Efficient Reproducible Floating Point Summation and BLAS

James Demmel
Willow Ahrens
Hong Diep Nguyen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-121
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html

June 18, 2016

Acknowledgement

This research is supported in part by
NSF grant NSF ACI-1339676,
DOE grants DOE DE-SC0010200, DOE DE-SC0008699,
DOE DE-SC0008700, and DOE AC02-05CH11231,
and DARPA grant HR0011-12-2-0016,
ASPIRE Lab industrial sponsors and affiliates
Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and Samsung. Other industrial sponsors include Mathworks and Cray.
Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

Efficient Reproducible Floating Point Summation and BLAS

Willow Ahrens, Hong Diep Nguyen, James Demmel

June 18, 2016

Abstract

We define reproducibility to mean getting bitwise identical results from multiple runs of the same program, perhaps with different hardware resources or other changes that should ideally not change the answer. Many users depend on reproducibility for debugging or correctness [1]. However, dynamic scheduling of parallel computing resources, combined with nonassociativity of floating point addition, makes attaining reproducibility a challenge even for simple operations like summing a vector of numbers, or more complicated operations like the Basic Linear Algebra Subprograms (BLAS). We describe an algorithm that computes a reproducible sum of floating point numbers, independent of the order of summation. The algorithm depends only on a subset of the IEEE Floating Point Standard 754-2008. It is communication-optimal, in the sense that it does just one pass over the data in the sequential case, or one reduction operation in the parallel case, requiring an "accumulator" represented by just 6 floating point words (more can be used if higher precision is desired). The arithmetic cost with a 6-word accumulator is $7n$ floating point additions to sum n words, and (in IEEE double precision) the final error bound can be up to 10^{-8} times smaller than the error bound for conventional summation. We describe the basic summation algorithm, the software infrastructure used to build reproducible BLAS (ReproBLAS), and performance results. For example, when computing the dot product of 4096 double precision floating point numbers, we get a 4x slow-down compared to Intel®Math Kernel Library (MKL) running on an Intel® Core i7-2600 CPU operating at 3.4 GHz and 256 KB L2 Cache.

Contents

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 which 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 [1]. 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, a different number of processors, different floating point semantics, and different math libraries (e.g. trigonometric functions), is beyond what we address here [4].

Instead, we limit ourselves to nonreproducibility caused by floating point summation. Since roundoff makes floating point summation non-associative, summing the same summands in different orders frequently gives different answers. And on a parallel machine with a variable (e.g. dynamically allocated) number of processors, 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. So our first goal is to present an algorithm for floating point summation that is reproducible, as well as efficient and accurate. Our second goal is to use this as a building block for a Reproducible BLAS library (ReproBLAS), that permits all the usual performance optimizations (e.g. tiling) used in highly tuned non-reproducible BLAS implementations. We will refine these high level goals into more detailed ones

listed below.

Our work builds on [5], which attained some but not all of our design goals. So in the following summary (for more details see Section 2), we point out which goals were attained in [5], and which ones are attained here for the first time.

1. *Reproducible summation, independent of summation order, assuming only a subset of the IEEE 754 Floating Point Standard.* This is our primary goal. In particular, we only use binary arithmetic (any precision) with the default round-to-nearest-even rounding mode. We need to make some assumptions on the maximum number of summands, which turn out to be 2^{64} and 2^{33} for double and single precision, resp., which is more than enough in practice. Our approach is based on [5], with enhancements described below.

2. *Accuracy at least as good as conventional summation, and tunable.* The internal representation used in [5] lets us choose the number of **bins** (defined in Section 4), which is proportional to the precision. We call the set of bins used to represent a sum an **indexed_type** (defined in Section 5). The indexed type serves as a "reproducible accumulator." Our default in double precision is to carry at least 80 bits of precision. But [5] did not explain how to accurately 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, that introduces an additional error in the computed sum S of at most 7 ulps (units in the last place) of the *exact* sum $\sum_{j=0}^{n-1} x_j$:

$$|S - \sum_{j=0}^{n-1} x_j| \le n2^{-80} \max |x_j| + 7\epsilon |\sum_{j=0}^{n-1} x_j| \tag{1.1}$$

   where $\epsilon = 2^{-53}$. In contrast, the sum $S_{[5]}$ computed by the algorithm in [5] only satisfies the error bound

$$|S_{[5]} - \sum_{j=0}^{n-1} x_j| \le n(2^{-80} + 5\epsilon) \max |x_j| \approx 5n\epsilon \max |x_j| \tag{1.2}$$

   which can be up to 10^8 times larger for very ill-conditioned problems (i.e. when there is a great deal of cancellation); see Section 6.9 for

details. And the error bound for the conventionally computed sum S_{cc} is even larger:

$$|S_{cc} - \sum_{j=0}^{n-1} x_j| \leq n\epsilon \sum_{j=0}^{n-1} |x_j| \leq n^2 \epsilon \max |x_j|$$

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); see Section 2 for details. The algorithm in [5] did not deal with exceptions, or very large intermediate results. In contrast, our algorithm guarantees that no overflows can occur until the final rounding to a single floating point number (assuming the above bounds on the number of summands). Summands that are `+Inf`, `-Inf` or `NaN` propagate in a reproducible way to the final sum. Underflows are handled reproducibly by assuming default IEEE 754 gradual underflow semantics, but a slightly modified version of our algorithm works 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 [5] has this desirable property.

5. *One reduction.* 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 [5] again has this desirable property.

6. *Use as little memory as possible, to enable tiling.* Many existing algorithms for reproducible summation represent intermediate sums by a data structure that we called a "reproducible accumulator" above. BLAS3 operations like matrix-multiplication require many such reproducible accumulators in order to exploit common optimizations like tiling; otherwise they are forced to run at the much slower speeds of

BLAS1 or BLAS2 (see the communication lower bound in Theorem 2.1 in Section 2). In order 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 the algorithm 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.

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 of deterministic order or summation, with non-deterministic paths requiring explicit user opt-in [6]. This approach is limited in the same ways as MKL with CNR. Another approach has been to compute a correctly rounded sum, which is reproducible by definition [7, 8, 9, 10]. But this approach involves either an accumulator long enough to accumulate all possible floating point sums exactly (violating Goal 6), and/or multiple passes over the data rewriting it to eliminate cancellation (violating Goal 4).

We summarize our algorithm informally as follows. The simplest approach would be to

1. compute the maximum absolute value M of all the summands (this is exact and so reproducible),

2. round all the summands to 1 ulp (unit in the last place) of M (this introduces error, but not worse than the usual error bound), and then

3. add the rounded summands (since their nonzero bits are aligned, they behave like fixed point numbers, so summation is exact and so reproducible, assuming we have $\log_2 n$ extra bits for carries).

The trouble with this simple approach is that it requires 2 or 3 passes over the data, or 3 communication steps in parallel. We can in fact do it in one pass over the data, or one communication step, essentially by interleaving the 3 steps above: We break the floating point exponent range into fixed **bins** all of some width W (see Figure 4.1). 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. We can then sum all slices corresponding the same bin exactly, again because we are (implicitly) doing fixed point arithmetic. But we do not need to sum the slices in all bins, only the bins corresponding to the largest few exponent ranges (the number of bins summed can be chosen based on the desired precision). Slices lying in bins with smaller exponents are discarded or not computed in the first place. Independent of the order of summation, we end up with the same sums of slices in the same bins, all computed exactly and so reproducibly, which we then convert to a standard floating point number. As we will see, this idea, while it sounds simple, requires significant effort to implement and prove correct.

The rest of this paper is organized as follows. Section 2 describes our design goals above in more detail. Section 3 begins by explaining some of the notation necessary to express the mathematical and software concepts discussed in this work. Section 4 gives a formal discussion of the binning scheme. As described above, a floating point number x is split along predefined boundaries (bins) into a sum of separate numbers (slices) such that the sum of slices equals (or approximates) x. We sum the slices corresponding to each bin separately and exactly. Section 5 describes the data structure (the **indexed type**) used to store the sums of the slices. Section 6 contains several algorithms for basic manipulations of an indexed type. These algorithms allow the user to, for instance, extract the slices of a floating point number and add them to the indexed type, to add two indexed types together, or to convert from the indexed type to a floating point number. We show that the absolute error of the reproducibly computed sum of double-precision floating point numbers $x_0, ..., x_{n-1}$ in a typical use case is bounded as in Equation (1.1) above. Details regarding the error bound are given in Sections 6.8 and 6.9. As discussed in Section 6.10, the indexed types are

capable of summing approximately 2^{64} `doubles` or 2^{33} `floats`. Section 7 gives several descriptions of algorithms that can be built from the basic operations described in Section 6. In particular, several sequential reproducible algorithms from the BLAS (Basic Linear Algebra Subprograms) are given. Throughout the text, relevant functions from ReproBLAS (the C library accompanying this work from which these algorithms are taken) are mentioned. The interfaces to these functions allow the user to adjust the accuracy of the indexed types used. ReproBLAS uses a custom build system, code generation, and autotuning to manage the software engineering complexities of performance-optimized code. A short description of ReproBLAS, including timing data and testing methods, is given in Section 8. ReproBLAS is available online at `http://bebop.cs.berkeley.edu/reproblas`.

2 Design Goals

We begin by defining reproducibility more carefully in Goal 1 of Section 1. At one extreme, which we call "complete reproducibility", it could mean getting bitwise identical answers on any computer, no matter what hardware resources are available, or how they are scheduled, for any size and ordering of inputs, that would get identical results in exact arithmetic. For example, this would mean that the matrix multiplication $(AP) \cdot (P^T B)$ would get the identical answer for any permutation matrix P, no matter how the matrices are laid out across any number of processors, etc.

A much less demanding definition would say that reproducibility is guaranteed only on one computer, assuming that the number of processors, and data ordering and layout on those processors, is fixed from run-to-run, so that the only potential source of nonreproducibility are the parallel reductions (say MPI reduce). This could potentially allow much faster local summations to be performed locally on each processor, with care taken only to make the parallel reductions reproducible. And if the number of summands is known to be small enough, a different, faster algorithm could be used, even if it gives a different reproducible answer than one that works for a larger number of summands.

The algorithms we present will support "complete reproducibility," assuming only that the number of double precision summands is at most 2^{64}, and that a limited subset of IEEE 754 floating point standard operations are available. We will present them in a modular way, that will also make

it easy to see how to apply them selectively, e.g. just to the parallel reductions in the example in the previous paragraph. We will call this "selective reproducibility". This addresses Goal 7.

Reproducibility also implies reproducible exception handling, which is Goal 3. To illustrate the challenges, let X be any finite floating point number greater than half the overflow threshold, and consider adding the numbers $[X, X, 1, -X, -X]$. Depending on the order of summation one might get $+Inf$, $-Inf$, NaN, 0 or 1 (the exact answer). In comparison, the standard error bound of conventional summation [11] would be about $16 \cdot \epsilon \cdot |X|$, where ϵ is the maximum relative error in a single floating point operation, in the absence of any exceptions. Thus, both 0 and 1 would be satisfactory answers, but the design space we explore in later sections includes other possible error bounds less than $16 \cdot \epsilon \cdot |X|$ in magnitude.

We note that reproducible exceptions in dot products are more challenging; consider the dot product of $[X, X, 1, -X, -X]$ and $[X, X, 1, X, X]$. There are two possible approaches: avoid all possible over/underflows until the final rounding back to a floating number (which would require an intermediate representation with more than double the exponent range), or allow each product to over/underflow, and reproducibly return a $\pm Inf$ or NaN. Since supporting twice the exponent range may noticeably increase the cost and memory requirements of our algorithm, with rare benefit, we choose the following goal: If none of the products $x_i \cdot y_i$ in a dot product (or other BLAS operation) overflows, then no exceptions will be generated unless the final sum overflows, in which case a $\pm Inf$ is reproducibly returned. If one or more $x_i \cdot y_i$ does overflow, then one of $+Inf$, $-Inf$ or NaN is reproducibly returned, depending on whether some $x_i \cdot y_i$ overflow just to $+Inf$, just to $-Inf$, or to both. We note that the BLAS1 routine nrm2, which computes $(\sum_i x_i^2)^{1/2}$, does require dealing with a wider exponent range in many cases (where the answer is otherwise unexceptional), so appropriate scaling is required for nrm2.

In addition to avoiding unnecessary exceptions and getting sufficient accuracy, performance is obviously important. There are a number of usage scenarios to consider, any subset of which might arise in a particular application. First, the data might all be on one processor, resident in any level of the memory hierarchy. When the data is all in L1 cache, say, then accessing the data is cheap, so this makes the bottleneck the cost of the arithmetic operations themselves. When the data is at a more distant level of the memory hierarchy, say DRAM, and too large to fit in L1, then the cost of accessing

the data will likely be dominant. But this scenario gives us the goal that any algorithm we propose should be able to work with one pass over the data, since otherwise communication (data-movement) costs will at least double that of straightforward summation. Another reason why doing one pass over the data is important is that in the dot product (or other BLAS operations), the summands $x_i \cdot y_i$ are created on-the-fly, and we do not want to to store or recompute them for multiple passes. This justifies Goal 4.

Analogously, we may suppose that the data is spread over multiple processors, where the communication bottleneck is the cost of a parallel reduction operation. So our performance goal is to require just one reduction, like straightforward summation, where we cannot make any assumptions about the shape of the reduction tree. This is Goal 5.

Of course, many BLAS need to perform not just one reproducible sum, but many using the same data. The Level 3 BLAS optimize their performance by tiling, e.g. multiplying matrices $A \cdot B = C$ by breaking A, B and C into smaller tiles that fit in higher levels of the memory hierarchy, and multiplying these smaller tiles without any more slow memory accesses. There are analogous optimizations in the parallel case.

