoSum}(x_i, x_j)$. In the case of an addition gate, only one output value is produced, and the rounding error incurred in that addition is said to be *discarded*. After all gates have been executed, the algorithm returns the values occupying the labeled positions on the right-hand side.

The intended operation of an FPAN is to compute a nonoverlapping floating-point expansion of the exact sum of its input values. Hence, an FPAN is required to satisfy two correctness conditions:

- The output values that an FPAN produces must be nonoverlapping for all possible input values.
- The rounding error terms that are discarded by addition gates must be small relative to the leading output term z_0. In particular, for an FPAN to have q-bit precision, the absolute value of the sum of all discarded terms must be bounded above by $2^{-q}|z_0|$.

Verifying these properties requires extensive case analysis of all possible rounding error patterns that can be created by a given sequence of sum and TwoSum operations. This combinatorial explosion of cases is challenging and tedious to analyze by hand; we refer the reader to the proof of Theorem 3.1 in Ref. [28] for an example of this phenomenon.

In Ref. [53], the present authors introduced an automated reasoning technique that efficiently verifies both types of FPAN correctness properties (nonoverlapping invariants and error bounds) using standard SMT solvers. The basic idea of this procedure is to construct an integer linear program that expresses the existence of a counterexample, i.e., an input that produces a large discarded error term or overlapping output terms. The variables of this linear program track the sign, exponent, and partial mantissa information for each floating-point value flowing through the network, while its equations and inequalities express constraints on the inputs and outputs of each gate that rule out impossible input-output pairs. We then query an SMT solver to determine whether this linear program is feasible. If it is infeasible, then no counterexample exists, which rigorously proves the desired FPAN correctness property.

This proof strategy, while highly effective, generates very complicated linear programs involving hundreds of variables and thousands of inequalities. A proof of infeasibility for such a system is an enormous computational artifact that is essentially unreadable to humans. For this reason, we are unable to include human-readable proofs of correctness for the FPANs presented in this paper. To facilitate independent verification of our results, we have made our verifier implementation freely available under a permissive open-source license [52], and we provide step-by-step instructions to reproduce our proofs in the appendices of this paper.

To guard against bugs in a particular SMT reasoning engine, our verifier works with any SMT solver that implements the SMT-LIB 2 standard interface. We have independently confirmed our

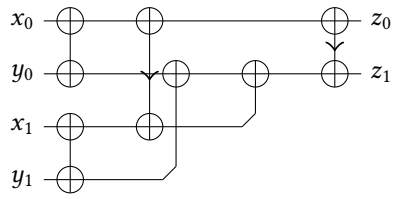


Figure 2: Provably optimal FPAN for addition of 2-term floating-point expansions. Here, (x_0, x_1) and (y_0, y_1) denote the input expansions to be added, and (z_0, z_1) denotes the output expansion. The absolute value of the sum of the discarded error terms is bounded above in magnitude by $2^{-(2p-1)}|x+y|$.

results using a variety of SMT solvers, including Z3 [9], CVC5 [3], MathSAT5 [8], Yices2 [12], OpenSMT [6], and Bitwuzla [44].

In addition to quantifying the number of bits of precision, a FPANs is also parameterized by its *size* (its total number of gates) and its *depth* (the number of gates encountered on the longest directed path from an input node to an output node). To maximize computational efficiency, it is desirable to minimize size and depth while maximizing precision.

4 Arithmetic Algorithms

With the basic notions of FPANs established, we are now prepared to state branch-free algorithms for addition, subtraction, multiplication, division, and square root of two-term, three-term, and four-term floating-point expansions.

4.1 Addition and Subtraction

Addition and subtraction are the most straightforward operations to implement using FPANs, which naturally compute high-precision sums. Given two floating-point expansions, $(x_0, x_1, \ldots, x_{n-1})$ and $(y_0, y_1, \ldots, y_{n-1})$, we construct an FPAN with $2n$ inputs

$$(x_0, \pm y_0, x_1, \pm y_1, \ldots, x_{n-1}, \pm y_{n-1}) \tag{10}$$

and n nonoverlapping outputs, with + signs chosen for addition and − signs chosen for subtraction. We assume the input expansions $(x_0, x_1, \ldots, x_{n-1})$ and $(y_0, y_1, \ldots, y_{n-1})$ to be nonoverlapping, which makes this task easier than the more general problem of accumulating $2n$ arbitrary inputs.

In Figures 2, 3, and 4, we present the smallest provably-correct FPANs we have discovered for addition of 2-term, 3-term, and 4-term floating-point expansions, with size and depth (6, 4), (14, 8), and (26, 11), respectively. These FPANs were produced by a heuristic search procedure, based on simulated annealing, in which random TwoSum gates were added to an empty FPAN until it passed the automatic verification procedure described in Ref. [53]. Then, random gates were added and removed, with the probability of removal gradually adjusted upwards over time, subject to the constraint that the resulting FPAN still pass verification.

We have proven that the two-term addition FPAN shown in Figure 2 is provably optimal by exhaustive enumeration of every FPAN with size ≤ 6 and depth ≤ 4. Every such FPAN, besides the one shown in Figure 2, either fails to produce a nonoverlapping result or computes a sum z with error $|z-(x+y)|$ strictly exceeding $2^{-(2p-1)}|x+y|$. Unfortunately, the exponential growth in the number of FPANs makes exhaustive enumeration intractable for the three-term and four-term FPANs shown in Figures 3. However, we have high confidence that these FPANs are indeed optimal based on the observation that our heuristic search procedure consistently and repeatably converges to these local minima.

One notable feature of the FPANs shown in Figures 2, 3, and 4 is that they all begin with an initial layer of TwoSum gates that pair the corresponding terms $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ between the two input expansions. Because TwoSum is a commutative operation, this structure guarantees that the sums computed by these FPANs are invariant under swapping $(x_0, x_1, \ldots, x_{n-1})$ with $(y_0, y_1, \ldots, y_{n-1})$. This is a desirable commutativity property that will be revisited in our subsequent discussion of multiplication.

