Article:
Mikaitis, M. orcid.org/0000-0001-8706-1436 (2024) Monotonicity of Multi-Term Floating-Point Adders. IEEE Transactions on Computers. ISSN 1557-9956

https://doi.org/10.1109/TC.2024.3371783

Monotonicity of Multi-Term Floating-Point Adders

Mantas Mikaitis

Abstract—In the literature on algorithms for computing multi-term addition $s_n = \sum_{i=1}^{n} x_i$ in floating-point arithmetic it is often shown that a hardware unit that has single normalization and rounding improves precision, area, latency, and power consumption, compared with the use of standard add or fused multiply–add units. However, non-monotonicity can appear when computing sums with a subclass of multi-term addition units, which is currently not explored in the literature. We prove that computing multi-term floating-point addition with $n \geq 4$, without normalization of intermediate quantities, can result in non-monotonicity—increasing one of the addends x_i decreases the sum s_n. Summation is required in dot product and matrix multiplication operations, operations that are increasingly appearing in the hardware of high-performance computers, and knowing where monotonicity is preserved can be of interest to the developers and users. Non-monotonicity of summation in existent hardware devices that implement a specific class of multi-term adders may have appeared unintentionally as a consequence of design choices that reduce circuit area and other metrics. To demonstrate our findings we simulate non-monotonic multi-term adders in MATLAB using the `CPFloat` custom-precision floating-point simulator.

Index Terms—Monotonicity, floating-point arithmetic, multi-term addition, dot product, matrix multiply.

1 INTRODUCTION

A real function f is monotonically nondecreasing on an interval $[a, b]$ if $f(x) \leq f(y)$ whenever $a \leq x \leq y \leq b$. In other words, when the argument of a monotonic function is increasing, the value of the function does not decrease. Similarly for a function that is monotonic nonincreasing: when the input argument is increasing the value of the function is not increasing. For the multivariate functions, a function is monotonic if it is monotonic for all the input values. For example, $f(x_1, x_2)$ is monotonic nondecreasing if for any $x_1 \leq x_1^*$ and $x_2 \leq x_2^*$ we have $f(x_1, x_2) \leq f(x_1^*, x_2^*)$.

Summation of a set of values is a multivariate function that is monotonic nondecreasing (just monotonic thereafter). Given a set of input values x_1, x_2, ..., x_n, summation is expressed as

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} x_i. \qquad (1)$$

This function is monotonically nondecreasing by the definition of the sum: take $x_n = a$ followed by $x_n = a + \varepsilon$, with $\varepsilon > 0$. Then $f(x_1, x_2, ..., a) = (\sum_{i=1}^{n-1} x_i) + a$, whereas $f(x_1, x_2, ..., a + \varepsilon) = (\sum_{i=1}^{n-1} x_i) + a + \varepsilon > (\sum_{i=1}^{n-1} x_i) + a$. Because of the commutativity of the sum this is true for all the input arguments.

Summation of values is at the core of scientific computing—it is required, for example, for calculating vector–vector products, matrix–vector and matrix–matrix multiplications, as well as in evaluating polynomials. Most computer software works with floating-point numbers [1], rather than exact numbers, so the addition operation is different from the exact addition and the monotonicity of summation should be tested rather than assumed to hold because it holds in exact arithmetic. This is similar to the properties of associativity, commutativity and distributivity—commutativity is generally preserved, but associativity and distributivity in basic arithmetic operations, when transitioning from exact to floating-point arithmetic, are not [2, Sec. 2.6].

Algorithm 1.1: Given numbers $\mathcal{S} = \{x_1, ..., x_n\}$, compute $s_n = \sum_{i=1}^{n} x_i$.

```
1 Repeat while $\mathcal{S}$ contains more than one element
2     From $\mathcal{S}$, remove two numbers $a$ and $b$
3     Put $\mathrm{fl}(a + b)$ to $\mathcal{S}$
4 The remaining element in $\mathcal{S}$ is $s_n$
```

From here we use the notation of the standard model [3, Sec. 2.2] of addition in precision-p arithmetic:

$$\mathrm{fl}(x + y) = (x + y)(1 + \delta), \quad |\delta| \leq u$$

where $u = 2^{-p}$, the unit roundoff, and $\mathrm{fl}(x)$ refers to normalizing (see Section 2.1) and rounding x to form a floating-point value defined above.

Most floating-point units on general-purpose hardware include 2-term adders that compute the sum of floating-point numbers $z = \mathrm{fl}(a + b)$ which is required by the IEEE 754 standard [1, Sec. 5.4.1]. This operation includes computing $a + b$ as though in infinite precision, normalizing the resultant significand if required and rounding it to obtain z [2, Sec. 7.3]. At software level, this operation is called repeatedly to compute sums of arbitrary length (Equation 1). A high-level algorithm [3, Sec. 4.2] is given as Algorithm 1.1. On line 3 the operation $\mathrm{fl}(a + b)$ includes rounding and normalization, which means that, when implemented this way, overall the n-term addition performs $n - 1$ such roundings and normalizations. It is safe to say that most software is implemented this way on the hardware that includes circuitry only for adding two numbers as per IEEE 754.

Summation of three, four or longer vectors of floating-point values can be classified under reduction operations which are recommended but not required by the IEEE 754

School of Computing, University of Leeds, Leeds, LS2 9JT, United Kingdom (`M.Mikaitis@leeds.ac.uk`). Preprint version: 4th of December, 2023.

standard [1, Sec. 9.4]. The guideline provided by IEEE 754 for implementing them is not significantly constrained.

Addition of more than three values can be achieved by an accumulator which takes as one of the inputs its output from the previous stage or by using multiple 2-term adders in series or in parallel (Figure 1). However, in hardware, a custom design is often considered to gain in speed and circuit area compared with the straightforward approach of using the standard 2-term adders. Algorithm 1.2 shows the main steps taken by most of the multi-term adders available in the literature (see Section 2.1 for the details on floating point). The main point to note is that normalization and rounding are performed once, at the end, rather than after each intermediate addition operation, as in Algorithm 1.1, which in literature on hardware appears to be beneficial for saving hardware resources and even increasing the accuracy. All of this applies to dot products and matrix multiplies, by replacing $\mathrm{fl}(a+b)$ with $\mathrm{fl}(\mathrm{fl}(a \times b)+c)$ or the fused multiply–add (FMA) $\mathrm{fl}(a \times b + c)$.

Algorithm 1.2: Given numbers $\mathcal{S} = \{x_1, ..., x_n\}$, with exponents $\{e_1, ..., e_n\}$ and significands $\{1.m_1, ..., 1.m_n\}$ compute $s_n = \sum_{i=1}^{n} x_i$.

```
Determine $e_{max} = \max(e_1, ..., e_n)$
Align all $1.m_i$ by shifting each $e_{max} - e_i$ steps right
Perform addition of aligned significands
Perform normalization and rounding to form $s_n$
```

As a side note, unnormalized floating-point arithmetic appeared as early as 1958 in the work of Metropolis and Ashenhurst [4]; the performance improvement compared with the normalized arithmetic was noted even then.

In this paper we identify four types of implementation for the Algorithm 1.2 and demonstrate that one class of designs for multi-term addition are non-monotonic, including various commercial hardware designs available and widely present on the machines in the TOP500[1]. We also suggest that this issue can be fixed by masking off the bottom bits when carries occur or by using an adder from a different class, which can be added into the future summation and dot product hardware designs as an option.

We mentioned that non-monotonic summation in floating-point arithmetic adds to the list of mathematical properties that are not preserved when switching from exact arithmetic. One other motivating point for studying this is reproducibility of numerical computations. Bit-wise reproducibility is not impossible on single-core CPUs that implement the IEEE 754 standard correctly and assuming special features such as 80-bit arithmetic are not enabled by compilers. If we run some code in IEEE 754 software, using basic operations of a floating-point unit (FPU), we get one behaviour, but if we run that code in hardware that performs summation non-monotonically, we may get unexpected results that may be hard to explain. We now demonstrate this with an example.

1.1 An example on current hardware

For the following we use NVIDIA A100 SXM 80GB GPU because we have access to it, however we predict that most commercial hardware in the market would not pass the following, and similar, tests.

A100 GPUs are equipped with matrix multiply hardware that can perform $D = A \times B + C$, where $A \in \mathbb{R}^{8\times 8}$ and $B \in \mathbb{R}^{8\times 4}$ are binary16 matrices, and $C, D \in \mathbb{R}^{8\times 4}$ are binary32 matrices [5, p. 20]. We showed before that matrix multipliers in this GPU have one extra bit in the alignment of binary32 significands [6]. We will focus on two resulting matrix elements, which perform dot products $d_{11} = a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{18}b_{81} + c_{11}$ and $d_{12} = a_{11}b_{12} + a_{12}b_{22} + \cdots + a_{18}b_{82} + c_{12}$ by most likely implementing a 9-term adder of products. We set $A, B = 1$ (matrices of ones) and $c_{11} = 33554430$ and $c_{12} = 33554432$. Computing $A \times B + C$ with the GPU matrix multipliers returns a matrix that has $d_{11} = 33554436$ and $d_{12} = 33554432$, demonstrating non-monotonic behaviour where an increase in one of the 9 addends decreases the sum. Note that we increased the addend by 2 and the sum was decreased by 4 since the amount we decrease by comes from the other 8 addends not contributing to the sum. Similar examples can be constructed around any powers of two, and at the edges of the dynamic range the quantities by which the sum changes by decreasing one of the addends would be larger in absolute terms. In the past we performed tests that indicated that NVIDIA V100 and the T4 GPUs have similar behaviour, although the V100 differs due to narrower internal accumulator in the multi-term addition [6]. NVIDIA H100 and the AMD matrix multipliers are yet to be tested.