We want our underlying reproducible summation algorithm to be compatible with this ubiquitous optimization. The issue is that the size of the "reproducible accumulator" needed to represent a single reproducible sum will limit the size of the C tile, which needs one reproducible accumulator per tile entry.

Here is a simple analysis of the communication cost as a function of the size of a reproducible accumulator. It is an extension of the usual analysis of conventional (non-reproducible) matrix multiplication that shows that 3 square tiles of A, B and C that take up all the fast memory minimizes communication between fast and slow memory. Suppose that fast memory contains W words, one reproducible accumulator takes R words, and the tile sizes of A, B and C are $m \times k$, $k \times n$ and $m \times n$, respectively. The goal is to maximize the number of arithmetic operations we can perform on 3 tiles, assuming that they fit in fast memory. The number of operations is $2mnk$, and the memory constraint is $mk + kn + Rmn \leq W$. The solution of this constrained optimization problem is: $m = n = \sqrt{(W/3)/R)}$ and $k = \sqrt{(W/3) \cdot R}$, so each tile takes up $W/3$ words of the fast memory, and the number of operations is $2mnk = 2(W/3)^{3/2}/\sqrt{R}$. This means that the total number of reads and writes to slow memory for tiled matrix multiply

grows proportionally to $\#\text{flops} \cdot \sqrt{R/W}$. In other words, the communication cost must grow proportionally to the square root of the size R of the reproducible accumulator. So we would like R to be as small as possible; this is Goal 6. This result can also be extended to bound the communication over a network in parallel matrix multiplication. In fact the above tiling attains a lower bound on the amount of communication of any implementation of classical $O(n^3)$ matrix multiply, where each entry of C is computed using a reproducible accumulator of R words. This assumes that a reproducible accumulator cannot be losslessly compressed before it is stored in slow memory. The proof is a simple extension of the communication lower bound proof in [12, 13]:

Theorem 2.1. *Suppose we implement classical $O(n^3)$ matrix multiplication $C = A \cdot B$ on a machine with a 2-level memory hierarchy ("fast" and "slow"), where the fast memory is of size W words. Each entry of A, B and C occupies 1 word, but we accumulate each C_{ij} in an (incompressible) reproducible accumulator of length R words before rounding the final result to one word. Then a lower bound on the number of words moved between fast and slow memory is $\Omega(\#flops \cdot \sqrt{R/W})$.*

Proof. The approach of [12, 13] is as follows: Think of the instructions performed by any implementation of matrix multiply as a sequence of loads, stores and arithmetic operations. Partition this sequence into "segments" consisting of consecutive instructions each containing exactly W load and store instructions (except possibly the last segment). Use the Loomis-Whitney inequality [14] to upper bound the number of arithmetic operations that can be performed during a segment by G. Then the number of (complete) segments must be at least $\lfloor \#flops/G \rfloor$, and since each (complete) segment does W loads and stores, the total number of loads and stores is at least $W\lfloor \#flops/G \rfloor$. The Loomis-Whitney inequality tells us that $G \leq (|A| \cdot |B| \cdot |C|)^{1/2}$, where $|A|$ is the number of entries of A available in fast memory during a segment (and analogously for $|B|$ and $|C|$). When each entry of A occupies one word, then $|A| \leq 2W$, because fast memory can contain at most W words at the start of the segment, and at most W more words can be read during the segment. If each entry of B and C also occupies one word, then $|B| \leq 2W$ and $|C| \leq 2W$, so $G \leq (2W)^{3/2}$, and $W\lfloor \#flops/G \rfloor = \Omega(\#flops/W^{1/2})$, the usual lower bound. But if each "entry" of C occupies R words, before it is finally rounded back to 1 word and (possibly) written to slow memory, then $|C| \leq 2W/R$ by analogous reason-

ing. Thus $G \le (|A| \cdot |B| \cdot |C|)^{1/2} \le (2W \cdot 2W \cdot 2W/R)^{1/2} = (2W)^{3/2}/R^{1/2}$, and the result follows. □

Furthermore, this result extends to all BLAS and direct linear algebra more generally, not just matrix-multiply [13], with the same lower bound. And it also extends to all algorithms that are (intrinsically) nested loops accessing arrays, where the subscripts are just linear combinations of the loop indices [15]. In this latter case, the communication cost will grow by some factor R^e, where the exponent e depends on the subscript expressions, and which array (or arrays) are represented by long accumulators. For example, for the direct N-body algorithm, which computes the forces between all pairs of particles and sums them, the communication cost will grow by the factor $R^1 = R$, which is even worse than linear algebra.

3 Notation and Background

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 multiples of r, $\{rz | z \in \mathbb{Z}\}$.

For all $r \in \mathbb{R}$, let $\lceil r \rceil$ be the minimum element $z \in \mathbb{Z}$ such that $z \ge r$.

For all $r \in \mathbb{R}$, let $\lfloor r \rfloor$ be the maximum element $z \in \mathbb{Z}$ such that $z \le r$.

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

$$\mathcal{R}_\infty(r, e) = \begin{cases} \lfloor r/2^e + 1/2 \rfloor 2^e \text{ if } r \ge 0 \\ \lceil r/2^e - 1/2 \rceil 2^e \text{ otherwise} \end{cases} \tag{3.1}$$

$\mathcal{R}_\infty(r, e)$ rounds r to the nearest multiple of 2^e, breaking ties away from 0. Properties of such rounding are shown in (3.2)

$$\begin{aligned} \left| r - \mathcal{R}_\infty(r, e) \right| &\le 2^{e-1} \\ \mathcal{R}_\infty(r, e) &= 0 \text{ if } |r| < 2^{e-1}. \end{aligned} \tag{3.2}$$

Let $\mathbb{F}_{b,p,e_{min},e_{max}}$ denote the set of floating-point numbers of **base** $b \in \mathbb{Z}$ ($b \ge 2$), **precision** $p \in Z$ ($p \ge 1$) and **exponent range** $[e_{min}, e_{max}]$ where $e_{min}, e_{max} \in \mathbb{Z}$ and $e_{min} \le e_{max}$. Each value $f \in \mathbb{F}_{b,p,e_{min},e_{max}}$ is represented by:

$$f = s \cdot m_0.m_1 \ldots m_{p-1} \cdot b^e,$$

where $s \in \{-1, 1\}$ is the **sign**, $e \in \mathbb{Z}, e_{\min} \leq e \leq e_{\max}$ is the **exponent** (also defined as $\exp(f)$), and $m = m_0.m_1 \ldots m_{p-1}, m_i \in \{0, 1, \ldots, b-1\}$ is the **significand** (also called the **mantissa**) of f. Assume that f is represented using the smallest exponent possible.

Although much of the analysis below can be applied to a general floating-point format, in the context of this paper we assume binary floating-point formats complying with the IEEE 754-2008 standard [16]. For simplicity as well as for readability, throughout this paper $\mathbb{F}_{b,p,e_{min},e_{max}}$ will be written simply as $\mathbb{F}$, referring to some IEEE 754-2008 binary floating-point format, i.e. $b = 2$ and $m_i \in \{0, 1\}$. All the analysis will be based on the corresponding parameters p, e_{min} and e_{max}.

Since a floating point number is always represented using the smallest possible exponent, the first bit m_0 is not explicitly stored in internal representation and is referred to as the "hidden" or "implicit" bit. Therefore only $p-1$ bits are used to represent the mantissa of f.

$f = 0$ if and only if all $m_j = 0$ and $e = e_{\min} - 1$. f is said to be **normalized** if $m_0 = 1$ and $e_{max} \geq e \geq e_{\min}$. f is said to be **unnormalized** if $m_0 = 0$ (unnormalized numbers can exist if the hidden bit convention is not followed), and **denormalized** if $m_0 = 0$ and $e = e_{\min} - 1$.

We assume rounding mode "to nearest" (no specific tie breaking behavior is required) and gradual underflow, although methods to handle abrupt underflow will be considered in Section 5.1.3. (At the end of Section 6.3, we briefly discuss the use of other rounding modes.)

$r \in \mathbb{R}$ 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 $f \in \mathbb{F}$, $\mathrm{ulp}(f)$, is the spacing between two consecutive floating point numbers of the same exponent as f. If f is normalized, $\mathrm{ulp}(f) = 2^{\exp(f)-p+1} = 2\epsilon 2^{\exp(f)}$ and $\mathrm{ulp}(f) \leq 2^{1-p}|f|$.

The unit in the first place of $f \in F$, $\mathrm{ufp}(f)$, is the value of the first significant bit of f. If f is normalized, $\mathrm{ufp}(f) = 2^{\exp(f)}$.

For all $f_0, f_1 \in \mathbb{F}$, $\mathrm{fl}(f_0 \text{ op } f_1)$ denotes the evaluated result of the expression $(f_0 \text{ op } f_1)$ in floating point arithmetic. If $(f_0 \text{ op } f_1)$ is representable, then $\mathrm{fl}(f_0 \text{ op } f_1) = (f_0 \text{ op } f_1)$. If rounding is "to nearest," and there is no overflow, then we have that $|\mathrm{fl}(f_0 \text{ op } f_1) - (f_0 \text{ op } f_1)| \leq 0.5\mathrm{ulp}(\mathrm{fl}(f_0 \text{ op } f_1))$.

This bound accounts for underflow as the magnitude of ulp(f) reflects the appropriate loss of accuracy when f is in the denormal range.

ReproBLAS is the library implementation of the algorithms defined later in this work.

As ReproBLAS is written in C, `float` and `double` refer to the floating point types specified in the 1989 C standard [17] and we assume that they correspond to the `binary-32` and `binary-64` types in the IEEE 754-2008 floating point standard [16].

The functions in ReproBLAS are named after their BLAS counterparts. As such, function names are prefixed by a one or two character code indicating the data type of their inputs and outputs. If a function's input and output data types differ, the function name is prefixed by the output data type code followed by the input data type code. The codes used are enumerated in Table 1. The indexed type is a reproducible floating point data type that will be described later in Section 5. As an example, an absolute sum

Table 1: ReproBLAS naming convention character codes

Data Type	Code
`double`	d
`double complex`	z
`float`	s
`float complex`	c
`double_indexed`	di
`double_complex_indexed`	zi
`float_indexed`	si
`float_complex_indexed`	ci

routine `asum` that returns a `double_indexed` and takes as input an array of `complex double` would be named `dizasum`. To be generic when referring to a routine, we will use the special character `x`, so that all `asum` routines that take floating point inputs and return indexed types can be referred to as `xixasum`.

All indices start at 0 in correspondence with the actual ReproBLAS implementation.

4 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 accumulated separately. We begin in Section 4.1 by explaining the particular set of ranges (referred to as bins, see Figure 4.1) used. Section 4.2 develops mathematical theory to describe the components (referred to as slices) corresponding to each bin. The data format (called Indexed Type) to represent bins will be explained in Section 5. In this section, we develop theory to concisely describe and prove correctness of algorithms throughout the paper (especially Algorithms 6.4 and 6.5).

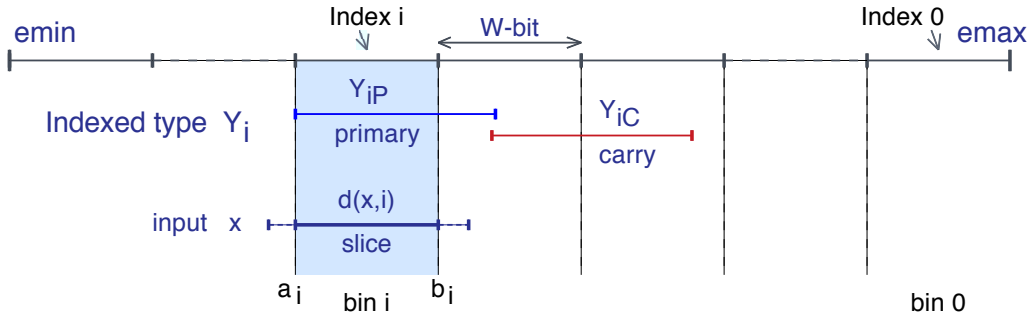

Figure 4.1: Indexed Floating-Point: binning process

4.1 Bins

We start by dividing the exponent range $(e_{\min} - p, ..., e_{\max} + 1]$ into **bins** $(a_i, b_i]$ of **width** W according to (4.1), (4.2), and (4.3). Such a range is used so that the largest and smallest (denormalized) floating point numbers may be approximated.

Definition 4.1.

$$i_{\max} = \lfloor (e_{\max} - e_{\min} + p - 1)/W \rfloor - 1 \tag{4.1}$$

$$a_i = e_{\max} + 1 - (i+1)W \ \textit{for}\ 0 \le i \le i_{\max} \tag{4.2}$$

$$b_i = a_i + W \tag{4.3}$$

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 both $b_{i_0} > b_{i_1}$ and $i_0 < i_1$).

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 both $b_{i_0} < b_{i_1}$ and $i_0 > i_1$).

We use $i \leq i_{\max} = \lfloor (e_{\max} - e_{\min} + p - 1)/W \rfloor - 1$ to ensure that $a_i > e_{\min} - p + 1$ as discussed in Section 5.1.2. This means that the greatest bin, $(a_0, b_0]$, is

$$(e_{\max} + 1 - W, e_{\max} + 1] \tag{4.4}$$

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

$$\Big(e_{\min}-p+2+\big((e_{\max}-e_{\min}+p-1) \mod W\big), e_{\min}-p+2+W+\big((e_{\max}-e_{\min}+p-1) \mod W\big)\Big] \tag{4.5}$$

Section 5.1.2 explains why the bottom of the exponent range

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

is ignored.