4.2 Multiplication

Our strategy for multiplication with FPANs is based on the distributive property. Recall that the value represented by the floating-point expansion $(x_0, x_1, \ldots, x_{n-1})$ is the number $x \coloneqq x_0 + x_1 + \cdots + x_{n-1}$. Hence, the exact value of the product of $(x_0, x_1, \ldots, x_{n-1})$ and $(y_0, y_1, \ldots, y_{n-1})$ can be written as a sum of n^2 terms:

$$xy = x_0y_0 + x_0y_1 + x_1y_0 + \cdots + x_{n-1}y_{n-1} \tag{11}$$

Each term in this expansion is a product of two machine-precision floating-point numbers, which can be exactly computed by the TwoProd algorithm. Thus, by computing all pairwise error-free products $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$, we can write the product xy as the exact sum of the $2n^2$ machine-precision floating point numbers $p_{0,0}, p_{0,1}, p_{1,0}, \ldots, p_{n-1,n-1}$ and $e_{0,0}, e_{0,1}, e_{1,0}, \ldots, e_{n-1,n-1}$. This strategy reduces multiplication of floating-point expansions to two steps:

- an initial expansion step that executes n^2 TwoProd operations; followed by
- an accumulation step that executes an FPAN with $2n^2$ inputs.

We can significantly reduce the number of operations in both of these steps by observing that certain product terms can always be safely discarded when the inputs are nonoverlapping. Let e_x and e_y denote the exponents of x_0 and y_0, respectively. To compute an n-term floating-point expansion of the exact product $z \coloneqq xy$, which satisfies $|z| \geq 2^{e_x+e_y}$, we can safely ignore any term whose exponent falls below $e_x+e_y-n(p+1)$. The nonoverlapping invariant implies that the exponent of x_i is at most $e_x - i(p+1)$, and similarly, the exponent of y_j is at most $e_y - j(p+1)$. Hence, the exponents of $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ are at most $e_x + e_y + 1 - (i+j)(p+1)$ and $e_x + e_y + 1 - (i+j+1)(p+1)$, respectively. Thus, we can safely ignore $p_{i,j}$ whenever $i + j \geq n$ and $e_{i,j}$ whenever $i + j + 1 \geq n$. This optimization simplifies the initial expansion step from n^2 TwoProd operations to $n(n-1)/2$ TwoProd operations and n machine-precision floating-point products, thereby reducing the number of FPAN inputs from $2n^2$ to n^2.

Some multiplication algorithms proposed in prior work exhibit the notable deficiency of violating the commutative property of multiplication [28]. In other words, the product of two floating-point expansions is computed differently if the inputs are swapped. Of course, it is well-known that floating-point arithmetic violates many of the algebraic properties enjoyed by exact real arithmetic,

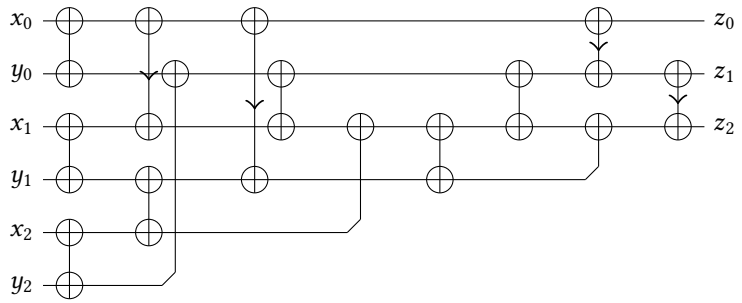


Figure 3: Conjecturally optimal FPAN with size 14 and depth 8 for addition of 3-term floating-point expansions. Here, (x_0, x_1, x_2) and (y_0, y_1, y_2) denote the input expansions to be added, and (z_0, z_1, z_2) denotes the output expansion. The absolute value of the sum of the discarded error terms is bounded above in magnitude by $2^{-(3p-3)}|x + y|$.

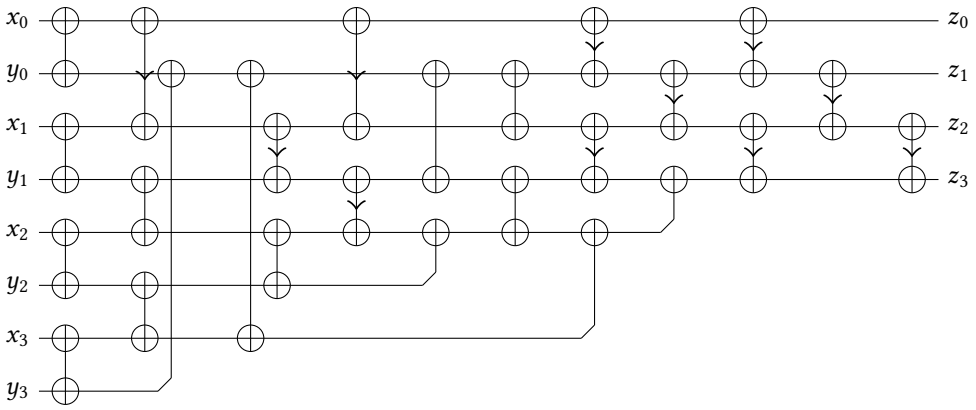


Figure 4: Conjecturally optimal FPAN with size 26 and depth 11 for addition of 4-term floating-point expansions. Here, (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) denote the input expansions to be added, and (z_0, z_1, z_2, z_3) denotes the output expansion. The absolute value of the sum of the discarded error terms is bounded above in magnitude by $2^{-(4p-4)}|x + y|$.

such as the associative and distributive properties. However, the lack of commutativity is particularly problematic in applications involving complex numbers because it causes the complex conjugate product $(a+bi)(a-bi)$ to have a small but nonzero imaginary part. This creates significant rounding artifacts that severely degrade the performance of certain numerical algorithms, such as eigensolvers.