1.2 Multi-term floating-point addition in literature

Hardware designs of multi-term adders (Algorithm 1.2), including those that are part of dot product and matrix multiply hardware, can be classified into four main categories. Some of the ways to build multi-term adders are demonstrated in the high-level diagrams of Figure 1.

We will use a term *fused*. From the user's perspective, in most cases it means that only one rounding error is incurred in the computation, except where stated otherwise.

1.2.1 Class I: Adders that use long accumulators

One approach is to retain all the bits in the summation of multiple values and round it once at the end. This is advocated by Kulisch [7, Sec. 8]. See the design-space exploration by Uguen and de Dinechin [8] for a detailed analysis of the costs. An implementation by Koenig, Bachrach, and Asanović [9] used 4288 bits internally for multiplying and accumulating binary64 values exactly. These kind of multi-term adders are fused because they contain only one rounding across the whole computation; however, keeping all of the bits can be expensive in circuit area and latency due to carry propagation.

While not directly a multi-term adder, the accumulator of binary16 products by Brunie [10] uses an exact 80-bit fixed-point internal format and therefore can be used to implement fused dot products or matrix multiplies with a

[1] https://top500.org/lists/top500/list/2023/11/

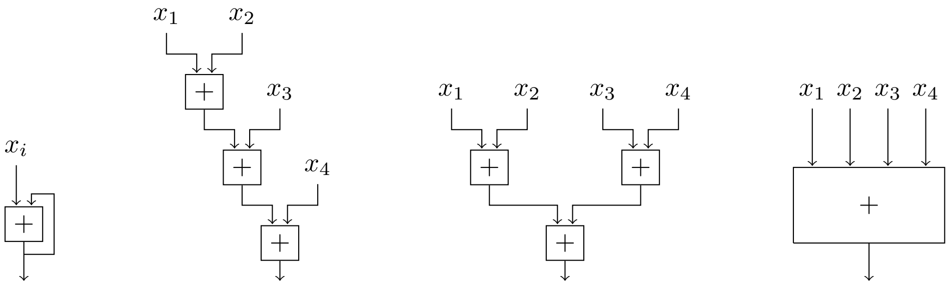

Fig. 1. Various methods for implementing 4-term addition in hardware. The left-most method, which is present in virtually all modern hardware, is to iterate over an IEEE 754 floating-point addition multiple times to sum a vector of numbers x; with this method, every iteration requires a new addition instruction to be fetched and executed. The second and third approaches demonstrate two ways to utilize multiple IEEE 754 floating-point adders—the first one simply chains four adders while the second one creates an adder tree to reduce the latency. In these three methods, the order of addition can be changed by rearranging the inputs. The remaining method is a black box which contains a specialized hardware design for performing the summation, not necessarily using the operations outlined in IEEE 754 and usually enforcing a specific order of summation. The first three approaches correspond to Class III adders and the last one encapsulates Classes I, II, and IV (Section 1.2).

single rounding error. Brunie [11] also proposed an architectural extension to CPUs which adds basic linear algebra instructions that work on matrices packed in general-purpose vector registers. The papers suggest that whether the accumulation is exact or not depends on the precision of input arguments and whether it is feasible to build the hardware required to accumulate exactly.

Burgess, Goodyer, Hinds, and Lutz [12] propose High-Precision Anchored (HPA) accumulators for accurate floating-point summation and suggest extensions to ARM Scalable Vector Extension (SVE) units to efficiently support them. The HPA number format would be used for computation, while the input and output data would still be in an IEEE 754 format, such as the binary64, which requires conversion to and from the HPA. The main concept is to convert a floating-point value into the HPA format by placing different parts of the significand into different registers, based on the significance of the bits when taking into account the exponent. HPA numbers are therefore stored in a wider format, across multiple registers. Similar to fixed-point representation, scaling is applied to choose the balance between the range and precision. HPA numbers can be configured to calculate correctly rounded sums of floating-point values, therefore we classify this software-hardware concept in the Class I of multi-term adders.

1.2.2 Class II: Adders that achieve correct rounding without the use of long accumulators

Tenca [13] provides an optimized algorithm for finding the largest exponent and choosing the amounts to shift the significands by. The general algorithm is not changed, however, with the main steps in Algorithm 1.2 still present. Tenca [13] actually proposes a fused design for performing $\mathrm{fl}(a+b+c)$ with only one rounding error, which complicates the problem in that bits that are shifted out in the significand alignment step have to be tracked. This is not what is implemented in the hardware; for example in the A100, which we used for the demonstration above, $n-1$ rounding or truncation errors are incurred when aligning and adding significands in limited precision.

Sohn and Swartzlander propose a series of fused operators, such as a two-term dot product [14], a three-term adder [15], and a four-term dot product [16]. A generalized n-term fused dot product architecture is explored by Tao et al. [17]. The goal is to implement fused operations, meaning that computation is not performed through standard hardware multipliers and adders joined together, but by making a new optimized unit without the intermediate rounding and normalization steps. Since these operators are fused, we do not expect non-monotonicity to appear when computing with them.

Multi-term adders also appear is in the hardware designs of the fast Fourier transform (FFT) operation. Swartzlander and Saleh [18] utilize a two-term adder for implementing a fused two-term dot product while Kaivani and Ko [19] discuss an implementation of FFT for which a five-term floating-point adder was used. The authors mention not using intermediate normalization and rounding blocks by implementing a custom-design five-term adder, which in turn allowed to reduce the area of the FFT design. This 5-term adder is fused, but it is not specified what the accumulator's size is and how the sticky bits are computed to replicate the exact accumulation.

A generalized algorithm by Boldo, Gallois-Wong, and Hillaire [20] computes a correctly rounded dot product of a series of fixed-point numbers with varied precisions. Instead of using a long accumulator that could cover all possible values, the algorithm uses some number of extra bits and round-odd rounding mode.

1.2.3 Class III: Adders that replicate software behaviour

Kim and Kim [21] propose a 4-term dot product unit without the intermediate normalization of sums but with intermediate rounding performed in correct places (by taking into account where the most significant nonzero bit is) to assure bit reproducible operation compared with an IEEE 754 software implementation. Even though the monotonicity is not addressed in this work, the implementation should be monotonic as it mimics a software implementation with the correctly rounded elementary operations. The application space is 3D graphics—for this a 4-term dot product in single precision is particularly useful and this is what the authors explored. It was noted that "*the exact bit-level matching between hardware units and software models is more important in*

3D graphics than the rounding errors to the real value." [21, p. 892] as the motivation for performing rounding in the intermediate calculations.

1.2.4 Class IV: Adders that use limited precision accumulator

In machine learning, Kaul et al. [22] discuss a generalized n-term dot product hardware design. It is proposed to split the calculation of the maximum exponent and the differences to all the other exponents into two phases, to reduce the critical path of the design. For the purposes of our study, the main feature of Kaul et al. [22] design is that it implements the behaviour of Algorithm 1.2: it aligns the products relative to the product with the maximum exponent, the alignment right-shifts are in limited precision and the addition is performed with extra bits for carries, with a single normalization step of the sum.

Lopes and Constantinides [23] have designed a configurable dot product unit which was tested on FPGAs for up to 150 terms and compared it with a basic implementation that uses a tree of multipliers and adders. The main feature of the design is that internally it uses a configurable precision fixed-point register to accumulate the products in before normalizing and rounding it to produce the floating-point answer. This design follows the general structure of Algorithm 1.2 with precision growth due to no intermediate normalization (as the authors point out, precision growth allows to avoid overflows), and therefore should be non-monotonic.

Hickmann et al. [24] present a 32×32 matrix multiply accelerator with 16-bit floating-point inputs and 32-bit outputs. The internal accumulation of products is limited to 37 bits. One interesting aspect of this work is that the term *fused* is used, but since only the products are exact, not the accumulation of them, this has a different meaning than the Class I/II designs which perform everything as though the computation is exact and rounded once. Another aspect worth noting is that this design explicitly adds the products a_ib_i of the 32×32 dot product operation before adding their sum to the accumulator c_i in order to reduce the error accumulation when the value in the accumulator is growing in magnitude.

Bertaccini et al. [25] developed a three-term addition unit for performing dot products or sums of floating-point values of various formats. The internal accumulator is expanded twice, after each addition, but it is not reported to be an exact accumulator that would cover all right-shift distances in the alignment of addends.

Lee et al. [26] implemented a four-core mixed-precision AI chip which includes a three-term floating-point adder as part of the FMMA (fused multiply–multiply–accumulate) instruction. The multiplicands are 8-bit floating-point values and the accumulator is 16-bit. The three-term addition is performed in 16-bit precision followed by normalization and rounding.