As discussed in [5], and explained again in Section 6.4, we must assume

$$W < p - 2 \tag{4.6}$$

As discussed in Section 5.1.1, we must also assume

$$2W > p + 1 \tag{4.7}$$

Notice that in order to satisfy (4.6) and (4.7), we must have

$$p \geq 8 \tag{4.8}$$

(4.8) is satisfied in all binary IEEE floating point formats, but in this paper we focus on single and double precision. We will use (4.8) in Sections 6.8 and 6.9.

ReproBLAS uses both `float` and `double` floating point types. The chosen division of exponent ranges for both types (as well as the `quad` floating point type, which we do not use in ReproBLAS but we include for future implementers) is shown in Table 2. The rationale behind choices for W and K is explained in Section 6.10.

Floating-Point Type	`float`	`double`	`quad`
$e_{\max}$	127	1023	16383
$e_{\min}$	-126	-1022	-16382
p	24	53	113
$e_{\min} - p$	-150	-1075	-16495
W	13	40	100
$i_{\max}$	20	51	327
$(a_0, b_0]$	$(115, 128]$	$(984, 1024]$	$(16284, 16384]$
$(a_{i_{\max}}, b_{i_{\max}}]$	$(-145, -132]$	$(-1056, -1016]$	$(-16416, -16316]$

Table 2: ReproBLAS Binning Scheme

4.2 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 4.1). 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. Specifically, the slice of $x \in \mathbb{R}$ in the bin $(a_i, b_i]$ is defined recursively as $d(x, i)$ in (4.9). 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 paper focuses on using a majority of floating point instructions for portability and for efficiency on architectures with different register sets for fixed and floating point operands. Floating point instructions also allow us to take advantage of the rounding operations built in to floating point arithmetic.

Definition 4.2.

$$\begin{aligned} d(x, 0) &= \mathcal{R}_\infty(x, a_0 + 1) \\ d(x, i) &= \mathcal{R}_\infty\big(x - \sum_{j=0}^{i-1} d(x, j), a_i + 1\big) \ \textit{for } i > 0. \end{aligned} \tag{4.9}$$

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 $d(x, 0) = \mathcal{R}_\infty(x, a_0 + 1)$.

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 largest finite floating point number, then $d(x,0) = \mathcal{R}_\infty(x, a_0+1)$ would be $2^{e_{max}+1}$. Overflow of this type is accounted for in Section 5.1.1. Technical detail of how to handle this special case during the binning process will be explained in Section 6.3.

Lemmas 4.1 and 4.2 follow from the definition of $d(x,i)$.

Lemma 4.1. *For all $i \in \{0, ..., i_{\max}\}$ 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 (3.2) we have $d(x,0) = \mathcal{R}_\infty(x, a_0+1) = 0$.

In the inductive step, we have $|x| < 2^{a_{i+1}} < \ldots < 2^{a_0}$ by (4.2) and by induction $d(x,i) = ... = d(x,0) = 0$. Thus,

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

Again, since $x < 2^{a_{i+1}}$, by (3.2) we have $d(x,i+1) = \mathcal{R}_\infty(x, a_{i+1}+1) = 0$. □

Lemma 4.2. *For all $i \in \{0, ..., i_{\max}\}$ and $x \in \mathbb{R}$ such that $|x| < 2^{b_i}$, $d(x,i) = \mathcal{R}_\infty(x, a_i+1)$.*

Proof. The claim is a simple consequence of Lemma 4.1.

By (4.2) and (4.3), $|x| < 2^{b_i} = 2^{a_{i-1}} < \ldots < 2^{a_0}$. Therefore Lemma 4.1 implies $d(x,0) = ... = d(x,i-1) = 0$ and we have

$$d(x,i) = \mathcal{R}_\infty\big(x - \sum_{j=0}^{i-1} d(x,j), a_i+1\big) = \mathcal{R}_\infty(x, a_i+1)$$

□

Lemma 4.1, Lemma 4.2, and (4.9) can be combined to yield an equivalent definition of $d(x,i)$ for all $i \in \{0, ..., i_{\max}\}$ and $x \in \mathbb{R}$.

$$d(x,i) = \begin{cases} 0 \text{ if } |x| < 2^{a_i} \\ \mathcal{R}_\infty(x, a_i+1) \text{ if } 2^{a_i} \le |x| < 2^{b_i} \\ \mathcal{R}_\infty\big(x - \sum_{j=0}^{i-1} d(x,j), a_i+1\big) \text{ if } 2^{b_i} \le |x| \end{cases} \tag{4.10}$$

Theorem 4.3 shows that sum of the slices of $x \in \mathbb{R}$ provides a good approximation of x.

Theorem 4.3. *For all* $i \in \{0, ..., i_{\max}\}$ *and* $x \in \mathbb{R}$, $|x - \sum_{j=0}^{i} d(x,j)| \leq 2^{a_i}$.

Proof. We apply (3.2) and (4.10)

$$\begin{aligned}\Big|x - \sum_{j=0}^{i} d(x,j)\Big| &= \Big|\big(x - \sum_{j=0}^{i-1} d(x,j)\big) - d(x,i)\Big| \\ &= \Big|\big(x - \sum_{j=0}^{i-1} d(x,j)\big) - \mathcal{R}_\infty\big(x - \sum_{j=0}^{i-1} d(x,j), a_i + 1\big)\Big| \leq 2^{a_i}\end{aligned}$$

□

Although the bins do not extend all the way to $e_{\min} - p$, we now show that the sum of the slices of some $x \in \mathbb{F}$ still offers a good approximation of x.

Using $W < p - 2$ and (4.5),

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

Hence,

$$a_{i_{\max}} \leq e_{\min} - 2$$

As a consequence, we can use Theorem 4.3 to say that for any $x \in \mathbb{R}$,

$$\Big|x - \sum_{i=0}^{i_{\max}} d(x,i)\Big| \leq 2^{a_{i_{\max}}} \leq 2^{e_{\min}-2} \tag{4.11}$$

This means that we can approximate x using the sum of its slices to the nearest multiple of $2^{e_{\min}-1}$.

As the slices of x provide a good approximation of x, the sum of the slices of some $x_0, ..., x_{n-1} \in \mathbb{R}$ 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 $x_0, ..., x_{n-1}$ 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.

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 that 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 4.4 shows a bound on $d(x, i)$.

Theorem 4.4. *For all $i \in \{0, ..., i_{\max}\}$ and $x \in \mathbb{R}$, $|d(x, i)| \leq 2^{b_i}$.*

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

If $i = 0$, then we have

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

Otherwise, we can apply (4.2) and (4.3) to Theorem 4.3 to get

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

As $2^{b_i} \in 2^{a_i+1}\mathbb{Z}$, (4.9) can be used

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

□

Combining Theorem 4.4 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.

5 The Indexed Type

The **indexed type** is used to represent the intermediate result of accumulation using Algorithms 6 and 7 in [5]. An indexed type Y is a data structure composed of several accumulator data structures $Y_0, ..., Y_{K-1}$. An indexed type with K accumulators is referred to as a K**-fold** indexed type. Due to their low accuracy, 1-fold indexed types are not considered.

Let Y be the indexed type corresponding to the reproducibly computed sum of $x_0, ..., x_{n-1} \in \mathbb{F}$. Y is referred to as the **indexed sum** of $x_0, ..., x_{n-1}$, a term which will be defined formally in Section 5.2.

Each accumulator Y_k is a data structure used to accumulate the slices of input in the bin $(a_{I+k}, b_{I+k}]$ where I is the **index** of Y and $k \geq 0$. The **width** of an indexed type is equal to the width of its bins, W. Recall the assumptions (4.6) and (4.7) made on the value of W.

The accumulators in an indexed type correspond to contiguous bins in decreasing order. If Y has index I, then $Y_k, k \in \{0, ..., K-1\}$ accumulates slices of input in the bin $(a_{I+k}, b_{I+k}]$. If I is so large that $I + K > i_{\max}$, then the extra $I + K - i_{\max}$ accumulators are unused.

In ReproBLAS, the data type used to sum `double` is named `double_indexed`, and likewise for `double complex`, `float`, and `float complex`. A K-fold indexed type can be allocated with the `idxd_xialloc` method, and set to zero with the `idxd_xisetzero` method. Both of these methods are defined in `idxd.h` (see Section 8 for details).

Section 5.1 elaborates on the specific fields that make up the indexed type and the values they represent. Sections 5.1.1, 5.1.2, 5.1.3, and 5.1.4 contain extensions of the indexed type to handle overflow, underflow, and exceptional values. Section 5.2 explains how the indexed type can be used to represent the sum of several floating point numbers.

5.1 Primary and Carry

As discussed in [5], indexed types are represented using floating point numbers to minimize traffic between floating point and integer arithmetic units.

In the ReproBLAS library, if an indexed type is used to sum `doubles`, then it is composed entirely of `doubles` and likewise for `floats`. ReproBLAS supports complex types as pairs of real and imaginary components (stored contiguously in memory). If an indexed type is used to sum complex `doubles` or `floats`, then it is composed of pairs (real part, imaginary part) of `doubles` or `floats` respectively. The decision to keep the real and imaginary components together (as opposed to keeping separate indexed types for real and imaginary parts of the sum) was motivated by a desire to process accumulators simultaneously with vectorized (SIMD) instructions.

The accumulators Y_k of an indexed type Y are each implemented using two underlying floating point fields. The **primary** field Y_{kP} is used during accumulation, 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 [5].

The numerical value $\mathcal{Y}_{kP}$ represented by data stored in the primary field Y_{kP} is an offset from $1.5\epsilon^{-1}2^{a_{I+k}}$ (corresponding to "$M_{[i]}$" at the beginning of Section IV.A. in [5]), where I is the index of Y, as shown in (5.1) below. Note that (5.1) only holds when $I + k > 0$. The special case of $I + k = 0$, where $1.5\epsilon^{-1}2^{a_0} > 2^{1+e_{max}}$ is not representable, will be handled in Section 5.1.1 below on Overflow.

$$\mathcal{Y}_{kP} = Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}} \tag{5.1}$$

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 $(a_{I+k}, b_{I+k}]$. It will be shown in Lemma 6.1 in Section 6.3 that if we represent each primary value $\mathcal{Y}_{kP}$ as in (5.1) and keep Y_{kP} within the range $(\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$, then Algorithm 6.4 in Section 6.3 extracts the slices of x in bins $(a_I, b_I], \ldots, (a_{I+K-1}, b_{I+K-1}]$ and adds them to $Y_{0P}, \ldots, Y_{K-1P}$ without error (and hence reproducibly) for all $x \in \mathbb{F}$, where $|x| < 2^{b_I}$.

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

In order to keep the primary fields in the necessary range while the slices

are accumulated and to keep the representation of Y_k unique, Y_{kP} is routinely renormalized to the range $[1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$. As will be shown in Section 6.4, that renormalization is required every 2^{p-W-2} iterations, so 2^{11} in double and 2^9 in single precision. This means the renormalization introduces a very low overhead to the overall running time. To renormalize, Y_{kP} is incremented or decremented by $0.25\epsilon^{-1}2^{a_{I+k}}$ as described in Algorithm 6.6, leaving the carry field Y_{kC} to record the number of such adjustments. Depending on the data format used to store Y_{kC}, the number of updates to one accumulator without overflow is limited, which determines the possible maximum number of inputs that can be reproducibly added to one accumulator. As will be explained in Section 6.4, Equation (6.2), using the same precision p as the primary field to store the carry field, the total number of inputs that can be reproducibly added to one accumulator is $(\epsilon^{-1} - 1)2^{p-W-2}$. This is approximately 2^{64} for `double` and 2^{33} for `float`. See Section 6.10 for a summary of restrictions. The numerical value $\mathcal{Y}_{kC}$ represented by data stored in the carry field Y_{kC} of an indexed type Y of index I is expressed in (5.2)

$$\mathcal{Y}_{kC} = (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} \tag{5.2}$$

Combining (5.1) and (5.2), we get that the value $\mathcal{Y}_k$ of the accumulator Y_k of an indexed type Y of index I is

$$\mathcal{Y}_k = \mathcal{Y}_{kP} + \mathcal{Y}_{kC} = (Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} \tag{5.3}$$

Therefore, using (5.3), the numerical value $\mathcal{Y}$ represented by data stored in a K-fold indexed type Y of index I (the sum of Y's accumulators) is

$$\mathcal{Y} = \sum_{k=0}^{K-1} \mathcal{Y}_k = \sum_{k=0}^{K-1} \big((Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC}\big) \tag{5.4}$$

It is worth noting that by keeping Y_{kP} within the range $(\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ for $I+k > 0$, the exponent of Y_{0P} is a_I+p when $I > 0$. The case of $I = 0$ will be explained in the below section of Overflow. Therefore it is unnecessary to store the index of an indexed type explicitly. As will be explained in Section 6.1, the index can be determined by simply examining the exponent of Y_{0P}, as all a_I are distinct and the mapping between the exponent of Y_{0P} and the index of Y is bijective.

5.1.1 Overflow

If an indexed type Y has index 0 and the width is W, then the value in the primary field Y_{0P} would be stored as an offset from $1.5\epsilon^{-1}2^{e_{\max}+1-W}$. However, $1.5\epsilon^{-1}2^{e_{\max}+1-W} > 2^{e_{\max}+1+(p-W)} > 2 \cdot 2^{e_{\max}}$ since $W < p-2$, so it would be out of the range of the floating-point system and not representable. Before discussing the solution to this overflow problem, take note of Theorem 5.1.

Theorem 5.1. *If $2W > p+1$, then for any indexed type Y of index I and any Y_{kP} such that $I+k \geq 1$, $|Y_{kP}| < 2^{e_{\max}}$.*

Proof. $a_1 = e_{\max}+1-2W$ by (4.2), therefore $a_1 < e_{\max}-p$ using $2W > p+1$ and since all quantities are integers, $a_1 \leq e_{\max}-p-1$. If $I+k \geq 1$, $a_{I+k} \leq a_1 \leq e_{\max}-p-1$ by (4.2).