In a similar fashion to the addition FPANs described in the preceding subsection, we can enforce the commutative property in our multiplication FPANs by adding an initial layer of TwoSum gates that pair the symmetric values $(p_{i,j}, p_{j,i})$ and $(e_{i,j}, e_{j,i})$ for all $i \neq j$. For addition FPANs, this initial commutativity layer happens to naturally occur in the optimal FPANs discovered by our heuristic search procedure. However, this does not naturally occur in multiplication FPANs, and we must deliberately impose the presence of the commutativity layer in our search procedure.

In Figures 5, 6, and 7, we present the smallest provably-correct FPANs we have discovered for commutative multiplication of 2-term, 3-term, and 4-term floating-point expansions. These FPANs have size and depth (3, 3), (12, 7), and (27, 10), respectively. As before, the 2-term multiplication FPAN is provably optimal by exhaustive enumeration, while the 3-term and 4-term FPANs are merely conjecturally optimal. We have high confidence in the optimality of

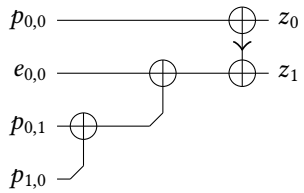


Figure 5: Provably optimal FPAN for commutative multiplication of 2-term floating-point expansions. Here, (x_0, x_1) and (y_0, y_1) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1) denotes the output expansion. The absolute value of the sum of the discarded error terms is bounded above by $2^{-(2p-3)}|xy|$.

our 3-term multiplication algorithm because our heuristic search procedure reliably converges to this particular FPAN, but for 4-term multiplication, we find a large pool of several thousand similar but non-isomorphic FPANs that all share the same size and depth. The

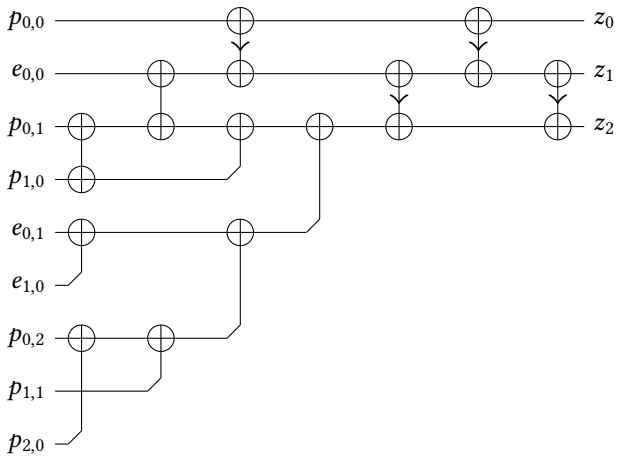


Figure 6: Conjecturally optimal FPAN with size 12 and depth 7 for commutative multiplication of 3-term floating-point expansions. Here, (x_0, x_1, x_2) and (y_0, y_1, y_2) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1, z_2) denotes the output expansion. The absolute value of the sum of the discarded terms is bounded above by $2^{-(3p-3)}|xy|$.

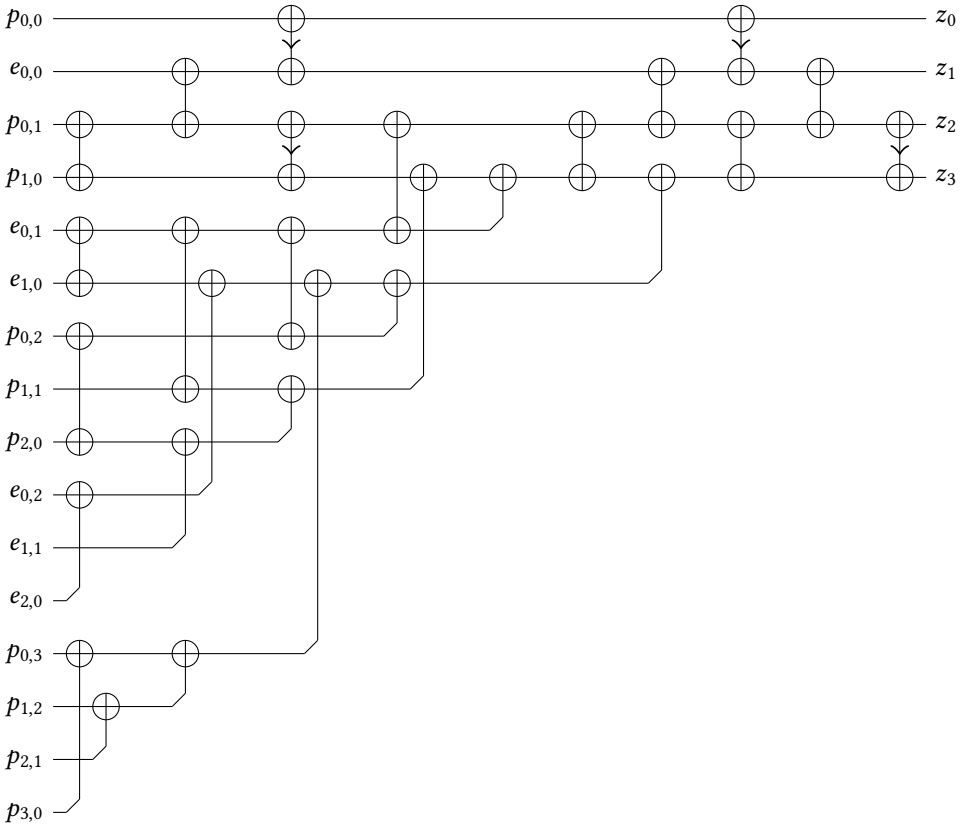


Figure 7: Conjecturally optimal FPAN with size 27 and depth 10 for commutative multiplication of 4-term floating-point expansions. Here, (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) denote the input expansions to be multiplied, $(p_{i,j}, e_{i,j}) \coloneqq \mathsf{TwoProd}(x_i, y_j)$ denote the FPAN inputs, and (z_0, z_1, z_2, z_3) denotes the output expansion. The absolute value of the sum of the discarded terms is bounded above by $2^{-(4p-4)}|xy|$.

