Algorithms for Efficient Reproducible Floating Point Summation

PETER AHRENS, Massachusetts Institute of Technology, USA
JAMES DEMMEL and HONG DIEP NGUYEN, University of California Berkeley, USA

We define "reproducibility" as getting bitwise identical results from multiple runs of the same program, perhaps with different hardware resources or other changes that should not affect the answer. Many users depend on reproducibility for debugging or correctness. However, dynamic scheduling of parallel computing resources, combined with nonassociative floating point addition, makes reproducibility challenging even for summation, or operations like the BLAS. We describe a "reproducible accumulator" data structure (the "binned number") and associated algorithms to reproducibly sum binary floating point numbers, independent of summation order. We use a subset of the IEEE Floating Point Standard 754-2008 and bitwise operations on the standard representations in memory. Our approach requires only one read-only pass over the data, and one reduction in parallel, using a 6-word reproducible accumulator (more words can be used for higher accuracy), enabling standard tiling optimization techniques. Summing n words with a 6-word reproducible accumulator requires approximately $9n$ floating point operations (arithmetic, comparison, and absolute value) and approximately $3n$ bitwise operations. The final error bound with a 6-word reproducible accumulator and our default settings can be up to 2^{29} times smaller than the error bound for conventional (recursive) summation on ill-conditioned `double`-precision inputs.

CCS Concepts: • **Mathematics of computing** → **Numerical analysis**; **Arbitrary-precision arithmetic**; *Mathematical software*; • **Computing methodologies** → *Parallel algorithms*; *Distributed computing methodologies*;

Additional Key Words and Phrases: Reproducible summation, binned number, binned summation, floating point number, floating point summation, reproducibility, parallel, computer arithmetic, summation

ACM Reference format:
Peter Ahrens, James Demmel, and Hong Diep Nguyen. 2020. Algorithms for Efficient Reproducible Floating Point Summation. *ACM Trans. Math. Softw.* 46, 3, Article 22 (July 2020), 49 pages.
https://doi.org/10.1145/3389360

This research is supported in part by a DOE Computational Science Graduate Fellowship DE-FG02-97ER25308, NSF grants NSF ACI-1339676, NSF DMS-1312831, DOE grants DOE DE-SC0010200, DOE DE-SC0008699, DOE DE-SC0008700, DOE AC02-05CH11231, an Intel ITSC grant, a Darpa XDATA grant, DARPA grant HR0011-12-2-0016, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Other industrial sponsors include Mathworks, Cray, and Aramco. Any opinions, findings, conclusions, or recommendations in this article are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.
Authors' addresses: P. Ahrens, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Room 32-G604, Cambridge, MA, 02139; email: pahrens@mit.edu; J. Demmel, University of California Berkeley, Computer Science Division, Mathematics, 564 Soda Hall, Berkeley, CA, 94720; email: demmel@berkeley.edu; H. D. Nguyen, Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, 94720; email: hdnguyen@eecs.berkeley.edu.

0098-3500/2020/07-ART22 $15.00
https://doi.org/10.1145/3389360

1 INTRODUCTION

Reproducibility, i.e., getting bitwise identical results from multiple runs of the same program, is important for several reasons. First, it can be very hard to debug and test a program if results (including errors) cannot be reproduced. Reproducibility may also be needed for producing correct results, such as simulations that produce rare events that must then be reproduced and studied more carefully, or when a quantity is computed redundantly on different processors and assumed to be identical in subsequent tests and branches. Reproducibility may also be required for contractual reasons when multiple parties must agree on a result (such as a simulation evaluating the earthquake safety of a proposed building design). Finally, reproducibility can be important for replicating previously published results. There have been numerous recent meetings at conferences addressing the need for reproducibility and proposed ways to achieve it [12]. In response to this demand, Intel has introduced a version of their Math Kernel Library (MKL) that supports reproducibility under certain restrictive conditions [2]. NVIDIA's CuBLAS routines are, by default, reproducible under the same conditions [3]. We will see that neither of these solutions meet all our design goals.

There are many potential sources of nonreproducibility, so we need to define the scope of our work. For example, taking the source code for an arbitrary parallel program on one machine and moving it to a different machine with a different compiler or compiler flags, different floating point semantics, and different math libraries (e.g., trigonometric functions) is beyond what we address here [7].

Instead, we limit ourselves to nonreproducibility caused by binary floating point summation (addressing decimal floating point summation is future work). Since roundoff makes floating point summation non-associative, summing the same summands in different orders frequently gives different answers. On a parallel machine with a variable (e.g., dynamically allocated) number of processors, a dynamic execution schedule, or even a sequential machine where different data alignments may cause a different use of SIMD instructions, the order of summation can easily vary from run-to-run or even subroutine-call-to-subroutine-call in the same run.

Our goal is to formally describe a data structure (and associated algorithms) for reproducible floating point summation that could be used in existing software patterns for summation. An **accumulator** is a data structure used to store the intermediate results of summation. A **reproducible accumulator** is an accumulator that can produce a reproducible sum [8–10, 14, 15, 21–23, 26]. Our reproducible accumulator and algorithms are efficient, accurate, and behave sensibly on all floating point values.

A modular reproducible accumulator can be used to construct reproducible versions of higher-order linear algebra operations, including matrix-matrix and matrix-vector multiplication, dot products, and stable norms of vectors. For example, to implement a reproducible dot product $x^T y$, we can sum the products $x_i y_i$ using our reproducible accumulator. Since we will always sum the same results $x_i y_i$, the output will be the same regardless of summation order. We can generalize this algorithm for the dot product to higher-order operations like matrix multiplication using several reproducible accumulators to represent the necessary partial inner products. Performance optimizations used in highly tuned non-reproducible BLAS implementations such as tiling might be performed using tiles of reproducible accumulators that can be added together pointwise. If more accuracy in the dot product is desired, we may use a "TwoMul" operation to multiply x_i and y_i to produce floating point numbers h_i and l_i such that $h_i + l_i = x_i y_i$ exactly, and sum the resulting h_i and l_i using a reproducible accumulator [28]. Since the h_i and l_i do not depend on the order of summation, this result should also be reproducible, and, since the pointwise multiplication was exact, the only error in the dot product will come from summation.

Our work builds on Reference [15], which attained some but not all of our design goals. So in the following summary, we point out which goals were attained in Reference [15] and which ones are attained here for the first time.

(1) *Reproducible summation, independent of summation order, assuming only a subset of the IEEE Floating Point Standard 754.*

Our algorithms, based on Reference [15] and originally inspired by References [30, 31], use only arithmetic operations on binary floating point numbers (any precision) and bitwise operations on their underlying representations in current IEEE Floating Point Standard 754-2008 [1]. We require only a "to-nearest" rounding mode; the default rounding mode (rounding to the nearest value, breaking ties to even values) is sufficient.

In Appendix A, we show how to use a new operation in the new Standard 754-2019 [5, Section 9.5], which was introduced to accelerate both our core algorithm [20, 29] and the widely used two-sum operation [11, 17, 19, 21, 24, 25, 31]. We note that this operation needs to use a new rounding mode, round "to-nearest" with ties broken *toward zero*, that was not previously required for any floating point operations. The new operation changes only the innermost loop.

(2) *Accuracy at least as good as recursive summation, and tunable.*

The size of the reproducible accumulator in Reference [15] may be selected based on the desired accuracy. However, Reference [15] did not give any algorithm to convert this reproducible accumulator to a single floating point result.

Our contribution here is a very simple and nearly optimally accurate algorithm for this conversion, which introduces an additional error in the computed sum S of at most 7 ulps (units in the last place) of the *exact* sum $\sum_{j=0}^{n-1} x_j$. Although we could perform the conversion exactly and produce a correctly rounded result, our algorithm is very efficient. It uses no more floating point operations than the naive conversion algorithm. Since our default accumulator has at least 80 bits of precision, after using our new conversion algorithm, we have

$$\left| S - \sum_{j=0}^{n-1} x_j \right| < n2^{-80} \max |x_j| + 7\epsilon \left| \sum_{j=0}^{n-1} x_j \right|, \tag{6.1}$$

where $\epsilon = 2^{-53}$. In contrast, the sum $S_{\text{naiveconv}}$ computed using a naive conversion algorithm only satisfies

$$\left| S_{\text{naiveconv}} - \sum_{j=0}^{n-1} x_j \right| < n(2^{-80} + 5\epsilon) \max |x_j| \approx 5n\epsilon \max |x_j|, \tag{6.2}$$

which can be over 2^{29} times larger for very ill-conditioned problems (i.e., when there is a great deal of cancellation, such as $n \cdot \max |x_j| \approx 2^{32} |\sum x_j|$ in this example).

The error bound for the conventionally (nonreproducibly) computed sum using **recursive summation** S_{recur} is even larger. By recursive summation, we mean that the sum is computed in the order: $S_{\text{recur}} = 0$, for $j = 0$ to $n-1$, $S_{\text{recur}} = S_{\text{recur}} \oplus x_j$ (where $\oplus$ represents floating point addition, discussed in more detail later).

$$\left| S_{\text{recur}} - \sum_{j=0}^{n-1} x_j \right| < n\epsilon \sum_{j=0}^{n-1} |x_j| \le n^2 \epsilon \max |x_j| \qquad [18, (2.6)]$$

These error bounds are discussed in more detail in Section 6.1.

Appendix A.6 also shows that the arithmetic cost for arbitrarily high precision is roughly the same as using our 6-word reproducible accumulator, plus a term that only grows proportionally to the size of the reproducible accumulator, not the number of summands n.

(3) *Handle overflow, underflow, and other exceptions reproducibly.*

It is easy to construct small sets of finite floating point numbers, where depending on the order of summation, one might compute `+Inf` (positive infinity), `-Inf` (negative infinity), `NaN` (Not-a-Number), 0, or 1 (the correct answer): Consider $[X, X, 1, -X, -X]$ where X is any finite floating point number greater than half the overflow threshold. The reproducible summation algorithm in Reference [15] (Algorithms 6 and 7) could not handle exceptions, very large summands, or very large intermediate results. In contrast, our algorithms guarantee that no overflows can occur until the final rounding to a single floating point number. Summands that are `+Inf`, `-Inf`, or `NaN` propagate in a reproducible way to the final sum.

We note that when we guarantee that a `NaN` will either always, or never, appear as the result of a reproducible sum, we are not making any guarantees about the bitwise representation of the resulting `NaN` being reproducible, just that the result is a `NaN`. This is because the IEEE Floating Point Standard 754 does not specify a unique representation for `NaN` (there are several) or guarantee which operand in the sum `NaN + NaN` will propagate to the result.

Underflows are handled reproducibly by assuming default gradual underflow semantics, but a slightly modified version of our algorithms work with abrupt or gradual underflow.

(4) *One read-only pass over the summands.*

This property is critical for efficient implementation of the BLAS, where summands (like $A_{ik} \cdot B_{kj}$ in matrix multiply $C = A \cdot B$) are computed once on the fly and discarded immediately. The algorithm in Reference [15] has this desirable property.

(5) *One reduction.*

A parallel reduction is an operation that uses a binary operator to combine multiple values (stored on different processors) into one. Any arbitrary reduction tree combining values and intermediates may be used. In parallel environments, the cost of a parallel reduction may dominate the total cost of a sum, so analogously to performing a single read-only pass over the data, it is important to be able to perform one parallel reduction operation (processing the data in any order) to compute a reproducible sum. The algorithm in Reference [15] again has this desirable property.

(6) *Use as little memory as possible to enable tiling.*

Many existing algorithms for reproducible summation represent intermediate sums with a reproducible accumulator. BLAS3 operations like matrix-multiplication require many reproducible accumulators to exploit common optimizations like tiling; otherwise, they are forced to run at the much slower speeds of BLAS1 or BLAS2.

To fit as many of these reproducible accumulators into the available fast memory as needed, they need to be as small as possible. Our default-sized reproducible accumulator occupies 6 `double` precision floating point words, which is small enough for these optimization purposes.

(7) *Use a modular design, so reproducible summation can be applied in a variety of use cases.*

Floating point summation is very common, so we want to make it easy to be reproducible whenever required. Also, in some applications only some parts of a sum may require modification to be reproducible. For example, in a parallel sum where the data

assigned to each processor is known to be fixed, only the parallel reduction of the partial sums from each processor may require modification.

Since we can add floating point numbers to our reproducible accumulator and we can add two of our reproducible accumulators together, our algorithms can be used in most software patterns for summation. Reference [15] also had these algorithms, but lacked a formal description of the accumulator data structure, how it was initialized, and how to convert it back to a floating point number. Our algorithm to add a floating point number to a reproducible accumulator is described in smaller, more modular functions than in Reference [15]. All of our algorithms have clear requirements and guarantees in terms of our reproducible accumulator.

We need to make some assumptions on the maximum number of summands. These turn out to be 2^{64} and 2^{33} for `double` and `single` precision, respectively, which is more than enough in practice, and 4× more than in Reference [15] (see Section 6.2).

Previous alternative approaches to reproducible summation fail to attain all these goals. For example, the Intel Math Kernel Library (MKL) introduced a feature called Conditional Numerical Reproducibility (CNR) in version 11.0 [2]. When enabled, the code returns the same results from run-to-run as long as calls to MKL come from the same executable and the number of computational threads used by MKL is constant. Performance is degraded by these features by at most a factor of 2. However, these routines cannot offer reproducible results if the data ordering or the number of processors changes, violating Goal 1 above. MKL also does not have a distributed memory implementation. NVIDIA's CuBLAS [3] routines are reproducible by default, because they use a deterministic order of summation. This approach is limited in the same ways as MKL with CNR. A **correctly rounded** sum is the sum computed exactly and then rounded to a floating point number. The correctly rounded sum is reproducible by definition, but this approach involves either an accumulator big enough to accumulate all possible floating point sums exactly [8, 10, 26] (violating Goal 6), and/or multiple passes over the data rewriting it to eliminate cancellation [8, 30] (violating Goal 4). Pre-rounding the sum to a fixed precision before summing exactly can reduce the necessary size of the superaccumulator used in exact summation; however, this also requires more than one pass over the data [14, 27, 31], violating Goal 4. A hardware instruction for computing pre-rounded sums is presented in Reference [27], and while pre-rounding violates Goal 4, such an instruction could be used as an alternative to floating point operations as the building block for our approach.

We summarize our reproducible summation algorithm informally as follows:

We break the range of floating point exponents into fixed **bins** all of some width W (see Figure 1). Each summand is then rewritten as the exact sum of a small number of **slices**, where each slice corresponds to the significant bits of the summand lying (roughly) in a bin. Since we choose W to be smaller than the number of floating point significand bits, we can then exactly sum a large number of slices corresponding to the same bin in one floating point number. We can also use this floating point number to hold any `+Inf`, `-Inf`, or `NaN` that we encounter along the way. To avoid overflow, we use scaling to represent the sum of slices in the most significant bin that is very close to the overflow threshold. Notice that we do not need to sum the slices in all bins, only the bins corresponding to the largest exponents (the number of bins summed can be chosen based on the desired accuracy). Slices lying in bins with smaller exponents are discarded or not computed in the first place. For example, if sums of slices were computed on separate processors and then combined, we would only keep the sums of slices corresponding to the most significant bins. Independent of the order of summation, or of parallelism, we end up with the same sums of slices in the same bins, all computed exactly and so reproducibly. Finally, we convert from our accumulator to a standard

floating point number by carefully summing the separate sums of slices in each bin. As we will see, this idea, while it sounds simple, requires significant effort to implement and prove correct.

The rest of this article is organized as follows: Section 2 introduces notation used later. Section 3 gives a formal discussion of the bins and slices on which our algorithms are based. Section 4 describes the data structure (the **binned number**) used to store the sums of the slices. Section 5 contains several algorithms for basic manipulations of a binned number. These algorithms allow the user to, for instance, extract the slices of a floating point number and add them to the binned number, to add two binned numbers together, or to convert from the binned number to a floating point number. By extract, we informally mean any one of several ways of computing a slice. Section 6 contains the analysis of our algorithms. Section 6.1 shows that the absolute error of the reproducibly computed sum is bounded as in Equation (6.1). Section 6.2 provides and explains our recommended default parameter settings and considers `bfloat16` summands and mixed precision. Appendix A contains notes on the design choices we made in formalizing our algorithms. For example, it shows how our algorithms can be implemented in different floating point rounding modes, with the new augmented addition operation in the IEEE 754-2019 standard, using simpler exception handling, to support denormalized numbers, using abrupt underflow, or to support very high accuracy. Appendix A.7 contains detailed operation counts for each of our presented algorithms.

2 NOTATION AND BACKGROUND

All indices start at 0 in correspondence with several programming language implementations.

Let $\mathbb{R}$ and $\mathbb{Z}$ denote the sets of real numbers and integers, respectively. For all $r \in \mathbb{R}$, let $r\mathbb{Z}$ denote the set of all integer multiples of r.

Let $\mathbb{F}$ denote the set of binary floating point numbers with **precision** p and **exponent range** $[e_{\min}, e_{\max}]$. Each $f \in \mathbb{F}$ is represented using three fields: the **sign** $s \in \{-1, +1\}$, the **significand** (also called **mantissa**) $m \in [0, 2)$, and the **exponent** $e \in [e_{\min} - 1, e_{\max} + 1]$. The sign is represented by one bit $s_0 \in \{0, 1\}$ where $s = 1 - 2 \cdot s_0$. The mantissa is represented by the binary fixed point number $m = m_0.m_1m_2 \ldots m_{p-1}$. The exponent is represented using the unsigned integer $e_0e_1e_2 \ldots$, where $e = (e_{\min} - 1) + (e_0e_1e_2 \ldots)$. We will use the function getexp(f) to refer to e.

When $e \in [e_{\min}, e_{\max}]$, f is interpreted as $f = s \cdot m \cdot 2^e$. In this case, f is said to be **normalized** if $m_0 = 1$ ($m \in [1, 2)$) and **unnormalized** if $m_0 = 0$ and $m \neq 0$.

When $e = e_{\min} - 1$, f is interpreted as $f = s \cdot m \cdot 2^{e_{\min}}$. In this case, f is said to be **denormalized** or **subnormal** if $m_0 = 0$ and $m \neq 0$.

Note that $\mathbb{F}$ includes the **exceptional** values `+Inf`, `-Inf`, and `NaN`. When $e = e_{\max} + 1$ and $m_1m_2 \ldots m_{p-1} = 0$, then $f = s \cdot$ `Inf`. When $e = e_{\max} + 1$ and $m_1m_2 \ldots m_{p-1} \neq 0$, then $f =$ `NaN` (a special value meaning "Not a Number"). We say f is **finite** when it is not exceptional. Equivalently, f is finite when $f \in \mathbb{F} \cap \mathbb{R}$.

In the IEEE 754-2008 Floating Point Standard [1], floating point numbers follow the "hidden bit" convention, meaning that $m_0 = 0$ only when $e = e_{\min} - 1$. This implies that f is normalized only when $e \in [e_{\min}, e_{\max}]$, denormalized only when $e = e_{\min} - 1$ and $m \neq 0$, and zero only when $e = e_{\min} - 1$ and $m = 0$. The "hidden bit" convention also disallows the existence of unnormalized numbers. Finally, it means that m_0 need not be stored, and the underlying bitwise representation of the floating point number is therefore $s_0e_0e_1e_2 \ldots m_1m_2 \ldots m_{p-1}$. We define bits$(f)$ as the unsigned integer corresponding to this bitwise representation of f.

$r \in \mathbb{R}$ is **representable** as a floating point number if there exists $f \in \mathbb{F}$ such that $r = f$ as real numbers. For all $r \in \mathbb{R}$, $e \in \mathbb{Z}$ such that $e_{\min} - p < e$ and $|r| < 2 \cdot 2^{e_{\max}}$, if $r \in 2^e\mathbb{Z}$ and $|r| \leq 2^{e+p}$, then r is representable.

Machine epsilon, ϵ, the difference between 1 and the greatest floating point number smaller than 1, is defined as $\epsilon = 2^{-p}$.

The **unit in the last place** of $f \in \mathbb{F}$, $\text{ulp}(f)$, is the spacing between two consecutive floating point numbers of the same exponent as f. Thus, $\text{ulp}(f) = 2^{\text{getexp}(f)-p+1} = 2\epsilon 2^{\text{getexp}(f)}$. If f is finite, $\epsilon|f| < \text{ulp}(f)$. If f is normalized, $\text{ulp}(f) \leq 2\epsilon|f|$.

The **unit in the first place** of $f \in \mathbb{F}$, $\text{ufp}(f)$, is the value of the first significant bit of f. In this text, we define $\text{ufp}(f)$ only on normalized floating point numbers as $\text{ufp}(f) = 2^{\text{getexp}(f)}$. Since f is normalized, $\text{ufp}(f) \leq |f|$.

We define the operations $\oplus$, $\ominus$, and $\odot$ to be the IEEE floating point addition, subtraction, and multiplication operations. Thus, e.g., $x \oplus y$ is an operation that returns a floating point approximation to the real number $x + y$. When $x + y$ lies between two floating point numbers, the **rounding mode** defines which number to choose. We assume any "to-nearest" IEEE rounding mode (no specific tie-breaking behavior is required; see Lemma 5.1 for how tie-breaking is avoided in the critical part of Algorithm 5.4). We assume **gradual underflow**, meaning that when the nearest floating point number to $x + y$ is subnormal, that number is returned by $\oplus$. If $\oplus$ were to return a zero instead of returning subnormal numbers, we would call this behavior **abrupt underflow**. Methods to handle abrupt underflow and alternate rounding modes will be considered in Appendix A.

For all $f_0, f_1 \in \mathbb{F}$ and, e.g., $\oplus$ and $+$, if $f_0 + f_1$ is representable, we have $f_0 \oplus f_1 = f_0 + f_1$. Under our rounding assumptions, if $f_0 \oplus f_1$ is finite, then we have that $|(f_0 \oplus f_1) - (f_0 + f_1)| \leq 0.5\text{ulp}(f_0 \oplus f_1)$. This bound accounts for underflow, as the magnitude of $\text{ulp}(f)$ reflects the appropriate loss of accuracy when f is in the denormal range.

We will refer to the various IEEE formats as follows: We refer to Binary16 as `half`, Binary32 as `single`, Binary64 as `double`, and Binary128 as `quad`.

We define $f|m$ and $f\&m$ as the floating point result of applying the logical "or" and "and" operators to the underlying bitwise representation of f using the integer mask m. We assume that f is stored in the standard binary interchange format [1]. If $>>$ and $<<$ are the logical shift operators on integers, we can implement $\text{ufp}(f)$ as $\text{ufp}(f) = f\&((e_{\max} - e_{\min} + 2) << (p-1))$, using the fact that the representation of the binary integer $(e_{\max} - e_{\min} + 2) << (p-1)$ can be used to mask the bits corresponding to the exponent field. Thus, we can implement $\text{getexp}(f)$ as $\text{getexp}(f) = (\text{bits}(f\&((e_{\max} - e_{\min} + 2) << (p-1))) >> (p-1)) + (e_{\min} - 1)$. Note that getexp differs from the IEEE operation *logB* on exceptional and tiny inputs, since $\text{getexp}(\texttt{+Inf}) = e_{\max} + 1$ while $logB(\texttt{+Inf}) = \texttt{+Inf}$ and $\text{getexp}(0) = e_{\min} - 1$ while $logB(0) = \texttt{-Inf}$. We can check if a floating point value is exceptional by checking that $\text{bits}(f\&((e_{\max} - e_{\min} + 2) << (p-1)))$ is equal to $(e_{\max} - e_{\min} + 2) << (p-1)$. This could be implemented with IEEE operations, as the logical negation of *isFinite*, or the logical disjunction of *isNan* or *isInf*, but we use our bitwise check for operation counting.