1.2.5 Adders that lie in multiple classes

Ledoux and Casas [27] proposed a hardware generator of general matrix multiply–accumulate (GEMM) accelerators. The generator is parameterized and provides a choice of numerical formats and the option for setting the sizes of internal accumulators, including making them long accumulators to accumulate products exactly. Irrespective of the setup of the accumulator, rounding and normalization from the internal hardware numerical format to some chosen standard format is performed at the end, once the whole dot product has been computed. The work does not mention the use of multi-term adders and implements GEMM accelerators through the accumulation of values by iterating through the hardware. Nevertheless, in terms of resultant numerical behaviour, this work potentially can generate hardware of classes I and IV listed above.

1.3 Commercial hardware

Table 1 lists hardware that is available and contains multi-term floating-point addition, as part of dot product and matrix operations.

Most of the companies do not provide information on low level numerical hardware details which makes it hard to classify them and say what numerical features, such as rounding and monotonicity, are present. An attempt can be made at deducing some of the features from the numerical results that are obtained when computing on these devices, as demonstrated with NVIDIA V100 GPUs by Hickann and Bradford [34] and with V100, T4, and the A100 by Fasi et al. [6]. Fasi et al. [35] have subsequently demonstrated, through error analysis, that low level features such as rounding can become significant when multiplying matrices with matrix arithmetic hardware.

1.4 Our contributions

In summary, the present manuscript's contributions to computer arithmetic and beyond are three-fold:

1) We identify conditions in which floating-point operations that involve multi-term addition can be non-monotonic—this allows to explain surprising numerical results of some of the commercial hardware and construct tests that can be used to look for non-monotonicity of summation within the vector and matrix operations in hardware devices. We show that Class IV operations are not monotonic, but Class I-III are and provide proofs in each case.
2) We demonstrate various applications that may be impacted.
3) We propose ways to modify architectures that contain units for adding multiple floating-point numbers in order for the computed approximations of sums to be monotonic.
4) The paper acts as a survey of hardware designs that are and are not monotonic, and fills an important gap in the literature by addressing the monotonicity of multi-term addition.

2 Background

2.1 Floating-point representation and arithmetic

We will be using the following IEEE-compliant floating-point systems and properties. A binary floating-point number x has the form $(-1)^s \times m \times 2^{e-p+1}$, where s is the sign bit, p is the precision, $m \in [0,\ 2^p - 1]$ is the integer

TABLE 1
List of devices that contain vector or matrix arithmetic hardware, such as dot product and matrix multiply. In the last column we make a prediction on the class of the multi-term addition based on the available information. †This number is determined only from the H100 whitepaper as no other information is available, to the best of our knowledge, on what inputs tensor cores take at hardware layer; *Two 8-bit floating-point formats are available, one with a 4-bit exponent and a 3-bit significand, and one with a 5-bit exponent and a 2-bit significand; ‡Configurable Floating Point 8-bit data type, with programmable bias.

Year	Device/Architecture	Input formats	Output formats	Multi-term adder terms	Throughput (max)	Predicted class
2016	Google TPUv2 [28]	bfloat16	binary32	-	46 Tflop/s	Class III
2017	Google TPUv3 [28]	bfloat16	binary32	-	123 Tflop/s	Class III
2018	NVIDIA V100	binary16	binary32	5	125 Tflops/s	Class IV
2018	Graphcore IPU1	binary16	binary32	-	125 Tflop/s	-
2020	Google TPUv4i [28]	bfloat16	binary32	4	138 Tflop/s	Class IV
2020	Graphcore IPU2	binary16	binary32	-	250 Tflop/s	-
2020	NVIDIA A100 [5]	bfloat16, binary16, binary64, TensorFloat-32	binary32/64	9	312 Tflop/s	Class IV
2021	AMD MI250X [29]	bfloat16, binary16, binary32, binary64	-	5	383 Tflop/s	-
2021	GroqChip [30]	binary16	binary32	160	188 Tflops/s	Class I or II
2022	NVIDIA H100	8-bit*, bfloat16, binary16, binary64, TensorFloat-32	binary32, binary64	17†	1978.9 Tflop/s	-
2022	Intel Ponte Vecchio [31]	bfloat16, binary16, binary64, TensorFloat-32	-	-	-	-
2016-2022	Intel AMX [32]	binary16	binary32	17	-	Class III
2023	Tesla Dojo [33]	CFP8‡, bfloat16	binary32	8	360 Tflops/s	Class IV

significand, and $e \in [e_{\min},\ e_{\max}]$, with $e_{\min} = 1 - e_{\max}$, is the integer exponent. In order for x to have a unique representation, the number system is *normalized* so that the most significant bit of m is set to 1 if $|x| \geq 2^{e_{\min}}$. Therefore, all floating-point numbers with $m \geq 2^{p-1}$ are normalized. Numbers below the smallest normalized number $2^{e_{\min}}$ in absolute value are called *subnormal numbers*, and are such that $e = e_{\min}$ and $0 < m < 2^{p-1}$. The set of floating-point numbers is denoted by $\mathbb{F}$.

The results of floating-point operations may not be normalized and must be normalized by shifting the significand left or right until it falls within $[2^{p-1},\ 2^{p} - 1]$ and adjusting the exponent accordingly. Those numbers that cannot be normalized in such a way, due to requiring exponents lower than the minimum exponent value, form subnormal numbers.

The IEEE 754 standard for floating-point arithmetic provides a limited set of requirements for reduction operations such as multi-term addition [1, Sec. 9.4] or vector and matrix operations: a particular order of adding the partial sums is not required, and the use of arbitrary precision accumulator is allowed. The standard does not specify: 1) whether this internal format should be normalized after each addition, 2) which rounding mode should be used, and 3) when the rounding should happen. IEEE 754-2019 [1] specifies six rounding modes for various purposes. These requirements provide a lot of freedom in implementation choices and can potentially introduce a wide array of different numerical behaviours. We identified four main classes of algorithms that are present in literature and made their way into various devices (Section 1.2).

2.2 Monotonicity of IEEE 754 arithmetics

In this section we demonstrate a few results about the approximation of a sum computed using the 2-term correctly rounded addition operation [1].

2.2.1 *Rounding*

The default rounding mode of IEEE 754 arithmetics is round-to-nearest ties-to-even (RN) and it can be shown that it is monotonic. Take $x, y \in \mathbb{R}$, $x \leq y$, and two neighbouring floating-point values in some precision-p arithmetic over $\mathbb{F}$, a and b. Assume that x and y lie between a and b such that $a \leq x \leq y \leq b$. Normalization of the floating-point significand [1] followed by rounding $x \in \mathbb{R}$ to $\mathbb{F}$ is denoted by $\mathrm{fl}(x)$ and in this case x and y can be rounded to a or b.

The definition of round-to-nearest does not allow non-monotonic behaviour: since $a \leq x, y \leq b$ and $x \leq y$, $\mathrm{fl}(x) \leq \mathrm{fl}(y)$ because $a = \mathrm{fl}(y) < \mathrm{fl}(x) = b$ contradicts the definition of round-to-nearest [1].

Other IEEE 754 [1] rounding modes can also be shown to preserve monotonicity: round-toward-zero (RZ), round-toward-negative (RD) and round-toward-positive (RU) are all monotonic because when they are used, $\mathrm{fl}(x) = \mathrm{fl}(y) = a$ or $\mathrm{fl}(x) = \mathrm{fl}(y) = b$.

2.2.2 *Addition of two operands*

Now we look at $f_{\mathrm{add}}(x, y) = \mathrm{fl}(x+y)$ where as per IEEE 754, $x + y$ is computed as though in infinite precision arithmetic and then rounded to the nearest value in some $\mathbb{F}$. Take a, b, and $c > b$, $s_1 = a+b$, and $s_2 = a+c$, then $s_1 < s_2$. Using the same approach as in Section 2.2.1 we can show that $\mathrm{fl}(s_1) \leq$

$\mathrm{fl}(s_2)$ and therefore that the addition of two operands in IEEE 754 arithmetics is also a monotonic function.

Theorem 2.1. *Addition of two operands,* $\mathrm{fl}(x+y)$ *with* $x, y \in \mathbb{R}$, *computed using the addition operation as defined in the IEEE 754 is monotonic with round-to-nearest, round-towards-zero, round-toward-negative, and round-toward-positive.*

Proof. Since the addition in IEEE 754 arithmetics is first performed as though in infinite precision, the computed quantities will be $\mathrm{fl}(s_1)$ and $\mathrm{fl}(s_2)$. Since $s_1 < s_2$ the reasoning is equivalent to that of Section 2.2.1 and $\mathrm{fl}(s_1) \leq \mathrm{fl}(s_2)$. □

2.2.3 Addition of three or more operands

Multi-term addition in IEEE 754 floating-point arithmetics is also monotonic. Let $x_1, \ldots, x_n \in \mathbb{R}$. Consider $\mathrm{fl}(\mathrm{fl}(x_1 + x_2) + x_3)$. Take $x_3 = a$ and set $s_1 = \mathrm{fl}(x_1 + x_2) + a$. Then take $x_3 = a+\varepsilon$, with $\varepsilon > \mathrm{ulp}(a)/2$, where $\mathrm{ulp}(a) = 2^{e_a - p+1}$ is the size of the gap between a and the following floating-point number, and set $s_2 = \mathrm{fl}(x_1+x_2)+a+\varepsilon$. Since $a+\varepsilon > a$ we get that $s_1 < s_2$ and we can show that $\mathrm{fl}(\mathrm{fl}(x_1+x_2)+x_3)$ is monotonic by showing that $\mathrm{fl}(s_1) \leq \mathrm{fl}(s_2)$ is, using the reasoning in Sec. 2.2.2.