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

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

□

By Theorem 5.1, if $2W > p+1$ then the only primary field that could possibly overflow is a primary field corresponding to bin 0, and all other primary fields have exponent less than $e_{\max}$. Therefore, we require $2W > p+1$ and express the value of the primary field corresponding to bin 0 as a scaled offset from $1.5 \cdot 2^{e_{\max}}$. Note that this preserves uniqueness of the exponent of the primary field corresponding to bin 0 because no other primary field has an exponent of $e_{\max}$. The value $\mathcal{Y}_{0P}$ stored in the primary field Y_{0P} of an indexed type Y of index 0 is expressed in (5.5).

$$\mathcal{Y}_{0P} = 2^{p-W+1}(Y_{0P} - 1.5 \cdot 2^{e_{\max}}) \tag{5.5}$$

Although the primary field corresponding to bin 0 is scaled, the same restriction (6.2) on n applies here as it does in the normal case. Therefore, the partial sums in reproducible summation can grow much larger ($(\epsilon^{-1}-1)2^{p-W-1}$ times 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 $n \leq (\epsilon^{-1}-1)2^{p-W-2}$, the reproducible summation won't overflow. However, if the inputs to summation are already infinite, the summation will overflow. In the case of a dot product, overflow will occur if the pairwise products themselves overflow (overflow only occurs during multiplication).

5.1.2 Gradual Underflow

Here we consider the effects of gradual underflow on algorithms described in [5] and how the indexed type allows these algorithms to work correctly. Although we will discuss abrupt underflow briefly in the next section, we will consider only gradual underflow in the remainder of the work.

Algorithms 6.5 for adding a floating point input to an indexed type in Section 6.3 and Algorithm 6.6 for renormalizing an indexed type in Section 6.4 require that the primary fields Y_{kP} are normalized to work correctly. Theorem 5.2 shows that the primary fields should always be normalized.

Theorem 5.2. *For any primary field Y_{kP} of an indexed type Y of index I where $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ ($Y_{0P} \in (2^{e_{\max}}, 2 \cdot 2^{e_{\max}})$ if Y has index 0), Y_{kP} is normalized.*

Proof. By (4.5),

$$a_{I+k} \geq a_{i_{\max}} = e_{\min} - p + 2 + \big((e_{\max} - e_{\min} + p - 1) \mod W\big) \geq e_{\min} - p + 2$$

Because $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ we have $\exp(Y_{kP}) = a_{I+k} + p > e_{\min} + 1$ so Y_{kP} is normalized. □

Algorithms 6.4 and 6.5 in Section 6.3 rely on setting the last bit of intermediate results before adding them to Y_{kP} in order to fix the direction of the rounding mode. However, if r is the quantity to be added to Y_{kP}, $\text{ulp}(r)$ must be less than rounding error in Y_{kP} when added to Y_{kP}. Mathematically, we will require $\text{ulp}(r) < 0.5\text{ulp}(Y_{kP})$ in order to prove Theorem 6.2 about the correctness of Algorithm 6.5. This is why we must enforce $a_{i_{\max}} \geq e_{\min} - p + 2$ so that the least significant bit of the least bin is larger than twice the smallest denormalized number.

This does not mean it is impossible to sum the input in the denormalized range. One simple way the algorithm could be extended to denormalized inputs would be to scale the least bins up, analogously to the way we handled overflow. Due to the relatively low priority for accumulating denormalized values, this method was not implemented in ReproBLAS. With respect to computing a reproducible dot product, we do not extend the underflow threshold to ensure the products of tiny values do not underflow. The underflow threshold is the same as in normal reproducible summation. Others may implement these features if they think it is important.

5.1.3 Abrupt Underflow

If underflow is abrupt, several approaches may be taken to modify the given algorithms to ensure reproducibility. We discuss these approaches in this section, but the rest of the work will consider only gradual underflow.

The most straightforward approach would be to accumulate input in the denormalized range by scaling the smaller inputs up. This has the added advantage of increasing the accuracy of the algorithm. A major disadvantage to this approach is the additional branching cost incurred due to the conditional scaling.

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 accumulated. 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 mantissa 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 indexed 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.

5.1.4 Exceptions

Indexed types are capable of representing exceptional cases such as `NaN` (Not a Number) and `Inf` (Infinity). An indexed type 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_{0P} yet (Y_{0P} is initialized to 0).

Since the Y_{kP} are kept within the range $(\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ (where I is the index) and are normalized (by Theorem 5.2), we have

$$Y_{kP} > 2^{e_{\min}}$$

Therefore the value of 0 in a primary field is unused in any previously specified context and may be used as a sentinel value. (As the exponent of 0 is distinct from the exponent of normalized values, the bijection between the index of an indexed type Y and the exponent of Y_{0P} is preserved)

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 or one or more of both `Inf` and `-Inf` have been added to Y. Note that there are several types of `NaN`. We do not differentiate among them. We consider all types `NaN` as identically `NaN`.

As the Y_{kP} are kept finite to store finite values, `Inf`, `-Inf`, and `NaN` are unused in any previously specified context and are valid sentinel values. (As the exponent of `Inf`, `-Inf`, and `NaN` is distinct from the exponent of finite values, the bijection between the index of an indexed type Y and the exponent of Y_{0P} is preserved)

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 indexed summation. As `Inf`, `-Inf`, and `NaN` add associatively, this behavior is reproducible.

Note that, as will be explained in Section 6, as long as the number of inputs is limited by $n \leq (\epsilon^{-1} - 1)2^{p-W-2}$ (approximately 2^{64} for `double` and 2^{33} for `float`) and all the inputs are finite, there will be no intermediate `NaN` or $\pm$`Inf` during the computation. For example, given $x \in \mathbb{F}$ the biggest representable value below the overflow threshold, our algorithm will compute a result of exactly 0 for the sum of input vector $[x, x, -x, -x]$ regardless of the order of evaluation. On the other hand, a standard recursive summation algorithm returns either $((x+x)-x)-x =$ `Inf`, $((-x-x)+x)+x = -$`Inf`, $((x-x)+x)-x = 0$, or $((x+x)+(-x-x)) =$ `NaN` depending on the order of evaluation.

It should also be noted here that it is possible to achieve a final result of $\pm$`Inf` when Y_{0P} is finite. This is due to the fact that the indexed representation can express values outside of the range of the floating point numbers

that it is composed with. More specifically, it is possible for the value $\mathcal{Y}$ represented by the indexed type Y to satisfy $|\mathcal{Y}| \geq 2 \cdot 2^{e_{\max}}$. The condition that $\mathcal{Y}$ is not representable is discovered when calculating $\mathcal{Y}$ (converting Y to a floating point number). The methods used to avoid overflow and correctly return the `Inf` or `-Inf` are discussed in Section 6.8.

There are several ways that `Inf`, `-Inf`, and `NaN` could be handled reproducibly. We have chosen to handle these values analogously to how they are handled in standard recursive summation. This results in a small additional performance cost due to special branches for exceptional values. Another way to handle these values would be to always return `NaN` when any summand is `Inf`, `-Inf`, or `NaN`. This would result in less branching because no explicit checks for exceptional values would be necessary (this is a side effect of the algorithm due to `NaN` propagation). We did not choose this option because there are ways to reduce the number of branches such that the additional cost of the branches is negligible when compared to the computational cost of the reproducible algorithm. These two approaches to handling exceptional values are discussed in more detail at the end of Section 6.3.

5.2 Indexed Sum

We have previously explained the indexed type, a data structure we will use for reproducible summation. We now define a quantity that can be expressed using the indexed type, called the **indexed sum**. The goal for Section 6 will be to show that we can indeed compute this quantity. Ultimately, Theorems 6.4, 6.5, and 6.6 will prove that Algorithms 6.7, 6.9, and 6.10 (respectively) can indeed compute the indexed sum. Here we focus on the definition of an indexed sum. As further motivation for computing the indexed sum, we show that if an algorithm returns the indexed sum of its inputs, it is reproducible.

Definition 5.1. *Assume* $n \leq (\epsilon^{-1} - 1)2^{p-W-2}$. *The K-fold* ***indexed sum*** *of finite* $x_0, ..., x_{n-1} \in \mathbb{F}$ *is defined to be a K-fold indexed type Y such that:*

$$\begin{aligned} Y_{kP} &\in \begin{cases} [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}}) \text{ if } I + k > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}}) \text{ if } I + k = 0 \end{cases} \\ \mathcal{Y}_k &= (Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC} \\ \mathcal{Y}_k &= \sum_{j=0}^{n-1} d(x_j, I + k) \end{aligned} \tag{5.6}$$

I *is the greatest integer such that* $\max(|x_j|) < 2^{b_I}$ *and* $I \leq i_{\max}$

We have repeated (5.3) *as a requirement above for clarity.*
The K-fold indexed sum of $x_0, ..., x_{n-1} \in \mathbb{F}$ *(with at least one exceptional value* `Inf`, `-Inf`, *or* `NaN`*) is defined to be a K-fold indexed type such that*

$$Y_{0P} = \begin{cases} \texttt{Inf} \text{ if there is at least one } \texttt{Inf} \text{ and no other exceptional values} \\ \texttt{-Inf} \text{ if there is at least one } \texttt{-Inf} \text{ and no other exceptional values} \\ \texttt{NaN} \text{ otherwise} \end{cases} \tag{5.7}$$

And the K-fold indexed sum of no numbers (the empty sum) is defined to be the K-fold indexed type such that

$$\begin{aligned} Y_{kP} &= 0 \\ Y_{kC} &= 0 \end{aligned} \tag{5.8}$$

We now show that the indexed sum is well-defined for finite summands and that each field in the indexed type corresponding to the summands $x_0, ..., x_{n-1}$ (in any order) is unique. We show this in Lemma 5.3.

Lemma 5.3. *Let Y be the indexed sum of some* $x_0, ..., x_{n-1} \in \mathbb{F}$*, where each* x_i *is a finite value and* $n \geq 1$. *Let* $\sigma_0, ..., \sigma_{n-1}$ *be some permutation of the first n nonnegative integers such that* $\{\sigma_0, ..., \sigma_{n-1}\} = \{0, ..., n-1\}$ *as sets. Let Z be the indexed sum of* $x_{\sigma_0}, ..., x_{\sigma_{n-1}}$.
For all k, $0 \leq k < K$, *we have that* $Y_{kP} = Z_{kP}$ *and* $Y_{kC} = Z_{kC}$.

Proof. 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 $I \leq i_{\max}$.

Using the associativity of addition,

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

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 $\mathcal{Y}_k = \mathcal{Z}_k$, (5.3) 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 (5.3) yields that $Y_{kP} = Z_{kP}$.

If $I = 0$, assume for contradiction that $Y_{0C} \neq Z_{0C}$. Since $\mathcal{Y}_0 = \mathcal{Z}_0$, (5.3) and (5.5) yield

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

and by (4.2) and (4.3), we have

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

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

Therefore, we have that $Y_{0C} = Z_{0C}$. Along with the fact that $\mathcal{Y}_0 = \mathcal{Z}_0$, application of (5.3) together with (5.5) yields that $Y_{0P} = Z_{0P}$. □

With Lemma 5.3, it is not hard to see that the more inclusive Theorem 5.4 applies to the indexed sum.

Theorem 5.4. *Let Y be the indexed sum of some $x_0, ..., x_{n-1} \in \mathbb{F}$. Let $\sigma_0, ..., \sigma_{n-1}$ be some permutation of the first n nonnegative integers such that $\{\sigma_0, ..., \sigma_{n-1}\} = \{0, ..., n-1\}$ as sets. Let Z be the indexed sum of $x_{\sigma_0}, ..., x_{\sigma_{n-1}}$.*
If all x_0 are finite or $n = 0$, we have that for all k, $0 \leq k < K$, $Y_{kP} = Z_{kP}$ and $Y_{kC} = Z_{kC}$. Otherwise, $Y_{0P} = Z_{0P}$ and Y_{0P} is exceptional.

Proof. If all x_i are finite and $n \geq 1$, then the claim holds by Lemma 5.3. If $n = 0$, then by (5.8) we have that for all k, $0 \leq k < K$, $Y_{kP} = Z_{kP} = 0$ and $Y_{kC} = Z_{kC} = 0$.

If at least one x_i is exceptional, then since the conditions in (5.7) depend only on the number of each type of exceptional value and not on their order, we have that $Y_{0P} = Z_{0P}$. Since all of the possible cases are exceptional, Y_{0P} is exceptional. □

Theorem 5.4 implies that any algorithm that can compute the indexed sum of a list of floating point numbers is a reproducible summation algorithm, as the indexed sum is well-defined, unique, and independent of the ordering of the summands.

6 Primitive Operations

Here we reorganize algorithms in [5] into a set of primitive operations on an indexed type. This set of operations is intended to make it easy to build higher level operations on different platforms, accommodating different sources of nonreproducibility. Two simple original algorithms relating to the index of an indexed type are given in Section 6.1. Theoretical summaries of algorithms (with some improvements) from [5] are provided in Sections 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7. To obtain a general completely reproducible algorithm for summation, one must design for reproducibility under both data ordering and reduction tree shape. Section 6.5 provides methods to sum numbers regardless of ordering (a more efficient algorithm is presented in Section 6.6), while Section 6.7 provides methods to sum numbers regardless of reduction tree shape.

Section 6.8 provides an original algorithm (with a greatly improved error bound) to obtain the value represented by an indexed type. Section 6.9 presents an original theorem regarding the sum of a decreasing sequence of floating point numbers and utilizes this theorem to obtain error bounds for reproducible summation algorithms. We summarize the restrictions governing the indexed type and these algorithms in Section 6.10.