We define the function $\mathcal{R}_{\pm\infty}(r, e), r \in \mathbb{R}, e \in \mathbb{Z}$ as

$$\mathcal{R}_{\pm\infty}(r, e) = \begin{cases} \lfloor r/2^e + 1/2 \rfloor 2^e \text{ if } r \geq 0, \\ \lceil r/2^e - 1/2 \rceil 2^e \text{ otherwise.} \end{cases} \tag{2.1}$$

$\mathcal{R}_{\pm\infty}(r, e)$ rounds r to the nearest multiple of 2^e, breaking ties away from 0. Properties of such rounding are shown in Equation (2.2). Let s_r be the sign of r, so $s_r = 1$ if $r \geq 0$ and $s_r = -1$ if $r < 0$. Let $z \in \mathbb{Z}$.

$$\begin{aligned} &|r - \mathcal{R}_{\pm\infty}(r, e)| \leq 2^{e-1} \\ &|r - \mathcal{R}_{\pm\infty}(r, e)| \leq |r| \\ &\quad \mathcal{R}_{\pm\infty}(r, e) \in 2^n\mathbb{Z} \text{ if } r \in 2^n\mathbb{Z}, n \in \mathbb{Z} \\ &\quad \mathcal{R}_{\pm\infty}(r, e) = 0 \text{ if } |r| < 2^{e-1} \\ &\quad \mathcal{R}_{\pm\infty}(r, e) = r + s_r 2^{e-1} \text{ if } |r - \mathcal{R}_{\pm\infty}(r, e)| = 2^{e-1}. \end{aligned} \tag{2.2}$$

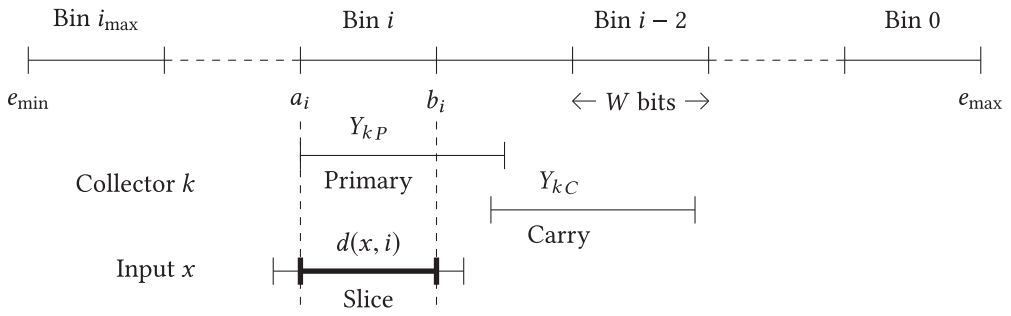

Fig. 1. The binning process.

3 BINNING

We achieve reproducible summation of floating point numbers through binning. Each number is split into several components corresponding to predefined exponent ranges, then the components corresponding to each range are summed separately. We begin in Section 3.1 by explaining the particular set of ranges (referred to as bins; see Figure 1) used. Section 3.2 develops mathematical theory to describe the components (referred to as slices) corresponding to each bin. The data structure (referred to as a binned number) to represent slices in bins will be explained in Section 4. In this section, we develop theory to concisely describe and prove correctness of the algorithms throughout the article.

3.1 Bins

We start by dividing an expanded range of floating point exponents, $(e_{\min} - p, \ldots, e_{\max} + 1]$ into **bins** $(a_i, b_i]$ of **width** W according to Equations (3.1), (3.2), and (3.3).

Definition 3.1. We define the bins $(a_i, b_i]$ used by our algorithms as follows:

$$0 \le i \le i_{\max} = \left\lfloor (e_{\max} - e_{\min} + p - 1)/W \right\rfloor - 1, \tag{3.1}$$

$$a_i = e_{\max} + 1 - (i+1)W, \tag{3.2}$$

$$b_i = a_i + W. \tag{3.3}$$

Note that Equations (3.2) and (3.3) imply that for $1 \le i \le i_{\max}$, $a_{i-1} = b_i$.

We say the bin $(a_{i_0}, b_{i_0}]$ is **greater** than the bin $(a_{i_1}, b_{i_1}]$ if $a_{i_0} > a_{i_1}$ (which is equivalent to either $b_{i_0} > b_{i_1}$ or $i_0 < i_1$).

We say the bin $(a_{i_0}, b_{i_0}]$ is **less** than the bin $(a_{i_1}, b_{i_1}]$ if $a_{i_0} < a_{i_1}$ (which is equivalent to either $b_{i_0} < b_{i_1}$ or $i_0 > i_1$).

The greatest bin, $(a_0, b_0]$, is

$$(e_{\max} + 1 - W, e_{\max} + 1] \tag{3.4}$$

and the least bin, $(a_{i_{\max}}, b_{i_{\max}}]$, is

$$(e_{\min} - p + 2 + ((e_{\max} - e_{\min} + p - 1) \mod W), e_{\min} - p + 2 + W + ((e_{\max} - e_{\min} + p - 1) \mod W]. \tag{3.5}$$

For reasons explained later, we require that

$$W < p - 2 \tag{3.6}$$

and

$$2W > p + 1. \tag{3.7}$$

Table 1. Proposed Binning Scheme

Floating Point Type	`single`	`double`	`quad`
$e_{\max}$	127	1,023	16,383
$e_{\min}$	−126	−1,022	−16,382
p	24	53	113
$e_{\min} - p$	−150	−1,075	−16,495
W	13	40	100
$i_{\max}$	20	51	327
$(a_0, b_0]$	(115, 128]	(984, 1,024]	(16,284, 16,384]
$(a_{i_{\max}}, b_{i_{\max}}]$	(−145, −132]	(−1,056, −1,016]	(−16,416, −16,316]

Notice that to satisfy Equations (3.6) and (3.7), we must have

$$p \geq 8. \tag{3.8}$$

Equation (3.8) is satisfied in all binary IEEE floating point formats.
Since

$$\begin{aligned} a_{i_{\max}} &= e_{\min} - p + 2 + \Big((e_{\max} - e_{\min} + p - 1) \mod W\Big) \\ &\geq e_{\min} - p + 2, \end{aligned} \tag{3.9}$$

we ignore the least exponents in our expanded range,

$$\Big(e_{\min} - p, e_{\min} - p + 2 + \Big((e_{\max} - e_{\min} + p - 1) \mod W\Big)\Big]. \tag{3.10}$$

However, Equations (3.6) and (3.5) ensure that our least bin will always extend below $e_{\min}$.

$$\begin{aligned} a_{i_{\max}} &= e_{\min} - p + 2 + \Big((e_{\max} - e_{\min} + p - 1) \mod W\Big) \\ &\leq e_{\min} - p + 2 + (W - 1) \\ &\leq e_{\min} - p + 2 + (p - 2 - 1 - 1) = e_{\min} - 2. \end{aligned} \tag{3.11}$$

A possible division of expanded exponent ranges for various binary IEEE floating point formats is shown in Table 1. The choices of W are discussed in detail in Section 6.2, when the effects of such choices can be more accurately described.

We do not consider `half` precision floating point numbers, because it is easier to sum them exactly. Since the `half` format has $e_{\min} = -14$, $e_{\max} = 15$, and $p = 11$, it can be represented in fixed-point arithmetic using 40 bits of precision. Thus, we could sum our `half` summands using a `double` and it would be exact, and so reproducible, to at least 2^{13} summands. One could sum even more of them exactly by converting the summands to (scaled) 64-bit integers.

3.2 Slices

Throughout the text, we will refer to the **slice** of some $x \in \mathbb{R}$ in the bin $(a_i, b_i]$ (see Figure 1). x can be split into several slices, each slice corresponding to a bin $(a_i, b_i]$ and expressible as the (possibly negated) sum of a subset of $\{2^e, e \in (a_i, b_i]\}$, such that the sum of the slices equals x exactly or provides a good approximation of x, possibly dropping bits at the very bottom of the exponent range, $(e_{\min}, e_{\min} - p]$. Specifically, the slice of $x \in \mathbb{R}$ in the bin $(a_i, b_i]$ is defined recursively as $d(x, i)$ in Equation (3.12). We must define $d(x, i)$ recursively, because it is not a simple bitwise extraction. The extraction is more complicated, because the splitting is performed using floating point instructions. There are many ways to implement the splitting (using only integer instructions, only floating point instructions, a mix of the two, or even special purpose hardware). This

article focuses on using a majority of floating point instructions, allowing us to take advantage of the rounding operations built in to floating point arithmetic.

Definition 3.2. We define the slice $d(x,i)$ of $x \in \mathbb{R}$ in the bin $(a_i, b_i]$ as follows:

$$\begin{aligned} d(x,0) &= \mathcal{R}_{\pm\infty}(x, a_0 + 1) \\ d(x,i) &= \mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right) \text{ for } i > 0. \end{aligned} \tag{3.12}$$

We make three initial observations on the definition of $d(x,i)$. First, we note that $d(x,i)$ is well defined recursively on i with base case $d(x,0) = \mathcal{R}_{\pm\infty}(x, a_0 + 1)$.

Next, notice that $d(x,i) \in 2^{a_i+1}\mathbb{Z}$.

Finally, it is possible that $d(x,0)$ may be too large to represent as a floating point number. For example, if x is the greatest finite floating point number, then $d(x,0) = \mathcal{R}_{\pm\infty}(x, a_0 + 1)$ would be $2^{e_{max}+1}$. We will have to handle the 0 bin using scaling.

Lemmas 3.1 and 3.2 follow from the definition of $d(x,i)$.

Lemma 3.1. *For all $i \in \{0, \ldots, i_{\max}\}$ and $x \in \mathbb{R}$ such that $|x| < 2^{a_i}$, $d(x,i) = 0$.*

Proof. We show the claim by induction on i.

In the base case, $|x| < 2^{a_0}$, by Equation (2.2), we have $d(x,0) = \mathcal{R}_{\pm\infty}(x, a_0 + 1) = 0$.

In the inductive step, we have $|x| < 2^{a_{i+1}} < \cdots < 2^{a_0}$ by Equation (3.2) and by induction $d(x,i) = \cdots = d(x,0) = 0$. Thus,

$$d(x, i+1) = \mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i} d(x,j), a_{i+1} + 1\right) = \mathcal{R}_{\pm\infty}(x, a_{i+1} + 1).$$

Again, since $x < 2^{a_{i+1}}$, by Equation (2.2), we have $d(x, i+1) = \mathcal{R}_{\pm\infty}(x, a_{i+1} + 1) = 0$. □

Lemma 3.2. *For all $i \in \{0, \ldots, i_{\max}\}$ and $x \in \mathbb{R}$ such that $|x| < 2^{b_i}$, $d(x,i) = \mathcal{R}_{\pm\infty}(x, a_i + 1)$.*

Proof. The claim is a simple consequence of Lemma 3.1.

By Equations (3.2) and (3.3), $|x| < 2^{b_i} = 2^{a_{i-1}} < \cdots < 2^{a_0}$. Therefore, Lemma 3.1 implies $d(x,0) = \cdots = d(x, i-1) = 0$ and we have

$$d(x,i) = \mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right) = \mathcal{R}_{\pm\infty}(x, a_i + 1).$$ □

Lemma 3.1, Lemma 3.2, and Equation (3.12) can be combined to yield an equivalent definition of $d(x,i)$ for all $i \in \{0, \ldots, i_{\max}\}$ and $x \in \mathbb{R}$.

$$d(x,i) = \begin{cases} 0 \text{ if } |x| < 2^{a_i} \\ \mathcal{R}_{\pm\infty}(x, a_i + 1) \text{ if } 2^{a_i} \le |x| < 2^{b_i} \\ \mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right) \text{ if } 2^{b_i} \le |x| \end{cases} \tag{3.13}$$

Theorem 3.3 shows that the sum of the slices of $x \in \mathbb{R}$ in the $i+1$ lowest bins approximates x to within an absolute error of 2^{a_i}.

Theorem 3.3. *For all $i \in \{0, \ldots, i_{\max}\}$ and $x \in \mathbb{R}$, $|x - \sum_{j=0}^{i} d(x,j)| \le 2^{a_i}$.*

Proof. We apply Equations (3.12) and (2.2)

$$\left|x - \sum_{j=0}^{i} d(x,j)\right| = \left|\left(x - \sum_{j=0}^{i-1} d(x,j)\right) - d(x,i)\right|$$
$$= \left|\left(x - \sum_{j=0}^{i-1} d(x,j)\right) - \mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right)\right| \leq 2^{a_i}. \qquad \square$$

Combining Theorem 3.3 with Equation (3.11), we see that for any $x \in \mathbb{R}$,

$$\left|x - \sum_{i=0}^{i_{\max}} d(x,i)\right| \leq 2^{a_{i_{\max}}} \leq 2^{e_{\min}-2}. \tag{3.14}$$

Although the bins do not extend all the way to $e_{\min} - p$, we can still approximate finite $x \in \mathbb{F}$ using the sum of its slices to the nearest multiple of $2^{e_{\min}-1}$ or better (see $a_{i_{\max}}$ in Table 1).

As the slices of x provide a good approximation of x, the sum of the slices of some $x_0, \ldots, x_{n-1} \in \mathbb{R}$ provide a good approximation of $\sum_{j=0}^{n-1} x_j$. This is the main idea behind the reproducible summation algorithm presented here. Since the largest nonzero slices of x provide the best approximation to x, we compute the sum of the slices of each $x_0, \ldots, x_{n-1}$ corresponding to the largest K bins such that at least one slice in the largest bin is nonzero. If such an approximation can be computed exactly, then it is necessarily reproducible. Notice that we do not necessarily compute the exact sum as in References [9, 21, 26]. We compute a well-defined approximation of the sum exactly.

If the sums of slices corresponding to each bin are kept separate, we can compute the reproducible sum iteratively, only storing sums of nonzero slices in the K largest bins seen so far. When a summand is encountered with nonzero slices in a larger bin than what has been seen previously, we abandon sums of slices in smaller bins to store the sums of slices in the larger ones.

Before moving on to discussions of how to store and compute the slices and sums of slices, we must show a bound on their size. Theorem 3.4 shows a bound on $d(x,i)$.

Theorem 3.4. *For all $i \in \{0, \ldots, i_{\max}\}$ and finite $x \in \mathbb{F}$, $|d(x,i)| \leq 2^{b_i}$.*

Proof. First, we show that $|x - \sum_{j=0}^{i-1} d(x,j)| \leq 2^{b_i}$.
If $i = 0$, we use Equation (3.4) to get

$$\left|x - \sum_{j=0}^{i-1} d(x,j)\right| = |x| < 2 \cdot 2^{e_{\max}} = 2^{b_0}.$$

Otherwise, we can apply Equations (3.2) and (3.3) and Theorem 3.3 to get

$$\left|x - \sum_{j=0}^{i-1} d(x,j)\right| \leq 2^{a_{i-1}} = 2^{b_i}.$$

As $2^{b_i} \in 2^{a_i+1}\mathbb{Z}$, Equation (3.12) can be used:

$$|d(x,i)| = \left|\mathcal{R}_{\pm\infty}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right)\right| \leq 2^{b_i}. \qquad \square$$

Combining Theorem 3.4 with the earlier observation that $d(x,i) \in 2^{a_i+1}\mathbb{Z}$, we see that the slice $d(x,i)$ can be represented by bits lying in the bin $(a_i, b_i]$ as desired.

4 THE BINNED NUMBER

The **binned number** is used to represent the intermediate sum during reproducible summation. A binned number Y is a data structure composed of several slice-collecting data structures $Y_0, \ldots, Y_{K-1}$. Each **collector** Y_k is a data structure used to extract and sum the slices of the input in the bin $(a_{I+k}, b_{I+k}]$ where I is the **index** of Y, $0 \leq I \leq i_{\max} - K + 1$ (so all of the bins in the binned number are defined) and $0 \leq k < K$.

A binned number with K collectors is referred to as a ***K*-fold** binned number. Due to their low accuracy, 1-fold binned numbers are not considered. K can be at most $i_{\max} + 1$, when all collectors are included. The collectors in a binned number correspond to contiguous bins in decreasing order. If Y has index I, then Y_k sums slices of the input in the bin $(a_{I+k}, b_{I+k}]$. The binned number corresponding to the reproducibly computed sum of $x_0, \ldots, x_{n-1} \in \mathbb{F}$ is referred to as the **binned sum** of $x_0, \ldots, x_{n-1}$.

Section 4.1 elaborates on the specific fields that make up the binned number and the values they represent in the finite case. Sections 4.1.1 and 4.1.2 show that the fields in the binned number are normalized floating point numbers below the overflow threshold, and Section 4.1.3 contains sentinel values for the binned number to handle exceptions. Section 4.2 explains why the index does not need to be stored explicitly. Section 4.3 explains how the binned number can be used to represent the sum of several floating point numbers.

4.1 Primary and Carry

The fields of the binned number are of the same floating point type as the numbers it is summing. The collectors Y_k of a binned number Y are each implemented using two underlying floating point fields. The **primary** field Y_{kP} is used during slice extraction, while the **carry** field Y_{kC} holds overflow from the primary field. Because primary fields are frequently accessed sequentially, the primary fields and carry fields are each stored contiguously in separate arrays. The notation for the primary field Y_{kP} and carry field Y_{kC} corresponds to the "S_j" and "C_j" of Algorithm 6 in Reference [15].

The numerical value $\mathcal{Y}_{kP}$ represented by finite data stored in the primary field Y_{kP} is an offset from $1.5\epsilon^{-1}2^{a_{I+k}}$, where I is the index of Y. We store the field using an offset to set the exponent of Y_{kP} so we may use Y_{kP} for both splitting and storage of summands. To keep the exponent of Y_{kP} constant, it must be constrained to a range of numbers with the same exponent as $1.5\epsilon^{-1}2^{a_{I+k}}$. Because the offset corresponding to $I + k = 0$ is too large to be representable as a floating point number, we must store the collector corresponding to index 0 using a scaled representation.

$$\begin{aligned} \mathcal{Y}_{kP} &= \begin{cases} Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}} & \text{if } I + k > 0 \\ 2^{p-W+1}(Y_{0P} - 1.5 \cdot 2^{e_{\max}}) & \text{if } I + k = 0 \end{cases} \\ Y_{kP} &\in \begin{cases} (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}}) & \text{if } I + k > 0 \\ (2^{e_{\max}}, 2 \cdot 2^{e_{\max}}) & \text{if } I + k = 0 \end{cases} \end{aligned} \tag{4.1}$$

Representing the primary field value as an offset from $1.5\epsilon^{-1}2^{a_{I+k}}$ simplifies the process of extracting the slices of input in bins $(a_{I+k}, b_{I+k}]$. If we have finite $r \in \mathbb{F}$, $|r| \leq 2^{b_{I+k}}$ and ensure that Equation (4.1) stays satisfied, then it will be shown that the operation $Y_{kP} = Y_{kP} + (r|1)$ in a "to-nearest" rounding mode will add to Y_{kP} the value $\mathcal{R}_{\pm\infty}(r, a_{I+k})$, where $I + k > 0$ and $r|1$ is r with the least significant bit of the significand set to 1. We will show how to use this observation to efficiently extract and sum the slices belonging to each collector in Algorithm 5.4.

Because $d(x, I + k) = 0$ for bins with $|x| < 2^{a_{I+k}}$, the values in the greatest K nonzero collectors can be computed reproducibly by computing the values in the greatest K collectors needed for the largest x seen so far. Upon encountering an $x \geq 2^{b_I}$, the collectors can then be shifted towards

index 0 as necessary. Since the maximum absolute value operation is always reproducible, so is the index of the greatest collector.

To keep the primary fields in the necessary range while the slices are extracted and to keep the representation of Y_k unique, when Y_{kP} strays too far from $1.5\epsilon^{-1}2^{a_{I+k}}$ it is renormalized to the range

$$Y_{kP} \in \begin{cases} [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}}) \text{ if } I+k>0 \\ [1.5\cdot 2^{e_{\max}}, 1.75\cdot 2^{e_{\max}}) \text{ if } I+k=0. \end{cases}$$

This renormalization is performed periodically during summation. To renormalize, Y_{kP} is incremented or decremented by $0.25\epsilon^{-1}2^{a_{I+k}}$, leaving the carry field Y_{kC} to record the number of such adjustments (where Y_{kC} is decremented or incremented by 1, respectively). The numerical value $\mathcal{Y}_{kC}$ represented by finite data stored in the carry field Y_{kC} of a binned number Y of index I is

$$\mathcal{Y}_{kC} = (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC}. \tag{4.2}$$

Combining Equations (4.1) and (4.2), we get that the value $\mathcal{Y}_k$ of the collector Y_k of a binned number Y of index I is

$$\mathcal{Y}_k = \mathcal{Y}_{kP} + \mathcal{Y}_{kC} = \begin{cases} Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}} + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} \text{ if } I+k>0 \\ 2^{p-W+1}(Y_{0P} - 1.5\cdot 2^{e_{\max}}) + (0.25\epsilon^{-1}2^{a_0})Y_{0C} \text{ if } I+k=0. \end{cases} \tag{4.3}$$

Therefore, using Equation (4.3), the numerical value $\mathcal{Y}$ represented by data stored in a K-fold binned number Y of index I (the sum of Y's collectors) is

$$\mathcal{Y} = \sum_{k=0}^{K-1} \mathcal{Y}_k = \sum_{k=0}^{K-1} \begin{cases} Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}} + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} \text{ if } I+k>0 \\ 2^{p-W+1}(Y_{0P} - 1.5\cdot 2^{e_{\max}}) + (0.25\epsilon^{-1}2^{a_0})Y_{0C} \text{ if } I+k=0. \end{cases} \tag{4.4}$$

Depending on the data format used to store Y_{kC}, the number of updates to one collector without overflow is limited, which determines the possible maximum number of inputs that can be reproducibly added to one collector. As Y_{kC} must be able to record additions of absolute value 1 without error, Y_{kC} must stay in the range $[-\epsilon^{-1}, \epsilon^{-1}]$. As the absolute value of a slice $d(x,i)$ is bounded by 2^{a_i+W} as in Theorem 3.4 and a value of 1 in the carry field has a value of 2^{a_i+p-2}, each collector is capable of representing the sum of at least

$$2^{2p-W-2} \tag{4.5}$$

slices, and thus the binned number can represent the sum of at least the same number of floating point numbers. Equation (4.5) is 2^{64} in `double` and 2^{33} in `single` precision using the values in Table 1.