FPAN we present in Figure 7 is one member of this conjecturally-optimal pool. It is presently unclear whether this unusually complex structure is an inherent mathematical property of four-term multiplication or is merely indicates that our search procedure has not yet converged.

4.3 Division and Square Root

With branch-free addition and multiplication algorithms in hand, division and square root can be implemented in a branch-free fashion using classical algorithms based on division-free Newton–Raphson iteration. This approach is well-known in the computer arithmetic literature, so we only state the core ideas in this paper for completeness, referring the reader to Ref. [30] for detailed analysis.

The basic principle of these algorithms is to apply the Newton–Raphson iterative root-finding method, defined by the recurrence

$$x_{n+1} \coloneqq x_n - \frac{f(x_n)}{f'(x_n)}, \tag{12}$$

to the function

$$f(x) = \frac{1}{x} - a \tag{13}$$

which has a unique root at $x = 1/a$ for nonzero a, or the function

$$f(x) = \frac{1}{x^2} - a \tag{14}$$

which has two roots at $x = \pm 1/\sqrt{a}$. Substituting these functions into Eq. 12 produces the iterative formula

$$x_{n+1} = x_n + x_n(1 - ax_n) \tag{15}$$

for computing inverses, and

$$x_{n+1} = x_n + \frac{1}{2}x_n(1 - ax_n^2) \tag{16}$$

for computing inverse square roots. (Note that multiplication by $1/2$ is an exact operation that can be applied termwise in binary floating-point arithmetic.) By taking the initial guess x_0 to be the machine-precision approximation $1.0 \oslash a$ or $1.0 \oslash \sqrt[3]{a}$, respectively, these iterative formulas allow rapid approximation of $1/a$ or $1/\sqrt{a}$, with the number of correct bits roughly doubling on every iteration. Finally, once $1/a$ or $1/\sqrt{a}$ is computed to the desired accuracy, we can obtain the quotient b/a by multiplying $1/a$ by b, or the square root $\sqrt{a}$ by multiplying $1/\sqrt{a}$ by a.

This technique can be optimized for use with floating-point expansions by reducing the number of terms used to represent the initial iterates. The initial approximation x_0 is only accurate to machine precision, so there is no need to store more than one term at this stage. The number of accurate bits doubles with each subsequent iteration, so the next iterate x_1 can be represented using a two-term expansion, then x_2 with a four-term expansion, and so on until the desired final precision is reached. The Karp–Markstein optimization [30] can also be applied to fuse the final Newton iteration with the multiplication of $1/a$ by b or $1/\sqrt{a}$ by a, eliminating several costly full-precision multiplications.

4.4 Limitations and Extensions

Although the arithmetic algorithms stated in this paper are provably correct for all finite nonoverlapping inputs, they do not strictly adhere to IEEE 754 semantics for the special values `−0.0` (negative zero) and `±Inf` (positive and negative infinity). This is because the underlying TwoSum, TwoProd, and FastTwoSum algorithms perform sequential inverse operations, such as adding and subtracting the same number, which are designed to compute rounding errors. When applied to special values, these operations discard the sign[2] of zero, turning `−0.0` into `+0.0`, and collapse `±Inf` into `NaN`, since subtracting infinity from itself yields an indeterminate form.

In our experience, this limitation does not affect most scientific applications, which do not make any substantial use of `−0.0` or `±Inf`. Indeed, scientific programmers routinely enable unsafe compiler optimizations, such as automatic vectorization, that assume these special values do not occur. Many common usage patterns, such as returning `NaN` to signal an error condition, are not affected by the loss of negative zero or infinity.

In cases where it is necessary to distinguish `−0.0` from `+0.0` or `±Inf` from `NaN`, strict IEEE 754 semantics can be restored using conditional move operations. We also note that this issue will not exist on future processors implementing the augmentedAddition and augmentedMultiplication operations proposed in the 2019 revision of IEEE 754 [21]. These operations, not yet available today, will replace TwoSum, TwoProd, and FastTwoSum with hardware operations that natively handle special values.

Another notable limitation is that floating-point expansions can only increase the precision, not the exponent range, of the underlying base type. In contrast to true IEEE quadruple precision, which supports a dynamic range of $2^{16384} \approx 10^{4932}$, expansions based on IEEE double precision can only store numbers up to $2^{1024} \approx 10^{308}$. This is unproblematic for computational science and engineering, since all physically measurable quantities[3] fall well within this range, but may pose an issue for applications in number theory and cryptography that work with immeasurably large numbers.

The inability to extend the exponent range, while unproblematic in double precision, severely limits the applicability of our algorithms to low-precision data types, which are increasingly popular for accelerating AI workloads. Their narrow exponent range causes floating-point expansions to lose precision past the machine underflow threshold, which typically occurs at roughly 4 terms in single precision and just 2 terms in half precision. It is an open research question whether branch-free algorithms can overcome this limitation. The only known methods that handle wide exponent ranges, such as the Ozaki scheme [45], use data-dependent branching with a dynamic number of terms and lose their performance advantage as the exponent range widens. Similarly, it is unknown whether matrix multiplication accelerators, such as tensor cores and neural processing units, can be adapted for extended-precision operations in a branch-free fashion.

Finally, when the result of a floating-point addition or multiplication operation is exactly $\pm 2^{e_{\max}}$ (i.e., `±DBL_MAX`), the TwoSum algorithm can internally overflow [4], causing it to erroneously return `NaN` instead of $\pm 2^{e_{\max}}$. This means the overflow threshold for

[2] IEEE 754 defines `−0.0` to be equal to `+0.0`, so these values are normally indistinguishable without the use of functions like `signbit` or `copysign` that inspect the underlying bitwise representation of a floating-point value.

[3] The dynamic range of any quantity in a physical simulation, such as length, is bounded by the ratio of the longest theoretically measurable length (the diameter of the observable universe) to the shortest theoretically measurable length (the Planck length). This ratio is roughly 5×10^{61}. The corresponding ratios for mass ($m_{\text{universe}}/m_{\text{Planck}} \approx 6 \times 10^{60}$) and time ($t_{\text{Hubble}}/t_{\text{Planck}} \approx 8 \times 10^{60}$) fall similarly well within the IEEE double precision overflow threshold (`DBL_MAX` $\approx 2^{1024} \approx 10^{308}$).

floating-point expansions is one machine epsilon narrower than the underlying base type.