This can be repeated for showing that $\mathrm{fl}(\mathrm{fl}(\cdots \mathrm{fl}(x_1 + x_2) + \cdots) + x_n)$ is also monotonic, and when any of the addends is increased, not necessarily the last one.

Theorem 2.2. *Summation* $\sum_i^n x_i$, *with* $x_i \in \mathbb{R}$ *and* $n \geq 3$, *computed using the floating-point addition operation as defined in the IEEE 754 is monotonic with round-to-nearest, round-towards-zero, round-toward-negative, and round-toward-positive, in any ordering, such as* $\mathrm{fl}(\mathrm{fl}(\cdots \mathrm{fl}(x_1 + x_2) + \cdots) + x_n)$.

Proof. First, consider computing $x_1 + x_2 + \cdots + x_n$ recursively as $\mathrm{fl}(\mathrm{fl}(\cdots \mathrm{fl}(x_1+x_2)+\cdots)+x_n)$. Then, consider increasing the last addend x_n. We can define the partial sum of the first $n-1$ addends as $s_p = \mathrm{fl}(\cdots \mathrm{fl}(x_1 + x_2)\cdots) + x_{n-1})$. Then consider two cases, $s_1 = \mathrm{fl}(s_p + x_n)$ and $s_2 = \mathrm{fl}(s_p + (x_n + \varepsilon))$. Then $s_1 \leq s_2$ through the result in Section 2.2.1.

Secondly, we can check what can happen when any of the addends x_i for $1 \leq i \leq n-1$ are increased before the sum is computed.

Take j to be the index of an addend which we modify. Let

$$x'_i = \begin{cases} x_i + \varepsilon, & i = j, \\ x_i, & i \neq j, \end{cases}$$

and define the partial sums as

$$s_i = \begin{cases} \mathrm{fl}(x_1), & i = 1, \\ \mathrm{fl}(s_{i-1} + x_i), & 2 \leq i \leq n, \end{cases}$$

and

$$s'_i = \begin{cases} \mathrm{fl}(x'_1), & i = 1, \\ \mathrm{fl}(s'_{i-1} + x'_i), & 2 \leq i \leq n. \end{cases}$$

We need to prove that $s'_n \geq s_n$. If $i < j$, then $s'_i = s_i$. Using Theorem 2.2 and the fact that $x'_j \geq x_j$ we can conclude that

$$s'_j = \mathrm{fl}(s'_{j-1} + x'_j) = \mathrm{fl}(s_{j-1} + x'_j) \geq \mathrm{fl}(s_{j-1} + x_j) = s_j.$$

For $j < i \leq n$, the result follows by induction:

$$s'_i = \mathrm{fl}(s'_{i-1} + x'_i) = \mathrm{fl}(s'_{i-1} + x_i) \geq \mathrm{fl}(s_{i-1} + x_i) = s_i.$$

The proof is analogous for other orderings of evaluation of the sum. □

An anonymous referee has pointed out that each ordering can be represented by a tree with vertices representing rounded operations. For each ordering we need two trees, one with and one without the ε update to one of the addends x_j. Similarly to the proof above, we can prove monotonicity for each operation and use induction to prove the monotonicity at all levels of the tree as the expression is being computed. This would allow us to confirm that $s'_n \geq s_n$ which are at the roots of the trees.

2.2.4 Multiplication

Now we look at $f_{\mathrm{mul}}(x, y) = \mathrm{fl}(x \times y)$ where as per IEEE 754, $x \times y$ is computed as though in infinite precision arithmetic and then rounded to the nearest value in some $\mathbb{F}$. Take a, b, $c > b$, $m_1 = a \times b$, and $m_2 = a \times c$. If $a > 0$, $m_1 < m_2$ (multiplication is monotonic increasing). If $a < 0$ we have $m_1 > m_2$ (monotonic decreasing).

Theorem 2.3. *Multiplication of two operands,* $\mathrm{fl}(x \times y)$ *with* $x \in \mathbb{R}$ *and* $y \in \mathbb{R}$, *computed using the floating-point multiplication operation as defined in the IEEE 754 is monotonic with round-to-nearest, round-towards-zero, round-toward-negative, and round-toward-positive.*

Proof. Since the multiplication in IEEE 754 arithmetics is first performed as though in infinite precision, the computed quantities will be $\mathrm{fl}(m_1)$ and $\mathrm{fl}(m_2)$. Since $m_1 < m_2$ the reasoning is equivalent to that of Section 2.2.1 and $\mathrm{fl}(m_1) \leq \mathrm{fl}(m_2)$. The proof is analogous for $a < 0$ which gives $\mathrm{fl}(m_1) \geq \mathrm{fl}(m_2)$. □

Theorem 2.4. *The inner product of column vectors* $a, b \in \mathbb{R}^n$, $a^T b$ *, computed using the floating-point multiplication and addition operations as defined in IEEE 754 with round-to-nearest, round-towards-zero, round-toward-negative, and round-toward-positive is monotonic for any ordering, such as* $\mathrm{fl}(\cdots \mathrm{fl}(\mathrm{fl}(a_1 \times b_1) + \mathrm{fl}(a_2 \times b_2)) + \cdots + \mathrm{fl}(a_n \times b_n))$.

Proof. The proof follows from the monotonicity of the scalar multiplication (Theorem 2.3) and the monotonicity of the n-term sum (Theorem 2.2). □

Since matrix multiplication is comprised of inner products, elementwise monotonicity results from the monotoncicity of the inner product operation. This proves the monotonicity properties of the units that lie in Class III (Section 1.2.3), which implement multi-term addition hardware to mimic the behaviour of IEEE 754, equivalent to normalizing and rounding after every operation.

2.3 Fused multi-term adders

Theorem 2.5. *Summation using fused multi-term adders which perform addition as though in infinite precision and then round once, is monotonic.*

Proof. Since fused multi-term adders compute as though the overall sum is computed in infinite precision and rounded once, $\mathrm{fl}(x_1 + x_n + \cdots + x_n)$, they are monotonic due to monotonicity of rounding, showed in Section 2.2.1. □

This proves the monotonicity properties of the Class I/II units (Sections 1.2.1 and 1.2.2).

3 Results

In this section we prove a few results about the Class IV multi-term floating-point adders (Section 1.2.4).

3.1 Modified IEEE 754 arithmetics: addition without normalization

We need a modified floating-point addition model to describe Class IV multi-term addition units with precision growth. We take the normalized significand of a floating-point number to be $2^{p-1} \leq m < 2^p$ [1]. In the binary representation of m the binary point is defined to be between the first and second left-most bits of m [1]. We now consider a modified version of IEEE 754 addition operation without this constraint, meaning that the normalization step in the addition is not performed. Specifically, we will focus on the normalization that requires the right-shift of the significand by one step (Figure 2). Namely, instead of having one bit to the left of the binary point we assume there are multiple bits for carries to propagate when the result of the partial summation reaches or crosses the powers of two. Equivalently, we can keep the normalization but add one bit of precision if the sum reaches the next power of two.

Take $a, b \in \mathbb{R}$. If $|a| < |b|$ swap them so that in general we assure $|a| \geq |b|$. Define $t = 2^{1+\lfloor \log_2 |a| \rfloor}$: this finds the absolute value of the power of two nearest to $|a|$ with $|a| < |t|$. Then the adder with precision increase (which describes an adder without the right-shift normalization) can be defined as

$$\mathrm{flr}(a+b) = \begin{cases} \mathrm{fl}_p(a+b) & \text{if } |a+b| < t, \\ \mathrm{fl}_{p+1}(a+b) & \text{if } |a+b| \geq t. \end{cases} \tag{2}$$

When we use this adder multiple times, for example to compute $\mathrm{flr}(\mathrm{flr}(x_1 + x_2) + x_3)$ any precision increase in the first adder is propagated into the next adder. Therefore this expression can grow precision from p to $p+1$, while n additions could grow precision from p to $p + \lceil \log_2 n \rceil$—we call this precision growth after each addition in a multi-term summation a *gradual precision growth*. In practice the final result may also be rounded to some desired target precision: $\mathrm{fl}(\mathrm{flr}(x_1 + x_2))$, $\mathrm{fl}(\mathrm{flr}(\mathrm{flr}(x_1 + x_2) + x_3))$, and so on. As an aside, this double rounding can cause issues with accuracy of the final result [36], [37], but this is not the cause of the non-monotonicity and is not addressed further in this paper. This model of addition does not model the lack of left-shift normalization (Figure 2); it is not required for the purposes of this article.

A similar device was used by Ashenhurst and Metropolis [38, p. 418] for error analysis of unnormalized floating-point arithmetic.

3.2 Monotonicity of the modified addition

It can be shown using the similar reasoning as in Section 2.2 that $\mathrm{fl}(\mathrm{flr}(x+y))$ and $\mathrm{fl}(\mathrm{flr}(\mathrm{flr}(x_1+x_2)+x_3))$ are monotonic.