4.1.1 Overflow. Here, we show that none of the primary fields in a binned number may overflow.

Theorem 4.1. *For any binned number Y of index I and any Y_{kP} satisfying Equation (4.1), if $I+k \geq 1$, $|Y_{kP}| < 2^{e_{\max}}$. If $I+k=0$, $|Y_{kP}| < 2\cdot 2^{e_{\max}}$.*

Proof. If $I+k \geq 1$, $a_1 = e_{\max} + 1 - 2W$ by Equation (3.2), therefore, $a_1 < e_{\max} - p$ using Equation (3.7) and, since all quantities are integers, $a_1 \leq e_{\max} - p - 1$. Thus, $a_{I+k} \leq a_1 \leq e_{\max} - p - 1$ by Equation (3.2).

By Equation (4.1), Y_{kP} is kept within the range $(\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$, therefore

$$|Y_{kP}| < 2\epsilon^{-1}2^{a_{I+k}} \leq 2^{1+p}2^{e_{\max}-1-p} = 2^{e_{\max}}.$$

If $I+k=0$, the result is given directly by Equation (4.1). □

4.1.2 Underflow. Although we sum numbers in the denormalized range, Algorithms 5.4 and 5.5 require that the primary fields Y_{kP} are normalized to work correctly. Theorem 4.2 shows that the primary fields will always be normalized.

THEOREM 4.2. *Any primary field Y_{kP} of a binned number Y of index I satisfying Equation (4.1) is normalized.*

PROOF. If $I + k \geq 1$, by Equation (4.1), we have $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ and we can use Equation (3.9) to show $\text{getexp}(Y_{kP}) = a_{I+k} + p > e_{\min} + 1$, so Y_{kP} is normalized.

If $I + k = 0$, by Equation (4.1), we have $Y_{kP} \in (2^{e_{\max}}, 2 \cdot 2^{e_{\max}})$ and clearly it is normalized. □

4.1.3 Exceptions. Binned numbers are capable of representing exceptional cases such as `+Inf`, `-Inf`, or `NaN`. A binned number Y stores its exception status in its first primary field Y_{0P}.

A value of 0 in Y_{0P} indicates that nothing has been added to Y yet (Y_{0P} is initialized to 0). By Equation (4.1), the value of 0 in a primary field is unused in any previously specified context and may be used as a sentinel value.

A value of `+Inf` or `-Inf` in Y_{0P} indicates that one or more `+Inf` or `-Inf` (and no other exceptional values) have been added to Y, respectively.

A value of `NaN` in Y_{0P} indicates that one or more `NaN` s have been added to Y and/or one or more of both `+Inf` and `-Inf` have been added to Y. Note that we treat all bitwise representations of `NaN` as identical.

This behavior follows the behavior for exceptional values in IEEE 754-2008 floating point arithmetic. The result of adding some exceptional values using floating point arithmetic, therefore, matches the result obtained from binned summation. As `+Inf`, `-Inf`, and `NaN` add associatively, this behavior is reproducible.

As the Y_{kP} are kept finite to store finite values and to indicate that nothing has been added to Y yet, `+Inf`, `-Inf`, and `NaN` are unused in any previously specified context and are valid sentinel values.

Notice that Equation (4.1) implies that the binned number is capable of expressing values that are too large to represent with the floating point type it is composed with. The partial sums in reproducible summation can grow much larger than the overflow threshold and then cancel back down. In fact, as long as the sum itself is below overflow (beyond the margin of error), the summands are finite, and the number of summands is bounded by Equation (4.5), the reproducible summation will not overflow.

However, as we have just described, if the inputs to summation are already infinite, the summation will return `+Inf` or `-Inf`.

If the final value of the summation is too large to express as a floating point number, we will also return `+Inf` or `-Inf`. We can determine whether or not the real number $\mathcal{Y}$ is too large to be representable in the desired floating point format when converting Y to a floating point number. This procedure is described in Section 5.8.

4.2 Implicit Index Storage

Here, we make the observation that there exists a bijection between the index I of a binned number Y and the exponent of Y_{0P}.

By Equation (4.1) and Theorem 4.1, we see that if $I = 0$, Y_{0P} has exponent $e_{\max}$, whereas if $I > 0$, $Y_{IP} < 2^{e_{\max}}$. Equation (4.1) and Theorem 4.2 imply that if $I > 0$, Y_{0P} has exponent $a_I + p$. If nothing has been added to Y, Y_{0P} is 0 and Theorem 4.2 tells us this is a previously unused exponent. Since all of the previous exponents have corresponded to finite values, the exceptional values have an unused exponent.

Thus, we do not need to explicitly store the index of a binned number, as it can be determined by examining the exponent of Y_{0P}. Algorithm 5.1 will show how to do this explicitly.

4.3 Binned Sum

We have previously explained the binned number, a data structure we will use for reproducible summation. We now define a quantity that can be expressed using the binned number, called the **binned sum**. We show that if an algorithm returns the binned sum of its inputs, it is reproducible.

Definition 4.1. Assume $1 \le n \le 2^{2p-W-2}$ (4.5). The K-fold **binned sum** of finite $x_0, \ldots, x_{n-1} \in \mathbb{F}$ is defined to be a K-fold binned number Y such that for all k where $0 \le k < K$, the following equations are satisfied:

$$
\begin{aligned}
&I \text{ is the greatest integer such that } \max(|x_j|) < 2^{b_I} \text{ and } I \le i_{\max} - K + 1 \\
&Y_{kP} \in \begin{cases} [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}}) \text{ if } I + k > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}}) \text{ if } I + k = 0 \end{cases} \\
&\mathcal{Y}_k = \sum_{j=0}^{n-1} d(x_j, I + k)
\end{aligned} \tag{4.6}
$$

Recall that Equation (4.3) defines the value $\mathcal{Y}_k$ in terms of I, Y_{kP}, and Y_{kC}.

The K-fold binned sum of $x_0, \ldots, x_{n-1} \in \mathbb{F}$ (with at least one exceptional value `+Inf`, `-Inf`, or `NaN`) is defined to be a K-fold binned number Y such that

$$
Y_{0P} = \begin{cases} \texttt{+Inf} \text{ if there is at least one } \texttt{+Inf} \text{ and no other exceptional values,} \\ \texttt{-Inf} \text{ if there is at least one } \texttt{-Inf} \text{ and no other exceptional values,} \\ \texttt{NaN} \text{ otherwise.} \end{cases} \tag{4.7}
$$

And the K-fold binned sum of no numbers (the empty sum) is defined to be the K-fold binned number Y such that

$$
Y_{0P} = 0. \tag{4.8}
$$

Theorem 4.3 shows that the binned sum is well-defined in the sense that the fields in the binned number corresponding to the summands $x_0, \ldots, x_{n-1}$ (in any order) are unique.

Theorem 4.3. *Let Y be the binned sum of some $x_0, \ldots, x_{n-1} \in \mathbb{F}$. Let $\sigma_0, \ldots, \sigma_{n-1}$ be some permutation of the first n nonnegative integers. Let Z be the binned sum of $x_{\sigma_0}, \ldots, x_{\sigma_{n-1}}$.*

If $n \neq 0$ and $x_0, \ldots, x_{n-1}$ are all finite, we have that for all k, $0 \le k < K$, $Y_{kP} = Z_{kP}$ and $Y_{kC} = Z_{kC}$. Otherwise, $Y_{0P} = Z_{0P}$ and Y_{0P} is either 0 or exceptional.

Proof. If $n = 0$, then by Equation (4.8), we have that $Y_{0P} = Z_{0P} = 0$.

If $n \ge 1$ and at least one x_i is exceptional, then, since the conditions in Equation (4.7) depend only on the number of each type of exceptional value and not on their order, we have that $Y_{0P} = Z_{0P}$. Since at least one x_i is exceptional, all of the possible cases in Equation (4.7) specify that Y_{0P} is exceptional.

The rest of the proof deals with the remaining case when $n \ge 1$ and all x_i are finite.

Since $\max(|x_j|) = \max(|x_{\sigma_j}|)$, both Y and Z have the same index I, since I is the greatest integer such that $\max(|x_j|) < 2^{b_I}$ and $I \le i_{\max} - K + 1$.

Using the associativity of addition of real numbers,

$$
\mathcal{Y}_k = \sum_{j=0}^{n-1} d(x_j, I + k) = \sum_{j=0}^{n-1} d(x_{\sigma_j}, I + k) = \mathcal{Z}_k.
$$

If $I + k \geq 1$, assume for contradiction that there exists some k, $0 \leq k < K$, such that $Y_{kC} \neq Z_{kC}$. Since $\mathcal{Y}_k = \mathcal{Z}_k$, Equation (4.3) yields

$$\begin{aligned}(Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} &= (Z_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Z_{kC},\\ Y_{kP} - Z_{kP} &= (0.25\epsilon^{-1}2^{a_{I+k}})(Z_{kC} - Y_{kC}),\\ |Y_{kP} - Z_{kP}| &\geq 0.25\epsilon^{-1}2^{a_{I+k}}.\end{aligned}$$

Since $Y_{kP} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$, $Z_{kP} \notin [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$, a contradiction.

Therefore, we have that $Y_{kC} = Z_{kC}$. Along with the fact that $\mathcal{Y}_k = \mathcal{Z}_k$, application of Equation (4.3) implies that $Y_{kP} = Z_{kP}$.

If $I = 0$, assume for contradiction that $Y_{0C} \neq Z_{0C}$. Since $\mathcal{Y}_0 = \mathcal{Z}_0$, Equation (4.3) yields

$$\begin{aligned}2^{p-W+1}(Y_{0P} - 1.5 \cdot 2^{e_{\max}}) + (0.25\epsilon^{-1}2^{a_0})Y_{0C} &= 2^{p-W+1}(Z_{0P} - 1.5 \cdot 2^{e_{\max}}) + (0.25\epsilon^{-1}2^{a_0})Z_{0C},\\ Y_{0P} - Z_{0P} &= 2^{W-p-1}(0.25\epsilon^{-1}2^{a_0})(Z_{0C} - Y_{0C}),\\ |Y_{0P} - Z_{0P}| &\geq 2^{W-p-1}(0.25\epsilon^{-1}2^{a_0}),\end{aligned}$$

and by Equation (3.4), we have

$$|Y_{0P} - Z_{0P}| \geq 0.25 \cdot 2^{e_{\max}}.$$

Since $Y_{0P} \in [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}})$, $Z_{kP} \notin [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}})$, a contradiction.

Therefore, we have that $Y_{0C} = Z_{0C}$. Along with the fact that $\mathcal{Y}_0 = \mathcal{Z}_0$, application of Equation (4.3) implies that $Y_{0P} = Z_{0P}$. □

Theorem 4.3 implies that any algorithm that can compute the binned sum of a list of floating point numbers is a reproducible summation algorithm, as the binned sum is well-defined, unique, and independent of the ordering of the summands.

5 OPERATIONS

Here, we present a set of basic operations on a binned number. Our intent is to make it easy to build high-level reproducible operations with different requirements for reproducibility, data ordering, and reduction tree shapes due to platform differences. For example, if on a parallel machine, the summands on each processor are created and summed deterministically, with the only source of nonreproducibility in the parallel reduction, then we want to be able to use reproducible summation only in the parallel reduction. If the subset of summands on each processor were different from run to run, we want to be able to use reproducible summation throughout the whole summation routine.

Two simple original algorithms relating to the index of a binned number are given in Section 5.1. In Sections 5.2 to 5.7, algorithms from Reference [15] have been modified to handle very large summands (summands close to the overflow threshold $\approx 2 \cdot 2^{e_{\max}}$) and exceptional cases. These algorithms have also been modified to use the bins presented in Section 3 and the binned number presented in Section 4. To obtain a general reproducible algorithm for summation, one must design for reproducibility under both data ordering and reduction tree shape. Section 5.5 provides algorithms to sum numbers regardless of ordering (a more efficient algorithm is presented in Section 5.6), while Section 5.7 provides methods to sum numbers regardless of reduction tree shape. Section 5.8 provides an original algorithm to obtain the floating point value represented by a binned number. This conversion algorithm is more accurate than Reference [15] and leads to a much improved error bound.

5.1 Index

When operating on binned numbers it is sometimes necessary to compute their index. Algorithm 5.1 yields the index of a binned number in constant time.

ALGORITHM 5.1. *Given a binned number Y, calculate its index I.*

```
function BinnedNumberIndex(Y)
    if Y_0P = 0 then
        return i_max + 1
    end if
    return ⌊(e_max + p − getexp(Y_0P) − W + 1)/W⌋          ▷ Index I of Y
end function
```

Ensure:

$$\text{Returned result } I \text{ is } \begin{cases} i_{\max} + 1 \text{ if } Y_{0P} = 0 \\ 0 \text{ if } Y_{0P} \text{ is not finite} \\ \text{the index of } Y \text{ otherwise} \end{cases}$$

Note that the floor function is necessary in Algorithm 5.1 to account for the case when Y has index 0, which has $\text{getexp}(Y_{0P}) = e_{\max}$ by Equation (4.1). This uses the assumptions (3.6) and (3.7) ($\frac{p+1}{2} < W < p - 2$), so $2 < p - W + 1 < W$.

Another useful operation is, given some finite $x \in \mathbb{F}$, to find the unique bin $(a_J, b_J]$ where J is the greatest integer such that $|x| < 2^{b_J}$ and $J \leq i_{\max} - K + 1$. Algorithm 5.2 yields such a J in constant time.

ALGORITHM 5.2. *Given $x \in \mathbb{F}$, calculate the greatest index sufficient for storing x in a binned number.*

```
function FloatingPointIndex(x)
    return max(0, min(i_max − K + 1, ⌊(e_max − getexp(x))/W⌋))   ▷ Index J of x
end function
```

Ensure:

$$\text{Returned result } J \text{ is } \begin{cases} 0 \text{ if } x \text{ is not finite} \\ \text{the greatest integer such that } |x| < 2^{b_J} \text{ and } J \leq i_{\max} - K + 1, \text{ otherwise} \end{cases}$$

The behavior of Algorithm 5.2 in the case when $x < 2^{a_{i_{\max}}}$ is consistent with the following algorithms, since values smaller than the least bin will not be extracted.

Both Algorithms 5.1 and 5.2 satisfy their exceptional cases, because the function getexp() is assumed to return $e_{\max} + 1$ when its argument is exceptional. These algorithms are used infrequently, usually being called once at the beginning of a routine.

Although the algorithms are presented using integer division for clarity, Appendix A.7 shows how each floored integer division can be replaced by an integer multiplication and shift in these particular circumstances for binned `single`, `double`, and `quad`.

5.2 Update

Sometimes it is necessary to adjust the index of Y. For example, when adding $x \in \mathbb{F}$ to a K-fold binned number Y of index I in Algorithm 5.4, we will make the assumption that $|x| < 2^{b_I}$, which might require decreasing I to increase b_I. As another example, a new binned number Y is initialized to have Y_{0P} set to 0; therefore, before adding any value to Y, we must **update** the primary and carry fields of Y first. The process of updating Y to the necessary index is summarized succinctly in Algorithm 5.3.

ALGORITHM 5.3. *Update K-fold binned sum Y of $x_0, \ldots, x_{n-1} \in \mathbb{F}$ to have an index J sufficient to store the binned sum of $x_0, \ldots, x_n \in \mathbb{F}$. Y may be modified by this function.*

1: ***function*** UPDATE(x_n, Y)
2: I = BINNEDNUMBERINDEX(Y)
3: L = FLOATINGPOINTINDEX(x_n)
4: ***if*** $L < I$ ***then*** $\triangleright$ $J = \min(I, L)$
5: $[Y_{\min(I-L,K)P}, \ldots, Y_{K-1P}] = [Y_{0P}, \ldots, Y_{K-1-\min(I-L,K)P}]$
6: $[Y_{0P}, \ldots, Y_{\min(I-L,K)-1P}] = \begin{cases} [1.5\epsilon^{-1}2^{a_L}, \ldots, 1.5\epsilon^{-1}2^{a_{\min(I,K+L)-1}}] & \text{if } L > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.5\epsilon^{-1}2^{a_{L+1}}, \ldots, 1.5\epsilon^{-1}2^{a_{\min(I,K+L)-1}}] & \text{otherwise} \end{cases}$
7: $[Y_{\min(I-L,K)C}, \ldots, Y_{K-1C}] = [Y_{0C}, \ldots, Y_{K-1-\min(I-L,K)C}]$
8: $[Y_{0C}, \ldots, Y_{\min(I-L,K)-1C}] = [0, \ldots, 0]$
9: ***end if***
10: ***end function***

Ensure: *If Y_{0P} was exceptional, it is unchanged. If Y_{0P} was finite and x_n is exceptional, Y_{0P} is finite. Otherwise, if x_n is finite, the following hold:*
Y has index J where J is the greatest integer such that $|x_n| < 2^{b_J}$, $\max(|x_j|) < 2^{b_J}$, and $J \leq i_{\max} - K + 1$.
$\mathcal{Y}_k = \sum_{j=0}^{n-1} d(x_j, J + k)$
$Y_{kP} \in \begin{cases} [1.5\epsilon^{-1}2^{a_{J+k}}, 1.75\epsilon^{-1}2^{a_{J+k}}) & \text{if } J + k > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}}) & \text{if } J + k = 0 \end{cases}$

The update operation is described in the "Update" section (lines 7 to 17) of Algorithm 6 in Reference [15]. We have modified this algorithm to use our new functions BINNEDNUMBERINDEX and FLOATINGPOINTINDEX and handle exceptional values.

Although the "Ensure" claim looks similar to Equation (4.6), the index J is not necessarily the same as I in Equation (4.6).

It should be noted that if Y_{0P} is 0, then the update initializes all K collectors of Y. If Y_{0P} is exceptional, the "Ensure" holds, since I is 0. If Y_{0P} was finite and x_n is `+Inf`, `-Inf`, or `NaN`, then the above "Ensure" statement holds, since the update only sets Y_{0P} to finite values.

If Y represents the binned sum of finite values and x_n is finite, then the index J sufficient to store the binned sum of $x_0, \ldots, x_n$ is the greatest integer such that $\max(|x_0|, \ldots, |x_n|) < 2^{b_J}$, which is the minimum of L and I. If $L \geq I$, then we do not need to adjust Y, since $I = J$. If $L < I$, then we must adjust the index of Y by shifting Y's K collectors to represent the sums of slices in greater bins. Since Y represents the sum of $x_0, \ldots, x_{n-1}$, the new collectors Y_k with $0 \leq k < I - J$ are shifted in with value 0 (as described in Equation (4.3)) as the slices of $x_0, \ldots, x_{n-1}$ in these higher bins are zero. More formally, we know these greater bins have value 0, because $|x_j| < 2^{b_I} \leq 2^{a_{I-1}} \leq 2^{a_{J+k}}$ so $\sum_{j=0}^{n-1} d(x_j, J + k) = 0$ by Lemma 3.1. Note that we lose the lesser collectors when we shift in the greater collectors.

5.3 Deposit

The **deposit** operation is used to extract the slices of a floating point number and add them to the appropriate collectors of a binned number. Here, we refer to the deposit operation as Algorithm 5.4. Algorithm 5.4 deals with very large inputs (i.e., with exponents near $e_{\max}$) and exceptions, unlike the simpler version described in the "Extract K first bins" section (lines 18 to 20) of Algorithm 6 in Reference [15], which produces `NaN` on very large inputs. Algorithm 6 in Reference [15] was originally inspired by Rump's algorithm "ExtractVector" for error-free vector transformation [30, 31]. To give some intuition about variable names, the variable S will hold the "sum" and r will

hold the "remainder" after splitting. Thus, we split some number $q + r$ into a leading part q and a trailing part r.

ALGORITHM 5.4. *Extract slices of $x \in \mathbb{F}$ and add them to a K-fold binned number Y. Here, $(r|1)$ represents the result of setting the last bit of the significand (m_{p-1}) of a floating point number r to 1. Y may be modified by this function.*

Require: *If Y_{0P} is finite, there exists I such that $|x| < 2^{b_I}$ and*

$$Y_{kP}, Y_{kP} + d(x, I + k) \in \begin{cases} (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}}) & \text{if } I + k > 0 \\ (2^{e_{\max}}, 2 \cdot 2^{e_{\max}}) & \text{if } I + k = 0, \end{cases} \tag{5.1}$$

for all k such that $0 \le k < K$.

Operations are performed in any "to-nearest" rounding mode (no specific tie-breaking behavior is required).

```
1:  function DEPOSIT(x, Y)
2:      if x is exceptional or Y_0P is exceptional then
3:          Y_0P = Y_0P ⊕ x
4:      else
5:          if BINNEDNUMBERINDEX(Y) = 0 then
6:              r = x ⊙ 2^(W−p−1)
7:              S = Y_0P ⊕ (r|1)
8:              q = S ⊖ Y_0P
9:              Y_0P = S
10:             q = q ⊙ 2^(p−W)
11:             r = x ⊖ q
12:             r = r ⊖ q
13:             k = 1
14:         else
15:             r = x
16:             k = 0
17:         end if
18:         while k ≤ (K − 2) do
19:             S = Y_kP ⊕ (r|1)
20:             q = S ⊖ Y_kP
21:             Y_kP = S
22:             r = r ⊖ q
23:             k = k + 1
24:         end while
25:         Y_kP = Y_kP ⊕ (r|1)
26:     end if
27: end function
```

Ensure: *If Y_{0P} or x was exceptional, x was added to Y_{0P} as a floating point number. Otherwise, for all k such that $0 \le k < K$, the amount added to Y_{kP} by this algorithm is exactly $d(x, I + k)$.*

Algorithm 5.4 is very similar to the "Extract K first bins" section (lines 18 to 20) of Algorithm 6 in Reference [15] except for when the index of Y is 0, which is rare. In that case, the first collector Y_0 will be scaled by a factor of 2^{W-p-1} so the value of the first primary field Y_{0P} stays in the range $[2^{e_{max}}, 2 \cdot 2^{e_{max}})$ to avoid overflow. The slices corresponding to the first collector will also need to

be scaled by the same factor before being added. Since the scaling is by a power of 2, it does not change any significands of either the primary field or the input value.

If the slice q is scaled back by 2^{p-W+1} and does not overflow, we can simply subtract q from x and continue with the rest of the algorithm. However, if x is equal to the biggest value below the overflow threshold, then $d(x, 0) = 2 \cdot 2^{e_{max}}$, scaling q back by 2^{p-W+1} would cause overflow. To handle this special case, instead of scaling q back by 2^{p-W+1}, we only scale q back by 2^{p-W} to obtain a value of $d(x, 0)/2$ and subtract q twice in lines 11 to 12 to compute r. Note that if an FMA (Fused Multiply-Add) instruction is available, we would not have to explicitly scale q back; one single FMA instruction suffices to compute $r = x - q \cdot 2^{p-W+1}$ without any overflow.

Algorithm 5.4 sets the last bit of intermediate results (i.e., forms $(r|1)$) before adding them to Y_{kP} to fix the direction in which ties are broken during rounding. To show why this is necessary for reproducibility, consider what would happen if we did not fix the direction of tie-breaking. Suppose $Y_{kP} = 1.5$, $x_1 = \text{ulp}(Y_{kP})$, and $x_2 = 0.5\text{ulp}(Y_{kP})$. Notice that $(Y_{kP} \oplus x_1) \oplus x_2 = 1.5 + 2\text{ulp}(1.5)$, whereas $(Y_{kP} \oplus x_2) \oplus x_1 = 1.5 + \text{ulp}(1.5)$. Since the addition of x_2 results in a tie, tie-breaking in round-to-nearest even depends on the current value of Y_{kP} and is nonreproducible.