5 Evaluation

To assess the performance of our new algorithms in a practical scientific computing context, we used them to implement four extended-precision BLAS kernels that exercise typical computational patterns found in scientific software.

- AXPY: vector-vector operations
- DOT: vector-vector reduction operations
- GEMV: matrix-vector operations
- GEMM: matrix-matrix operations

We developed *MultiFloats*, a prototype C++ library that implements these BLAS kernels on two-term (quadruple precision), three-term (sextuple precision), and four-term (octuple precision) expansions using the addition and multiplication FPANs shown in Figures 2–7. Our library provides a template datatype `MultiFloat<T,N>` parameterized by an underlying floating-point type `T` and a floating-point expansion length $N = 1, 2, 3, 4$. (`MultiFloat<T,1>` is simply an alias for `T`.)

Note that allowing the user to select the underlying floating-point type `T` significantly enhances the portability of our library. For example, datatypes like `MultiFloat<float,4>` can be used to provide extended-precision arithmetic on machines that lack double-precision hardware. On the other hand, processors with native support for IEEE quadruple precision can use `MultiFloat<quad,2>` to provide fast octuple-precision arithmetic. Nonetheless, we expect `MultiFloat<T,N>` to be used with `T = double` on the vast majority of contemporary high-performance computer hardware.

We compared the performance of our library, MultiFloats, to the following suite of extended-precision arithmetic libraries:

- GMP 6.3.0 [15]
- MPFR 4.2.1 [14]
- FLINT (formerly known as Arb) 3.2.1 [17, 27]
- Boost.Multiprecision 1.86 [37]
- QD[4] 2.3.23 [2]
- CAMPARY[5] 01.06.17 [29]
- libquadmath[6] 14.2 [16]

The latest version of each library available at the time of writing was selected for testing. This is an exhaustive list of all extended-precision floating-point libraries that we are aware of, excluding (1) libraries for base-10 floating-point arithmetic, such as mpdecimal; (2) libraries that merely wrap the interface of another library, such as MPFR++ and mppp; (3) libraries that are not thread-safe, such as CLN; (4) libraries targeting dynamic languages, such as mpmath and bignumber.js; and (5) unmaintained libraries that no longer compile on modern hardware, including CUMP and MPRES-BLAS. We also exclude XBLAS [34] from consideration because its interface does not allow extended-precision numbers to be passed into or out of the library.

[4]QD only supports two-term and four-term floating-point expansions.

[5]CAMPARY provides two sets of arithmetic algorithms: a "certified" set that is provably correct but uses branching, and a "fast" set that is branch-free but known to be incorrect on some classes of inputs. In some cases, the "fast" algorithms exhibit catastrophic loss of precision, degrading the accuracy of the result to machine precision. We benchmark only the "certified" algorithms to provide a fair comparison to our algorithms, which are provably correct on all inputs.

[6]libquadmath is the library used to provide the built-in `__float128` type in the GCC and Clang compilers. It only supports IEEE quadruple-precision arithmetic and does not provide any other precision levels.

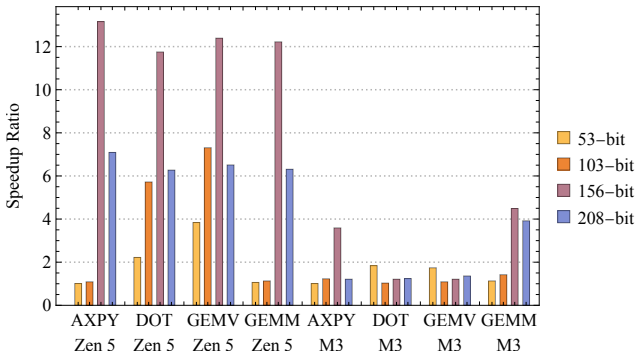


Figure 8: Ratio of peak performance achieved by our FPAN-based algorithms over the next best-performing multiprecision library, measured on AMD Zen 5 (Ryzen 9 9950X) and Apple M3 Pro processors. In all cases, the ratio exceeds 1, indicating that our algorithms are faster than all existing multiprecision libraries.

To ensure optimal conditions for fair comparison, each combination of library and BLAS kernel was compiled using the latest available versions of both the GCC (14.2) and Clang (20.1) compilers, using both medium (`-O2`) and full (`-O3`) optimization levels, with all available ISA extensions (`-march=native`) enabled on the most recent high-performance processor microarchitectures (AMD Zen 5 and Apple M3) available to us at the time of writing. All BLAS kernels were implemented with identical parallelization strategies, using `ij` loop ordering for GEMV and `ikj` loop ordering for GEMM. In addition, each kernel was run in both thread-per-physical-core and thread-per-logical-core configurations using the OpenMP thread affinity API (`OMP_PROC_BIND`).

In Figures 8, 9, and 10, we report the maximum AXPY, DOT, GEMV, and GEMM performance achieved by each library on one-term (double precision), two-term (quadruple precision), three-term (sextuple precision), and four-term (octuple precision) floating-point expansions. For libraries not based on floating-point expansions, we statically specified an equivalent number of bits of precision (53, 103, 156, and 208 bits, respectively) based on our FPAN error bounds. The numbers reported in these tables represent the maximum computational throughput, in billions of extended-precision operations per second, achieved over all possible choices of compiler, optimization level, and thread count. To eliminate the effect of memory bandwidth, we measured performance on the largest matrix and vector sizes that each library can fit into L3 cache.

We adopt the usual convention in numerical linear algebra that one operation consists of one multiplication followed by one addition. Thus, given vectors of size n and matrices of size $n \times n$, AXPY and DOT execute n operations, GEMV executes n^2 operations, and GEMM executes n^3 operations. Note that each extended-precision operation conists of several dozen to several hundred native machine-precision FLOPs.