6.1 Index

When operating on indexed types it is sometimes necessary to compute their index. Algorithm 6.1 yields the index of an indexed type in constant time.

Algorithm 6.1 is available in ReproBLAS as `idxd_xmindex` in `idxd.h` (see Section 8 for details).

Algorithm 6.1. Given an indexed type Y, calculate its index I

Require:

$Y_{0P} \in (\epsilon^{-1}2^{a_I}, 2\epsilon^{-1}2^{a_I})$ where a_I is defined in (4.2)

1: **function** IINDEX(Y)
2: **return** $\lfloor (e_{\max} + p - \exp(Y_{0P}) - W + 1)/W \rfloor$ ▷ Index I of Y
3: **end function**

Ensure:

Returned result I is the index of Y.

Note that the floor function is necessary in Algorithm 6.1 to account for the case where Y has index 0, which has $\exp(Y_{0P}) = 2^{e_{\max}}$ as discussed in Section 5.1.1. This uses the assumption that $\frac{p+1}{2} < W < p - 2$, so $3 < p - W + 1 < W$. Note also that the function exp() used above is assumed to return the biased exponent of Y_{0P}, so that IINDEX returns 0 when Y_{0P} is `Inf`, `-Inf`, or `NaN` and $i_{\max}$ when Y_{0P} is 0 or denormalized.

Another useful operation is, given some $x \in \mathbb{F}$, to find the unique bin $(a_J, b_J]$ where J is the greatest integer such that $|x| < 2^{b_J}$ and $J \leq i_{\max}$. Algorithm 6.2 yields such a J in constant time. Algorithm 6.2 is available in ReproBLAS as `idxd_xindex` in `idxd.h` (see Section 8 for details).

Algorithm 6.2. Given $x \in \mathbb{F}$, calculate the largest integer J such that $2^{b_J} > |x|$ and $J \leq i_{\max}$

Require: $i_{\max}$ is defined in (4.1), a_J is defined in (4.2).

1: **function** INDEX(x)
2: **if** $x = 0$ **then**
3: **return** $i_{\max}$
4: **end if**
5: **return** $\min(i_{\max}, \lfloor (e_{\max} - \exp(x))/W \rfloor)$ ▷ Index J of x
6: **end function**

Ensure:

J is the greatest integer such that $|x| < 2^{b_J}$ and $J \leq i_{\max}$.

Note again that the function exp() used above is assumed to return the biased exponent of x, so that INDEX returns 0 when x is `Inf`, `-Inf`, or `NaN` and $i_{\max}$ when $x < 2^{a_{i_{\max}}}$. This behavior is consistent with the following algorithms since values smaller than the least bin will not be accumulated.

Algorithms 6.1 and 6.2 are used infrequently, usually being called once at the beginning of a routine.

6.2 Update

Sometimes it is necessary to adjust the index of Y. For example, in Algorithm 6.5, when adding $x \in \mathbb{F}$ to a K-fold indexed type Y of index I, we will make the assumption that $|x| < 2^{b_I}$, which might require decreasing I to increase b_I. As another example, a new indexed type Y is always initialized to have all primary and carry fields set to 0, therefore before adding any value to Y it is required to adjust the primary and carry fields of Y first.

This adjustment is called an **update**. The process of updating Y to the necessary index is summarized succinctly in Algorithm 6.3. Algorithm 6.3 is available in ReproBLAS as `idxd_xixupdate` in `idxd.h` (see Section 8 for details).

Algorithm 6.3. Update K-fold indexed type Y of index I to have an index J such that $|x| < 2^{b_J}$.

Require:

Y is the indexed sum of some $x_0, ..., x_{n-1} \in \mathbb{F}$

1: **function** UPDATE(K, x, Y)
2: $I = \text{IINDEX}(\text{Y})$
3: $J = \text{INDEX}(\text{x})$
4: **if** $J < I$ **then**
5: $[Y_{\min(I-J,K)_P}, ..., Y_{K-1_P}] = [Y_{0P}, ..., Y_{K-1-\min(I-J,K)_P}]$
6: $[Y_{0_P}, ..., Y_{\min(I-J,K)-1_P}] = [1.5\epsilon^{-1}a_J, ..., 1.5\epsilon^{-1}a_{\min(I,K+J)-1}]$
7: $[Y_{\min(I-J,K)_C}, ..., Y_{K-1_C}] = [Y_{0C}, ..., Y_{K-1-\min(I-J,K)_C}]$
8: $[Y_{0_C}, ..., Y_{\min(I-J,K)-1_C}] = [0, ..., 0]$
9: **end if**
10: **end function**

Ensure:

Y has index J where J is the greatest integer such that $|x| < 2^{b_J}$, $\max(|x_j|) < 2^{b_J}$, and $J < i_{\max}$.

$$\mathcal{Y}_k = \sum_{j=0}^{n-1} d(x_j, J+k)$$

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

The update operation is described in the "Update" Section (lines 7-17) of Algorithm 6 in [5].

It should be noted that if Y_{0P} is 0, then the update is performed as if $I+K<J$. If Y_{0P} is `Inf`, `-Inf`, or `NaN`, then Y is not modified by an update. If J is such that $J+K>i_{\max}$, then $Y_{i_{\max}-JP},...,Y_{K-1P}$ are set to $1.5\epsilon^{-1}2^{a_{i_{\max}}}$ and the values in these accumulators are ignored.

If Y represents the indexed sum of finite values, then existing accumulators of Y may be shifted towards index 0, losing the lesser bins. New accumulators are shifted into Y with value 0 in Y_{kC} and $1.5\epsilon^{-1}2^{a_{\min(I,J)+k}}$ in Y_{kP}.

The new accumulators Y_k with $0\leq k<I-J$ must represent 0 because $|x_j|<2^{b_I}\leq a_{I-1}\leq a_{J+k}$ so $\sum_{j=0}^{n-1} d(x_j,J+k)=0$ by Lemma 4.1.

To speed up this operation, the factors $1.5\epsilon^{-1}a_j$ for all valid $j\in Z$ are stored in a precomputed array.

6.3 Deposit

The **deposit** operation (here referred to as Algorithm 6.5, which deals with overflow, unlike a simpler version described in the "Extract K first bins" Section (lines 18-20) of Algorithm 6 in [5]) is used to extract the slices of a floating point number and add them to the appropriate accumulators of an indexed type.

Algorithm 6.4 deposits floating point numbers in the case that there is no overflow (the indexed type has an index greater than 0).

Algorithm 6.4. Extract slices of $x \in \mathbb{F}$, where $|x| < 2^{b_I}$, in bins $(a_I, b_I], ..., (a_{I+K-1}, b_{I+K-1}]$ and add to indexed type Y. Here, $(r|1)$ represents the result of setting the last bit of the significand (m_{p-1}) of floating point number r to 1. This is a restatement of lines 18-20 of Algorithm 6 in [5].

Require:

No overflow occurs.

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

$|x| < 2^{b_I}$.

$Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ at all times. (The carry fields Y_{kC} will be used to ensure this in Algorithm 6.6)

1: **function** DepositRestricted(K, x, Y)
2: $r = x$
3: **for** $k = 0$ to $(K - 2)$ **do**
4: $S = Y_{kP} + (r|1)$
5: $q = S - Y_{kP}$
6: $Y_{kP} = S$
7: $r = r - q$
8: **end for**
9: $Y_{K-1P} = Y_{K-1P} + (r|1)$
10: **end function**

Ensure:

The amount added to Y_{kP} by this algorithm is exactly $d(x, I + k)$.

The last bit of r is set to break ties when rounding "to nearest" so that the amount added to Y_{kP} does not depend on the size of Y_{kP} so far. Algorithm 6.4 costs $3(K - 1) + 1 = 3K - 2$ FLOPs, not counting the or-bit operation. The following theorem proves the "Ensure" claim at the end of Algorithm 6.4.

Lemma 6.1. *Let Y be an K-fold indexed type of index I. Assume that we run Algorithm 6.4 on Y and some $x \in \mathbb{F}$, $|x| < 2^{b_I}$. If all requirements of the algorithm are satisfied, then the amount added to Y_{kP} is exactly $d(x, I + k)$.*

Proof. Throughout the proof, assume that the phrase "for all k" means "for all $k \in \{0, ..., K - 1\}$." Assume also that r_k and S_k refer to the value of r and S after executing line 4 in the k^{th} iteration of the loop. Finally, assume Y_{kP} refers to the initial value of Y_{kP} and S_k refers to the final value of Y_{kP}. Therefore, $S_k - Y_{kP}$ is the amount added to Y_{kP}.

Note that lines 4-7 correspond to Algorithm 4 of [5]. Therefore, if $\text{ulp}(Y_{kP}) = \text{ulp}(S_k)$ and $\text{ulp}(r_k) < 0.5\text{ulp}(Y_{kP})$, Corollary 3 of [5] applies and we have that $S_k - Y_{kP} \in \text{ulp}(Y_{kP})\mathbb{Z} \in 2^{a_{I+k}+1}\mathbb{Z}$ and that $|r_{k+1}| \leq 0.5\text{ulp}(Y_{kP}) = 2^{a_{I+k}}$.

As it is assumed $Y_{kP}, S_k \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$, we have $\text{ulp}(S_k) = \text{ulp}(Y_{kP})$ for all k.

We show $|r_k| \leq 2^{b_{I+k}} = 2^{a_{I+k-1}}$ for all k inductively. As a base case, $r_0 = x$ (from line 2) so $|r_0| = |x| < 2^{b_I}$. As an inductive step, assume $|r_k| \leq 2^{b_{I+k}}$. We must show $\text{ulp}(r_k) < 0.5\text{ulp}(Y_{kP})$.

By Theorem 5.2 we have that Y_{kP} is normalized and therefore $\text{ulp}(Y_{kP}) = 2^{a_{I+k}+1}$. If r_k is normalized, then because $\text{ulp}(r_k) \leq 2^{1-p}|r_k| \leq 2^{b_{I+k}-(p-1)} = 2^{a_{I+k}+W-(p-1)}$, and $W < p-2$, we have $\text{ulp}(r_k) \leq 2^{a_{I+k}-1} < 0.5\text{ulp}(Y_{kP})$. (This case is considered in [5]). If r_k is denormalized, $\text{ulp}(r_k) = 2^{e_{\min}-p+1}$ since the unit in the last place of a denormalized number is always equal to $2^{e_{\min}-p+1}$. Using (4.5), $\text{ulp}(r_k) = 2^{e_{\min}-p+1} \leq 2^{e_{\min}-p+1+((e_{\max}-e_{\min}+p-1) \mod W)} = 2^{a_{i_{\max}}-1} \leq 2^{a_{I+k}-1} < 0.5\text{ulp}(Y_{kP})$.

Therefore we have $\text{ulp}(r_k) < 0.5\text{ulp}(Y_{kP})$. Thus, Corollary 3 of [5] applies and we have that $|r_{k+1}| \leq 0.5\text{ulp}(Y_{kP}) = 2^{a_{I+k}}$. This completes the induction.

Next, we show $S_k - Y_{kP} = \mathcal{R}_\infty(r_k, a_{I+k}+1)$. As Corollary 3 of [5] applies for all k, then $S_k - Y_{kP} \in 2^{a_{I+k}+1}\mathbb{Z}$. By Theorem 3 of [5], $r_{k+1} = r_k - (S_k - Y_{kP})$. Since $|r_{k+1}| \leq 2^{a_{I+k}}$, we consider two cases.

If $|r_k - (S_k - Y_{kP})| < 2^{a_{I+k}}$, then $S_k - Y_{kP} = \mathcal{R}_\infty(r_k, a_{I+k}+1)$.

If $|r_k - (S_k - Y_{kP})| = 2^{a_{I+k}}$, then $S_k - Y_{kP} \in \{r_k + 2^{a_{I+k}}, r_k - 2^{a_{I+k}}\}$. As $S_k = \text{fl}(Y_{kP} + (r_k|1))$, we have $|S_k - Y_{kP} - (r_k|1)| \leq 0.5\text{ulp}(S_k) = 2^{a_{I+k}}$. As $\text{ulp}(S_k) = \text{ulp}(Y_{kP}) = 2^{a_{I+k}+1}$, we also have that $r_k \in 2^{a_{I+k}}\mathbb{Z}$ and because $\text{ulp}(r_k) < 2^{a_{I+k}}$, $|(r_k|1) - r_k| > 0$ (with $(r_k|1) - r_k$ taking the same sign as r_k). If $r_k > 0$, then $(S_k - Y_{kP}) = r_k + 2^{a_{I+k}}$ (otherwise we will have $|S_k - Y_{kP} - (r_k|1)| = |r_k - 2^{a_{I+k}} - (r_k|1)| > 2^{a_{I+k}}$). If $r_k < 0$, then $(S_k - Y_{kP}) = r_k - 2^{a_{I+k}}$ (otherwise we will have $|S_k - Y_{kP} - (r_k|1)| = |r_k + 2^{a_{I+k}} - (r_k|1)| > 2^{a_{I+k}}$). Therefore, $S_k - Y_{kP} = \mathcal{R}_\infty(r_k, a_{I+k}+1)$.

We can now show $r_{k+1} = x - \sum_{i=0}^{I+k} d(x,i)$ and $S_k - Y_{kP} = d(x, I+k)$ for all k by induction on k.

In the base case, $S_0 - Y_{0P} = \mathcal{R}_\infty(r_0, a_I+1) = \mathcal{R}_\infty(x, a_I+1)$. As $|x| < 2^{b_I}$, Lemma 4.2 implies $S_0 - Y_{0P} = d(x, I)$. By Theorem 3 of [5], $r_1 = r_0 - (S_0 - Y_{0P}) = x - d(x, I)$. By assumption and (4.2) and (4.3), $|x| < 2^{b_I} \leq 2^{a_i}$ for all $i \in \{0, ..., I-1\}$, and therefore by Lemma 4.1, $r_1 = x - \sum_{i=0}^{I} d(x,i)$.