Theorem 3.1. *Addition of two operands,* $\mathrm{flr}(x + y)$ *with* $x \in \mathbb{R}$ *and* $y \in \mathbb{R}$*, is monotonic with round-to-nearest, round-towards-zero, round-toward-negative, and round-toward-positive.*

Proof. First, if the internal adder does not grow precision, the final rounding does not have any effect and the summations are monotonic as shown in Section 2.2. Let us consider the monotonicity of $\mathrm{fl}(\mathrm{flr}(x + y))$ when the precision grows by one bit. Take $s_1 = \mathrm{flr}(a+b)$ and $s_2 = \mathrm{flr}(a+c)$ with $c > b$. Due to monotonicity of rounding, $s_1 \leq s_2$, and therefore $\mathrm{fl}(s_1) \leq \mathrm{fl}(s_2)$. □

Theorem 3.2. *Addition of three operands,* $\mathrm{flr}(\mathrm{flr}(x_1 + x_2) + x_3)$ *with* $x_i \in \mathbb{R}$*, is non-monotonic with round-to-nearest, round-towards-zero, and round-toward-negative (*$x_i > 0$*) or round-toward-positive (*$x_i < 0$*), except if the final rounding* $\mathrm{fl}(\mathrm{flr}(\mathrm{flr}(x_1 + x_2) + x_3))$ *to the starting precision is performed.*

Proof. Take a, b, and c where b is a power of two, a is the floating-point number preceding b, and c is the floating-point number following b, such that $a < b < c$. We consider positive values, but the proof for negative values is analogous. Also, take $\varepsilon = \frac{c-b}{2}$. Then, $\mathrm{flr}(b + \varepsilon) = b$ with RN, RD, and RZ, as is $\mathrm{flr}(\mathrm{flr}(b + \varepsilon) + \varepsilon) = b$. However, $\mathrm{flr}(a + \varepsilon) = b$ and precision grows by one bit. Due to precision growth, $\mathrm{flr}(\mathrm{flr}(a + \varepsilon) + \varepsilon) > b$. Therefore monotonicity is not preserved. However, the final rounding $\mathrm{fl}(\mathrm{flr}(\mathrm{flr}(a + \varepsilon) + \varepsilon)) = b$ and overall the monotonicity is preserved. With RU monotonicity is present even without the final rounding. □

However, as we now show, a sum that includes $n > 3$ terms computed with non-normalized additions modelled by Equation 2 can be non-monotonic in general.

Theorem 3.3. *Summation* $\mathrm{flr}(\cdots \mathrm{flr}(x_1+x_2)+\cdots)+x_n)$*, with* $x_i \in \mathbb{R}$ *and* $n \geq 4$ *is not monotonic with round-to-nearest, round-towards-zero, and round-toward-negative (*$x_i > 0$*) or round-toward-positive (*$x_i < 0$*), with and without the final rounding to the starting precision.*

Proof. Take three consecutive positive floating-point values in some precision-p arithmetic, a, b, and c with b a power of two. Then consider evaluating a 4-term summation $\mathrm{flr}(\mathrm{flr}(\mathrm{flr}(x + \varepsilon) + \varepsilon) + \varepsilon)$ with $x, \varepsilon > 0$ (similar example can be shown for $x, \varepsilon < 0$). In precision-p arithmetic, with round to nearest ties to even, we can show that $\mathrm{flr}(b+\varepsilon) = b$ for $\varepsilon \leq (c-b)/2$, while in precision-$(p+1)$ arithmetic $\mathrm{flr}(b+\varepsilon) = b$ for $\varepsilon \leq (c-b)/4$. Also, in precision-p arithmetic $a + (c-b)/2 = b$.

Take $\varepsilon = (c-b)/2$ and consider two cases.

1) $x = b$, then $\mathrm{flr}(\mathrm{flr}(\mathrm{flr}(b + \varepsilon) + \varepsilon) + \varepsilon) = b$ (all in precision-p).
2) $x = a$, then the first addition $\mathrm{flr}(a + \varepsilon) = b$ (and precision increases to $p+1$ since b is a power of two). Following that, the second addition $\mathrm{flr}(b+\varepsilon) = b+\varepsilon$ as well as the third addition $\mathrm{flr}(b+\varepsilon+\varepsilon) = c$ (since we are in precision-$(p+1)$).

Since the sum evaluates to b when $x = b$ and to c when $x = a < b$, we have shown that the 4-term sum in this modified arithmetic is non-monotonic. The final rounding would not change the result because $\mathrm{fl}(\mathrm{flr}(b + \varepsilon + \varepsilon)) = c$ since 2ε is a value stored in the bits to the left of the rounding point. □

Corollary 3.3.1. *The inner product of vectors* $a, b \in \mathbb{R}^n$*,* $\mathrm{fl}(\cdots \mathrm{flr}(\mathrm{fl}(a_1 \times b_1) + \mathrm{fl}(a_2 \times b_2)) + \cdots + \mathrm{fl}(a_n \times b_n))$ *for* $n \geq 4$*, with round-to-nearest, round-toward-zero, and round-toward-negative (*$a_i \times b_i > 0$*) or round-toward-positive (*$a_i \times b_i < 0$*) is non-monotonic.*

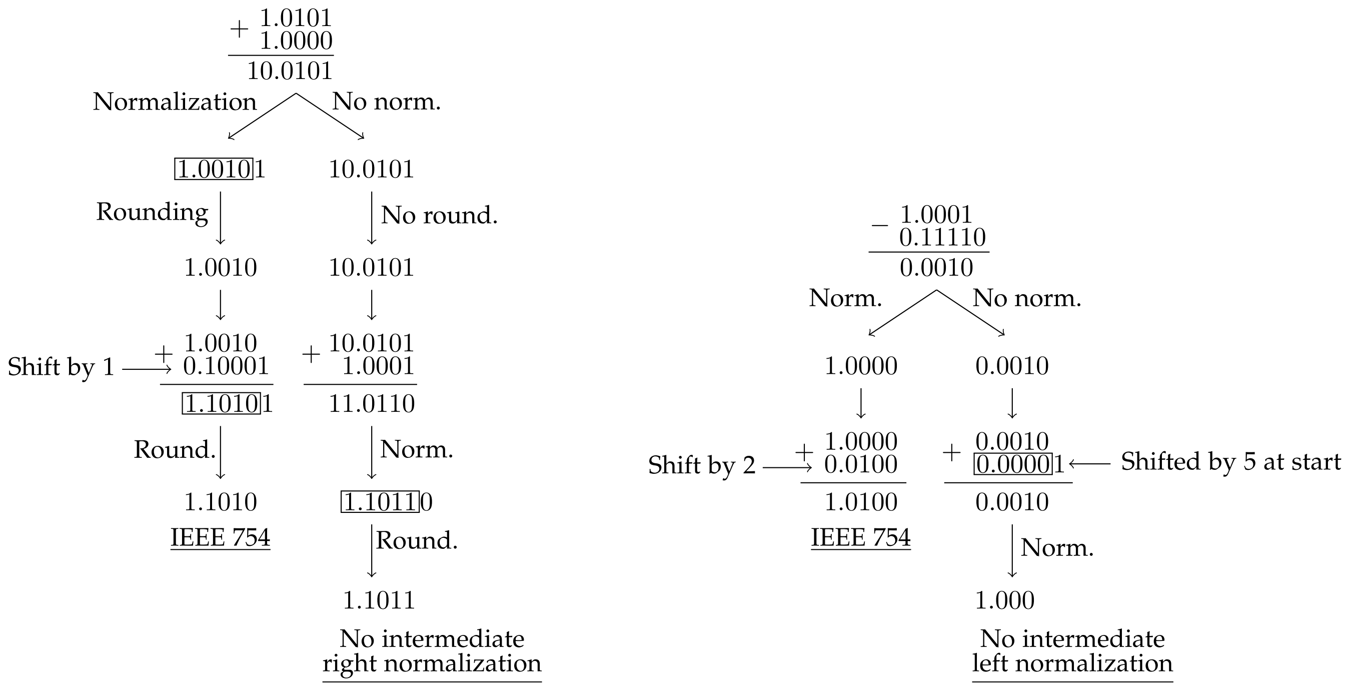

Fig. 2. Example summation of three precision-p numbers (significands showed) in IEEE 754 arithmetic and a Class IV multi-term adder without the intermediate normalization and rounding. In the multi-term adder the carry bits on the left are kept but the bits past precision p in the fraction are discarded; the normalization and rounding steps are performed at the end, after all the addition operations have been completed. We take $p = 5$. On the left is the case in which the significand grows and requires a right-shift to be normalized. On the right is the case with a significant cancellation [2, p. 242] which requires multiple left-shifts to normalize. Notice that the former improves the accuracy of the second addition operation, while the latter makes it worse for the multi-term adder compared with the sum computed using IEEE 754 2-term addition operations. IEEE 754 arithmetic uses round-to-nearest even-on-ties in this example. The monotonicity issue is caused by the lack of right-shift normalization and Equation 2 models the adder that lacks only this normalization in order to simplify.

Proof. Set all elements of b to 1. Then the proof of Theorem 3.3 concludes this proof. □

Corollary 3.3.2. *Matrix-vector multiplication Ax and matrix-matrix multiplication AB, where $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $B \in \mathbb{R}^{n\times l}$, $n \geq 4$, with round-to-nearest, round-toward-zero, and round-toward-negative ($a_{ik} \times b_{kj} > 0$) or round-toward-positive ($a_{ik} \times b_{kj} < 0$) are element-wise non-monotonic.*

Proof. Each element of the output vector or matrix is computed by the inner product. □

3.3 Impact of the order of addition

Class IV multi-term adders align all input significands relative to the largest magnitude addend. This is performed so that all the significands can be right-shifted, with lower order bits dropped or rounded. If the alignments were performed relative to an addend of arbitrary choice, shifting both to the left and right would be required. When the shifts are to the left, shifted bits cannot be dropped and would have to be preserved, which would introduce a high hardware cost similar to Class I/II adders. As many bits from the left shifts would have to be preserved as the highest difference between the exponents of addends.