On AMD Zen 5, our new FPAN-based algorithms significantly outperformed all competing libraries in all benchmarks, often by

AMD Zen 5 AXPY Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**135.22**	**35.35**	**11.32**	**5.60**
GMP	0.67	0.64	0.63	0.63
MPFR	1.45	1.13	0.75	0.50
FLINT	1.39	1.01	0.86	0.79
Boost.Multiprecision	1.33	0.61	0.36	0.33
QD	N/A	24.13	N/A	0.50
CAMPARY	133.80	32.44	0.35	0.24
libquadmath	N/A	1.05	N/A	N/A

AMD Zen 5 DOT Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**117.35**	**30.87**	**11.75**	**5.77**
GMP	0.65	0.64	0.64	0.63
MPFR	1.44	1.16	0.78	0.55
FLINT	1.62	1.21	1.00	0.92
Boost.Multiprecision	1.40	0.63	0.34	0.32
QD	N/A	4.66	N/A	0.51
CAMPARY	52.84	5.40	0.36	0.25
libquadmath	N/A	1.13	N/A	N/A

AMD Zen 5 GEMV Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**225.18**	**38.87**	**12.14**	**5.86**
GMP	0.66	0.66	0.66	0.64
MPFR	1.51	1.21	0.79	0.59
FLINT	1.63	1.22	0.98	0.90
Boost.Multiprecision	1.34	0.63	0.38	0.33
QD	N/A	4.68	N/A	0.51
CAMPARY	58.65	5.32	0.36	0.25
libquadmath	N/A	1.12	N/A	N/A

AMD Zen 5 GEMM Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**328.98**	**42.18**	**12.34**	**5.93**
GMP	0.62	0.61	0.61	0.60
MPFR	1.50	1.18	0.79	0.55
FLINT	1.61	1.22	1.01	0.94
Boost.Multiprecision	1.30	0.63	0.37	0.31
QD	N/A	26.47	N/A	0.51
CAMPARY	310.29	37.42	0.36	0.25
libquadmath	N/A	1.13	N/A	N/A

Figure 9: Measured CPU performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 16-core AMD Zen 5 CPU (Ryzen 9 9950X). "N/A" entries indicate lack of library support for a specific precision level.

Apple M3 AXPY Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**15.12**	**4.60**	**1.47**	**0.29**
GMP	0.15	0.16	0.16	0.16
MPFR	0.69	0.56	0.41	0.24
FLINT	0.29	0.22	0.19	0.18
Boost.Multiprecision	0.59	0.33	0.18	0.15
QD	N/A	2.40	N/A	0.17
CAMPARY	14.93	3.75	0.27	0.16
libquadmath	N/A	N/A	N/A	N/A

Apple M3 DOT Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**12.50**	**1.19**	**0.52**	**0.31**
GMP	0.16	0.16	0.16	0.16
MPFR	0.73	0.66	0.43	0.25
FLINT	0.44	0.30	0.27	0.23
Boost.Multiprecision	0.62	0.34	0.18	0.15
QD	N/A	1.16	N/A	0.17
CAMPARY	6.81	0.94	0.24	0.16
libquadmath	N/A	N/A	N/A	N/A

Apple M3 GEMV Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**15.59**	**1.26**	**0.51**	**0.34**
GMP	0.16	0.16	0.16	0.16
MPFR	0.78	0.68	0.42	0.25
FLINT	0.45	0.32	0.27	0.23
Boost.Multiprecision	0.59	0.33	0.18	0.15
QD	N/A	1.16	N/A	0.17
CAMPARY	8.95	0.95	0.25	0.14
libquadmath	N/A	N/A	N/A	N/A

Apple M3 GEMM Performance

Library	53-bit	103-bit	156-bit	208-bit
MultiFloats (ours)	**46.53**	**6.78**	**2.02**	**0.98**
GMP	0.16	0.16	0.16	0.16
MPFR	0.84	0.69	0.45	0.25
FLINT	0.48	0.32	0.27	0.25
Boost.Multiprecision	0.61	0.32	0.18	0.14
QD	N/A	2.76	N/A	0.17
CAMPARY	41.10	4.77	0.27	0.19
libquadmath	N/A	N/A	N/A	N/A

Figure 10: Measured CPU performance, in billions of extended-precision operations per second, of multiprecision libraries at 53-bit, 103-bit, 156-bit, and 208-bit precision on a 12-core ARMv8.6-A CPU (Apple M3 Pro). "N/A" entries indicate lack of library support for a specific precision level.

more than an order of magnitude. Only two libraries, QD and CAMPARY, achieved comparable AXPY and GEMM performance in the two-term case by using previously known, albeit suboptimal, branch-free algorithms. They are unable to match our two-term

AMD RDNA3 GPU Performance

Kernel	1-Term	2-Term	3-Term	4-Term
AXPY	44.25	21.63	15.77	9.71
DOT	84.83	56.72	38.14	28.44
GEMV	170.77	92.37	28.42	31.92
GEMM	466.43	277.37	170.50	81.11

Figure 11: Measured GPU performance, in billions of extended-precision operations per second, of our FPAN-based algorithms on an AMD RDNA 3 GPU (RX 7900 XTX).

DOT and GEMV performance because they do not provide SIMD reduction operators and their code is too complex for either GCC 14.2 or Clang 20.1 to automatically vectorize. Moreover, at three-term (156-bit) and four-term (208-bit) precision levels, no competing libraries come within a factor of 10× or 5× of our algorithms in any of the four tested kernels. We also observe that our algorithms exhibit a modest but consistent trend of increasing computational throughput across vector-vector operations (AXPY and DOT), matrix-vector operations (GEMV), and matrix-matrix operations (GEMM), representing different points on a roofline curve.

On Apple M3, our FPAN-based algorithms also outperformed all competing libraries in all benchmarks, though the ratio of improvement is less dramatic. Compared to AMD Zen 5, this architecture deprioritizes SIMD performance (128-bit NEON vs. 512-bit AVX), so efficiently-vectorizable branch-free algorithms experience a smaller performance uplift compared to branching scalar code. Nonetheless, our algorithms are still consistently the fastest, and some order-of-magnitude improvements over existing libraries are still observed.