When adding r to Y_{kP}, setting the last bit of r only avoids ties in rounding when $\text{ulp}(r)$ is less than the rounding error in Y_{kP}. Mathematically, we will require $\text{ulp}(r) < 0.5\text{ulp}(Y_{kP})$ to prove Theorem 5.2. This is why we must enforce $a_{i_{\max}} \geq e_{\min} - p + 2$ so the smallest denormalized number is smaller than half of the least significant bit of the least bin. To show why this is necessary for accuracy, consider $Y_{kP} = 1.25$ and $r = 0.5 + \text{ulp}(Y_{kP})$. Notice that $\text{ulp}(r) = 0.5\text{ulp}(Y_{kP})$. While $Y_{kP} \oplus r = 1.75 + \text{ulp}(1.5) = Y_{kP} + r$ exactly, $Y_{kP} \oplus (r|1) = 1.75 + 2\text{ulp}(1.5)$, an error of $\text{ulp}(1.5)$ when we could have achieved an exact result! In this case, setting the last bit of r has introduced a tie into our rounding procedure.

Note that we present line 5 using BinnedNumberIndex for clarity, but an equivalent condition can be written using getexp, since we are only checking that Y_{0P} has a large enough exponent for Y to have an index of 0. For instruction-counting purposes in Appendix A.7, we count line 5 as the equivalent condition, $e_{\max} + p - 2W + 1 < \text{getexp}(Y_{0P})$.

Algorithm 5.4 uses at most $3K + 1 = 10$ floating point operations when $K = 3$, $K + 9 = 12$ integer operations, and 2 branches, potentially changing floating point and integer registers $2K + 3 = 9$ times. If the index of Y is known to be greater than 0 and Y and x are known to be finite, then no branches are necessary and we need only $3K - 2 = 7$ floating point operations and $K = 3$ integer operations. Notice that we do not count operations related to the loops in Algorithm 5.4, since they may be unrolled as K is constant. A full count of operations is given in Appendix A.7.

In Appendix A.6, we will show that the assumption (3.7) implies that depositing any floating point number will modify at most 3 collectors, since one floating point number can have at most 3 nonzero slices. Appendix A.6 contains discussion of how to modify Algorithm 5.4 to only deposit to those collectors, such that the cost of using the modified algorithm will still be 7 FLOPs independent of $K \geq 3$, i.e., independent of how much accuracy is desired.

Showing the correctness of Algorithm 5.4 is a nontrivial task. However, the main piece of the argument is described in Lemma 5.1, which explains how the last bit of r is set to break ties when rounding to-nearest so the amount added to Y_{kP} does not depend on the size of Y_{kP} so far.

Lemma 5.1. *Let $Y_{kP}, r \in \mathbb{F}$ be such that $Y_{kP}, Y_{kP} + \mathcal{R}_{\pm\infty}(r, e + 1) \in (2^{e+p}, 2 \cdot 2^{e+p})$ where $\text{ulp}(r) < 0.5\text{ulp}(Y_{kP})$. Then the operation $S = Y_{kP} \oplus (r|1)$ computed with any "to-nearest" rounding mode sets S to $Y_{kP} + \mathcal{R}_{\pm\infty}(r, e + 1)$ exactly.*

Proof. We will make repeated use of Equation (2.2) and the fact that because $\text{ulp}(r) < 0.5\text{ulp}(Y_{kP})$, $\text{ulp}(r) \leq 2^{e-1}$. Let s_r be the sign of r, so $s_r = 1$ if $r \geq 0$ and $s_r = -1$ if $r < 0$.

We start by showing that $|\mathcal{R}_{\pm\infty}(r, e + 1) - (r|1)| < 2^e$.

If $r = (r|1)$, then $|\mathcal{R}_{\pm\infty}(r, e+1) - (r|1)| = |\mathcal{R}_{\pm\infty}(r, e+1) - r| \leq 2^e$. Notice that we cannot have $|\mathcal{R}_{\pm\infty}(r, e+1) - r| = 2^e$, because this would imply that $r \in 2^e\mathbb{Z}$ when $r - \text{ulp}(r) \in 2\text{ulp}(r)\mathbb{Z}$ and $\text{ulp}(r) \leq 2^{e-1}$.

If $r \neq (r|1)$, this means that $r \in 2\text{ulp}(r)\mathbb{Z}$. Since we have $|\mathcal{R}_{\pm\infty}(r, e+1) - r| \leq 2^e$, we consider two cases.

If $|\mathcal{R}_{\pm\infty}(r, e+1) - r| = 2^e$, then $\mathcal{R}_{\pm\infty}(r, e+1) = r + s_r 2^e$. Since $(r|1) \neq r$, $(r|1) = r + s_r\text{ulp}(r)$, so $|\mathcal{R}_{\pm\infty}(r, e+1) - (r|1)| = |s_r 2^e - s_r\text{ulp}(r)| = |2^e - \text{ulp}(r)| < 2^e$, since $\text{ulp}(r) \leq 2^{e-1}$.

Otherwise, $|\mathcal{R}_{\pm\infty}(r, e+1) - r| < 2^e$. Since $\mathcal{R}_{\pm\infty}(r, e+1) \in 2^{e+1}\mathbb{Z} \subseteq 2\text{ulp}(r)\mathbb{Z}$ and $r \in 2\text{ulp}(r)\mathbb{Z}$, we have that $|\mathcal{R}_{\pm\infty}(r, e+1) - r| \in 2\text{ulp}(r)\mathbb{Z}$. Thus, $|\mathcal{R}_{\pm\infty}(r, e+1) - r| \leq 2^e - 2\text{ulp}(r)$. Since $|r - (r|1)| \leq \text{ulp}(r)$, $|\mathcal{R}_{\pm\infty}(r, e+1) - (r|1)| \leq 2^e - \text{ulp}(r) < 2^e$ by the triangle inequality.

Since we have just shown $|\mathcal{R}_{\pm\infty}(r, e+1) - (r|1)| < 2^e$, we have $|(Y_{kP} + \mathcal{R}_{\pm\infty}(r, e+1)) - (Y_{kP} + (r|1))| < 2^e$. Since $Y_{kP} + \mathcal{R}_{\pm\infty}(r, e+1) \in 2^{e+1}\mathbb{Z}$ and $Y_{kP} + \mathcal{R}_{\pm\infty}(r, e+1) \in (2^{e+p}, 2 \cdot 2^{e+p})$, $Y_{kP} \oplus (r|1)$ computed with any to-nearest rounding mode sets S to $Y_{kP} + \mathcal{R}_{\pm\infty}(r, e+1)$ exactly. □

With Lemma 5.1 in hand, we can show the correctness of Algorithm 5.4.

Theorem 5.2. *If the requirements of Algorithm 5.4 are satisfied, then after running the algorithm the "Ensure" claim holds.*

Proof. If Y_{0P} or x is exceptional, the proof is trivial. We therefore focus on the case when neither is exceptional. Let I be defined as in the requirements.

The proof proceeds by induction on k for all $0 \leq k < K$. We show that at the top of the loop on line 18, $I + k > 0$ and for all $l < k$, the amount $d(x, I + l)$ has been added to $\mathcal{Y}_{kP}$ and that $r = x - \sum_{i=0}^{I+k-1} d(x, i)$. Note that this claim applies even at the end of the loop when the loop condition is false.

We start by showing the base case both when $I = 0$ and when $I > 0$.

If $I = 0$, the algorithm will execute lines 6 to 12. Equation (5.1) provides us with the fact that $Y_{0P}, Y_{0P} + d(x, 0) \in (2^{e_{\max}}, 2 \cdot 2^{e_{\max}})$. Note that $d(x, 0) = \mathcal{R}_{\pm\infty}(x, a_0 + 1)$ by Equation (3.12).

If $|x| < 2^{a_0}$, then we have $d(x, 0) = 0$ by Lemma 3.1. After line 6, we also have that $|r| < 2^{a_0 - p + W - 1} = 2^{e_{\max} - p}$. Thus, Lemma 5.1 implies that line 7 adds $\mathcal{R}_{\pm\infty}(r, e_{\max} - p + 1) = 0 = d(x, 0)/2^{p-W+1}$ to Y_{kP}, and, therefore, $d(x, 0)$ is added to $\mathcal{Y}_{kP}$.

If $|x| \geq 2^{a_0}$, then line 6 sets $r = x/2^{p-W+1}$ exactly, as there is no underflow. Since x is finite, $|x| < 2 \cdot 2^{e_{\max}}$ and thus $|r| < 2^{e_{\max}+1-p+W-1} < 2^{e_{\max}-2}$ by Equation (3.6). Thus, $\text{ulp}(r) < 0.5\text{ulp}(Y_{0P})$ and Lemma 5.1 implies that in line 7, $\mathcal{R}_{\pm\infty}(r, a_0 + 1 - p + W - 1) = d(x, 0)/2^{p-W+1}$ is added to Y_{kP}, and, therefore, $d(x, 0)$ is added to $\mathcal{Y}_{kP}$.

By Equation (3.12), Theorem 3.4, and Equation (3.6), $d(x, 0)/2^{p-W+1} \in 2^{a_0+1-p+W-1}\mathbb{Z} = 2^{e_{\max}-p+1}\mathbb{Z}$ and $|d(x, 0)/2^{p-W+1}| \leq 2^{e_{\max}+1-p+W-1} < 2^{e_{\max}-2}$. Thus, $S - Y_{0P} = d(x, 0)/2^{p-W+1}$ is representable and q is computed exactly in line 8. Again by Theorem 3.4, $|d(x, 0)/2| \leq 2^{e_{\max}}$. Thus, in line 10, we have that $q = d(x, 0)/2$ exactly, since we scale by a power of two. By Theorem 3.3, $|x - d(x, 0)| \leq |x|$.

If $x \geq 0$, we have $x \geq x - d(x, 0)/2 \geq x - d(x, 0) \geq -x$.

If $x \leq 0$, we have $-x \geq x - d(x, 0) \geq x - d(x, 0)/2 \geq x$.

In either case, we have $|x - d(x, 0)/2| \leq |x|$.

Using Equation (3.6) and the fact that $|x| < 2^{b_0}$, $d(x, 0)/2 = \mathcal{R}_{\pm\infty}(x, a_0 + 1)/2 \in 2^{a_0}\mathbb{Z} \subset 2^{b_0-p+1}\mathbb{Z} \subseteq \text{ulp}(x)\mathbb{Z}$. Therefore, $x - d(x, 0)/2 \in \text{ulp}(x)\mathbb{Z}$. Combined with $|x - d(x, 0)/2| \leq |x|$ this implies that $x - d(x, 0)/2$ is representable, and that $r = x - q$ exactly in line 11. Again, since $d(x, 0)/2, x - d(x, 0)/2 \in \text{ulp}(x)\mathbb{Z}$ and $|x - d(x, 0)| \leq |x|$, $x - d(x, 0)$ is representable and $r = r - q$ exactly in line 12.

Thus, when $I = 0$, the inductive claim is satisfied the first time, we hit the loop in line 18.

If $I > 0$, we can satisfy the inductive claim by setting $k = 0$ and $r = x - \sum_{i=0}^{I+k-1} d(x,i) = x$.

We have shown the base case and proceed to the inductive step that relies on the lines 19 to 22.

By Equation (5.1), $Y_{kP}, Y_{kP} + d(x, I+k) \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$. If r is normalized, by Theorem 3.3, $|r| = |x - \sum_{i=0}^{I+k-1} d(x,j)| \le 2^{a_{I+k-1}}$. Thus, using Equation (3.6), $\mathrm{ulp}(r) \le 2^{a_{I+k-1}-p+1} = 2^{a_{I+k}+W-p+1} < 2^{a_{I+k}-1} < 0.5\mathrm{ulp}(Y_{kP})$. If r is denormalized, we have $\mathrm{ulp}(r) = 2^{e_{\min}-p+1}$, the unit in the last place of a denormalized number. Using Equation (3.9), $\mathrm{ulp}(r) = 2^{e_{\min}-p+1} \le 2^{a_{i_{\max}}-1} \le 2^{a_{I+k}-1} < 0.5\mathrm{ulp}(Y_{kP})$. Thus, we have $\mathrm{ulp}(r) < 0.5\mathrm{ulp}(Y_{kP})$ and by Lemma 5.1 and Equation (3.12), line 19 sets $S = Y_{kP} + \mathcal{R}_{\pm\infty}(r, a_{I+k}+1) = Y_{kP} + \mathcal{R}_{\pm\infty}(x - \sum_{i=0}^{I+k-1} d(x,j), a_{I+k}+1) = Y_{kP} + d(x, I+k)$. Since $d(x, I+k) \in 2^{a_{I+k}+1}\mathbb{Z}$ and $|d(x, I+k)| \le 2^{b_{I+k}} = 2^{a_{I+k}+W} < 2^{a_{I+k}+p-2}$ by Equation (3.6) and Theorem 3.4, line 20 sets $q = d(x, I+k)$ exactly. Note that $r = x - \sum_{i=0}^{I+k-1} d(x,j)$ before executing line 22. Since $d(x, I+k) = \mathcal{R}_{\pm\infty}(x - \sum_{i=0}^{I+k-1} d(x,j), a_{I+k}+1)$, Equation (2.2) gives us that $|x - \sum_{i=0}^{I+k} d(x,j)| \le |x - \sum_{i=0}^{I+k-1} d(x,j)|$ and $d(x, I+k) \in \mathrm{ulp}(x - \sum_{i=0}^{I+k-1} d(x,j))\mathbb{Z}$. Thus, $r = x - \sum_{i=0}^{I+k} d(x,j)$ exactly in line 22.

Thus, we have shown the claim for all k, $0 \le k < K$. Since the line 25 is the same as line 19 and the same assumptions are satisfied, the proof is complete. □

5.4 Renormalize

When depositing values into a K-fold binned number Y of index I, Algorithm 5.4 assumes Equation (5.1) throughout the routine. To enforce this condition, the binned number must be **renormalized** to recenter Y_{kP} within a smaller range by shifting value from $\mathcal{Y}_{kP}$ to $\mathcal{Y}_{kC}$. The renormalization procedure is shown in Algorithm 5.5.

ALGORITHM 5.5. *Renormalize a K-fold binned number Y. Y may be modified by this function.*

Require: *If Y_{0P} is finite and nonzero,*

$$Y_{kP} \in \begin{cases} [1.25\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}}) & \text{if } I+k > 0 \\ [1.25 \cdot 2^{e_{\max}}, 2 \cdot 2^{e_{\max}}) & \text{if } I+k = 0 \end{cases} \tag{5.2}$$

for all k such that $0 \le k < K$.

```
1:  function RENORMALIZE(Y)
2:      if Y_0P is finite and Y_0P ≠ 0 then
3:          for k = 0 to K − 1 do
4:              if Y_kP < 1.5 ⊙ ufp(Y_kP) then
5:                  Y_kP = Y_kP(⊕0.25 ⊙ ufp(Y_kP))
6:                  Y_kC = Y_kC ⊖ 1
7:              end if
8:              if Y_kP ≥ 1.75 ⊙ ufp(Y_kP) then
9:                  Y_kP = Y_kP(⊖0.25 ⊙ ufp(Y_kP))
10:                 Y_kC = Y_kC ⊕ 1
11:             end if
12:         end for
13:     end if
14: end function
```

Ensure: *If Y_{0P} was finite and nonzero, the values $\mathcal{Y}_k$ are unchanged and*

$$Y_{kP} \in \begin{cases} [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}}) & \text{if } I+k > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}}) & \text{if } I+k = 0 \end{cases} \tag{5.3}$$

for all k such that $0 \le k < K$. Otherwise, Y_{0P} is unchanged.

The renormalization operation is described in the "Carry-bit Propagation" section (lines 21 to 32) of Algorithm 6 in Reference [15], although it has been slightly modified not to include an extraneous case that covered an unused range of Y_{kP} (lines 25 to 27). Algorithm 5.5 must check for exceptional values, because lines 3 to 12 could change `+Inf` or `-Inf` to `NaN` depending on how ufp() behaves when given exceptional values. In total, Algorithm 5.5 uses $7K + 1 = 22$ FLOPs and $2K + 1 = 7$ conditional branches.

To show the reasoning behind the assumptions in Algorithm 5.5, we state Theorem 5.3.

THEOREM 5.3. *Assume $x_0, x_1, \ldots x_{n-1} \in \mathbb{F}$ are successively deposited (using Algorithm 5.4) in a K-fold binned number Y of index I where $\max |x_j| < 2^{b_I}$ and Y_{0P} is finite and nonzero. If Y initially satisfies Equation (5.3) and*

$$n \leq 2^{p-W-2},$$

then after all of the deposits, Y satisfies Equation (5.2).

Note that by Equation (3.6), $2^{p-W-2} > 1$.

PROOF. As the proof when $I + k = 0$ is almost identical to the case when $I + k > 0$, we consider here only the case when $I + k > 0$. First, note that $|d(x_j, I + k)| \leq 2^{b_{I+k}}$ by Theorem 3.4, where $d(x_j, I + k)$ is the amount added to Y_{kP} on iteration k.

By Theorem 5.2, DEPOSIT (Algorithm 5.4) extracts and adds the slices of x_j exactly (assuming $Y_{kP}, Y_{kP} + d(x_j, I + k) \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ at each step, which can be shown by applying this proof inductively to each j). By Theorem 3.4,

$$\left| \sum_{j=0}^{n-1} d(x_j, I + k) \right| \leq n2^{b_{I+k}} = n2^W 2^{a_{I+k}}.$$

If $n \leq 2^{p-W-2}$, then after the nth deposit

$$\begin{aligned} Y_{kP} &\in \left[(1.5\epsilon^{-1} - n2^W)2^{a_{I+k}}, (1.75\epsilon^{-1} + n2^W)2^{a_{I+k}} \right) \\ &\in [1.25\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}}). \end{aligned}$$

□

The UPDATE operation (Algorithm 5.3) initializes collectors with Y_{0P} such that Y satisfies Equation (5.3). If a binned number Y initially satisfies Equation (5.3) and we deposit at most 2^{p-W-2} floating point numbers into it, then Theorem 5.3 shows that after all of the deposits, Y satisfies Equation (5.2). Therefore, after another renormalization, Y satisfies Equation (5.3).

Note that the possible maximum number of summands, Equation (4.5) is slightly bigger than that of Reference [15]. Reference [15] proved the maximum number of summands using the renormalization function. Since Reference [15] used an unnecessary renormalization case on lines 25 to 27 of Algorithm 6 that incremented the carry field by 2, the bound was not as tight as it could be.

5.5 Add a Floating Point Number to a Binned Sum

Algorithm 5.6 allows the user to add a single floating point number to a binned sum. By running this algorithm iteratively on each element of a vector, the user can make a naive local sum. However, a more efficient summation algorithm is presented in Section 5.6, making Algorithm 5.6 more useful for small sums or sums where the summands are not gathered into a vector.

ALGORITHM 5.6. *Add $x_n \in \mathbb{F}$ to the K-fold binned sum Y of $x_0, \ldots, x_{n-1} \in \mathbb{F}$. (If $n = 0$, this implies that Y_{0P} is 0 and Y will be initialized in line 2.) Y may be modified by this function.*

```
1: function AddFloatingPointToBinnedSum(x_n, Y)
2:     Update(x_n, Y)
3:     Deposit(x_n, Y)
```

4: RENORMALIZE(*Y*)
5: ***end function***
Ensure: *Y is the K-fold binned sum of* $x_0, \ldots, x_n$.

As stated in the introduction under Goal 3, in contrast to Algorithm 6 of Reference [15], Algorithm 5.6 handles exceptions, very large summands, and very large intermediate results properly.

The following theorem proves the "Ensure" claim at the end of Algorithm 5.6.

THEOREM 5.4. *If the requirements of Algorithm 5.6 are satisfied, then after running the algorithm the "Ensure" claim holds.*

PROOF. As Y is the binned sum of $x_0, \ldots, x_{n-1}$, the requirements of UPDATE (Algorithm 5.3) are satisfied. The "Ensure" claim of UPDATE satisfies the requirements of DEPOSIT (Algorithm 5.4). Theorem 5.3 and the "Ensure" claim of DEPOSIT satisfy the requirements of RENORMALIZE (Algorithm 5.5). If any of the x_j were exceptional, it is trivial to verify that Y is now the binned sum of $x_0, \ldots, x_n$. We focus on the case when all x_j are finite.

After UPDATE, we have that I is the greatest integer such that $\max(|x_j|) < 2^{b_I}$ and $I \leq i_{\max} - K + 1$. Note that the index of Y is unchanged throughout the rest of the algorithm. After DEPOSIT, we have that $\mathcal{Y}_k = \sum_{j=0}^{n-1} d(x_j, I + k)$ and this is explicitly unchanged by RENORMALIZE. After RENORMALIZE, we have Equation (5.3), completing the requirements described in Equation (4.6) for Y to be the binned sum of $x_0, \ldots, x_n$. □

5.6 Sum Floating Point Numbers with a Binned Sum

Algorithm 5.7 is a binned summation algorithm that allows the user to efficiently add a vector of floating point numbers $x_m, \ldots, x_{m+n-1} \in \mathbb{F}$ to the binned sum Y of some $x_0, \ldots, x_{m-1} \in \mathbb{F}$.

As mentioned in Section 5.4, it is not necessary to perform a renormalization for every deposit, as would be done if Algorithm 5.6 were applied iteratively on each element of $x_m, \ldots, x_{m+n-1}$. At most 2^{p-W-2} values can be deposited in the binned number before having to perform the renormalization. Therefore, we have created Algorithm 5.7, a more efficient version of Algorithm 5.6 for when we need to reproducibly sum a local vector of floating point numbers. As Algorithm 5.7 computes a binned sum, it can be performed on the $x_m, \ldots, x_{m+n-1}$ in any order. However, for the simplicity of presenting the algorithm, it is depicted as running linearly from m to $m + n - 1$. Algorithm 5.7 uses only one binned number to hold the intermediate result of the recursive summation, and the vast majority of required instructions in the algorithm are due to DEPOSIT (Algorithm 5.4). An operation count is considered at the end of the section.

ALGORITHM 5.7. *Add* $x_m, \ldots, x_{m+n-1} \in \mathbb{F}$ *to the the K-fold binned sum Y of* $x_0, \ldots, x_{m-1} \in \mathbb{F}$. *(If* $m = 0$*, this implies that* Y_{0P} *is 0 and Y will be initialized in line 5.) Y may be modified by this function.*

1: ***function*** SUMFLOATINGPOINTWITHBINNEDSUM($[x_m, \ldots, x_{m+n-1}]$, *Y*)
2: $j = 0$
3: ***while*** $j < n$ ***do***
4: $nb = \min(n, j + 2^{p-W-2})$
5: UPDATE($\max(|x_{m+j}|, \ldots, |x_{m+nb-1}|)$, *Y*)
6: ***while*** $j < nb$ ***do***
7: DEPOSIT(x_{m+j}, *Y*)
8: $j = j + 1$
9: ***end while***
10: RENORMALIZE(*Y*)
11: ***end while***
12: ***return*** *Y*

13: ***end function***
Ensure: *Y is the K-fold binned sum of* $x_0, \ldots, x_{m+n-1}$.