After the significands are aligned relative to the largest magnitude addend's exponent, if there is no normalization of intermediate sums, the addition acts like the addition of fixed-point values and is associative; the order of performing additions in such a method will not impact the final result. Whether all the addends are added in series or in parallel also does not have any impact on the final result.

When adding a series of positive floating-point values, doing so in an increasing rather than a decreasing order in absolute value reduces the worst-case error bound [3, Sec. 4.2]; this ordering may not reduce the *actual error* [39, Sec. 2]. When considering both positive and negative numbers, the decreasing order can yield better accuracy in the face of cancellation [39, Sec. 2]. However, the hardware starts at the largest magnitude addend, as discussed above; therefore it can be interesting to check the accuracy of the summation, which we do in the following section. Note that that when there is no intermediate normalization implemented, the addition is in effect growing precision, which may improve the accuracy.

4 Numerical experiments

We have simulated the Class IV multi-term adders with the gradual precision growth in MATLAB, using the custom precision simulator `CPFloat` [40]. The code for reproducing the results is available[2].

4.1 Order of addends and associativity

In Figure 3 we plot relative errors $\frac{|s_n - \widehat{s_n}|}{s_n}$ for adding n floating-point values in binary16 arithmetic ($\widehat{s_n}$), with

[2] https://github.com/north-numerical-computing/multi-operand-add-monotonicity

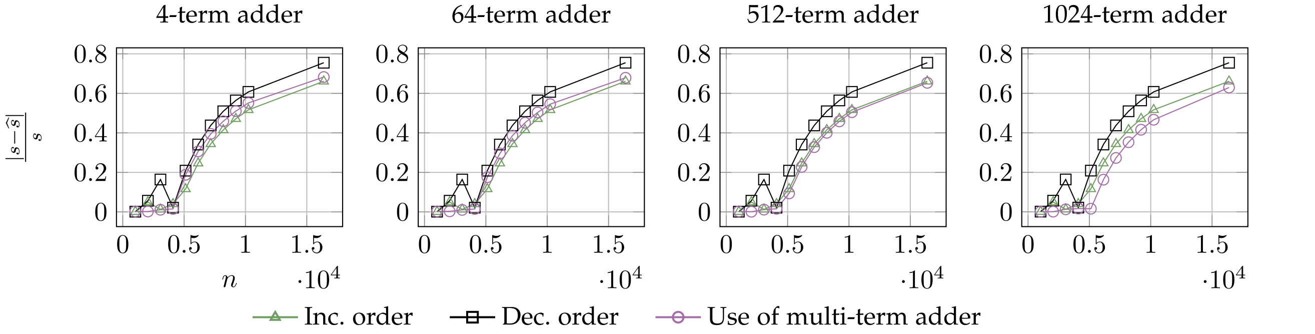

Fig. 3. Summation of positive random binary16 vectors of increasing length. Three summation algorithms are used: recursive summation in increasing order of magnitude using the binary16 IEEE 754 arithmetic, recursive summation in decreasing order of magnitude using the binary16 IEEE 754 arithmetic, and recursive blocking summation using Class IV multi-term adders of various sizes. Relative errors are measured by comparing with the summation of the same values in binary64 arithmetic.

$p = 11$ and $e_{max} = 15$, compared with the sum in binary64 arithmetic (s_n). The addends are pseudo-random numbers generated in MATLAB using the *mrg32k3a* random number generator with a seed of 500, in the range $(0, 0.001)$. We vary n and perform summations with recursive summation algorithm with addends in the decreasing and increasing orders, as well as in the original order with the multi-term adders with terms 4, 64, 512, and 1024. As expected, the decreasing ordering results in the largest errors. In most cases, multi-term adders are worse or very close to recursive summation with increasing ordering of addends. However, with the 1024-term adder, sums become more accurate. This can be explained through precision growth—with the larger adders there is a possibility for more precision growth, which improves accuracy. In this experiment we observed precision to grow to the total of 19 bits, from the default 11.

In `test_associativity.m` we generated a vector of 64 pseudo-random binary16 values, randomly permuted them 10^4 times and each time computed the sum in IEEE 754 arithmetic and the model of a 64-term Class IV adder. Checking the range of computed sums we found that it is non-zero for the IEEE 754 arithmetic and zero for the 64-term Class IV adder, confirming non-associativity and associativity, respectively.

4.2 Monotonicity

For the purposes of demonstrating non-monotonicity of the Class IV adders, we compute

$$\mathrm{fl}(\cdots \mathrm{fl}(x_1 + x_2) + \cdots) + x_n)$$

and

$$\mathrm{fl}(\mathrm{flr}(\cdots \mathrm{flr}(x_1 + x_2) + \cdots) + x_n))$$

in three small floating-point systems: $p = 3$, $e_{max} = 3$; $p = 4$, $e_{max} = 3$; and $p = 5$, $e_{max} = 4$. We construct the sum for severe non-monotonicity to appear, as follows. First, we set all $x_i = 0.25$ and then vary x_1 by changing it to the adjacent floating-point value towards $+\infty$ until all representable values are covered. On each iteration we sum the values x_i with the two different addition models, the multi-term adder with precision growth (Class IV) and the IEEE 754 adder with normalization and rounding after each addition operation (Class III). We report the value of the sum as well as the relative error compared with the same sum performed in binary64 arithmetic. The results are plotted in Figure 4.

First, consider the first column of diagrams in Figure 4. In the top diagram, we see that from the beginning the sum saturates to some quantity and for a while stagnates with the IEEE 754 arithmetic. Relative error starts increasing. When the sum reaches this point, all remaining addends (set to 0.25) are rounded down and do not contribute to the sum. With the multi-term adder this does not occur because precision grows on powers of two, allowing the sum to keep changing as x_1 is being increased. At a certain point, when x_1 crosses the value at which the sum stagnates, the overall value of the sum becomes x_1 and both arithmetics align. At the beginning of this, non-monotonicity appears in the multi-term addition. As precision is increased (rows of the matrix of diagrams in Figure 4), the point at which the IEEE 754 stagnates, and the point at which the multi-term addition shows non-monotonicity, moves to higher values of the sum.

Other columns in Figure 4 correspond to the larger number of terms being added. The main observation is that with more terms the severity of non-monotonicity increases because a larger number of terms can grow the sum more in the range where IEEE 754 arithmetic stagnates and the multi-term adder grows precision.

5 Discussion

5.1 Where non-monotonicity can cause issues

In this subsection we discuss various algorithms that may be impacted by the non-monotonicity of floating-point arithmetic.

In a 1967 paper, Forsythe [41] notes that despite the lack of associativity and distributivity in floating-point addition and multiplication, one can do good analysis provided only that the arithmetic is monotonic. At that time monotonicity was not always preserved in arithmetics, like it was after the IEEE 754 was released: *"Such properties seem elemental but they are extremely helpful. And they are surprisingly absent!" [41, Sec. 4].*

Demmel, Dhillon, and Ren [42] explore the bisection algorithm for finding eigenvalues of real symmetric tridiagonal matrices. The algorithm relies on the $\mathrm{count}(x)$ to