Finally, in Figure 11, we report the performance of our algorithms on an AMD RDNA3 GPU using the ROCm 6.4.1 toolchain. Unlike our CPU benchmarks, our GPU implementation uses `T = float` as the underlying base type instead of `T = double` because this architecture lacks double precision units. We observe significant performance uplift over CPUs, particularly for high-precision GEMM operations, which are more than an order of magnitude faster. These experiments demonstrate that the branch-free nature of our algorithms makes them highly suitable for GPUs, in addition to their utility for extending the precision of single-precision hardware.

6 Related Work

Sorting Networks. FPANs are closely related to sorting networks, and the graphical FPAN notation employed in this paper is heavily inspired by the diagrammatic representation of sorting networks. Although they compute different operations, both are branch-free algorithms that sort or accumulate a fixed number of inputs by performing pairwise operations in a data-parallel fashion. This close relationship inspires many natural research questions connecting the well-established theory of sorting networks to the relatively unexplored theory of FPANs. For example, techniques for proving lower bounds on the size or depth of sorting networks may be transferable to FPANs, and there may exist an analogue of the 0-1 principle. In addition, methods for discovering and optimizing sorting networks may prove useful in searching for larger FPANs that handle more inputs.

Compensated Algorithms. Beyond FPANs, error-free transformations are also employed in a class of floating-point algorithms called *compensated algorithms*, such as Kahan–Babuška–Neumaier summation. Unlike floating-point expansions, which involve a fixed number of terms, these algorithms operate on a variable number of inputs and only attempt to partially track and correct rounding errors, with weaker worst-case precision guarantees.

Systems for Dynamic and Adaptive Precision Tuning. The high performance of FPAN-based algorithms makes them attractive candidates for implementation in systems like Precimonuous [47], Shaman [11], and ADAPT [39] for dynamic and adaptive precision tuning of floating-point applications. High-performance libraries outfitted with extended-precision computational kernels could be used to produce high-quality reference results and strong correctness checks for low- and mixed-precision implementations.

Program Synthesis. Our method for discovering and verifying FPANs is an example of search-based program synthesis, where a search procedure is used to discover a program that satisfies both correctness and performance requirements. Search-based synthesis methods have been used to superoptimize assembly code [38, 48], deep learning computations [26, 26], cryptographic primitives [32], and quantum algorithms [46, 51]. These techniques combine a fast heuristic search that uses testing to identify plausible candidate programs with a full formal verification procedure that confirms whether the candidate is correct. Our method uses a different search procedure than previous work (simulated annealing) combined with a novel and highly elaborate verifier [53].

7 Conclusion

We have presented a novel approach to extended-precision floating-point arithmetic using branch-free algorithms with formally verified error bounds. This unique combination of properties is enabled by the use of new building blocks, called floating-point accumulation networks (FPANs), along with a new analysis technique based on automatic theorem provers. We argue that these properties, along with the remarkably high performance demonstrated in our benchmarks, make our algorithms uniquely suitable for high-performance scientific computation.

References

[1] Ryan Abbott, William Detmold, Fernando Romero-López, Zohreh Davoudi, Marc Illa, Assumpta Parreño, Robert J. Perry, Phiala E. Shanahan, and Michael L. Wagman. 2023. Lattice quantum chromodynamics at large isospin density. *Phys. Rev. D* 108 (Dec 2023), 114506. Issue 11. doi:10.1103/PhysRevD.108.114506

[2] David H Bailey. 2020. Reproducibility and variable precision computing. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 483–490. doi:10.1177/1094342020938424 arXiv:https://doi.org/10.1177/1094342020938424

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[4] Sylvie Boldo, Stef Graillat, and Jean-Michel Muller. 2017. On the Robustness of the 2Sum and Fast2Sum Algorithms. *ACM Trans. Math. Softw.* 44, 1, Article 4 (July 2017), 14 pages. doi:10.1145/3054947

[5] Sylvie Boldo, Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. 2017. Formal Verification of a Floating-Point Expansion Renormalization Algorithm. In *Interactive Theorem Proving*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer International Publishing, Cham, 98–113.

[6] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus) *(TACAS'10)*. Springer-Verlag, Berlin, Heidelberg, 150–153. doi:10.1007/978-3-642-12002-2_12

[7] Mythile C, Jaisiva S, Arunadevi A, and Sudhapriya K. 2024. Examining Floating Point Precision in Contemporary FPGAs. In *2024 Third International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*. 1–7. doi:10.1109/ICEEICT61591.2024.10718633

[8] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–107.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[10] T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (June 1971), 224–242. doi:10.1007/BF01397083

[11] Nestor Demeure. 2021. *Gestion du compromis entre la performance et la précision de code de calcul*. Theses. Université Paris-Saclay. https://theses.hal.science/tel-03116750

[12] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 737–744.

[13] Giovanni Fantuzzi, David Goluskin, and Jean-Bernard Lasserre. 2025. Polynomial Optimization for Nonlinear Dynamics: Theory, Algorithms and Applications. *Oberwolfach Reports* 21, 3 (Feb. 2025), 1975–2032. doi:10.4171/owr/2024/35

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007), 13–es. doi:10.1145/1236463.1236468

[15] Free Software Foundation. 2023. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Free Software Foundation. https://gmplib.org/ Accessed: 2025-04-14.

[16] GCC Team. 2023. *libquadmath: The GCC Quad-Precision Math Library*. GNU Compiler Collection. https://gcc.gnu.org/onlinedocs/libquadmath/ Accessed: 2025-04-14.

[17] William Hart, Fredrik Johansson, and Sebastian Pancratz. 2023. *FLINT: Fast Library for Number Theory*. FLINT Project. https://flintlib.org/ Accessed: 2025-04-14.