Algorithm 5.7 is similar to Algorithm 6 in Reference [15], but requires no restrictions on the size or type (exceptional or finite) of inputs $x_0, \ldots, x_{m+n-1}$, since the methods it calls are implemented differently.

THEOREM 5.5. *If the requirements of Algorithm 5.7 are satisfied, then after running the algorithm the "Ensure" claim holds.*

PROOF. We show inductively that after each execution of line 10, Y is the binned sum of $x_0, \ldots, x_{m+j-1}$. Throughout the proof, assume that the value of all variables is specific to the given stage of execution.

In the case when at least one of $x_0, \ldots, x_{m+j-1}$ is exceptional, it is trivial to verify that the constituent functions behave correctly even if the max on line 5 does not propagate exceptions. We therefore focus on finite $x_0, \ldots, x_{m+j-1}$.

As a base case, on the first iteration of the loop on line 3, j is 0 and Y is given to be the binned sum of $x_0, \ldots, x_{m-1}$.

In subsequent iterations of the loop, we assume that at line 5, Y is the binned sum of $x_0, \ldots, x_{m+j-1}$.

In this case, the proof of Theorem 5.4 applies to lines 5 to 10 (keeping in mind that at most 2^{p-W-2} deposits are performed and by the "Ensure" claim of Algorithm 5.3, each finite x_{m+j} deposited satisfies $|x_{m+j}| < 2^{b_I}$). Therefore, after line 10, Y is the binned sum of $x_0, \ldots, x_{m+j-1}$. □

Note that the constant 2^{p-W-2} in line 4 is at its maximum value, and smaller values may be used to fit data into a cache.

As the binned sum is unique and independent of the ordering of its summands (Theorem 4.3), Algorithm 5.7 is reproducible for any permutation of its inputs.

At this point, an operation count should be considered. Since Algorithm 5.7 only performs the UPDATE and RENORMALIZE operations once for every 2^{p-W-2} times the DEPOSIT operation is performed (that is, $2^{53-40-2} = 2^{11}$ times for `double` precision and $2^{24-13-2} = 2^9$ times for `single` precision in our recommended parameter settings), the cost of Algorithm 5.7 is mostly due to the DEPOSIT operation and the absolute value and maximum instructions. Thus, our recommended parameter settings (discussed in Section 6.2) represent a tradeoff between accuracy and how often our numbers need renormalization. Algorithm 5.7 uses $n(2^{W-p+2}(7K+2)+3K+3) \approx 12.011n$ floating point operations, $n(2^{W-p+2}(K+19)+K+9) \approx 12.011n$ integer operations, $n(2^{W-p+2}(2K+3)+2K+3) \approx 9.004n$ potential register changes, and $2n(2^{W-p+2}(K+2)+1) \approx 2.005n$ branches. In the absence of exceptional and very large summands, the algorithm needs just $n(2^{W-p+2}(7K+2)+3K) \approx 9.011n$ floating point operations, $n(2^{W-p+2}(K+19)+K) \approx 3.011n$ integer operations, $2n(2^{W-p+1}(2K+3)+K) \approx 6.004n$ potential register changes, and $2^{W-p+3}n(K+2) \approx 0.005n$ branches. A full count of operations is given in Appendix A.7. Note that the very similar Algorithms 5 and 6 of Reference [15] have been shown experimentally to run within a factor of 1.2 to 1.6 of the runtime of conventional summation (see Figures 4 and 5 in Reference [15]).

5.7 Add a Binned Sum to a Binned Sum

An operation to produce the sum of two binned numbers is necessary to perform a reduction. For completeness, we include the algorithm here, although apart from the simplified renormalization algorithm, it is equivalent to Algorithm 7 in Reference [15].

ALGORITHM 5.8. *Add the K-fold binned sum* Z *of* $x_n, \ldots, x_{n+m-1} \in \mathbb{F}$ *to the K-fold binned sum* Y *of* $x_0, \ldots, x_{n-1} \in \mathbb{F}$. Y *may be modified by this function.*

1: ***function*** AddBinnedSumToBinnedSum(Y, Z)
2: ***if*** $Y_{0P} = 0$ ***then***
3: $Y = Z$
4: ***return***
5: ***end if***
6: ***if*** $Z_{0P} = 0$ ***then***
7: ***return***
8: ***end if***
9: I = BinnedNumberIndex(Y)
10: J = BinnedNumberIndex(Z)
11: ***if*** $J < I$ ***then***
12: $R = Z$
13: AddBinnedSumToBinnedSum(R, Y)
14: $Y = R$
15: ***return***
16: ***end if***
17: ***for*** $k = J - I$ *to* $K - 1$ ***do***
18: ***if*** $k = J = 0$ ***then***
19: $Y_{0P} = Y_{0P} \oplus (Z_{0P} \ominus 1.5 \cdot 2^{e_{\max}})$
20: ***else***
21: $Y_{kP} = Y_{kP} \oplus (Z_{k+I-JP} \ominus 1.5\epsilon^{-1}2^{a_{I+k}})$
22: ***end if***
23: $Y_{kC} = Y_{kC} \oplus Z_{k+I-JC}$
24: ***end for***
25: Renormalize(Y)
26: ***end function***

Ensure: *Y is the K-fold binned sum of $x_0, \ldots, x_{n+m-1}$.*

Algorithm 5.8 is identical (although simplified) to lines 1 to 18 of Algorithm 7 in Reference [15], but the Renormalize at the end is different (as we have a new renormalize operation). The definition of the binned number allows this same algorithm to work with exceptional values without modification.

This algorithm allows the user to perform reductions of arbitrary shapes. The user can use Algorithm 5.7 to sum locally, since it is faster, and use Algorithm 5.8 to merge the results of the local summations in an arbitrary reduction tree. Because the binned sum is well-defined, all reduction trees will produce the same binned sum.

Theorem 5.6. *If the requirements of Algorithm 5.8 are satisfied, then after running the algorithm the "Ensure" claim holds.*

Proof. If Y_{0P} or Z_{0P} are 0, then the algorithm correctly sets Y to the value of Z or Y (respectively).

If both Y_{0P} and Z_{0P} are exceptional, then BinnedNumberIndex (Algorithm 5.1) will return $I = J = 0$. The first iteration of the loop of line 17 will then set Y_{0P} to $Y_{0P} \oplus Z_{0P} \ominus 1.5 \cdot 2^{e_{\max}}$, which (since $1.5 \cdot 2^{e_{\max}}$ is finite) is equal to $Y_{0P} + Z_{0P}$, as desired.

If only one of Y_{0P} or Z_{0P} is exceptional, then BinnedNumberIndex will return $I = 0$ or $J = 0$ (respectively). The first iteration of the loop of line 17 will set Y_{0P} to the sum of the exceptional Y_{0P} or Z_{0P} (respectively) and some finite values. This sum is equal to the exceptional value. Therefore, if only one of Y_{0P} or Z_{0P} is exceptional, Y_{0P} is set to Y_{0P} or Z_{0P} (respectively), as desired.

We now focus on the case when both Y_{0P} and Z_{0P} are finite.

We must first prove that the addition in line 21 is exact. As it is almost identical, we leave out the case when $I + k = 0$ and focus on the case when $I + k > 0$. Since J is the index of Z, the index of Z_{k+I-J_P} is $J + (k + I - J) = I + k$. It means that $Z_{k+I-J_P} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$ and $Z_{k+I-J_P} \in 2^{a_{I+k}}\mathbb{Z}$. Therefore, $Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}} \in 2^{a_{I+k}}\mathbb{Z}$ and $Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}} \in [0, 0.25\epsilon^{-1}2^{a_{I+k}})$. This means $Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}}$ is representable and is exactly computed. Moreover, we have $Y_{kP} \in 2^{a_{I+k}}\mathbb{Z}$ and $Y_{kP} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$. Therefore $Y_{kP} + (Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}}) \in 2^{a_{I+k}}\mathbb{Z}$, and $Y_{kP} + (Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}}) \in [1.5\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$. This means $Y_{kP} + (Z_{k+I-J_P} - 1.5\epsilon^{-1}2^{a_{I+k}})$ is representable and is exactly computed, and that the requirements of Renormalize (Algorithm 5.5) apply. Thus, after line 25, we have Equation (5.3).

We assume that $n + m \leq 2^{2p-W-2}$ (4.5) and, therefore, $Y_{kC} + Z_{k+I-J_C}$ is exactly computed. We then have that $\mathcal{Y}_k = \sum_{j=0}^{m+n-1} d(x_j, I + k)$.

It is given that I is the greatest integer such that $|x_j| < 2^{b_I}$ for all $j, 0 \leq j \leq n-1$ and that J is the greatest integer such that $|x_j| < 2^{b_J}$ for all $j, n \leq j \leq n+m-1$ where both I and J are at most $i_{\max} - K + 1$. Since $I < J$, I is the greatest integer such that $|x_j| < 2^{b_I}$ for all $j, 0 \leq j \leq n+m-1$ and $I \leq i_{\max} - K + 1$.

This completes the requirements described in Equation (4.6) for Y to be the binned sum of $x_0, \ldots, x_{n+m-1}$. □

5.8 Convert a Binned Sum to a Floating Point Number

Converting a binned sum to a floating point number amounts to carefully evaluating Equation (4.4), which is just a sum of scaled and offset fields of the binned number. Since Theorem 4.3 guarantees that all fields in the binned sum are reproducible, any deterministic method to evaluate Equation (4.4) will also be reproducible.

For now, let Y be the binned sum of finite $x_0, \ldots, x_{n-1} \in \mathbb{F}$.

Reference [15] offered no method to convert from a binned number to a floating point result, so we compare our method to the naive evaluation of Equation (4.4) by straightforward recursive summation. The following algorithms in this section are original.

If we recursively sum the terms in Equation (4.4) in an arbitrary deterministic order and use the standard recursive summation error bound [18, (2.6)], we can only apply the error bound (6.2) in Section 6.1. In contrast, if we sum the terms in the following summation order motivated by Reference [13, Theorem 1]

$$\begin{aligned}
&\underbrace{(((\ldots(}_{2K}\mathcal{Y}_{0C}) \oplus \mathcal{Y}_{1C}) \quad \oplus \mathcal{Y}_{0P}) \quad \oplus \mathcal{Y}_{2C}) \quad \oplus \mathcal{Y}_{1P}) \quad \oplus \ldots \\
&\qquad \ldots \oplus \mathcal{Y}_{kC}) \quad \oplus \mathcal{Y}_{k-1P}) \oplus \mathcal{Y}_{k+1C}) \oplus \mathcal{Y}_{kP}) \quad \oplus \ldots \\
&\qquad \ldots \oplus \mathcal{Y}_{K-2C}) \oplus \mathcal{Y}_{K-3P}) \oplus \mathcal{Y}_{K-1C}) \oplus \mathcal{Y}_{K-2P}) \oplus \mathcal{Y}_{K-1P}),
\end{aligned} \tag{5.4}$$

then we compute the sum (4.4) with a relative error of at most about 7ϵ (modulo over/underflow) and we can apply the error bound (6.1) in Section 6.1. As discussed in more detail in Section 6.1, when there is a lot of cancellation (when $\sum |x_j| \ll n \cdot \max(|x_j|)$) the error bound (6.1) can be as much as 2^{29} times smaller than the error bound (6.2) (assuming `double` precision and our standard choices of $W = 40$ and $K = 3$ explained in Section 6.2).

Unfortunately, simply evaluating Equation (4.4) in the order specified by Equation (5.4) does not guard against unnecessary overflows. An **unnecessary overflow** is an overflow in an algorithm when there would be no overflow in the final result if intermediate calculations were performed with a large enough exponent range. In this section, we present two versions of our conversion algorithm. Algorithm 5.9 uses a floating point format with an expanded exponent range to

guarantee that no unnecessary overflow occurs, and Algorithm 5.10 uses the same floating point format as in the rest of the algorithm, along with scaling, to avoid unnecessary overflow.

We will refer to the floating point type that we use to hold the sum during computation as the **intermediate** floating point type. Let the precision of the intermediate floating point type be p_{interm}. Let the exponent range of the intermediate floating point type be $[e_{\text{interm,min}}, e_{\text{interm,max}}]$. Let the machine epsilon of the intermediate type be $\epsilon_{\text{interm}} = 2^{-p_{\text{interm}}}$. The intermediate type must satisfy $p_{\text{interm}} \geq p$, $e_{\text{interm,min}} \leq e_{\text{min}}$, and $e_{\text{interm,max}} \geq e_{\text{max}}$. Additionally, if $e_{\text{interm,max}}$ is large enough the intermediate type can contain the intermediate sum without special scaling to avoid unnecessary overflow.

THEOREM 5.7. *Let Y be a binned sum. Then, we have*

$$\max |\mathcal{Y}_{kP}| \leq 2^{a_{I+k}+p-1}, \tag{5.5}$$

$$\max |\mathcal{Y}_{kC}| \leq 2^{a_{I+k}+2p-2}, \tag{5.6}$$

and

$$\max |\mathcal{Y}| \leq 2^{e_{\max}-W+2p}, \tag{5.7}$$

where $\mathcal{Y}_{kP}$, $\mathcal{Y}_{kC}$, and $\mathcal{Y}$ are given by Equations (4.1), (4.2), and (4.4).

PROOF. By Equation (4.2), we have

$$\mathcal{Y}_{kC} = (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC}.$$

Therefore,

$$\max |\mathcal{Y}_{kC}| \leq 2^{a_{I+k}+2p-2}$$

By Equation (4.1), we have

$$\mathcal{Y}_{kP} = Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}.$$

We also fix the exponent of Y_{kP} as in Equation (4.6), which yields

$$Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}}).$$

Therefore,

$$\max |\mathcal{Y}_{kP}| < 2^{a_{I+k}+p-1}.$$

By Equations (3.2) and (4.4), we then have

$$\begin{aligned}
|\mathcal{Y}| &\leq \sum_{k=0}^{K-1} \max |\mathcal{Y}_{kC}| + \sum_{k=0}^{K-1} \max |\mathcal{Y}_{kP}| \\
&< \sum_{k=0}^{i_{\max}} 2^{a_k+2p-2} + \sum_{k=0}^{i_{\max}} 2^{a_k+p-1} \\
&= \sum_{k=0}^{i_{\max}} 2^{e_{\max}-(k+1)W+1+2p-2} + \sum_{k=0}^{i_{\max}} 2^{e_{\max}-(k+1)W+1+p-1} \\
&\leq \frac{2^{e_{\max}-W+2p-1}}{1-2^{-W}} + \frac{2^{e_{\max}-W+p}}{1-2^{-W}} \\
&\leq 2^{e_{\max}-W+2p},
\end{aligned}$$

where in the last two steps, we used Equations (3.8) and (3.7). □

If the maximum exponent of the intermediate floating point type satisfies $e_{\text{interm,max}} \geq e_{\max} - W + 2p$, then Equation (5.7) implies that no special cases to guard against overflow are needed. Theorem 6.1 in Section 6.1.1 implies that the computed sum is accurate to within a factor of $1 + 7\epsilon$

of the exact sum; therefore, the exponent of the computed sum will stay less than or equal to $e_{\max} - W + 2p$ and will not overflow. Note that $e_{\max} - W + 2p > e_{\max}$.

Algorithm 5.9 represents a conversion routine in such a case.

ALGORITHM 5.9. *Convert K-fold binned sum Y to $x \in \mathbb{F}$. x may be modified by this function.*

Require: *The variable z is stored using an intermediate floating point type satisfying $p_{\text{interm}} \geq p$, $e_{\text{interm,min}} \leq e_{\min}$, and $e_{\text{interm,max}} \geq e_{\max} - W + 2p$.*
Operations are performed in some "to-nearest" rounding mode (no specific tie-breaking behavior is required).

```
1:  function ConvertBinnedSumToFloatingPoint(x, Y)
2:      if Y_0P is exceptional or Y_0P = 0 then
3:          x = Y_0P
4:          return
5:      end if
6:      z = Y_0C
7:      for k = 1 to K − 1 do
8:          z = z ⊕ Y_kC
9:          z = z ⊕ Y_(k−1)P
10:     end for
11:     z = z ⊕ Y_(K−1)P
12:     x = z
13: end function
```

Ensure: *If Y_{0P} is 0 or exceptional, then $x = Y_{0P}$. Otherwise, x is equal to the value (cast to the original floating point format, overflowing if necessary) that results from evaluating Equation (5.4) using an intermediate floating point format with enough exponent range to avoid intermediate overflow.*

As explained in Section 4, a value of 0 in the primary field of the first bin means that no numbers have been added to Y. In addition, as explained in Section 5.3, exceptional values (`+Inf`, `-Inf`, and `NaN`) are added directly to the primary field of the first bin Y_{0P}. Therefore, exceptional values are reproducibly propagated through Y_{0P}, which will be returned as the computed result after the final conversion. More precisely, a result of `NaN` means that there is at least one `NaN` in the input or there are both `+Inf` and `-Inf` in the input. A result of `+Inf` or `-Inf` means that there is one or more values of `+Inf` or `-Inf` of the same sign in the input, and the rest are of finite value.

Note that an overflow situation in Algorithm 5.9 is reproducible as the fields in Y are reproducible. z is deterministically computed from the fields of Y, and the condition that z overflows when being converted back to the original floating point type in line 12 is reproducible.

If an intermediate floating point type with an exponent range containing $[e_{\min}, e_{\max} - W + 2p]$ is not available and the lowest bin has index 0, a rare case, the fields of Y must be scaled down by some factor during addition and the sum scaled back up when subsequent additions can no longer affect an overflow situation.

If the scaled sum is to overflow, then its unscaled absolute value will be greater than or equal to $2 \cdot 2^{e_{\max}}$ and it will overflow regardless of the values of any $\mathcal{Y}_{kP}$ or $\mathcal{Y}_{kC}$ with $|\mathcal{Y}_{kP}| < 0.5\epsilon_{\text{interm}} 2^{e_{\max}}$ or $|\mathcal{Y}_{kC}| < 0.5\epsilon_{\text{interm}} 2^{e_{\max}}$. If the floating point sum has exponent greater than or equal to $e_{\max}$, then these numbers are not large enough to have any effect when added to the sum. If the sum has exponent less than $e_{\max}$, then additions of these numbers cannot cause the exponent of the sum to exceed $e_{\max}$ for similar reasons.

As the maximum exponent of the exact sum is at most $2^{e_{\max} - W + 2p}$, a sufficient scaling factor is 2^{2p-W}, meaning that the maximum exponent of the exact scaled sum is at most $e_{\max}$. Again using

a forward reference of Theorem 6.1 in Section 6.1.1, the computed scaled sum will stay close to the exact sum and will not overflow.

When $\max |\mathcal{Y}_{kP}| < 0.5\epsilon_{\text{interm}} 2^{e_{\max}}$ and $\max |\mathcal{Y}_{kC}| < 0.5\epsilon_{\text{interm}} 2^{e_{\max}}$, the sum may be scaled back up and the remaining numbers added without scaling. Notice that no overflow can occur during addition in this algorithm. If an overflow is to occur, it will happen only when scaling back up. As the fields in the binned number are reproducible, such an overflow condition is reproducible.

If the sum is not going to overflow, then the smaller values must be added as unscaled numbers to avoid underflow.

The inequalities (5.5) and (5.6) from Theorem 5.7 give us a good way to check when we can scale up the terms in the sum, as they are strictly decreasing (among primary and carry values). As W and p are known, the branch conditions in Algorithm 5.10 can be greatly simplified. The conditions are left as is to make it more clear what is being compared.

Algorithm 5.10 represents a conversion routine in the case when a floating point type with an expanded exponent range is not available.

ALGORITHM 5.10. *Convert K-fold binned sum Y to $x \in \mathbb{F}$. x may be modified by this function.*

Require: *The variable z is stored using an intermediate floating point type satisfying $p_{\text{interm}} \geq p$, $e_{\text{interm,min}} \leq e_{\min}$, and $e_{\text{interm,max}} \geq e_{\max}$.*
Operations are performed in some "to-nearest" rounding mode (no specific tie-breaking behavior is required).

```
function ConvertBinnedSumToFloatingPointWithScaling(x, Y)
    if Y_0P is exceptional or Y_0P = 0 then
        x = Y_0P
        return
    end if
    I = BinnedNumberIndex(Y)
    k = 1
    if a_I + 2p − 2 > e_max − p_interm − 1 then
        z = (Y_0C ⊙ 2^(W−2p))
        while k ≤ K − 1 and (a_I+k + 2p − 2 ≥ e_max − p_interm − 1 or a_I+k−1 + p − 1 ≥ e_max − p_interm − 1) do
            z = z ⊕ Y_kC ⊙ 2^(W−2p)
            z = z ⊕ Y_k−1P ⊙ 2^(W−2p)
            k = k + 1
        end while
        if a_I+K−1 + p − 1 ≥ e_max − p_interm − 1 then
            z = z ⊕ Y_K−1P ⊙ 2^(W−2p)
            x = z ⊙ 2^(2p−W)
            return
        end if
        z = z ⊙ 2^(2p−W)
    else
        z = Y_0C
    end if
    while k ≤ K − 1 do
        z = z ⊕ Y_kC
        z = z ⊕ Y_k−1P
        k = k + 1
```

28: ***end while***
29: $z = z \oplus \mathcal{Y}_{K-1P}$
30: $x = z$
31: ***end function***

Ensure: *If Y_{0P} is 0 or exceptional, then $x = Y_{0P}$. Otherwise, x is equal to the value (cast to the original floating point format, overflowing if necessary) that results from evaluating Equation (5.4) using an intermediate floating point format with enough exponent range to avoid intermediate overflow.*

If a binned number is composed of `single`, then `double` provides sufficient precision and exponent range to use as an intermediate type and Algorithm 5.9 may be used to convert to a floating point number. However, if a binned number is composed of `double`, many machines may not have any higher precision available. We therefore perform the sum using `double` as an intermediate type. As this does not extend the exponent range, we must use Algorithm 5.10 for the conversion.

6 ANALYSIS

Here, we formally analyze the error in our reproducible summation algorithm and summarize the usage of a binned number. Section 6.1 presents an original theorem regarding the error in a sum of a decreasing sequence of floating point numbers and utilizes this theorem to obtain error bounds for our algorithms. Section 6.2 explains limits on the usage of binned numbers in terms of user parameters and recommends some default parameters.

6.1 Error Bounds

Here, we derive an error bound on the final floating point answer obtained through binned summation as presented in this work. We compare this to the error bound on binned summation when a naive conversion routine is used. By **naive conversion**, we refer to a conversion routine that sums the contributions from each field of the binned number (as in Equation (4.4)) in some fixed, arbitrary order instead of the order specified by Equation (5.4). We also compare to the standard error bound on recursive summation. We present our error bounds in Section 6.1.1 and their proofs in Section 6.1.2.