In the inductive step, assume $r_{k+1} = x - \sum_{i=0}^{I+k} d(x,i)$. Then by definition,

$$S_{k+1}{-}Y_{k+1P} = \mathcal{R}_\infty(r_{k+1}, a_{I+k+1}{+}1) = \mathcal{R}_\infty\big(x{-}\sum_{i=0}^{I+k} d(x,i), a_{I+k+1}{+}1\big) = d(x, I{+}k{+}1)$$

And by Theorem 3 of [5],

$$r_{k+2} = r_{k+1}{-}(S_{k+1}{-}Y_{k+1P}) = \big(x{-}\sum_{i=0}^{I+k} d(x,i)\big){-}d(x, I{+}k{+}1) = x{-}\sum_{i=0}^{I+k+1} d(x,i)$$

□

Of course, what remains to be seen is how we can extract and add the components of a floating point number to an indexed type Y of index 0, i.e. when overflow is an issue. Algorithm 6.5 shows the adaptation of Algorithm 6.4 for indexed types of index 0. Algorithm 6.5 is available in ReproBLAS as `idxd_xixdeposit` in `idxd.h` (see Section 8 for details).

Algorithm 6.5. Extract components of $x \in \mathbb{F}$, where $|x| < 2^{b_I}$, in bins $(a_I, b_I], \ldots, (a_{I+K-1}, b_{I+K-1}]$ and add to indexed type Y of index I. Here, $(r|1)$ represents the result of setting the last bit of the significand (m_{p-1}) of floating-point r to 1.

Require:

All requirements (except for the absence of overflow, which we will ensure) from Algorithm 6.4 except that Y_{0P} must now be kept within the range $(2^{e_{\max}}, 2 \cdot 2^{e_{\max}})$ if Y has index 0.

1: **function** DEPOSIT(K, x, Y)
2: $I =$ IINDEX(Y)
3: **if** I $= 0$ **then**
4: $r = x/2^{p-W+1}$
5: $S = Y_{0P} + (r|1)$
6: $q = S - Y_{0P}$
7: $Y_{0P} = S$
8: $q = q \cdot 2^{p-W}$
9: $r = x - q$
10: $r = r - q$
11: **for** $k = 1$ to $(K-2)$ **do**
12: $S = Y_{kP} + (r|1)$
13: $q = S - Y_{kP}$
14: $Y_{kP} = S$
15: $r = r - q$
16: **end for**
17: $Y_{K-1P} = Y_{K-1P} + (r|1)$
18: **else**
19: DEPOSITRESTRICTED(K, x, Y)
20: **end if**
21: **end function**

Ensure:

No overflow occurs during the algorithm.
The amount added to Y_{kP} is exactly $d(x, I+k)$ if $I + k \neq 0$.
The amount added to Y_{0P} is exactly $d(x, 0)/2^{p-W+1}$ if $I = 0$.

Algorithm 6.5 is identical to Algorithm 6.4 except for when the index of Y is 0, which is rare. In that case, the first accumulator Y_0 will be scaled by a factor of 2^{W-p-1} so that 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 accumulator 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 mantissas of both the primary field and the input value. The binning process as well as the correctness analysis are therefore similar to Algorithm 6.4. If the slice q is scaled back by 2^{p-W+1} and subtracted from x then the rest of the algorithm doesn't change in the absence of overflow. 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 perform twice the subtraction $x - q$ to compute r. Note that if an FMA (Fused-Multiply Adder) is available, we would not have to explicitly scale q back, one single FMA instruction suffices to compute $r = x - q * 2^{p-W+1}$ without any overflow.

In the rare case when the index of Y is 0, Algorithm 6.5 costs $3 * (K - 2) + 7 = 3K + 1$ FLOPs. Otherwise it has the same cost of $3K - 2$ FLOPs as Algorithm 6.4. Theorem 6.2 shows that Algorithm 6.5 enjoys the necessary properties.

Theorem 6.2. *Let Y be a K-fold indexed type of index I. Assume that we run Algorithm 6.5 on Y and some $x \in \mathbb{F}$, $|x| < 2^{b_I}$. If all requirements of the algorithm are satisfied, then the "Ensure" claim of Algorithm 6.5 holds.*

Proof. As noted earlier, in order to prove the correctness of Algorithm 6.5, we only need to show that $r = x - d(x,0)$ in line 10 for the case $\text{IINDEX}(Y) = 0$.

In line 6, we have that $q = S - Y_{0P} = d(x,0)/2^{p-W+1}$. By Theorem 4.4, $d(x,0) \leq 2 \cdot 2^{e_{\max}}$. We then have that in line 8, since $q = (d(x,0)/2^{p-W+1})2^{p-W} \leq 2^{e_{\max}}$ there is no overflow and as we scale by a power of two, $q = d(x,0)/2$ exactly. Again we divide into two cases based on the size of x.

If $|x| < 2^{a_0}$, we have $d(x,0) = 0$ by Lemma 4.1 and therefore $r = x - d(x,0) = x$ exactly in both line 9 and line 10.

If $|x| \geq 2^{a_0}$, we have $|x - d(x,0)| \leq 2^{a_0}$ by Theorem 4.3. Therefore, we have $|x| \geq |x - d(x,0)|$.

If $x > 0$, we have

$$x \geq x - d(x,0)/2 \geq x - d(x,0) \geq -x$$

If $x < 0$, we have

$$-x \geq x - d(x,0) \geq x - d(x,0)/2 \geq x$$

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

Since $d(x,0)/2 = \mathcal{R}_\infty(x, a_0+1)/2 \in 2^{a_0}\mathbb{Z} \in 2\epsilon 2^{b_0}$ (As $W < p-2$) and $x \leq 2^{b_0}$, $d(x,0)/2 \in \text{ulp}(x)\mathbb{Z}$ and 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 9.

Again since $d(x,0)/2, x - d(x,0)/2 \in \text{ulp}(x)\mathbb{Z}$, $x - d(x,0) \in \text{ulp}(x)\mathbb{Z}$ and since $|x - d(x,0)| \leq |x|$, $x - d(x,0)$ is representable and $r = r - q$ exactly in line 10.

At this point, since $r = x - d(x,0)$ and $|r| \leq 2^{a_0}$, no more overflow can occur in the algorithm and since the algorithm at this point is identical to Algorithm 6.4, the proof of Lemma 6.1 applies. □

Modifying Algorithm 6.5 to correctly handle exceptional values is easy to implement. At the beginning of Algorithm 6.5, we may simply check x and Y_{0P} for the exceptional values `Inf`, `-Inf`, and `NaN`. If any one of x or Y_{0P} is indeed exceptional, we add x to the (possibly finite) Y_{0P}. Otherwise, we deposit the finite value normally.

Note that if we ignore the explicit check for exceptional values, even though Algorithm 6.5 does not correctly propagate floating-point exceptions, computed results are still reproducible. If any input value is $\pm$`Inf`, the orbit operation at line 5 of Algorithm 6.5 will return a `NaN`, which in turn will be propagated and results in a `NaN` value for Y_{0P}. It means that if there is any exceptional value in input data, the final computed result will be a `NaN`, which is also reproducible.

Although checking for exceptional values explicitly is expensive, the cost can be reduced if several values are to be summed in the same method. We can run a block of summation assuming that there are no exceptional values, and then check at the end of the block if the final computed result is `NaN`. If it is indeed `NaN`, we can compute the exceptional result directly, without using any of the primitive operations.

Finally, we briefly discuss whether other rounding modes could be used. So far, we have assumed rounding "to nearest" with any tie breaking behavior, and used the operation $S = Y_{kP} + (r|1)$ in line 4 of Algorithm 6.4 to break ties by rounding up if r is nonnegative, and rounding down if r is negative. The mathematical property we need for reproducibility is to break ties in a way that does not depend on the mantissa of Y_{kP}. Another way to achieve this would be to compute $S = Y_{kP} + r$ rounding to nearest and breaking ties away from zero (RNA for short). RNA is specified in the IEEE 754 floating

point standard, but only required for decimal arithmetic [16]. A property of this approach is that the 3 main lines of the inner loop of Algorithm 6.4 would become $S = Y_{kP} + r$, $q = S - Y_{kP}$, and $r = r - q$, which perform the well-known two_sum operation, which has many other uses [18, 19, 9, 10, 20].

6.4 Renormalize

When depositing values into a K-fold indexed type Y of index I, Algorithms 6.4 and 6.5 assume that

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

throughout the routine. To enforce this condition, the indexed type must be **renormalized** at least every 2^{p-W-2} deposit operations, as will be shown in Theorem 6.3. The renormalization procedure is shown in Algorithm 6.6, which works for all indices $0 \leq I \leq i_{max}$. Algorithm 6.6 is available in ReproBLAS as `idxd_xirenorm` in `idxd.h` (see Section 8 for details).

Algorithm 6.6. Renormalize a K-fold indexed type Y of index I.

Require:

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

1: **function** RENORM(K, Y)
2: **for** $k = 0$ to $K-1$ **do**
3: **if** $Y_{kP} < 1.5\cdot \text{ufp}(Y_{kP})$ **then**
4: $Y_{kP} = Y_{kP} + 0.25\cdot \text{ufp}(Y_{kP})$
5: $Y_{kC} = Y_{kC} - 1$
6: **end if**
7: **if** $Y_{kP} \geq 1.75\cdot \text{ufp}(Y_{kP})$ **then**
8: $Y_{kP} = Y_{kP} - 0.25\cdot \text{ufp}(Y_{kP})$
9: $Y_{kC} = Y_{kC} + 1$
10: **end if**
11: **end for**
12: **end function**

Ensure:

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

The values $\mathcal{Y}_k$ are unchanged. Recall that by (5.3), if $I+k>0$,

$$\mathcal{Y}_k = \mathcal{Y}_{kP} + \mathcal{Y}_{kC} = (Y_{kP} - 1.5\epsilon^{-1}2^{a_{I+k}}) + (0.25\epsilon^{-1}2^{a_{I+k}})Y_{kC}$$

The renormalization operation is described in the "Carry-bit Propagation" Section (lines 21 to 32) of Algorithm 6 in [5], although it has been slightly modified so as not to include an extraneous case. Indexed types with exceptional values do not need renormalization. Algorithm 6.6 can be modified to handle indexed types with exceptional values by doing nothing when such types are encountered (depending on how ufp() behaves when given exceptional values, Algorithm 6.6 could change $\pm$`Inf` to `NaN`). In total, Algorithm 6.6 costs $3K$ FLOPs with a maximum of $2K$ conditional branches.

To show the reasoning behind the assumptions in Algorithm 6.6, we prove Theorem 6.3.

Theorem 6.3. *Assume $x_0, x_1, ...x_{n-1} \in \mathbb{F}$ are successively deposited (using Algorithm 6.5) in a K-fold indexed type Y of index I where $\max|x_j| < 2^{b_I}$. If Y initially satisfies*

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

and $n \leq 2^{p-W-2}$, then after all of the deposits,

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

Proof. As the proof when $I + k = 0$ is almost identical to the case where $I + k > 0$, we consider here only the case that $I + k > 0$. First, note that $|d(x_j, I+k)| \leq 2^{b_{I+k}}$ by Theorem 4.4, where $d(x_j, I+k)$ is the amount added to Y_{kP} on iteration k.

By Theorem 6.2, DEPOSIT (Algorithm 6.5) extracts and adds the slices of x_j exactly (assuming $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$ at each step, which will be shown),

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

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

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

□

If an indexed type Y initially satisfies $Y_{kP} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$ (such a condition is satisfied upon initialization of a new accumulator of Y during the updating process as will be explained in Section 6.2) and we deposit at most 2^{p-W-2} floating point numbers into it, then Theorem 6.3 shows that after all of the deposits, $Y_{kP} \in [1.25\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$. Therefore, after another renormalization, the primary fields would once again satisfy $Y_{kP} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$. The limit on the number of floating

point inputs that can be accumulated without executing a renormalization operation (2^{p-W-2}) also requires that $p - W - 2 > 0$, or

$$W < p - 2. \tag{6.1}$$

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 each renormalization results in addition not in excess absolute value of 1 to Y_{kC}, a maximum of $\epsilon^{-1} - 1$ renormalizations may be performed, meaning that an indexed type is capable of representing the sum of at least

$$(\epsilon^{-1} - 1)2^{p-W-2} \approx 2^{2p-W-2} \tag{6.2}$$

floating point numbers. The value of $(\epsilon^{-1} - 1)2^{p-W-2}$ is approximately 2^{64} in double and 2^{33} in single precision using the values in Table 2.

Note that this value of maximum number of additions is slightly bigger than that of [5] since we exclude the extraneous case which caused the increment of the carry field to at most 2 in absolute value per each renormalization. These bounds also exceed the largest integer that can be represented in the same-sized integer format, which helps justify the choice of W.

6.5 Add Float To Indexed

Algorithm 6.7 allows the user to add a single floating point number to an indexed 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 6.6, making Algorithm 6.7 more useful for small sums or sums where the summands are not gathered into a vector. This method is available in ReproBLAS as `idxd_xixadd` in `idxd.h` (see Section 8 for details).

Algorithm 6.7. Add floating point $x_n \in F$ to K-fold indexed sum Y

Require: Y is the indexed sum of $x_0, ..., x_{n-1} \in F$. (If $n = 0$, this implies that all fields of Y are 0 and Y will be initialized in line 2)

1: **function** ADD FLOAT TO INDEXED(K, x_n, Y)
2: UPDATE(K, x_n, Y)
3: DEPOSIT(K, x_n, Y)
4: RENORM(K, Y)
5: **end function**

Ensure: Y is the indexed sum of $x_0, ..., x_n$.