[18] Yun He and Chris H. Q. Ding. 2001. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. *The Journal of Supercomputing* 18, 3 (01 Mar 2001), 259–277. doi:10.1023/A:1008153532043

[19] Y. Hida, X.S. Li, and D.H. Bailey. 2001. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001.* 155–162. doi:10.1109/ARITH.2001.930115

[20] Nicholas J. Higham and Theo Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414. doi:10.1017/S0962492922000022

[21] IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. doi:10.1109/IEEESTD.2019.8766229

[22] Konstantin Isupov, Vladimir Knyazkov, and Alexander Kuvaev. 2020. Design and implementation of multiple-precision BLAS Level 1 functions for graphics processing units. *J. Parallel and Distrib. Comput.* 140 (2020), 25–36. doi:10.1016/j.jpdc.2020.02.006

[23] Manish Kumar Jaiswal and Ray C.C. Cheung. 2012. Area-Efficient FPGA Implementation of Quadruple Precision Floating Point Multiplier. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 376–382. doi:10.1109/IPDPSW.2012.46

[24] Manish Kumar Jaiswal and Hayden K.-H So. 2016. Architecture for quadruple precision floating point division with multi-precision support. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 239–240. doi:10.1109/ASAP.2016.7760807

[25] Manish Kumar Jaiswal and Hayden K.-H. So. 2018. An Unified Architecture for Single, Double, Double-Extended, and Quadruple Precision Division. *Circuits, Systems, and Signal Processing* 37, 1 (Jan. 2018), 383–407. doi:10.1007/s00034-017-0559-9

[26] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[27] F. Johansson. 2017. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Trans. Comput.* 66 (2017), 1281–1292. Issue 8. doi:10.1109/TC.2017.2690633

[28] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. 2017. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. *ACM Trans. Math. Softw.* 44, 2, Article 15res (Oct. 2017), 27 pages. doi:10.1145/3121432

[29] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. 2016. CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In *Mathematical Software – ICMS 2016*, Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese (Eds.). Springer International Publishing, Cham, 232–240.

[30] Alan H. Karp and Peter Markstein. 1997. High-precision division and square root. *ACM Trans. Math. Softw.* 23, 4 (Dec. 1997), 561–589. doi:10.1145/279232.279237

[31] Donald E. Knuth. 1969. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley. https://www.worldcat.org/oclc/310551264

[32] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. *Proc. ACM Program. Lang.* 7, PLDI, Article 158 (June 2023). doi:10.1145/3591272

[33] Christoph Quirin Lauter. 2005. *Basic building blocks for a triple-double intermediate format*. Research Report RR-5702, LIP RR-2005-38. INRIA, LIP. 67 pages. https://inria.hal.science/inria-00070314

[34] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 152–205. doi:10.1145/567806.567808

[35] Ding Ma and Michael A. Saunders. 2015. *Solving Multiscale Linear Programs Using the Simplex Method in Quadruple Precision*. Springer International Publishing, 223–235. doi:10.1007/978-3-319-17689-5_9

[36] Ding Ma, Laurence Yang, Ronan M. T. Fleming, Ines Thiele, Bernhard O. Palsson, and Michael A. Saunders. 2017. Reliable and efficient solution of genome-scale models of Metabolism and macromolecular Expression. *Scientific Reports* 7, 1 (Jan. 2017). doi:10.1038/srep40863

[37] John Maddock and Christopher Kormanyos. 2023. *Boost.Multiprecision: A Multiprecision Arithmetic Library for C++*. Boost C++ Libraries. https://www.boost.org/doc/libs/release/libs/multiprecision/ Accessed: 2025-04-14.

[38] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.

[39] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 614–626. doi:10.1109/SC.2018.00051

[40] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic*. Springer International Publishing. doi:10.1007/978-3-319-76526-6

[41] Jean-Michel Muller and Laurence Rideau. 2022. Formalization of Double-Word Arithmetic, and Comments on "Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic". *ACM Trans. Math. Softw.* 48, 1, Article 9 (Feb. 2022), 24 pages. doi:10.1145/3484514

[42] Ole Møller. 1965. Quasi double-precision in floating point addition. *BIT Numerical Mathematics* 5, 1 (March 1965), 37–50. doi:10.1007/BF01975722

[43] Alessio Netti, Yang Peng, Patrik Omland, Michael Paulitsch, Jorge Parra, Gustavo Espinosa, Udit Agarwal, Abraham Chan, and Karthik Pattabiraman. 2023. Mixed precision support in HPC applications: What about reliability? *J. Parallel and Distrib. Comput.* 181 (2023), 104746. doi:10.1016/j.jpdc.2023.104746

[44] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 3–17. doi:10.1007/978-3-031-37703-7_1

[45] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. 2024. DGEMM on integer matrix multiplication unit. *Int. J. High Perform. Comput. Appl.* 38, 4 (July 2024), 297–313. doi:10.1177/10943420241239588

[46] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. 2024. Quanto: Optimizing quantum circuits with automatic generation of circuit identities. *Quantum Science and Technology* 9, 4 (2024), 045009.

[47] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2503210.2503296

[48] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

[49] Gerhard W. Veltkamp. 1968. *ALGOL Procedures voor het Berekenen van een Inwendig Product in Dubbele Precisie*. Technical Report 22. Technische Hogeschool Eindhoven.

[50] Gerhard W. Veltkamp. 1969. *ALGOL Procedures voor het Rekenen in Dubbele Lengte*. Technical Report 21. Technische Hogeschool Eindhoven.

[51] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. 2022. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 625–640.

[52] David K. Zhang. 2025. FPANVerifier. https://github.com/dzhang314/FPANVerifier.

[53] David K. Zhang and Alex Aiken. 2025. Automatic Verification of Floating-Point Accumulation Networks. In *Computer Aided Verification*, Ruzica Piskac and Zvonimir Rakamarić (Eds.). Springer Nature Switzerland, Cham, 215–237.

[54] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A. Saunders, Stephen J. Thomas, and Slaven Peleš. 2022. Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers. *Parallel Comput.* 111 (2022), 102870. doi:10.1016/j.parco.2021.102870