6.1.1 *Statement of Error Bounds.*

Theorem 6.1 (Error in floating point result of binned summation). *Consider the K-fold binned sum Y of finite floating point numbers $x_0, \ldots, x_{n-1} \in \mathbb{F}$. We denote the exact sum $\sum_{j=0}^{n-1} x_j$ by T, the value of the binned sum as obtained by evaluating Equation (4.4) exactly by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained using an appropriate algorithm from Section 5.8 (Algorithm 5.9 or 5.10) by $\overline{\mathcal{Y}}$. Assuming the final answer does not overflow,*

$$\begin{aligned}\left|T - \overline{\mathcal{Y}}\right| &< \left(1 + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}\right)\left(n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right)\right) + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}|T| \\ &\approx n2^{W(1-K)} \max|x_j| + 7\epsilon|T|. \end{aligned} \tag{6.1}$$

Note that Equation (6.1) requires $K \geq 2$ to get a useful error bound. We can compare Equation (6.1) to the error bound obtained if Algorithms 5.9 and 5.10 evaluated Equation (4.4) in some order other than Equation (5.4). In this case, the conversion step from binned sum to floating point number is less accurate.

Lemma 6.2 (Error in floating point result of binned summation with naive conversion). *If in Theorem 6.1 we replace $\overline{\mathcal{Y}}$ with the floating point approximation of $\mathcal{Y}$ obtained by recursive summation (in some fixed, arbitrary order) of the contributions of each field of the binned number*

(4.4), then assuming the final answer does not overflow,

$$\begin{aligned}\left|T - \overline{\mathcal{Y}}\right| &< n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) \\ &\quad + \left(\frac{(2K-1)\epsilon}{1-(2K-1)\epsilon}\right)\left(\sum_{k=0}^{K-1}|\mathcal{Y}_{kP}| + \sum_{k=0}^{K-1}|\mathcal{Y}_{kC}|\right) \\ &\approx n \cdot \max|x_j|\left(2^{W(1-K)} + (2K-1)\epsilon\right). \end{aligned} \tag{6.2}$$

Equation (6.2) is not as tight as Equation (6.1) and grows linearly, instead of shrinking, as the user increases K in an attempt to increase accuracy. We can also compare Equation (6.1) to the error bound of recursive summation.

Lemma 6.3 (Error in floating point result of recursive summation). *Consider the recursive sum $\overline{\mathcal{Y}}$ of finite floating point numbers $x_0, \ldots, x_{n-1} \in \mathbb{F}$ in some arbitrary order. We denote the exact sum $\sum_{j=0}^{n-1} x_j$ by T. Assuming there is no overflow or underflow,*

$$\left|T - \overline{\mathcal{Y}}\right| < n\epsilon \sum_{j=0}^{n-1}|x_j| \leq n^2\epsilon \max|x_j|. \qquad \text{[18, (2.6)]}$$

We now compare the error bounds for the user who uses the values $W = 40$, $K = 3$ that we recommend for `double` ($p = 53$) in Section 6.2. Note that the term $7\epsilon|\sum_{j=0}^{n-1} x_j|$ is only seven times larger than the smallest possible error bound on rounding the exact sum of the x_j to the nearest floating point value. To compare the other terms, bound (6.1) grows like $2^{-80}n \cdot \max|x_j|$, whereas bound (6.2) grows like $5\epsilon n \cdot \max|x_j| = 5 \cdot 2^{-53}n \cdot \max|x_j|$, which is over 2^{29} times larger when the exact sum is tiny ($|T| < 2^{-32}n \cdot \max|x_j|$). To compare with Reference [18, (2.6)], we bound $\sum_{j=0}^{n-1}|x_j|$ by $n \cdot \max|x_j|$, which corresponds to the case when the input data are almost equal in magnitude. In such a regime, the error bound of the standard recursive summation [18, (2.6)] grows like n^2 instead of n, which becomes arbitrarily worse than both bounds (6.1) and (6.2) as the number of input values n grows. In the case when $\sum_{j=0}^{n-1}|x_j| \approx \max|x_j|$, for example when there are just a few large values and the others are small, then bounds [18, (2.6)] and (6.2) are almost of the same order of magnitude, which is still worse than Equation (6.1) by a factor of about 2^{26} when the exact sum is tiny (when $|T| < 2^{-30}n \cdot \max|x_j|$). Figure 2 compares the error bounds visually.

6.1.2 Proofs of Error Bounds. There are two sources of error in the final floating point sum produced through binned summation. The first is from the creation of a binned sum. The second is from the conversion from binned sum to a floating point number.

Lemma 6.4 analyzes the error from the creation of the binned sum.

Lemma 6.4. *Consider the K-fold binned sum Y of finite floating point numbers $x_0, \ldots, x_{n-1} \in \mathbb{F}$. We denote the exact sum $\sum_{j=0}^{n-1} x_j$ by T and the value of the binned sum as obtained by evaluating Equation (4.4) exactly by $\mathcal{Y}$. Then, we have:*

$$|T - \mathcal{Y}| \leq n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right). \tag{6.3}$$

Proof. The case of all zero input data is trivial, therefore, we assume that $\max|x_j|$ is nonzero. Let I be the index of Y. Let L be $I + K - 1$, the index of the least bin in Y. By Lemma 3.1 for all $i < I$ the slice of any x_j in bin i is $d(x_j, i) = 0$. Thus, Theorem 3.3 yields,

$$\begin{aligned}\left|x_j - \sum_{i=I}^{L} d(x_j, i)\right| &= \left|x_j - \sum_{i=0}^{L} d(x_j, i)\right| \\ &\leq 2^{a_L}. \end{aligned}$$

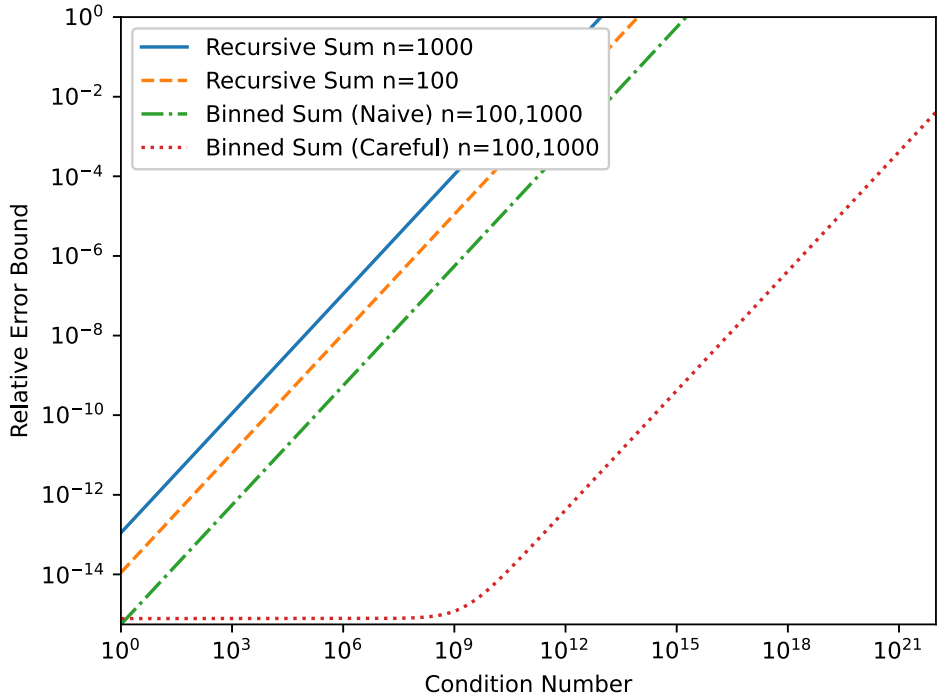

Fig. 2. Relative error bounds. Relative error bounds ($\frac{\text{absolute error bound}}{|\sum_{j=0}^{n-1} x_j|}$) in calculating $\sum_{j=0}^{n-1} x_j$ for different condition numbers (which we define as $\frac{n \cdot \max |x_j|}{|\sum_{j=0}^{n-1} x_j|}$). The condition number is a measure of how much cancellation occurs in the sum. "Binned Sum (Careful)" corresponds to Equation (6.1) divided by the exact sum. "Binned Sum (Naive)" corresponds to Equation (6.2) divided by the exact sum. "Recursive Sum" corresponds to Reference [18, (2.6)] divided by the exact sum and due to a dependence on n multiple error bounds are shown. The maximum relative error shown is 1, since a relative error of 1 gives no guarantee of accuracy. It is assumed that we sum using `double` ($p = 53$), $K = 3$, and $W = 40$.

Either $2^{a_L} \leq 2^{W(1-K)} \max |x_j|$ or $2^{a_L} > 2^{W(1-K)} \max |x_j|$.

Consider the latter. Assume for contradiction that $L < i_{\max}$. Then, we have that $\max |x_j| < 2^{a_L + W(K-1)} = 2^{a_{L-K+1}} = 2^{a_I} = 2^{b_{I+1}}$. By Equation (4.6), I is the greatest integer such that $\max |x_j| < 2^{b_I}$ and $I \leq i_{\max} - K + 1$, a contradiction, since $I + 1$ also satisfies these conditions. Thus, L must be $i_{\max}$, implying that

$$2^{a_L} = 2^{a_{i_{\max}}} \leq 2^{e_{\min}-2},$$

where Equation (3.11) was used in the last inequality.

Therefore,

$$2^{a_L} \leq \max(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}).$$

Since the summation in each bin Y_i is exact, we have

$$\begin{aligned} |T - \mathcal{Y}| &= \left| \sum_{j=0}^{n-1} x_j - \sum_{i=I}^{L} \sum_{j=0}^{n-1} d(x_j, i) \right| = \left| \sum_{j=0}^{n-1} \left(x_j - \sum_{i=I}^{L} d(x_j, i) \right) \right| \\ &\leq n 2^{a_L} \\ &\leq n \cdot \max(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}). \end{aligned}$$

□

Now that we have shown a bound on the difference between the exact sum and the binned sum, we must show a bound on the difference between the binned sum and the final result returned by Algorithms 5.9 and 5.10. These conversion algorithms convert a binned number to a floating point number by carefully evaluating Equation (4.4) in the order specified by Equation (5.4). To show the accuracy of Algorithms 5.9 and 5.10, we first establish Lemma 6.5, an error bound on the sum of floating point numbers in order of strictly decreasing exponent, and then show that the ordering in Equation (5.4) satisfies the conditions of this bound.

It should be noted that Lemma 6.5 is similar to Reference [13, Theorem 1], but requires less intermediate precision by exploiting additional structure of the input data. It is possible that future implementers may make modifications to the binned number (adding multiple carry fields, changing the binning scheme, etc.) such that the summation of its fields cannot be reordered to satisfy the assumptions of Lemma 6.5. In such an event, Reference [13, Theorem 1] provides more general ways to sum the fields while still maintaining accuracy.

Intuition is shared between Reference [13, Theorem 1] and Lemma 6.5. Both rely on the observation that during the recursive sum of floating point numbers in decreasing order of their exponents, the value of the partial sum when the first error is encountered is very close to the true value of the sum, and subsequent additions will be increasingly inconsequential, since the exponents are decreasing. When an error occurs during rounding, the partial sum must have had a magnitude large enough that the last bits needed to be truncated. This means that the exponent of the partial sum must be greater than the exponent of whatever summand induced the error, and since the exponents are decreasing, greater than the exponents of all subsequent summands after the first error. Since our summands are decreasing in magnitude very quickly, we can geometrically bound the error of the remaining additions. In fact, we show that within just four more additions after the first error, the remaining summands will not change the value of the partial sum. We need to use assumptions (3.6) and (3.7) to show how the exponents of the summands will decrease as quickly as is required. In the proof, we specify that the input floating point numbers may be unnormalized, as we only need to use the upper bound on the magnitude of the exponents and the lower bound on the units in the last place. The real inputs may be normalized as long as an unnormalized representation exists.

Lemma 6.5. *We are given n finite floating point numbers $f_0, \ldots, f_{n-1}$ for which there exist (possibly unnormalized) finite floating point numbers $f'_0, \ldots, f'_{n-1}$ of the same precision such that*

(1) $f_j = f'_j$ *for all* $j \in \{0, \ldots, n-1\}$,
(2) $\text{getexp}(f'_0) > \cdots > \text{getexp}(f'_{n-1})$,
(3) $\text{getexp}(f'_j) \geq \text{getexp}(f'_{j+2}) + \lceil \frac{p+1}{2} \rceil$ *for all* $j \in \{0, \ldots, n-3\}$.

Let $S_0 = \overline{S_0} = f_0$, $S_j = S_{j-1} + f_j$, *and* $\overline{S_j} = \overline{S_{j-1}} \oplus f_j$ *(assuming rounding "to-nearest," breaking ties arbitrarily) so* $S_{n-1} = \sum_{j=0}^{n-1} f_j$. *Then, in the absence of overflow and underflow, we have*

$$\left| S_{n-1} - \overline{S_{n-1}} \right| < \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} |S_{n-1}| \approx 7\epsilon |S_{n-1}|.$$

Proof. Throughout the proof, let $f_j = 0$ if $j > n-1$ so $S_\infty = S_{n-1}$ and $\overline{S_\infty} = \overline{S_{n-1}}$.

Let m be the location of the first error such that $S_{m-1} = \overline{S_{m-1}}$ and $S_m \neq \overline{S_m}$.

If no such m exists, then the computed sum is exact ($S_{n-1} = \overline{S_{n-1}}$) and we are done.

If such an m exists, then because $\text{getexp}(f'_0) > \cdots > \text{getexp}(f'_m)$, $f_0, \ldots, f_m \in \text{ulp}(f'_m)\mathbb{Z}$. Thus, $S_m \in \text{ulp}(f'_m)\mathbb{Z}$.

We now show $|S_m| > 2 \cdot 2^{\text{getexp}(f'_m)}$. Assume for contradiction that $|S_m| \leq 2 \cdot 2^{\text{getexp}(f'_m)}$. Because $S_m \in \text{ulp}(f'_m)\mathbb{Z}$, this would imply that S_m is representable as a floating point number, a

contradiction as $\overline{S_m} \neq S_m$. Therefore, we have

$$|S_m| > 2 \cdot 2^{\text{getexp}(f'_m)}. \tag{6.4}$$

Because $\text{getexp}(f'_m) > \text{getexp}(f'_{m+1})$,

$$|f_{m+1}| < 2 \cdot 2^{\text{getexp}(f'_m)-1} = 2^{\text{getexp}(f'_m)}. \tag{6.5}$$

Because $\text{getexp}(f'_m) \geq \text{getexp}(f'_{m+2}) + \lceil \frac{p+1}{2} \rceil$ and $\text{getexp}(f'_0) > \cdots > \text{getexp}(f'_{n-1})$,

$$\begin{aligned}\left|\sum_{j=m+2}^{n-1} f_j\right| &\leq \sum_{j=m+2}^{n-1} |f_j| < \sum_{j=m+2}^{n-1} 2 \cdot 2^{\text{getexp}(f'_j)} \leq \sum_{j=m+2}^{n-1} 2 \cdot 2^{\text{getexp}(f'_m)-\lceil \frac{p+1}{2} \rceil-(m+2-j)} \\ &< \sum_{j=0}^{\infty} \left(2\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)-j} = \left(4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)}. \end{aligned} \tag{6.6}$$

We can combine Equations (6.5) and (6.6) to obtain

$$\left|\sum_{j=m+1}^{n-1} f_j\right| \leq \sum_{j=m+1}^{n-1} |f_j| < 2^{\text{getexp}f'_m} + \left(4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)} = \left(1 + 4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)}. \tag{6.7}$$

By Equations (6.4) and (6.7),

$$\begin{aligned}|S_{n-1}| = \left|\sum_{j=0}^{n-1} f_j\right| &\geq \left|\sum_{j=0}^{m} f_j\right| - \left|\sum_{j=m+1}^{n-1} f_j\right| = |S_m| - \left|\sum_{j=m+1}^{n-1} f_j\right| \\ &\geq 2 \cdot 2^{\text{getexp}(f'_m)} - \left(1 + 4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)} = \left(1 - 4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)}. \end{aligned} \tag{6.8}$$

By Equations (6.8) and (6.6),

$$\left|\sum_{j=m+2}^{n-1} f_j\right| < \left(4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)} \leq \frac{4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}} \left|\sum_{j=0}^{n-1} f_j\right|. \tag{6.9}$$

By Equations (6.8) and (6.7),

$$\left|\sum_{j=m+1}^{n-1} f_j\right| \leq \sum_{j=m+1}^{n-1} |f_j| \leq \left(1 + 4\sqrt{\epsilon}\right) 2^{\text{getexp}(f'_m)} \leq \frac{1 + 4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}} \left|\sum_{j=0}^{n-1} f_j\right|. \tag{6.10}$$

And by Equations (6.8) and (6.10),

$$|S_m| \leq \left|\sum_{j=0}^{n-1} f_j\right| + \left|\sum_{j=m+1}^{n-1} f_j\right| \leq \left(1 + \frac{1 + 4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}}\right) \left|\sum_{j=0}^{n-1} f_j\right| = \frac{2}{1 - 4\sqrt{\epsilon}} \left|\sum_{j=0}^{n-1} f_j\right|. \tag{6.11}$$

By definition, $\overline{S_{m+4}}$ is the computed sum of $\overline{S_m}$, $f_{m+1}, \ldots, f_{m+4}$ using the standard recursive summation technique. The quantity θ_n is defined in Reference [18] such that $|\theta_n| \leq n\epsilon/(1 - n\epsilon)$.

Using Reference [18, (2.4)],

$$\begin{aligned}\left|\overline{S_m} + \sum_{j=m+1}^{m+4} f_j - \overline{S_{m+4}}\right| &\leq \left|(\overline{S_m} + f_{m+1})\theta_4 + \sum_{j=m+2}^{m+4} f_j\theta_{m+5-j}\right| \\ &\leq |\theta_4|\left|\overline{S_m} + f_{m+1}\right| + \sum_{j=m+2}^{m+4} |\theta_{m+5-j}||f_j| \\ &\leq \frac{4\epsilon}{1-4\epsilon}\left|\overline{S_m} + f_{m+1}\right| + \frac{3\epsilon}{1-3\epsilon}\sum_{j=m+2}^{m+4}|f_j| \\ &\leq \frac{4\epsilon}{1-4\epsilon}\left(\left|\overline{S_m} - S_m\right| + |S_m + f_{m+1}|\right) + \frac{3\epsilon}{1-3\epsilon}\sum_{j=m+2}^{n-1}|f_j|.\end{aligned}$$

Since $S_{n-1} = S_m + f_{m+1} + \sum_{j=m+2}^{n-1} f_j$, we have

$$|S_m + f_{m+1}| = \left|S_{n-1} - \sum_{j=m+2}^{n-1} f_j\right| \leq |S_{n-1}| + \sum_{j=m+2}^{n-1}|f_j|.$$

Therefore,

$$\left|\overline{S_m} + \sum_{j=m+1}^{m+4} f_j - \overline{S_{m+4}}\right| \leq \frac{4\epsilon}{1-4\epsilon}\left|S_m - \overline{S_m}\right| + \frac{4\epsilon}{1-4\epsilon}|S_{n-1}| + \frac{7\epsilon}{1-4\epsilon}\sum_{j=m+2}^{n-1}|f_j|.$$

Using the triangle inequality, we have

$$\begin{aligned}\left|S_{m+4} - \overline{S_{m+4}}\right| = \left|S_m + \sum_{j=m+1}^{m+4} f_j - \overline{S_{m+4}}\right| &\leq \left|S_m - \overline{S_m}\right| + \left|\overline{S_m} + \sum_{j=m+1}^{m+4} f_j - \overline{S_{m+4}}\right| \\ &\leq \left(1 + \frac{4\epsilon}{1-4\epsilon}\right)\left|S_m - \overline{S_m}\right| + \frac{4\epsilon}{1-4\epsilon}|S_{n-1}| + \frac{7\epsilon}{1-4\epsilon}\sum_{j=m+2}^{n-1}|f_j|.\end{aligned}$$

Since $\overline{S_{m-1}} = S_{m-1}$, we have that $|S_m - \overline{S_m}| = |(\overline{S_{m-1}} + f_m) - \overline{S_m}| \leq \epsilon|S_m|$ and therefore

$$\begin{aligned}\left|S_{m+4} - \overline{S_{m+4}}\right| &\leq \frac{1}{1-4\epsilon}\epsilon|S_m| + \frac{4\epsilon}{1-4\epsilon}|S_{n-1}| + \frac{7\epsilon}{1-4\epsilon}\sum_{j=m+2}^{n-1}|f_j| \\ &\leq \frac{\epsilon}{1-4\epsilon}\left(|S_m| + 4|S_{n-1}| + 7\sum_{j=m+2}^{n-1}|f_j|\right).\end{aligned}$$

And by Equations (6.11) and (6.9),

$$\begin{aligned}\left|S_{m+4} - \overline{S_{m+4}}\right| &\leq \frac{\epsilon}{1-4\epsilon}\left(\frac{2}{1-4\sqrt{\epsilon}}|S_{n-1}| + 4|S_{n-1}| + 7\frac{4\sqrt{\epsilon}}{1-4\sqrt{\epsilon}}|S_{n-1}|\right) \\ &= \frac{\epsilon}{1-4\epsilon}\left(\frac{6+12\sqrt{\epsilon}}{1-4\sqrt{\epsilon}}|S_{n-1}|\right) = \frac{6\epsilon}{(1-2\sqrt{\epsilon})(1-4\sqrt{\epsilon})}|S_{n-1}| \\ &< \frac{6\epsilon}{1-6\sqrt{\epsilon}}|S_{n-1}|. \qquad (6.12)\end{aligned}$$

Notice that

$$\text{getexp}(f'_m) \geq \text{getexp}(f'_{m+2}) + \left\lceil \frac{p+1}{2} \right\rceil \geq \text{getexp}(f'_{m+4}) + 2\left\lceil \frac{p+1}{2} \right\rceil > \text{getexp}(f'_{m+5}) + 2\left\lceil \frac{p+1}{2} \right\rceil.$$

Therefore,

$$\text{getexp}(f'_m) \geq \text{getexp}(f'_{m+5}) + p + 2. \tag{6.13}$$

Because $\text{getexp}(f'_0) > \cdots > \text{getexp}(f'_{n-1})$, Equation (6.13) yields

$$\left| \sum_{j=m+5}^{n-1} f_j \right| \leq \sum_{j=m+5}^{n-1} |f_j| < \sum_{j=m+5}^{n-1} 2 \cdot 2^{\text{getexp}(f'_m)-p-2-(j-(m+5))} < \sum_{j=0}^{\infty} 2^{\text{getexp}(f'_m)-p-1-j} = \epsilon 2^{\text{getexp}(f'_m)}. \tag{6.14}$$

Using Equations (6.14) and (6.8),

$$\left| \sum_{j=m+5}^{n-1} f_j \right| < \epsilon 2^{\text{getexp}(f'_m)} \leq \frac{\epsilon}{1-4\sqrt{\epsilon}} |S_{n-1}|. \tag{6.15}$$

By Equations (6.12) and (6.15)

$$\begin{aligned} \left| S_{n-1} - \overline{S_{m+4}} \right| &\leq |S_{n-1} - S_{m+4}| + \left| S_{m+4} - \overline{S_{m+4}} \right| \\ &\leq \left| \sum_{j=m+5}^{n-1} f_j \right| + \frac{6\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}| \\ &\leq \frac{\epsilon}{1-4\sqrt{\epsilon}} |S_{n-1}| + \frac{6\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}| \\ &< \frac{7\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}|. \end{aligned} \tag{6.16}$$

When combined with Equation (6.8), this gives

$$\begin{aligned} \left| \overline{S_{m+4}} \right| &> \left(1 - \frac{7\epsilon}{1-6\sqrt{\epsilon}} \right) |S_{n-1}| \\ &\geq \left(1 - \frac{7\epsilon}{1-6\sqrt{\epsilon}} \right) \left(1 - 4\sqrt{\epsilon} \right) 2^{\text{getexp}(f'_m)} \\ &= \left(1 - 4\sqrt{\epsilon} - \frac{7\epsilon\left(1-4\sqrt{\epsilon}\right)}{1-6\sqrt{\epsilon}} \right) 2^{\text{getexp}(f'_m)}, \end{aligned}$$

which can be simplified to

$$\left| \overline{S_{m+4}} \right| > 2^{\text{getexp}(f'_m)-1} \tag{6.17}$$

because $\epsilon \leq 2^{-7}$, which is satisfied because of assumption (3.8).