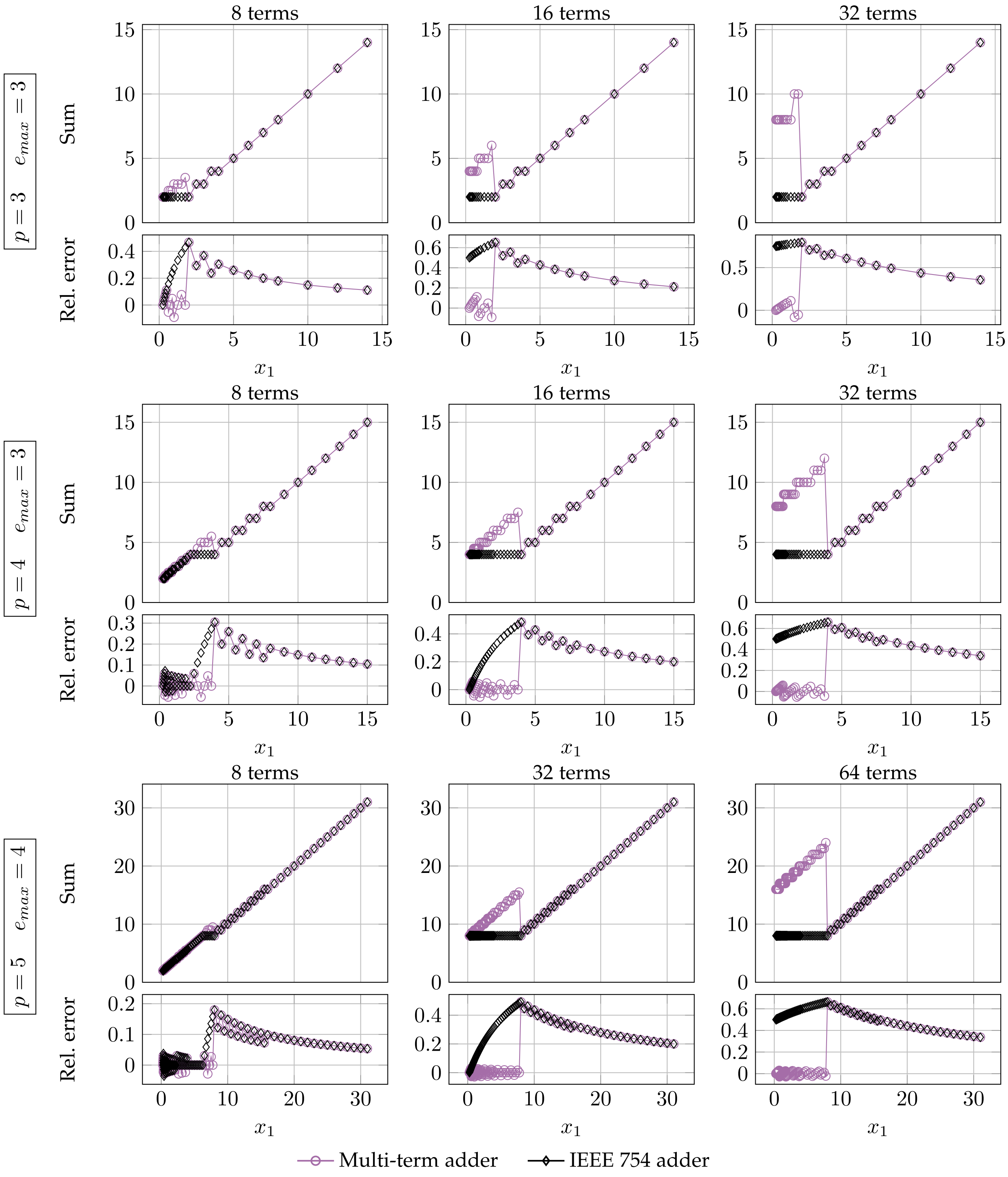

Fig. 4. Summation with various floating-point formats and two types of addition: multi-term addition with precision growth (Class IV) and IEEE 754 addition with normalization and rounding after each operation. Value of of the sum (top of each subdiagram) and the relative error (bottom) compared with the sum performed in the binary64 arithmetic. The x-axis corresponds to the quantity of x_1 which we vary, with the rest of $x_i = 0.25$.

Algorithm 5.1: $\mathrm{Count}(x, T)$: return the number of eigenvalues of a real symmetric tridiagonal matrix T.

```
$count \leftarrow 0$;
$d \leftarrow 1$;
for $i \leftarrow 1$ to $n$ do
    $d \leftarrow a_i - x - \frac{b_{i-1}^2}{d}$;
    if $d < 0$ then
        $count \leftarrow count + 1$;
return $count$;
```

count the number of eigenvalues that are smaller than x (Algorithm 5.1). This function is monotonic, but when implemented in floating-point arithmetic, various implementation details can introduce non-monotonicity, which in turn can cause wrong results by finding negative number of eigenvalues $(\mathrm{count}(x_b) - \mathrm{count}(x_a)) < 0$ over a specific range x_a to x_b with $x_b > x_a$. We have shown that non-monotonicity can appear in the three-term adder if no final rounding is performed (Theorem 3.2), which would invalidate the Assumption 1A [42, p 120] and introduce a source of non-monotonicity in an implementation of $\mathrm{count}(x)$.

Higham [3, Sec. 2.6] demonstrates that monotonicity of rounding may be an important property when solving a quadratic equation $ax^2 - 2bx + c = 0$. Computing an expression $\sqrt{b^2 - ac}$ is required and if $b^2 = ac$ but $\mathrm{fl}(b^2) < b^2$, computing $\mathrm{fl}(\mathrm{fl}(b^2) - ac)$ with an FMA can result in a negative number passed into the square root function. While this would not require a 4-term or wider adder, which we explore in this paper, if an expression $\sqrt{\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} b_i}$, where a and b are vectors, which uses the multi-term adder twice and then subtracts the sums, appears in applications, a similar issue to that discussed by Higham can appear. We provide an example in `test_sqrt.m`. We use a binary32 arithmetic with $p = 24$ and $e_{max} = 127$, and $n = 8$. Then take

$$a = [1, 1, 1, 1, 1, 1, 1, 16777216]$$

and

$$b = [1, 1, 1, 1, 1, 1, 1, 16777214].$$

Computing sums recursively left to right with IEEE 754 additions in order we get $\sqrt{4}$, while computing them with the 8-term Class IV adder we get $\sqrt{-4}$ due to the non-monotonicity in the sum. The correct result, computed in binary64 arithmetic, is $\sqrt{2}$. Note that with the IEEE 754 binary32 arithmetic we can change the order of addends and impact the result:

$$a = [16777216, 1, 1, 1, 1, 1, 1, 1]$$

$$b = [16777214, 1, 1, 1, 1, 1, 1, 1]$$

gives $\sqrt{0}$, but

$$a = [16777216, 1, 1, 1, 1, 1, 1, 1]$$

$$b = [1, 1, 1, 1, 1, 1, 1, 16777214]$$

gives $\sqrt{-4}$. However, as discussed, with the Class IV adder the order does not impact the final result and this example results in $\sqrt{-4}$ irrespective of it; as a result it cannot be fixed at software layer by reordering.

In computing elementary functions often monotonicity is one of the properties that is desired [43]. Silverstein et al. [44] in the report on the UNIX math library `libm`, note: *"Monotonicity failures can cause problems, for example, in evaluating divided diferences. The only function we have observed to violate monotonicity for C/SVR4 is* `lgamma`*. We would not be surprised to learn of others."*. This is of general interest, and we are not aware of a link between the non-monotonicity of multi-term adders and mathematical functions.

Interval arithmetic can be impacted by non-monotonicity of floating-point. Rump [45, Sec. 2.1] gives an example of computing lower and upper limits of the interval of matrix multiplication of point matrices (lower and upper limits are equal) through the change of rounding modes in floating-point arithmetic. Similar to Rump's example, in `test_interval.m` we compute the interval of summation operation, through round-toward-negative and round-toward-positive, for lower and upper limits, respectively. First we sum a vector $a = [16777216, 1, 1, 1, 1, 1, 1, 1]$ which yields an interval $[16777216, 16777230]$. We then decrease a_1 and perform the same for $b = [16777214, 1, 1, 1, 1, 1, 1, 1]$ which yields an interval $[16777220, 16777222]$. By decreasing one of the arguments we expect the interval to shift down the real axis, but the lower end shifts up and the interval becomes narrower due to precision growth in the multi-term adder. As before, it is possible to cause a similar issue with the IEEE 754 arithmetic, by changing the order of the addends.

5.2 Possible solutions

Having gradual precision growth in the dot product can be beneficial in getting more accurate results. However, if monotonicity is needed, a straightforward solution would be to always normalize after each addition to stop the gradual precision growth in the internal accumulator. This can replicate the behaviour of a software implementation of summation based on standard IEEE 754 operations. The hardware cost would most likely be increased substantially.

Another approach could be to detect where precision growth occurs by monitoring the carry-out bits in the significands of the sums and then informing any further additions to cancel an appropriate number of bits from the bottom of the significand and not take them into account when adding. However, this is highly dependent on what particular implementations are doing and would probably impact *guard*, *round* and *sticky* bits which might or might not be used in the intermediate additions inside the multi-term adder, depending on the rounding properties needed. The additional logic for this may make the Class IV adders as expensive as Class I/II adders, but we leave this for a separate hardware-oriented study.

As a summary:

- Class I/II adders perform the summation as though only one rounding error is induced at the end, and with these adders monotonicity and associativity of summation are preserved.
- Class III adders perform the summation which numerically behaves as a software implementation with

the IEEE 754 addition operations, with normalization and rounding after each. With these adders, associativity is not preserved, but monotonicity is.
- With the Class IV adders, monotonicity is not preserverd, as we have demonstrated, but the associativity is.

One can choose an appropriate implementation based on the numerical properties required in a particular architecture.

6 Conclusion

IEEE 754-2019 provides the following guidance for implementing multi-term summation or dot product operations (collectively called *reduction operations*) [1, Sec. 9.4]: "*Language standards should define the following reduction operations for all supported arithmetic formats. Unlike the other operations in this standard, these operate on vectors of operands in one format and return a result in the same format. Implementations may associate in any order or evaluate in any wider format.*". In the present manuscript we analysed how various hardware designs in the literature as well as in the available hardware implement this. We demonstrated that there are four classes of implementation, each with different hardware complexity and numerical properties.

Focusing on Class IV multi-term addition, where the significand alignment of the summands is performed in limited precision and the sum is performed without the normalization, all resulting in precision growth during the summation, we proved that non-monotonicity can occur. We also showed that Class I–III devices are not subject to this. Our results should assist in understanding numerical differences between different implementations and show what is needed in order to preserve the property of monotonicity in floating-point multi-term addition. This applies to dot products and matrix multiplication operations, which use multi-term addition. The results do not necessarily apply only to hardware as the same low level floating-point algorithms could be implemented in software, for example on devices that do not contain floating-point units.

Finally, our results indicate that monotonicity should be considered in the next iteration of the IEEE 754 standard and the new standard, P3109[3], of low-precision floating-point formats. Suitable recommendations for the reduction operations may need to be provided when the preservation of monotonicity is needed.