The following theorem proves the "Ensure" claim at the end of Algorithm 6.7.

Theorem 6.4. *If Y is the K-fold indexed sum of $x_0, ..., x_{n-1}$, then after running Algorithm 6.7 on Y and some $x_n \in \mathbb{F}$, Y is the indexed sum of $x_0, ..., x_n$.*

Proof. As Y is the indexed sum of $x_0, ..., x_{n-1}$, the requirements of UPDATE (Algorithm 6.3) are satisfied. Therefore, after UPDATE completes, we have that the index of Y is the greatest integer I such that for all j, $0 \leq j \leq n$, $|x_j| < 2^{b_I}$ and $I \leq i_{\max}$. We also have that

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

and

$$Y_{kP} \in \begin{cases} [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}}) \text{ if } I + k > 0 \\ [1.5 \cdot 2^{e_{\max}}, 1.75 \cdot 2^{e_{\max}}) \text{ otherwise} \end{cases} \tag{6.3}$$

(Unless Y_{0P} was `Inf`, `-Inf`, or `NaN` before the update operation, in which case Y was unchanged and is still exceptional)

Therefore, the requirements of DEPOSIT (Algorithm 6.5) are satisfied and after it is complete, we have that

$$\mathcal{Y}_k = \sum_{j=0}^{n} d(x_j, I + k) \tag{6.4}$$

(Unless Y_{0P} was `Inf`, `-Inf`, or `NaN` before the update operation, in which case Y_{0P} should reflect the correct value. Since the renormalization step does not act on exceptional indexed types, we are done in this case.)

Finally, since (6.3) held before the deposit operation, Theorem 6.3 holds and after the deposit (in Section 6.6 we will use the fact that up to 2^{p-W-2} deposits may be performed here as long as each new x_j is such that $|x_j| < 2^{b_I}$), the requirements of RENORM (Algorithm 6.6) hold and we have that once again

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

Since the renormalization step does not affect the values of the accumulators or the index of the indexed type, (6.4) still holds and all of the properties of the indexed sum are satisfied. □

A special usage of Algorithm 6.7 is to convert a single floating point number to an indexed type. Converting a floating point number to an indexed type should produce, for transparency and reproducibility, the indexed sum of the single floating point number. The procedure is very simply summarized by Algorithm 6.8, and is available in ReproBLAS as `idxd_xixconv` in `idxd.h` (see Section 8 for details).

Algorithm 6.8. Convert floating point x to a K-fold indexed type Y.

1: **function** ConvertFloatToIndexed(K, x, Y)
2: $Y = 0$
3: Add Float To Indexed(K, x, Y)
4: **end function**

Ensure: Y is the indexed sum of x.

6.6 Sum

Algorithm 6.9 is an indexed summation algorithm that allows the user to efficiently add a vector of floating point numbers $x_m, ..., x_{m+n-1} \in \mathbb{F}$ to the indexed sum Y of some $x_0, ..., x_{m-1} \in F$.

As mentioned in Section 6.4, it is not necessary to perform a renormalization for every deposit, as would be done if Algorithm 6.7 were applied iteratively on each element of $x_m, ..., x_{m+n-1}$. At most 2^{p-W-2} values can be deposited in the indexed type before having to perform the renormalization. Therefore we have an improved version of Algorithm 6.7 when we need to sum a vector of floating point numbers. Algorithm 6.9 summarizes the optimized version for the reproducible local sum. It is available as `idxdBLAS_xixsum` in `idxdBLAS.h` (see Section 8 for details). As Algorithm 6.9 computes an indexed sum, it can be performed on the $x_m, ..., x_{m+n-1}$ in any order. However, for the simplicity of presenting the algorithm, it is depicted as running linearly from m to $m+n-1$. Algorithm 6.9 uses only one indexed type to hold the intermediate result of the recursive summation, and the vast majority of time in the algorithm is spent in Deposit (Algorithm 6.5).

Algorithm 6.9. If Y is the K-fold indexed sum of some $x_0, ..., x_{m-1} \in \mathbb{F}$, produce the K-fold indexed sum of $x_0, ..., x_{m+n-1} \in \mathbb{F}$. (If $m = 0$, this implies that all fields of Y are 0 and Y will be initialized in line 5) This is similar to Algorithm 6 in [5], but requires no restrictions on the size or type (exceptional or finite) of inputs $x_0, ..., x_{m+n-1}$.

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

Ensure:
Y is the unique indexed sum of $x_0, ..., x_{m+n-1}$.

If a single floating point result is desired, it may be obtained from Y using Algorithm 6.12 described in Section 6.8.

Theorem 6.5. *Assume that we have run Algorithm 6.9 on the K-fold indexed sum Y of $x_0, ..., x_{m-1} \in \mathbb{F}$ and on $x_m, ..., x_{m+n-1} \in \mathbb{F}$. If all requirements of the algorithm are satisfied, then the "Ensure" claim at the end of the algorithm holds.*

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

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

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

In this case, the proof of Theorem 6.4 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 6.3, each finite x_{m+j} deposited satisfies $|x_{m+j}| < 2^{b_I}$).

Therefore, after line 10, Y is the indexed sum of $x_0, ..., x_{m+j-1}$ □

Note that after computing max() in line 5, we know whether or not the indexed type will have index 0, and can call either DEPOSIT or DEPOSITRESTRICTED accordingly (the latter being a faster routine). We also know from the max whether or not an `Inf`, `-Inf`, or `NaN` is present, and can skip the DEPOSIT and RENORM procedures and compute the exceptional result directly. Also 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. In ReproBLAS, this constant is autotuned (as discussed in Section 8).

As the indexed sum is unique and independent of the ordering of its summands (Theorem 5.4), Algorithm 6.9 is reproducible for any permutation of its inputs.

At this point, an operation count should be considered. Since Algorithm 6.9 only performs the update and renormalization 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 the current implementation of ReproBLAS), the cost of Algorithm 6.9 is mostly due to the deposit operation. Therefore, in the absence of overflow, Algorithm 6.9 costs $\approx (3K-1)n$ FLOPs counting the maximum absolute value operation as 1 FLOP, which is $\approx 8n$ FLOPs for the default value of K ($K = 3$) used by ReproBLAS. In the rare case of index 0 for the first bin, the cost is slightly higher since the first bin needs to be scaled down to avoid overflow, which increases the total cost of Algorithm 6.9 to $\approx (3K+2)n$ FLOPs.

We make the observation that it is possible to implement a reproducible absolute sum by applying Algorithm 6.9 to the the absolute values of entries of an input vector.

It is also possible to compute a reproducible dot product of vectors $\vec{x}$ and $\vec{y}$. To modify Algorithm 6.9 for this purpose, we need only to change line 7 to deposit the product of the j^{th} entry of $\vec{x}$ and the j^{th} entry of $\vec{y}$ and line 5 to calculate the maximum absolute value of these products. More examples of how this routine can be used for reproducible operations are given in Section 7.

A few implementation details should be covered regarding our implementation of the complex dot product. When multiplying the complex numbers $(a+bi)$ and $(c+di)$, we assume that the product is obtained by evaluating $(ac-bd)+(ad+bc)i$ straightforwardly. This means that $(\texttt{Inf}+\texttt{Inf}i)(\texttt{Inf}+0i)$ produces $\texttt{NaN}+\texttt{NaN}i$. This is the Fortran convention for complex multiplica-

tion, not the C convention. In C, complex multiplication is implemented with a function call to catch the exceptional cases and give more sensible results. In the example above, we get $\texttt{Inf} + \texttt{Inf}i$. We refer the curious reader to part G.5.1 of the C99 standard [21] where this issue is discussed. We choose the Fortran definition of multiplication because the Reference BLAS is written in Fortran.

6.7 Add Indexed to Indexed

An operation to produce the sum of two indexed types 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 [5]. This method is available in ReproBLAS as `idxd_xixiadd` in `idxd.h` (see Section 8 for details).

Algorithm 6.10. Given a K-fold indexed type Y of index I and a K-fold indexed type Z of index J, add Z to Y.

Require:

Y is the indexed sum of some $x_0, ..., x_{n-1} \in F$.

Z is the indexed sum of some $x_n, ..., x_{n+m-1} \in F$.

1: **function** AddIndexedToIndexed(K, 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: $I =$ IIndex(Y)
10: $J =$ IIndex(Z)
11: **if** $J < I$ **then**
12: $R = Z$
13: AddIndexedToIndexed(K, 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: $Y_{0P} = Y_{0P} + (Z_{0P} - 1.5 \cdot 2^{e_{\max}})$
20: **else**
21: $Y_{kP} = Y_{kP} + (Z_{k+I-JP} - 1.5\epsilon^{-1}2^{a_{I+k}})$
22: **end if**
23: $Y_{kC} = Y_{kC} + Z_{k+I-JC}$
24: **end for**
25: Renorm(K, Y)
26: **end function**

Ensure:

Y is set to the indexed sum of $x_0, ..., x_{n+m-1}$.

$Y_{kP} \in [1.5\epsilon^{-1}2^{a_{\min(I,J)+k}}, 1.75\epsilon^{-1}2^{a_{\min(I,J)+k}})$

Theorem 6.6. *If the requirements of Algorithm 6.10 are satisfied, then the "Ensure" claim holds.*

Proof. If Y or Z are 0, then the algorithm correctly sets Y to the value of Z

or Y (respectively).

If both Y and Z are exceptional, then Algorithm 6.1 will return $I = J = 0$. The first iteration of the loop of line 17 will then set Y_{0P} to $Y_{0P} + Z_{0P} + 1.5 \cdot 2^{e_{\max}}$, which (since $1.5 \cdot 2^{e_{\max}}$ is finite) is equal to $Y_{0P} + Z_{0P}$, as desired.

If only one of Y or Z is exceptional, then Algorithm 6.1 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 or Z is exceptional, Y is set to Y or Z (respectively), as desired.

We now focus on the case where both Y and Z are finite.

We must first prove that the addition in line 21 is exact. As it is almost identical, we leave out the case where $I + k = 0$ and focus on the case where $I + k > 0$. Since J is the index of Z, the index of Z_{k+I-JP} is $J + (k + I - J) = I + k$. It means that $Z_{k+I-JP} \in [1.5\epsilon^{-1}2^{a_{I+k}}, 1.75\epsilon^{-1}2^{a_{I+k}})$ and $Z_{k+I-JP} \in 2^{a_{I+k}}\mathbb{Z}$. Therefore $Z_{k+I-JP} - 1.5\epsilon^{-1}2^{a_{I+k}} \in 2^{a_{I+k}}\mathbb{Z}$ and $Z_{k+I-JP} - 1.5\epsilon^{-1} \in [0, 0.25\epsilon^{-1}2^{a_{I+k}})$. This means $Z_{k+I-JP} - 1.5\epsilon^{-1}$ 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 $Y_{kP} + (Z_{k+I-JP} - 1.5\epsilon^{-1}2^{a_{I+k}}) \in 2^{a_{I+k}}\mathbb{Z}$, and $Y_{kP} + (Z_{k+I-JP} - 1.5\epsilon^{-1}2^{a_{I+k}}) \in [1.5\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$. This means $Y_{kP} + (Z_{k+I-JP} - 1.5\epsilon^{-1}2^{a_{I+k}})$ is representable and is exactly computed, and that the requirements of RENORM (Algorithm 6.6) apply.

We then have that after line 25,

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

We assume that $n + m \leq (\epsilon^{-1} - 1)2^{p-W-2}$ and therefore $Y_{kC} + Z_{k+I-JC}$ 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 $|x_j| < 2^{b_I}$ for all $j, n \leq j \leq n + m - 1$. It is also given that $I, J \leq i_{\max}$ 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 $I \leq i_{\max}$. □

6.8 Convert Indexed to Float

After computing a reproducible indexed sum, we need to reproducibly and accurately convert the result to a single floating point number. There are two sources of error in the final floating point sum produced through indexed summation. The first is from the creation of an indexed sum (whose error bound is analyzed in Lemma 6.8 in Section 6.9). The second is from the conversion from indexed sum to floating point number using either Algorithm 6.11 or Algorithm 6.12 (whose error bound is analyzed in Lemma 6.10 in Section 6.9). Theorem 5.4 guarantees that all fields in the indexed sum are reproducible. Therefore, as long as the fields are operated on deterministically by the final conversion back to original floating-point format, any method to evaluate (5.4) accurately and without unnecessary overflow is suitable.

To provide motivation for our intricate conversion routines, we forward-reference several results from Section 6.9. Assume that Y is the indexed sum of some $x_0, ..., x_{n-1} \in \mathbb{F}$ (for now, assume no exceptional values).