Using Equation (6.13), for all $j \geq m+5$, we have

$$|f_j| < 2 \cdot 2^{\text{getexp}(f'_j)} \leq 2 \cdot 2^{\text{getexp}(f'_m)-p-2} = \epsilon \cdot 2^{\text{getexp}(f'_m)-1}. \tag{6.18}$$

And by Equations (6.18) and (6.17), all additions after f_{m+4} have no effect (since we are rounding to-nearest) and we have $\overline{S_{n-1}} = \overline{S_{m+4}}$. This, together with Equation (6.16), implies

$$\left| S_{n-1} - \overline{S_{n-1}} \right| < \frac{7\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}|.$$

The proof is complete. □

Now that we have Lemma 6.5, all that remains is to show that it applies to Algorithms 5.9 and 5.10. Since both algorithms add the numbers according to Equation (5.4), we must show that this ordering satisfies the assumptions of Lemma 6.5.

Lemma 6.6. *Consider the K-fold binned sum Y of index I of finite floating point numbers $x_0, \ldots, x_{n-1} \in \mathbb{F}$. We denote the value of the binned sum as obtained by evaluating Equation (4.4) exactly by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained using an appropriate algorithm from Section 5.8 (Algorithm 5.9 or 5.10) by $\overline{\mathcal{Y}}$. Assuming the final answer does not overflow,*

$$\left|\mathcal{Y} - \overline{\mathcal{Y}}\right| < \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}|\mathcal{Y}|.$$

Proof. We first show how to interpret the $\mathcal{Y}_{kP}$ and $\mathcal{Y}_{kC}$ as unnormalized floating point numbers and sort their exponents independently of the actual values of the fields. Note that this interpretation is to support reasoning about Equation (5.4) and does not affect the representation format of the data itself, since IEEE floating point formats do not permit unnormalized numbers beside exceptional values and denormalized numbers. Consider a K-fold binned number Y of index I. Since each value $\mathcal{Y}_{kP}$ in a primary field Y_{kP} is represented by an offset from $1.5\epsilon^{-1}2^{a_{I+k}}$ and $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$, with perhaps some scaling applied in the event that $I = 0$, $\mathcal{Y}_{kP}$ can be expressed exactly using an unnormalized floating point number $\mathcal{Y}'_{Pk}$ with an exponent of $a_{I+k} + p - 1$. As each carry field Y_{kC} is a count of renormalization adjustments later scaled by $0.25\epsilon^{-1}2^{a_{I+k}}$, $\mathcal{Y}_{kC}$ can be expressed exactly using an unnormalized floating point number $\mathcal{Y}'_{kC}$ with an exponent of $a_{I+k} + 2p - 3$.

First, we have $\mathrm{getexp}(\mathcal{Y}'_{kP}) > \mathrm{getexp}(\mathcal{Y}'_{k+1P})$ and $\mathrm{getexp}(\mathcal{Y}'_{kC}) > \mathrm{getexp}(\mathcal{Y}'_{k+1C})$ because $a_{I+k} > a_{I+k+1}$.

Next, note that

$$\mathrm{getexp}(\mathcal{Y}'_{kC}) = a_{I+k} + 2p - 3$$

and

$$\mathrm{getexp}(\mathcal{Y}'_{k-1P}) = a_{I+k-1} + p - 1 = a_{I+k} + W + p - 1.$$

Therefore, $\mathrm{getexp}(\mathcal{Y}'_{kC}) > \mathrm{getexp}(\mathcal{Y}'_{k-1P})$ by Equation (3.6).

Finally, note that

$$\mathrm{getexp}(\mathcal{Y}'_{k-2P}) = a_{I+k-2} + p - 1 = a_{I+k} + 2W + p - 1.$$

Therefore, $\mathrm{getexp}(\mathcal{Y}'_{kC}) < \mathrm{getexp}(\mathcal{Y}'_{k-2P})$ by Equation (3.7).

Combining the above inequalities, we see that the exponents of all the $\mathcal{Y}'_{kP}$ and $\mathcal{Y}'_{kC}$ are distinct and can be sorted as follows:

$$\begin{aligned}
&\mathrm{getexp}(\mathcal{Y}'_{0C}) > \mathrm{getexp}(\mathcal{Y}'_{1C}) \quad > \mathrm{getexp}(\mathcal{Y}'_{0P}) \quad > \mathrm{getexp}(\mathcal{Y}'_{2C}) \quad > \mathrm{getexp}(\mathcal{Y}'_{1P}) \quad > \ldots \\
&\qquad \ldots > \mathrm{getexp}(\mathcal{Y}'_{kC}) \quad > \mathrm{getexp}(\mathcal{Y}'_{k-1P}) > \mathrm{getexp}(\mathcal{Y}'_{k+1C}) > \mathrm{getexp}(\mathcal{Y}'_{kP}) \quad > \ldots \\
&\qquad \ldots > \mathrm{getexp}(\mathcal{Y}'_{K-2C}) > \mathrm{getexp}(\mathcal{Y}'_{K-3P}) > \mathrm{getexp}(\mathcal{Y}'_{K-1C}) > \mathrm{getexp}(\mathcal{Y}'_{K-2P}) > \mathrm{getexp}(\mathcal{Y}'_{K-1P})
\end{aligned}$$

Note that the above ordering is the same as that in Equation (5.4).

These unnormalized floating point numbers may, for convenience of notation, be referred to in decreasing order of unnormalized exponent as $\gamma'_0, \ldots, \gamma'_{2K-1}$.

We have just shown that

$$\mathrm{getexp}(\gamma'_0) > \cdots > \mathrm{getexp}(\gamma'_{2K-1}), \tag{6.19}$$

where γ_j denotes the normalized representation of the γ'_j. Note that $\gamma_j = \gamma'_j$ as real numbers and that $\mathrm{getexp}(\gamma_j) \leq \mathrm{getexp}(\gamma'_j)$.

If γ_j is a primary field, then either γ_{j+1} or γ_{j+2} is a primary field (with the exception of γ_{2K-1}). If γ_j is a carry field, then either γ_{j+1} or γ_{j+2} is a carry field (with the exception of γ_{2K-3}, but if we use the fact that $p \geq 8$ (3.8), we have $\text{getexp}(\gamma'_{2K-3}) = a_{I+K-1} + 2p - 3 \geq a_{I+K-1} + p + \lceil \frac{p+1}{2} \rceil - 1 = \text{getexp}(\gamma'_{2K-1}) + \lceil \frac{p+1}{2} \rceil$). Therefore, as $2W > p + 1$ and $W < p - 2$, for all $j \in \{0, \ldots, 2K - 3\}$

$$\text{getexp}(\gamma'_j) \geq \text{getexp}(\gamma'_{j+2}) + W \geq \text{getexp}(\gamma'_{j+2}) + \left\lceil \frac{p+1}{2} \right\rceil. \tag{6.20}$$

$\mathcal{Y}'_{kP}$ and $\mathcal{Y}'_{kC}$ can be expressed exactly using floating point numbers of the same precision as Y_{kP} and Y_{kC} (except in the case of overflow, in which a scaled version may be obtained), and such exact floating point representations can be obtained using Equations (4.1) and (4.2). By Equations (6.19) and (6.20), Lemma 6.5 applies to Equation (5.4) to yield

$$\left| \mathcal{Y} - \overline{\mathcal{Y}} \right| < \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} |\mathcal{Y}|. \qquad \square$$

With Lemmas 6.4 and 6.6, we have all the necessary ingredients to give the final error bounds. Reference [15] discusses the absolute error between the binned sum and the exact sum (addressed here by Lemma 6.4), but does not give a method to compute a floating point approximation of the binned sum. No error bound on the final floating point answer was given. Theorem 6.1 extends the error bound of Reference [15] all the way to the final return value of the algorithm, combining the results of Lemmas 6.4 and 6.6.

Proof of Theorem 6.1. Lemma 6.4 gives us

$$|T - \mathcal{Y}| \leq n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right).$$

Lemma 6.6 gives us

$$\left| \mathcal{Y} - \overline{\mathcal{Y}} \right| < \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} |\mathcal{Y}|.$$

By the triangle inequality,

$$|\mathcal{Y}| \leq |T| + |T - \mathcal{Y}| < n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right) + |T|.$$

The above results can be used to obtain Equation (6.1), the absolute error of the floating point approximation of a binned sum $|T - \overline{\mathcal{Y}}|$:

$$\begin{aligned}
\left| T - \overline{\mathcal{Y}} \right| &\leq |T - \mathcal{Y}| + \left| \mathcal{Y} - \overline{\mathcal{Y}} \right| \\
&< n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} |\mathcal{Y}| \\
&< n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right) \\
&+ \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} \left(n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right) + |T| \right) \\
&< \left(1 + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}\right) \left(n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right) \right) + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}} |T|. \qquad \square
\end{aligned}$$

Proof of Lemma 6.2. Lemma 6.4 gives us

$$|T - \mathcal{Y}| \leq n \cdot \max\left(2^{W(1-K)} \max |x_j|, 2^{e_{\min}-2}\right).$$

The standard error bound on recursive floating point summation [18, (2.6)] gives us

$$\left| \mathcal{Y} - \overline{\mathcal{Y}} \right| < \left(\frac{(2K-1)\epsilon}{1 - (2K-1)\epsilon} \right) \left(\sum_{k=0}^{K-1} |\mathcal{Y}_{kP}| + \sum_{k=0}^{K-1} |\mathcal{Y}_{kC}| \right).$$

Combining with the triangle inequality, we obtain

$$\begin{aligned}\left|T - \overline{\mathcal{Y}}\right| &< n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) \\ &\quad + \left(\frac{(2K-1)\epsilon}{1-(2K-1)\epsilon}\right)\left(\sum_{k=0}^{K-1}|\mathcal{Y}_{k\,P}| + \sum_{k=0}^{K-1}|\mathcal{Y}_{k\,C}|\right) \\ &\approx n \cdot \max|x_j|\left(2^{W(1-K)} + (2K-1)\epsilon\right).\end{aligned}$$

□

A user of binned summation may wish to have an expression for the error bound relative to the result $\overline{\mathcal{Y}}$, and not the exact sum T (since T is probably not available in practice). We show this error bound with Lemma 6.7:

Lemma 6.7 (Error in floating point result of binned summation (relative to result)). *Consider the K-fold binned sum Y of finite floating point numbers $x_0, \ldots, x_{n-1} \in \mathbb{F}$. We denote the exact sum $\sum_{j=0}^{n-1} x_j$ by T, the value of the binned sum as obtained by evaluating Equation (4.4) exactly by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained using an appropriate algorithm from Section 5.8 (Algorithm 5.9 or 5.10) by $\overline{\mathcal{Y}}$. Assuming the final answer does not overflow,*

$$\begin{aligned}\left|T - \overline{\mathcal{Y}}\right| &< n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1-6\sqrt{\epsilon}-7\epsilon}\left|\overline{\mathcal{Y}}\right| \\ &\approx n2^{W(1-K)} \max|x_j| + 7\epsilon\left|\overline{\mathcal{Y}}\right|. \end{aligned} \tag{6.21}$$

Proof. By the triangle inequality,

$$|\mathcal{Y}| \le \left|\overline{\mathcal{Y}}\right| + \left|\mathcal{Y} - \overline{\mathcal{Y}}\right|.$$

Applying Lemma 6.6 yields

$$|\mathcal{Y}| < \left|\overline{\mathcal{Y}}\right| + \frac{7\epsilon}{1-6\sqrt{\epsilon}}|\mathcal{Y}|.$$

After simplification,

$$|\mathcal{Y}| < \left(\frac{1}{1-\frac{7\epsilon}{1-6\sqrt{\epsilon}}}\right)\left|\overline{\mathcal{Y}}\right| = \frac{1-6\sqrt{\epsilon}}{1-6\sqrt{\epsilon}-7\epsilon}\left|\overline{\mathcal{Y}}\right|.$$

The above results can be used to obtain Equation (6.21), the absolute error of the floating point approximation of a binned sum $|T - \overline{\mathcal{Y}}|$:

$$\begin{aligned}\left|T - \overline{\mathcal{Y}}\right| &\le |T - \mathcal{Y}| + \left|\mathcal{Y} - \overline{\mathcal{Y}}\right| \\ &< n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1-6\sqrt{\epsilon}}|\mathcal{Y}| \\ &< n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1-6\sqrt{\epsilon}}\left(\frac{1-6\sqrt{\epsilon}}{1-6\sqrt{\epsilon}-7\epsilon}\left|\overline{\mathcal{Y}}\right|\right) \\ &= n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1-6\sqrt{\epsilon}-7\epsilon}\left|\overline{\mathcal{Y}}\right|.\end{aligned}$$

□

6.2 Suggested Parameters and Algorithmic Limits

Here, we clarify the limits of binned summation with respect to the parameters we have given. Certain key quantities are summarized in Table 2. As we will see, there are tradeoffs among accuracy, performance, memory, and the maximum number of summands as we vary W and K, so we explain these tradeoffs and how we made our final recommendations for W and K in Table 2.

Table 2. Suggested Parameter Settings

Data Type	`single`	`double`	`quad`
Default K	3	3	3
$K_{\min}$	2	2	2
$K_{\max}$	21	52	328
Default W	13	40	100
$W_{\min}$	13	28	58
$W_{\max}$	21	50	110
Endurance	2^9	2^{11}	2^{11}
Capacity	2^{33}	2^{64}	2^{124}
Error in computing $T = \sum_{j=0}^{n-1} x_j$ where $\overline{T} = n \cdot \max \lvert x_j \rvert$	$2^{-26}\overline{T} + 2^{-21}\lvert T \rvert$	$2^{-80}\overline{T} + 2^{-50}\lvert T \rvert$	$2^{-220}\overline{T} + 2^{-110}\lvert T \rvert$

As analyzed in Theorem 6.1, the maximum error in a K-fold binned number with W-bit bins is $2^{W(1-K)}$, so the minimum K with useful accuracy is 2. The error decreases as W and K increase. The maximum useful and allowed K is $i_{\max} + 1$, as this covers all of the bins and computes the exact sum (to within a relative error of 7ϵ, subject to overflow and underflow, as discussed in Section 6.1). We want to choose W and K so the error is at least as good as recursive summation using the underlying floating point format, implying $W(K-1)$ should be at least $p = 24$ in `single`, $p = 53$ in `double`, and $p = 113$ in `quad`.

By Equation (3.6), $W < p - 2$. By Equation (3.7), $2W > p + 1$. The combination of these two implies $p \geq 8$ (3.8). They also imply $(p+1)/2 < W < p - 2$, or $13 \leq W \leq 21$ in `single`, $28 \leq W \leq 50$ in `double`, and $58 \leq W \leq 110$ in `quad`.

Since $W < p$, and we require $W(K-1) \geq p$, we recommend $K = 3$ to minimize memory and operation counts.

By Theorem 5.3, a maximum of 2^{p-W-2} elements may be deposited into $Y_{k\,p}$ between renormalizations. This number is referred to as the **endurance** of a binned number. The larger the endurance, the closer the operation count of Algorithm 5.7 gets to the lower bound given by the cost of the Deposit operation. This motivates choosing W to be less than its upper bound of $p - 3$.

By Equation (4.5), a binned number is capable of representing the sum of at least 2^{2p-W-2} floating point numbers. This number is referred to as the **capacity** of the binned number. In principle, one could choose W so the capacity just exceeded the number of summands in a particular application, but in practice one might not know the number of summands ahead of time, so reproducibility is better guaranteed by choosing W small enough so the capacity is at least the largest number of expected summands, which we choose to be 2^{32} in `single` (so $2 \cdot 24 - W - 2 \geq 32$, or $W \leq 14$) and 2^{64} in `double` (so $2 \cdot 53 - W - 2 \geq 64$, or $W \leq 40$). These capacities are chosen to exceed the largest unsigned integer that can be represented in the same-sized integer format, so choosing W any smaller is unlikely to be useful. In `quad`, such a number would be impractically large (most systems use 64-bit pointers), so we will not constrain our choice by the corresponding inequality $2 \cdot 113 - W - 2 \geq 128$, or $W \leq 96$.

The binned number will, when used correctly, avoid intermediate overflow assuming the limits in the previous paragraphs are honored.

As discussed in Section 4.1.2, we assume gradual underflow and guarantee that our algorithms will round underflows to at least the nearest $2^{e_{\min}-1}$.

Now, we consider choosing W and K in the case of `single`. Our constraints above leave two choices: $W = 13$ or $W = 14$. We choose $W = 13$ to maximize endurance, which is 2^9, and so minimize cost. This also yields a capacity of 2^{33}.

In the case of double, our constraints limit W to the range $28 \leq W \leq 40$. When $W = 40$, the endurance is already quite large, 2^{11}, so increasing the endurance further (i.e., decreasing W) is unlikely to improve performance. The capacity is 2^{64}, which is as large as necessary. So, we choose $W = 40$ to maximize accuracy.

In the case of quad, we consider W greater than 96, so in the range $96 \leq W \leq 110$, since a capacity of 2^{128} is much larger than necessary. Increasing W reduces both the capacity and endurance, and also increases accuracy, so a reasonable tradeoff seems to be $W = 100$, which reduces the endurance to 2^{11}, the same as for double, leaving a capacity of (a still very large) 2^{124}.

In Reference [15], similar algorithms (Algorithms 5 and 6) were implemented with $K = 3$, $W = 40$ and performed renormalization once every 2,048 summands, running within 1.2 to 1.6 times the runtime of conventional recursive summation. We therefore assume that amortization over 2,048 summands is sufficient, but of course the precise selection of parameters should be specific to the application and based on experimentation, which is out of the scope of this article.

As mentioned in Section 3.1, the exponent range is so small in half that fixed-point arithmetic is likely a more efficient reproducible summation scheme for this format. Thus, we do not discuss parameters for half.

Although we have restricted our attention to cases where the fields of the binned number are of the same type as the numbers it is summing, our analysis does not preclude widening the summands to a larger floating point format before reproducible summation. If our summands are bfloat16 (a recently proposed floating point format with the same exponent range as single, but with reduced precision $p = 8$ to save space and increase efficiency [4]), then building a binned number out of bfloat16 would prove difficult, since we would be restricted to $W = 5$ and the capacity would therefore be limited to $2^9 = 512$. Instead, we could sum using a single precision binned number. If our summands are single, we could sum using a double precision binned number with $K = 2$, which could be faster, because it requires fewer instructions (although these instructions involve the more expensive double precision numbers). Since these tradeoffs depend heavily on architectural effects, we leave an exploration of these possibilities to future work.

7 CONCLUSIONS AND FUTURE WORK

The algorithms we have presented have been shown to sum binary IEEE 754-2008 floating point numbers accurately and reproducibly and require only a subset of the IEEE Floating Point Standard 754-2008 [1] and bitwise operations on the standard representations in memory, satisfying Goal 1. Our algorithms are more accurate than recursive summation, and by varying K the user can tune the accuracy (Goal 2). The algorithms avoid intermediate overflow and work on exceptional cases such as +Inf, -Inf, and NaN (Goal 3). Our algorithms require only one pass over the data, and only one parallel reduction is required (Goals 4 and 5). Our reproducible accumulator requires only six floating point words (Goal 6) and has been designed to work in existing software patterns for summation (Goal 7). We believe our approach is the first to satisfy all of our design goals.

We have specified all of the necessary steps to carry out reproducible summation in practice, including initialization of the accumulator, addition of floating point numbers to an accumulator, addition of an accumulator to an accumulator, and conversion from the intermediate binned number to a floating point result. It should be possible to use the algorithms presented here to create a user-friendly interface to a reproducible accumulator that could hide almost all of the complexity of reproducible summation while maintaining the flexibility of conventional summation.

In the future, we will show how our methods have been used to create reproducible absolute sums, dot products, norms, matrix-vector products, matrix-matrix products, and so on, in an optimized library called ReproBLAS.

APPENDIX

A ALTERNATE IMPLEMENTATIONS AND INSTRUCTION COUNTS

We made several design choices when formalizing our reproducible summation scheme. Here, we explore a design space of alternative implementations that could have performance advantages, depending on the architecture, and how exceptions may be handled. We describe variations on the algorithm described so far, which use different tie-breaking rounding modes (A.1), use the augmented addition operation in the new IEEE-754-2019 standard (A.2), handle exceptions differently but still reproducibly (A.3), deal with the entire range of denormalized numbers (A.4), deal with abrupt underflow (A.5), and how to get arbitrarily high precision while still doing only $9n + O(K)$ FLOPs (A.6). We also include a detailed accounting of instructions used by our algorithms (A.7).

A.1 Rounding Modes

In Algorithm 5.4, we must use a "to-nearest" rounding mode to exactly represent the error in our addition in lines 8 and 20. Although we require a to-nearest rounding mode, we explicitly do not require any specific tie-breaking behavior, because we were able to set the last bit of r in lines 7, 19, and 25. However, we could avoid setting the last bit of r if we had a tie-breaking behavior that did not depend on the significand of Y_{kP}, so the slices would be well-defined and the algorithm would be reproducible. (If the tie-breaking behavior depends on the significand of Y_{kP} and we do not break ties by setting the last bit of r, then the order in which numbers are added to Y_{kP} could change the amount that is added.)

With alternate tie-breaking behaviors, slices would be defined as

$$d(x,i) = \begin{cases} 0 \text{ if } |x| < 2^{a_i} \\ \mathcal{R}(x, a_i + 1) \text{ if } 2^{a_i} \le |x| < 2^{b_i} \\ \mathcal{R}\left(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\right) \text{ if } 2^{b_i} \le |x|, \end{cases} \tag{A.1}$$

where $\mathcal{R}$ is a rounding function that rounds to-nearest with tie-breaking dependent on the rounding mode used. Table A.1 contains the various tie-breaking behaviors of $\mathcal{R}$ for different rounding modes. Except for the first line in the table (which is the subject of this article), the tie-breaking of $\mathcal{R}$ can be derived from the tie-breaking in floating point using the facts that Y_{kP} and $Y_{kP} \oplus r$ are always both strictly positive.