7 Acknowledgements

The author is grateful to M. Fasi for the help on the proof of Theorem 2.2 and other comments and suggestions, and N. J. Higham, Nicolas Brunie and three anonymous referees for comments and suggestions.

8 Funding

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P020720/1, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The author also acknowledges the support of the School of Computing, University of Leeds.

[3] https://sagroups.ieee.org/p3109wgpublic/

References

[1] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Jul. 2019.

[2] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser, 2018.

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[4] N. Metropolis and R. L. Ashenhurst, "Significant digit computer arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-7, no. 4, pp. 265–267, 1958.

[5] *NVIDIA A100 Tensor Core GPU architecture*. NVIDIA, 2020, NVIDIA whitepaper v1.0. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[6] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical behavior of NVIDIA tensor cores," *PeerJ Comput. Sci.*, vol. 7, pp. e330(1–19), Feb. 2021.

[7] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. De Gruyter, 2013. [Online]. Available: https://doi.org/10.1515/9783110301793

[8] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator," Mar. 2017, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01488916

[9] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, 2017, pp. 114–121.

[10] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, 2017, pp. 106–113.

[11] ——, "Towards the basic linear algebra unit : Replicating multidimensional FPUs to accelerate linear algebra applications," in *2020 54th Asilomar Conference on Signals, Systems, and Computers*, 2020, pp. 1283–1290.

[12] N. Burgess, C. Goodyer, C. N. Hinds, and D. R. Lutz, "High-precision anchored accumulators for reproducible floating-point summation," *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 967–978, 2019.

[13] A. F. Tenca, "Multi-operand floating-point addition," in *2009 19th IEEE Symposium on Computer Arithmetic*, Portland, OR, USA, Jun. 2009, pp. 161–168.

[14] J. Sohn and E. E. Swartzlander, "Improved architectures for a floating-point fused dot product unit," in *2013 IEEE 21st Symposium on Computer Arithmetic*, 2013, pp. 41–48.

[15] ——, "A fused floating-point three-term adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 10, pp. 2842–2850, 2014.

[16] ——, "A fused floating-point four-term dot product unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, pp. 370–378, 2016.

[17] Y. Tao, G. Deyuan, F. Xiaoya, and J. Nurmi, "Correctly rounded architectures for floating-point multi-operand addition and dot-product computation," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 346–355.

[18] E. E. Swartzlander and H. H. Saleh, "FFT implementation with fused floating-point operations," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 284–288, 2012.

[19] A. Kaivani and S.-B. Ko, "Area efficient floating-point FFT butterfly architectures based on multi-operand adders," *Electronics Letters*, vol. 51, no. 12, pp. 895–897, 2015. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/el.2015.0342

[20] S. Boldo, D. Gallois-Wong, and T. Hilaire, "A correctly-rounded fixed-point-arithmetic dot-product algorithm," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, 2020, pp. 9–16.

[21] D. Kim and L. Kim, “A floating-point unit for 4D vector inner product with reduced latency,” *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 890–901, Jul. 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.210

[22] H. Kaul, M. Anders, S. Mathew, S. Kim, and R. Krishnamurthy, “Optimized fused floating-point many-term dot-product hardware for machine learning accelerators,” in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 84–87.

[23] A. Roldao Lopes and G. A. Constantinides, “A fused hybrid floating-point and fixed-point dot-product for FPGAs,” in *Reconfigurable Computing: Architectures, Tools and Applications*, P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–168.

[24] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, “Intel Nervana Neural Network Processor-T (NNP-T) fused floating point many-term dot product,” in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, 2020, pp. 133–136.

[25] L. Bertaccini, G. Paulin, T. Fischer, S. Mach, and L. Benini, “MiniFloat-NN and ExSdotp: An ISA extension and a modular open hardware unit for low-precision training on RISC-V cores,” in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 1–8.

[26] S. K. Lee, A. Agrawal, J. Silberman, M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, M. Cohen, S. M. Mueller, J. Oh, M. Lutz, J. Jung, S. Koswatta, C. Zhou, V. Zalani, M. Kar, J. Bonanno, R. Casatuta, C.-Y. Chen, J. Choi, H. Haynie, A. Herbert, R. Jain, K.-H. Kim, Y. Li, Z. Ren, S. Rider, M. Schaal, K. Schelm, M. R. Scheuermann, X. Sun, H. Tran, N. Wang, W. Wang, X. Zhang, V. Shah, B. Curran, V. Srinivasan, P.-F. Lu, S. Shukla, K. Gopalakrishnan, and L. Chang, “A 7-nm four-core mixed-precision AI chip with 26.2-TFLOPS hybrid-FP8 training, 104.9-TOPS INT4 inference, and workload-aware throttling,” *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 182–197, 2022.

[27] L. Ledoux and M. Casas, “A generator of numerically-tailored and high-throughput accelerators for batched GEMMs,” in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–10.

[28] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, “Ten lessons from three generations shaped google’s TPUv4i : Industrial product,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[29] *AMD Instinct™ MI200 Series Accelerator*. Advanced Micro Devices, Inc., Nov. 2021. [Online]. Available: https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf

[30] D. Abts, G. Kimmell, A. Ling, J. Kim, M. Boyd, A. Bitar, S. Parmar, I. Ahmed, R. DiCecco, D. Han, J. Thompson, M. Bye, J. Hwang, J. Fowers, P. Lillian, A. Murthy, E. Mehtabuddin, C. Tekur, T. Sohmers, K. Kang, S. Maresh, and J. Ross, “A software-defined tensor streaming multiprocessor for large-scale machine learning,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 567–580.

[31] W. Gomes, A. Koker, P. Stover, D. Ingerly, S. Siers, S. Venkataraman, C. Pelto, T. Shah, A. Rao, F. Mahony, E. Karl, L. Cheney, I. Rajwani, H. Jain, R. Cortez, A. Chandrasekhar, B. Kanthi, and R. Koduri, “Ponte Vecchio: A multi-tile 3D stacked processor for exascale computing,” in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 42–44.

[32] Intel Corporation, “Intel architecture instruction set extensions and future features programming reference,” Dec. 2022. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[33] E. Talpes, D. D. Sarma, D. Williams, S. Arora, T. Kunjan, B. Floering, A. Jalote, C. Hsiong, C. Poorna, V. Samant, J. Sicilia, A. K. Nivarti, R. Ramachandran, T. Fischer, B. Herzberg, B. McGee, G. Venkataramanan, and P. Banon, “The microarchitecture of dojo, tesla’s exa-scale computer,” *IEEE Micro*, pp. 1–5, 2023.

[34] B. Hickmann and D. Bradford, “Experimental analysis of matrix multiplication functional units,” in *Proceedings of the 26th IEEE Symposium on Computer Arithmetic*, Kyoto, Japan, Oct. 2019, pp. 116–119.

[35] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis, “Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores,” *SIAM J. Sci. Comput.*, vol. 45, no. 1, pp. C1–C19, Feb. 2023.

[36] P. Roux, “Innocuous double rounding of basic arithmetic operations,” *J. Formaliz. Reason.*, vol. 7, no. 1, pp. 131–142, Jul. 2014.

[37] S. M. Rump, “IEEE754 precision-k base-β arithmetic inherited by precision-m base-β arithmetic for $k < m$,” *ACM Trans. Math. Software*, vol. 43, no. 3, p. 1–15, Jan. 2017.

[38] R. L. Ashenhurst and N. Metropolis, “Unnormalized floating point arithmetic,” *J. ACM*, vol. 6, no. 3, p. 415–428, Jul. 1959.

[39] N. J. Higham, “The accuracy of floating point summation,” *SIAM J. Sci. Comput.*, vol. 14, no. 4, pp. 783–799, Jul. 1993.

[40] M. Fasi and M. Mikaitis, “CPFloat: A C library for simulating low-precision arithmetic,” *ACM Trans. Math. Softw.*, vol. 49, no. 2, Jun. 2023.

[41] G. E. Forsythe, “Today’s computational methods of linear algebra,” *SIAM Rev.*, vol. 9, pp. 489–515, 1967.

[42] J. W. Demmel, I. Dhillon, and H. Ren, “On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic,” *Electron. Trans. Numer. Anal.*, vol. 3, pp. 116–149, 1995.

[43] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, 3rd ed. Boston, MA, US: Birkhäuser, 2016.

[44] J. D. Silverstein, S. E. Sommars, and Y.-C. Tao, “The unix system math library, a status report,” in *USENIX - Winter’90*, 1990.

[45] S. Rump, “INTLAB - INTerval LABoratory,” in *Developments in Reliable Computing*, T. Csendes, Ed. Dordrecht: Kluwer Academic Publishers, 1999, pp. 77–104, http://www.tuhh.de/ti3/rump/.

Mantas Mikaitis received a B.Sc. (Hons.) degree in Computer Science in 2016 and a PhD degree in Computer Science in 2020, both from the University of Manchester, Manchester, United Kingdom. Between 2019 and 2020 he was an EPSRC Doctoral Prize Fellow and between 2020 and 2022 a Research Associate, both within the Numerical Linear Algebra group at the University of Manchester. He is currently a Lecturer at the School of Computing of the University of Leeds, Leeds, United Kingdom. His research interests include various aspects of computer arithmetic and mathematical software.