By using a different to-nearest rounding mode, we would accumulate different slices, but because our rounding mode is to-nearest, the theorems and lemmas in Sections 3 and 4 would still hold, albeit with a new definition of slices. These sections contain the basic results to prove the rest of the

Table A.1. Slice Definitions for Various Tie-breaking Behaviors

Tie-breaking of Floating Point	Lines 7, 19, and 25 of Algorithm 5.4	Tie-breaking of $\mathcal{R}$
Any	$S = Y_{kP} \oplus (r\|1)$	Away From 0
To s_r	$S = Y_{kP} \oplus r$	Away From 0
To $s_r \cdot s_{Y_{kP}}$	$S = Y_{kP} \oplus r$	Away From 0
To $-s_r$	$S = Y_{kP} \oplus r$	Towards 0
To $-s_r \cdot s_{Y_{kP}}$	$S = Y_{kP} \oplus r$	Towards 0
Away From 0	$S = Y_{kP} \oplus r$	Towards ∞
Towards ∞	$S = Y_{kP} \oplus r$	Towards ∞
Towards 0	$S = Y_{kP} \oplus r$	Towards $-\infty$
Towards $-\infty$	$S = Y_{kP} \oplus r$	Towards $-\infty$

Note that $s_{Y_{kP}}$ is the sign of Y_{kP} and s_r is the sign of r.

article, so we can say that any of the above rounding modes and implementations of Algorithm 5.4 would lead to a reproducible summation scheme. However, tie-breaking rules produce different slices; they might not produce the same reproducible sums.

A.2 Augmented Operations

If we use a rounding mode where lines 7, 19, and 25 of Algorithm 5.4 can be $S = Y_{kP} \oplus r$ (see Table 3 for options), then we can use an operation for **augmented addition** in the new IEEE Floating Point Standard 754-2019 [5] to speed up Algorithm 5.4. Such an operation $f(Y_{kP}, r)$ would return (S, z) such that $S = Y_{kP} \oplus r$ and $z = Y_{kP} + r - (Y_{kP} \oplus r)$ where $\oplus$ uses some "to-nearest" rounding mode (ties are broken toward 0 in Reference [5]). Thus, lines 19 to 22 could be replaced by

$$(Y_{kP}, r) = f(Y_{kP}, r)$$

and line 25 could also be replaced by the same. Since the error in the addition is computed exactly by f, we would no longer need special scaling to avoid overflow and lines 6 to 12 could be replaced by

$$\begin{aligned} r &= x \odot 2^{W-p-1}, \\ (Y_{0P}, r) &= f(Y_{0P}, r), \\ r &= r \odot 2^{p-W+1}. \end{aligned}$$

Thus, if augmented addition is implemented as a single floating point instruction, it reduces the cost of the algorithm to $K = 3$ FLOPs (without any $(r|1)$ operations) when $I \neq 0$. If it is implemented as two floating point instructions (one for the sum and one for the error), it reduces the cost of the algorithm to $2K - 1 = 5$ FLOPs (without any $(r|1)$ operations) when $I \neq 0$ (notice that we do not need the error in the last sum). If $I = 0$, we would need to use two extra scaling operations.

A.3 Simple `+Inf`, `-Inf`, and `NaN` Handling

Although we require strict handling of `+Inf`, `-Inf`, and `NaN`, we observe that this can also be handled lazily. If Y_{0P} is exceptional, then Algorithm 5.3 leaves it unchanged. We can remove the check on line 2 of Algorithm 5.4 and we will execute lines 6 to 12 or lines 19 to 22. Lines 7 or 19 will set the last bit of a summand `+Inf` or `-Inf` so it is `NaN`. Thus, at the end of our summation algorithm, we will get a result of `NaN` if and only if any of our summands were exceptional (which is a reproducible behavior). Note that in this implementation, we will only get a result of `+Inf` or `-Inf` if the sum was too large to represent. If we wish, we can go back whenever we see a result of `NaN` and determine the true exceptional result.

If we are using a rounding mode where lines 7, 19, and 25 of Algorithm 5.4 can be $S = Y_{kP} + r$, then we do not need the check on line 2, because r is added directly to Y_{0P} as desired.

Even if we must explicitly check for exceptional input, we can move the check from inside Algorithm 5.4 to Algorithm 5.7, since this algorithm already passes over the data to determine the maximum absolute value of the input on line 5. If we see that our input is exceptional, we can compute our result directly with few conditional branches.

A.4 Adding Denormalized Floating Point Numbers

Algorithm 5.4 in Section 5.3 relies on setting the last bit of intermediate results to fix the direction of the rounding mode. However, if r is the quantity to be added to Y_{kP}, $\mathrm{ulp}(r)$ must be less than rounding error in Y_{kP} when added to Y_{kP}. Mathematically, we require $\mathrm{ulp}(r) < 0.5\mathrm{ulp}(Y_{kP})$ to prove Theorem 5.2 about the correctness of Algorithm 5.4. This is why we must enforce $a_{i_{\max}} \geq e_{\min} - p + 2$ so the least significant bit of the least bin is larger than twice the smallest denormalized number.

If we use a rounding mode that does not require setting the last bit of r in lines 7, 19, and 25 as discussed above in Section A.1, we no longer need $\text{ulp}(r) < 0.5\text{ulp}(Y_{kP})$. Therefore, we can add one or two more bins so we can accumulate slices in the bin (3.10).

If we cannot use a different rounding mode, we could also accumulate input in the least bins by scaling them up, analogously to how we handled summands close to $2 \cdot 2^{e_{\max}}$. We would scale r up before adding it to the least bins in Algorithm 5.4. A disadvantage to this approach is the additional branching cost incurred due to the conditional scaling of what might be multiple bins.

Another approach, if the rounding mode cannot be changed, is to add a special bin to accumulate the input ignored by the other bins. Its associated collector would have a primary field that has exponent $e_{\min}$ and would not be stored with any bias. In line 25 of Algorithm 5.4, we must check to see if we are adding to this least bin and if we are, simply execute the statement $Y_{kP} = Y_{kP} \oplus r$ instead. The finite inputs $r \in \mathbb{F}$ to this bin would satisfy $|r| \leq 2^{a_{i\max}} = e_{\min} - p + 2 + \left((e_{\max} - e_{\min} + p - 1) \mod W\right) \leq e_{\min} - p + W + 1$. Since for all $r \in \mathbb{F}$ we have that $r \in 2^{e_{\min}-p+1}\mathbb{Z}$, this bin would be able record at least 2^{p-W-2} additions exactly, after which point it could be renormalized similarly to the other bins.

Note that the assumptions in Section 6.1 do not necessarily hold when another bin is added, so the conversion routine may have to be modified.

A.5 Abrupt Underflow

If underflow is abrupt (meaning that results that would normally round to a number with magnitude less than $2^{e_{\min}}$ are instead rounded to zero), several approaches may be taken to modify the given algorithms to ensure reproducibility. Abrupt underflow is defined as an alternate exception handling mode in Section 8.2 of the IEEE 754 standard [1, 5]. Since "denormals are zero" (treating input denormal numbers as zero) mode is not defined in the standard, we do not consider it.

Again, the most straightforward approach would be to extract input in the denormalized range by scaling the smaller inputs up. This has the added advantage of increasing the accuracy of the algorithm to sum the entire denormal range.

A more efficient way to solve the problem would be to set the least bin to have $a_{i_{\max}} = e_{\min}$. This means that all the values smaller than $2^{e_{\min}}$ will not be extracted. This could be accomplished either by keeping the current binning scheme and having the least bin be of a width not necessarily equal to W, or by shifting all other bins to be greater. The disadvantage of shifting the other bins is that it may cause multiple greatest bins to overflow, adding multiple scaling cases. Setting such a least bin would enforce the condition that no underflow occurs, since all intermediate sums are either 0 or greater than the underflow threshold. The denormal range would be discarded.

Setting the least bin is similar to zeroing out the significand bits of each summand that correspond to values $2^{(e_{\min}-1)}$ or smaller. However, performing such a bitwise manipulation would likely be more computationally intensive and would not map as intuitively to our binning process.

In the case that reproducibility is desired on heterogeneous machines, where some processors may handle underflow gradually and others abruptly, the approach of setting a least bin is recommended. The binned sum using this scheme does not depend on whether or not underflow is handled gradually or abruptly, so the results will be the same regardless of where they are computed.

Again, remember that adding another bin may break assumptions in Section 6.1.

A.6 High Accuracy

Allowing the user to adjust accuracy yields an interesting tradeoff between performance and accuracy. Using only the existing interface, a basic superaccumulator [26] can be built by setting K

to its maximum value so almost the entire floating point range is summed. A careful examination of the error bound (6.1) shows that this would give almost exact results regardless of the dynamic range of the sum.

The Deposit operation (Algorithm 5.4) calculates K slices of each summand. This is an efficient approach when K is small. However, if we use a very large value of K, this is inefficient, since many of these slices will be zero. Lemma A.1 shows that each floating point number corresponds to at most three consecutive bins that may have nonzero slices. Therefore, we can optimize Deposit for a high-accuracy use case by only calculating these slices.

Lemma A.1. *Let $x \in \mathbb{F}$ be finite and let J be the largest integer such that $|x| < 2^{b_J}$. If $J \leq i_{\max} - 2$, then only the 3 slices $d(x, J), d(x, J+1), d(x, J+2)$ may be nonzero.*

Proof. If $J > 0$, $|x| < 2^{a_{J-1}}$ and by Lemma 3.1, $d(x, i) = 0$ for all $0 \leq i < J$.

Since J is maximal, $|x| \geq 2^{a_J}$. Then, we have that $\text{ulp}(x) \geq \epsilon|x| \geq 2^{a_J - p}$. Therefore, $\text{ulp}(x) \geq 2^{a_{J+2}+2W-p} > 2^{a_{J+2}+1}$ by Equation (3.7). Since $x \in \text{ulp}(x)\mathbb{Z}$ and $d(x, j) \in 2^{a_j+1}\mathbb{Z}$, $x - \sum_{j=0}^{J+2-1} d(x, j) \in \min(2^{a_{J+1}+1}, \text{ulp}(x))\mathbb{Z}$. Therefore, $d(x, J+2) = \mathcal{R}_{\pm\infty}(x - \sum_{j=0}^{J+2-1} d(x, j), a_{J+2} + 1) = x - \sum_{j=0}^{J+2-1} d(x, j)$. Thus, $d(x, i) = 0$ for all $J + 2 < i \leq i_{\max}$. □

We may now consider an optimized version of the Deposit operation (Algorithm 5.4) for a high-accuracy use case. We would first use FloatingPointIndex (Algorithm 5.2) to calculate J in Lemma A.1. Since the slices in bins J, $J+1$, and $J+2$ are the only slices that may be nonzero, we only need to run Algorithm 5.4 on the collectors corresponding to those bins. Although there is a branching cost to finding these collectors, running Algorithm 5.4 on three consecutive collectors takes only 7 FLOPs. Thus, if we assume both the number of summands n and the endurance 2^{p-W-2} are much larger than $K \leq i_{\max} + 1$, Sum (Algorithm 5.7) would still only require approximately $9n$ FLOPs in addition to the cost of running FloatingPointIndex on each input.

A.7 Instruction Counts

For completeness, Table A.2 displays the number of required instructions for the operations listed in the text. Note that some architectures use the same registers for integer and floating point quantities, and so perform no register changes. We do not include the cost of table lookups of known quantities or data structure field access in, e.g., the Update operation. We assume that it takes 1 FLOP to convert each field in Y to a field of $\mathcal{Y}$ using Equations (4.1) and (4.2), as primary fields need an offset and carry fields need a scale. We also do not count the cost of reading the input summands in the SumFloatingPointWithBinnedSum operation.

The BinnedNumberIndex and FloatingPointIndex operations each include an integer division. These are the only operations to include such an instruction. Integer division is much more expensive than other integer operations on many architectures, leading to the development of algorithms for division using "reciprocal multiplication" in situations where the divisor is constant [6, 16]. However, not only do our divisions involve a constant divisor, W, but the dividends are bounded and nonnegative. This allows us to further simplify such techniques and replace each division with an integer multiply and shift by precomputed constants. Assume that the dividend is some integer e. We know that in each division, $0 < e \leq E = e_{\max} - e_{\min} + p - W + 2$. Let 2^α be the smallest power of two, which is at least $E \cdot W$. Then let $\beta = \lceil 2^\alpha / W \rceil$. We claim that

$$(e \cdot \beta) >> \alpha = \lfloor e/W \rfloor. \tag{A.2}$$

For a short proof, notice that

$$(e \cdot \beta) >> \alpha = \lfloor e \cdot \lceil 2^\alpha / W \rceil / 2^\alpha \rfloor \geq \lfloor e/W \rfloor$$

Table A.2. Instruction Counts

Operation	Floating Point Operations	Integer Operations	Potential Register Changes	Branches
ufp		1	2	
getexp		3	1	
is exceptional?		2	1	
BinnedNumberIndex	1	6	1	1
FloatingPointIndex		8	1	
Update	1	16	2	3
Deposit	$3K + 1 = 10$	$K + 9 = 12$	$2K + 3 = 9$	2
Deposit (x and Y finite)	$3K + 1 = 10$	$K + 4 = 7$	$2K + 1 = 7$	1
Deposit (x and Y finite, I nonzero)	$3K - 2 = 7$	$K = 3$	$2K = 6$	
Renormalize	$7K + 1 = 22$	$K + 3 = 6$	$2K + 1 = 7$	$2K + 1 = 7$
AddFloatingPointTo BinnedSum	$10K + 3 = 33$	$2K + 28 = 34$	$4K + 6 = 18$	$2K + 6 = 12$
SumFloatingPointWith BinnedSum	$n(2^{W-p+2}(7K + 2) + 3K + 3) \approx 12.011n$	$n(2^{W-p+2}(K + 19) + K + 9) \approx 12.011n$	$n(2^{W-p+2}(2K + 3) + 2K + 3) \approx 9.004n$	$2n(2^{W-p+2}(K + 2) + 1) \approx 2.005n$
SumFloatingPointWith BinnedSum (x and Y finite)	$n(2^{W-p+2}(7K + 2) + 3K + 3) \approx 12.011n$	$n(2^{W-p+2}(K + 19) + K + 4) \approx 7.011n$	$n(2^{W-p+2}(2K + 3) + 2K + 1) \approx 7.004n$	$n(2^{W-p+3}(K + 2) + 1) \approx 1.005n$
SumFloatingPointWith BinnedSum (x and Y finite, I nonzero)	$n(2^{W-p+2}(7K + 2) + 3K) \approx 9.011n$	$n(2^{W-p+2}(K + 19) + K) \approx 3.011n$	$2n(2^{W-p+1}(2K + 3) + K) \approx 6.004n$	$2^{W-p+3}n(K + 2) \approx 0.005n$
AddBinnedSumTo BinnedSum	$10K + 9 = 39$	$3K + 29 = 38$	$2K + 5 = 11$	$3K + 11 = 20$
ConvertBinnedSumTo FloatingPoint	$4K = 12$	3	1	1
ConvertBinnedSumTo FloatingPointWithScaling	$5K + 1 = 16$	$K + 11 = 14$	2	$K + 4 = 7$

Our default settings are $K = 3$, $W = 40$, $p = 53$, and are treated as constant. Operations involving only constant numbers are not counted. "Potential register changes" counts the reinterpretations of integers as floats and vice versa according to the standard binary interchange format. We count comparisons and max and min as operations on the types being compared, and logical operations as integer operations. Branches are counted separately from the computation of the conditions in the branch. When different branch paths have different operation counts, we pick the larger count. We do not count the cost of loop control flow, because most loops can be unrolled, so the amortized loop control cost is negligible. Accesses to arrays, indexing calculations, lookup tables of known values, and field accesses into data structures are not counted. Three versions of SumFloatingPointWithBinnedSum and Depositare listed, corresponding to the cases when x and y are known to be finite and the case when I is not initially 0 and x is known to be small enough not to trigger the $I = 0$ branch. The cost of SumFloatingPointWithBinnedSum reflects the amortized cost over each summand (constant overhead is not counted).

and

$$e \cdot \lceil 2^\alpha / W \rceil / 2^\alpha < e \cdot (2^\alpha / W + 1)/2^\alpha \leq (e + 1)/W.$$

Since W is a positive integer and e is a nonnegative integer, there are no integers between e/W and $(e + 1)/W$, so our proof is complete.

Notice that for this trick to work, all intermediate quantities must be representable in the underlying integer format. Using Tables 1 and 2 and taking a maximum over all possible values of W, we see that the maximum value of $e \cdot \beta$ is 134,656, 8,470,528, and 2,150,629,376 for `single`, `double`, and `quad`, respectively. Since the maximum representable unsigned 32-bit integer value is 4,294,967,295, this trick can be performed with either 32- or 64-bit integers. Thus, we count our

Table A.3. Magic Numbers Used in (A.2) Corresponding to Default Values of W

	single	double	quad
W_{default}	13	40	100
E	266	2,060	32,780
α	12	17	22
β	316	3,277	41,944
$E * \beta$	84,056	6,750,620	1,374,924,320
$\max_W E * \beta$	134,656	8,470,528	2,150,629,376

The last row shows the largest value that needs to be representable for the algorithm to work for all W.

two integer divisions as two simple integer operations each. Table A.3 shows some of the relevant values in our division algorithm for the default values of W.

REFERENCES

[1] IEEE. 2008. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug. 2008), 1–70. DOI: https://doi.org/10.1109/IEEESTD.2008.4610935

[2] Intel. 2018. Developer Reference for Intel® Math Kernel Library 2018 - C | Intel® Software. Retrieved from https://software.intel.com/en-us/download/developer-reference-for-intel-math-kernel-library-2018-c.

[3] NVIDIA. 2018. NVIDIA® cuBLAS. Retrieved from http://docs.nvidia.com/cuda/cublas/index.html.

[4] Intel. 2019. bfloat16 - HardwareNumerics Definition. Retrieved from https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf.

[5] IEEE. 2019. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019* (July 2019), 1–84. DOI: https://doi.org/10.1109/IEEESTD.2019.8766229

[6] R. Alverson. 1991. Integer division using reciprocals. In *Proceedings of the Symposium on Computer Arithmetic (ARITH'91)*. 186–190. DOI: https://doi.org/10.1109/ARITH.1991.145558

[7] A. Arteaga, O. Fuhrer, and T. Hoefler. 2014. Designing bit-reproducible portable high-performance applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'14)*. 1235–1244. DOI: https://doi.org/10.1109/IPDPS.2014.127

[8] C. Chohra, P. Langlois, and D. Parello. 2015. Efficiency of reproducible level 1 BLAS. In *Proceedings of the Conference on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN'15)*. Springer, Cham, 99–108. DOI: https://doi.org/10.1007/978-3-319-31769-4_8

[9] C. Chohra, P. Langlois, and D. Parello. 2016. Reproducible, accurately rounded and efficient BLAS. In *Proceedings of the Euro-Par Parallel Processing Workshops*. Springer, Cham, 609–620. DOI: https://doi.org/10.1007/978-3-319-58943-5_49

[10] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. 2015. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Comput.* 49 (Nov. 2015), 83–97. DOI: https://doi.org/10.1016/j.parco.2015.09.001

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

[12] J. Demmel, G. Gopalakrishnan, M. Heroux, W. Keyrouz, and K. Sato. 2015. Reproducibility of high performance codes and simulations: Tools, techniques, debugging. In *Proceedings of the SC 2015 Birds of a Feather Sessions*. Retrieved from https://gcl.cis.udel.edu/sc15bof.php.

[13] J. Demmel and Y. Hida. 2004. Accurate and efficient floating point summation. *SIAM J. Sci. Comput.* 25, 4 (Jan. 2004), 1214–1248. DOI: https://doi.org/10.1137/S1064827502407627

[14] J. Demmel and H. D. Nguyen. 2013. Fast reproducible floating-point summation. In *Proceedings of the Symposium on Computer Arithmetic (ARITH'13)*. 163–172. DOI: https://doi.org/10.1109/ARITH.2013.9

[15] J. Demmel and H. D. Nguyen. 2015. Parallel reproducible summation. *IEEE Trans. Comput.* 64, 7 (July 2015), 2060–2070. DOI: https://doi.org/10.1109/TC.2014.2345391

[16] T. Granlund and P. L. Montgomery. 1994. Division by invariant integers using multiplication. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'94)*. 61–72. DOI: https://doi.org/10.1145/178243.178249

[17] Y. Hida, X. S. Li, and D. H. Bailey. 2001. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the Symposium on Computer Arithmetic (ARITH'01)*. 155–162. DOI: https://doi.org/10.1109/ARITH.2001.930115

[18] N. Higham. 1993. The accuracy of floating point summation. *SIAM J. Sci. Comput.* 14, 4 (July 1993), 783–799. DOI: https://doi.org/10.1137/0914050

[19] N. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9780898718027

[20] D. G. Hough. 2019. The IEEE standard 754: One for the history books. *Computer* 52, 12 (Dec. 2019), 109–112. DOI: https://doi.org/10.1109/MC.2019.2926614

[21] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat. 2015. ExBLAS: Reproducible and accurate BLAS library. In *Proceedings of the SC 2015 Numerical Reproducibility at Exascale Workshops (NRE'15)*. Retrieved from https://hal.archives-ouvertes.fr/hal-01202396.

[22] R. Iakymchuk, D. Defour, S. Collange, and S. Graillat. 2015. Reproducible and accurate matrix multiplication. In *Proceedings of the Conference on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN'15)*. Springer, Cham, 126–137. DOI: https://doi.org/10.1007/978-3-319-31769-4_11

[23] R. Iakymchuk, D. Defour, S. Collange, and S. Graillat. 2015. Reproducible triangular solvers for high-performance computing. In *Proceedings of the International Conference on Information Technology - New Generations (ITNG'15)*. 353–358. DOI: https://doi.org/10.1109/ITNG.2015.63

[24] W. Kahan. 1965. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM* 8, 1 (Jan. 1965). DOI: https://doi.org/10.1145/363707.363723

[25] D. E. Knuth. 1969. *The Art of Computer Programming 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA.

[26] U. Kulisch. 2012. *Computer Arithmetic and Validity: Theory, Implementation, and Applications* (2nd ed.). Walter de Gruyter.

[27] D. R. Lutz and C. N. Hinds. 2017. High-precision anchored accumulators for reproducible floating-point summation. In *Proceedings of the Symposium on Computer Arithmetic (ARITH'17)*. 98–105. DOI: https://doi.org/10.1109/ARITH.2017.20

[28] J.-M. Muller, N. Brunie, F. Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. 2018. *Handbook of Floating-Point Arithmetic* (2nd ed.). Birkhäuser Basel. Retrieved from http://www.springer.com/us/book/9783319765259.

[29] J. Riedy and J. Demmel. 2018. Augmented arithmetic operations proposed for IEEE-754 2018. In *Proceedings of the Symposium on Computer Arithmetic (ARITH'18)*. 45–52. DOI: https://doi.org/10.1109/ARITH.2018.8464813

[30] S. M. Rump. 2009. Ultimately fast accurate summation. *SIAM J. Sci. Comput.* 31, 5 (Jan. 2009), 3466–3502. DOI: https://doi.org/10.1137/080738490

[31] S. M. Rump, T. Ogita, and S. Oishi. 2010. Fast high precision summation. *Nonlin. Theor. Applic. IEICE* 1 (2010), 2–24. DOI: https://doi.org/10.1587/nolta.1.2

Received June 2016; revised February 2020; accepted March 2020