If we simply convert an indexed sum to a single floating point number by evaluating (5.4) in an arbitrary order and apply the standard summation error bound given by [22], the error in the final answer $\overline{\mathcal{Y}}$ (the computed floating point approximation of $\mathcal{Y}$) is only bounded by (here we state a bound from Lemma 6.12 of Section 6.9)

$$\begin{aligned}
\Big|\sum_{j=0}^{n-1} x_j - \overline{\mathcal{Y}}\Big| &\leq n \cdot \max\big(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\big) \\
&\quad + \left(\frac{(2K-1)\epsilon}{1-(2K-1)\epsilon}\right)\Big(\sum_{k=0}^{K-1}|\mathcal{Y}_{kP}| + \sum_{k=0}^{K-1}|\mathcal{Y}_{kC}|\Big) \\
&\approx n \cdot \max|x_j|\big(2^{W(1-K)} + (2K-1)\epsilon\big) \qquad (6.29)
\end{aligned}$$

However, we can prove a much tighter error bound if we evaluate (5.4) by recursively summing in the following order:

$$\begin{aligned}
&\underbrace{(((...(}_{2K}\mathcal{Y}_{0C}) + \mathcal{Y}_{1C}) \quad + \mathcal{Y}_{0P}) \quad + \mathcal{Y}_{2C}) \quad + \mathcal{Y}_{1P}) \quad + ... \\
&\qquad ... + \mathcal{Y}_{kC}) \quad + \mathcal{Y}_{k-1P}) + \mathcal{Y}_{k+1C}) + \mathcal{Y}_{kP}) \quad + ... \\
&\qquad ... + \mathcal{Y}_{K-2C}) + \mathcal{Y}_{K-3P}) + \mathcal{Y}_{K-1C}) + \mathcal{Y}_{K-2P}) + \mathcal{Y}_{K-1P}) \qquad (6.5)
\end{aligned}$$

We will show that if we evaluate (5.4) in the order specified by (6.5), then the error is bounded by (here we state a bound from Theorem 6.11 in Section 6.9)

$$|\sum_{j=0}^{n-1} x_j - \overline{\mathcal{Y}}| < \left(1 + \frac{7\epsilon}{1-6\sqrt{\epsilon}}\right)\left(n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right)\right) + \frac{7\epsilon}{1-6\sqrt{\epsilon}}|\sum_{j=0}^{n-1} x_j| \tag{6.25}$$

$$\approx n2^{W(1-K)} \max|x_j| + 7\epsilon|\sum_{j=0}^{n-1} x_j| \tag{6.26}$$

which is 8 orders of magnitude smaller than the bound in (6.29) when the true sum is tiny (assuming double precision and our standard choices of $W = 40$ and $K = 3$). This improvement in the error bound comes at no additional cost in terms of the number of required floating point operations (but does require more conditional branches). We refer the reader to Section 6.9 for a more detailed discussion of differences in error bounds between these methods.

Unfortunately, simply evaluating (5.4) in the order specified by (6.5) does not guard against unnecessary overflow. In this section, we explore ways to evaluate (6.5) while avoiding overflow, either by assuming a data type with a larger exponent range is available (Algorithm 6.11) or by using appropriate scaling (Algorithm 6.12). 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 ρ. The intermediate type must have at least as much precision and exponent range as the original floating point type. The necessary exponent range depends on how large the intermediate sum can become. This leads us to Lemma 6.7.

Lemma 6.7. *The absolute value of an indexed type (where the value is given by* (5.4)*) is at most* $2^{e_{\max}-W+2p}$.

Proof. By (5.2), 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 (5.1), 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 (5.6), 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 (4.2) and (5.4), we then have

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

Notice that in the last two steps we used the facts that $p \geq 8$ (4.8) and $W \geq 2$ (since $2W > p+1$ (4.7)). □

If the intermediate floating point type has a maximum exponent greater than or equal to $e_{\max} - W + 2p > e_{\max}$, then no special cases to guard against overflow are needed. It will be shown in Section 6.9 that the computed sum is accurate to within a small factor of the true sum; the exponent of the computed sum will stay less than or equal to $e_{\max} - W + 2p$ and will not overflow.

Algorithm 6.11 represents a conversion routine in such a case.

Algorithm 6.11. Convert K-fold indexed type Y of index I to floating point x.

Require:

Y is an indexed sum.

z is a floating point type with at least the original precision and maximum exponent greater than or equal to $e_{\max} - W + 2p$

1: **function** ConvertIndexedToFloat(K, x, Y)
2: **if** Y_{0P} is 0, `NaN` or $\pm$`Inf` **then**
3: $x = Y_{0P}$
4: **return**
5: **end if**
6: $z = \mathcal{Y}_{0C}$
7: **for** $k = 1$ to $K - 1$ **do**
8: $z = z + \mathcal{Y}_{kC}$
9: $z = z + \mathcal{Y}_{k-1P}$
10: **end for**
11: $z = z + \mathcal{Y}_{K-1P}$
12: $x = z$
13: **end function**

Ensure:

If Y_{0P} is `Inf`, `-Inf`, `NaN`, or 0, 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 (6.5) using an intermediate floating point format with enough exponent range to avoid intermediate overflow.

As explained in Section 5, 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 6.3, exceptional values (`NaN`, $\pm$`Inf`) 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 $\pm$`Inf` means that there is one or more values of $\pm$`Inf` of the same sign in the input, the rest are of finite value.

Note that an overflow situation in Algorithm 6.11 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 maximum exponent greater than or equal to $e_{\max} - W + 2p$ 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 effect an overflow situation.

If the scaled sum is to overflow, then its unscaled 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 $|\mathcal{Y}_{kP}| < 0.5 \cdot 2^{-\rho} 2^{e_{\max}}$ or $|\mathcal{Y}_{kC}| < 0.5 \cdot 2^{-\rho} 2^{e_{\max}}$ (remember that ρ is the intermediate floating point type's precision). If the floating point sum has exponent greater than or equal to $e_{\max}$ 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 true sum is at most $2^{e_{\max}-W+2p}$, a sufficient scaling factor is 2^{2p-W}, meaning that the maximum exponent of the true scaled sum is at most $e_{\max}$. It will be shown in Section 6.9 that the computed sum is accurate to within a small factor of the true sum; the exponent of the computed scaled sum will stay less than or equal to $e_{\max}$ and will not overflow.

When $\max |\mathcal{Y}_{kP}| < 0.5 \cdot 2^{-\rho} 2^{e_{\max}}$ and $\max |\mathcal{Y}_{kC}| < 0.5 \cdot 2^{-\rho} 2^{e_{\max}}$, 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 indexed type 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.

From the proof of Lemma 6.7, we have

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

and

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

These inequalities 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 6.12 can be greatly simplified. The conditions are left as is to make it more clear what is being compared.

Algorithm 6.12 represents a conversion routine in the case where a floating point type without a wider exponent is available.

Algorithm 6.12. Convert a K-fold indexed type Y of index I to floating point x.

Require:

Y is an indexed sum.

z is a floating point number with at least the original precision and exponent range. Assume z has precision $\rho \geq p$.

1: **function** ConvertIndexedToFloatNative(K, x, Y)
2: **if** Y_{0P} is 0, `NaN` or $\pm$`Inf` **then**
3: $x = Y_{0P}$
4: **return**
5: **end if**
6: $k = 1$
7: **if** $a_I + 2p - 2 > e_{\max} - \rho - 1$ **then**
8: $z = (\mathcal{Y}_{0C}/2^{2p-W})$
9: **while** $k \leq K-1$ and $(a_{I+k}+2p-2 \geq e_{\max}-\rho-1$ or $a_{I+k-1}+p-1 \geq e_{\max} - \rho - 1)$ **do**
10: $z = z + \mathcal{Y}_{kC}/2^{2p-W}$
11: $z = z + \mathcal{Y}_{k-1P}/2^{2p-W}$
12: $k = k + 1$
13: **end while**
14: **if** $a_{I+K-1} + p - 1 \geq e_{\max} - \rho - 1$ **then**
15: $z = z + \mathcal{Y}_{K-1P}/2^{2p-W}$
16: $x = z \cdot 2^{2p-W}$
17: **return**
18: **end if**
19: $z = z \cdot 2^{2p-W}$
20: **end if**
21: **while** $k \leq K - 1$ **do**
22: $z = z + \mathcal{Y}_{kC}$
23: $z = z + \mathcal{Y}_{k-1P}$
24: $k = k + 1$
25: **end while**
26: $z = z + \mathcal{Y}_{K-1P}$
27: $x = z$
28: **end function**

Ensure:

If Y_{0P} is `Inf`, `-Inf`, `NaN`, or 0, 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 (6.5) using an intermediate floating point format with enough exponent range to avoid intermediate overflow.

If an indexed type is composed of `float`, then `double` provides sufficient precision and exponent to use as an intermediate type and Algorithm 6.11 may be used to convert to a floating point number. However, if an indexed type 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 6.12 for the conversion.

In ReproBLAS, the appropriate conversion for the given data type is available as `idxd_xxiconv` in `idxd.h` (see Section 8 for details). These conversion routines sum `float` with `double` (using Algorithm 6.11) and sum `double` with `double` (using Algorithm 6.12).

6.9 Error Bound

There are two sources of error in the final floating point sum produced through indexed summation. The first is from the creation of an indexed sum. The second is from the conversion from indexed sum to a floating point number.

Lemma 6.8 analyzes the error from the creation of the indexed sum.

Lemma 6.8. *Consider the K-fold indexed sum Y of finite floating point numbers $x_0, \ldots, x_{n-1}$. We denote the true sum $\sum_{j=0}^{n-1} x_j$ by T and the true value of the indexed sum as obtained using (5.4) by $\mathcal{Y}$. Then we have:*

$$|T - \mathcal{Y}| < n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) \tag{6.6}$$

Proof. The case of all zero input data is trivial, therefore we assume that $\max|x_j|$ is nonzero. We also assume here no overflow or underflow. Let I be the index of Y, which is also the index of $\max|x_j|$, so that $2^{b_I} > \max|x_j| \geq 2^{a_I}$. Therefore for all $i < I$ the slice of any x_j in bin i is $d(x_j, i) = 0$. The index of the smallest bin of Y is $I + K - 1$. According to Theorem 4.3, we have

$$\begin{aligned} |x_j - \sum_{i=I}^{I+K-1} d(x_j, i)| = |x_j - \sum_{i=0}^{I+K-1} d(x_j, i)| &\leq 2^{a_{I+K-1}} = 2^{a_I - (K-1)W} \\ &\leq 2^{W(1-K)} \max|x_j|. \end{aligned}$$

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

$$|T - \mathcal{Y}| = |\sum_{j=0}^{n-1} x_j - \sum_{i=I}^{I+K-1} \sum_{j=0}^{n-1} d(x_j, i)| = |\sum_{j=0}^{n-1} (x_j - \sum_{i=I}^{I+K-1} d(x_j, i))|$$
$$\leq n 2^{W(1-K)} \max |x_j|. \tag{6.7}$$

However, this bound does not consider underflow. By (4.11), a small modification yields a bound that considers underflow

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

□

Now that we have shown a bound on the difference between the true sum and the indexed sum, we must bound the difference between the indexed sum and the final result returned by Algorithms 6.11 and 6.12. As discussed in Section 6.8, to convert an indexed type to a floating point number, one must evaluate (5.4) accurately and without unnecessary overflow. This amounts to summing the fields of the indexed type. The main idea behind this second bound is to bound the error in the sum of floating point numbers in order of decreasing exponent, and then show that the ordering in (6.5) satisfies the conditions of this bound.

We first state and prove Lemma 6.9, the error bound on sums of floating point numbers in order of decreasing exponent.

It should be noted that Lemma 6.9 is similar to that of Theorem 1 from [23], but requires less intermediate precision by exploiting additional structure of the input data. It is possible that future implementers may make modifications to the indexed type (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 6.9. In such an event, [23] provides more general ways to sum the fields while still maintaining accuracy.

Lemma 6.9. *We are given n floating point numbers $f_0, \ldots, f_{n-1}$ for which there exist (possibly unnormalized) floating point numbers $f'_0, \ldots, f'_{n-1}$ of the same precision such that*

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

Let $S_0 = \overline{S_0} = f_0$, $S_j = S_{j-1} + f_j$, and $\overline{S_j} = fl(\overline{S_{j-1}} + f_j)$ (assuming rounding to nearest, breaking ties arbitrarily) so that $S_{n-1} = \sum_{j=0}^{n-1} f_j$. 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 that $S_\infty = S_{n-1}$ 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 $\exp(f'_0) > ... > \exp(f'_m)$, $f_0, ..., f_m \in \text{ulp}(f'_m)\mathbb{Z}$. Thus, $S_m \in \text{ulp}(f'_m)\mathbb{Z}$.

We now show $|S_m| > 2 \cdot 2^{\exp(f'_m)}$. Assume for contradiction that $|S_m| \leq 2 \cdot 2^{\exp(f'_m)}$. 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

$$|S_m| > 2 \cdot 2^{\exp(f'_m)} \tag{6.8}$$

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

$$|f_{m+1}| < 2 \cdot 2^{\exp(f'_m - 1)} = 2^{\exp(f'_m)} \tag{6.9}$$

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

$$\begin{aligned} \Big| \sum_{j=m+2}^{n-1} f_j \Big| &\leq \sum_{j=m+2}^{n-1} |f_j| < \sum_{j=m+2}^{n-1} 2 \cdot 2^{\exp(f'_j)} \leq \sum_{j=m+2}^{n-1} 2 \cdot 2^{\exp(f'_m) - \lceil \frac{p+1}{2} \rceil - (m+2-j)} \\ &< \sum_{j=0}^{\infty} \left(2\sqrt{\epsilon}\right) 2^{\exp(f'_m)-j} = \left(4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} \end{aligned} \tag{6.10}$$

We can combine (6.9) and (6.10) to obtain

$$\Big| \sum_{j=m+1}^{n-1} f_j \Big| \leq \sum_{j=m+1}^{n-1} |f_j| < 2^{\exp f'_m} + \left(4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} = \left(1 + 4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} \tag{6.11}$$

By (6.8) and (6.11),

$$\begin{aligned} |S_{n-1}| = \Big|\sum_{j=0}^{n-1} f_j\Big| &\geq \Big|\sum_{j=0}^{m} f_j\Big| - \Big| \sum_{j=m+1}^{n-1} f_j\Big| = |S_m| - \Big| \sum_{j=m+1}^{n-1} f_j\Big| \\ &\geq 2 \cdot 2^{\exp(f'_m)} - \left(1 + 4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} = \left(1 - 4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} \end{aligned} \tag{6.12}$$

By (6.12) and (6.10),

$$\Big| \sum_{j=m+2}^{n-1} f_j \Big| < \left(4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} \leq \frac{4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}} \Big|\sum_{j=0}^{n-1} f_j\Big| \tag{6.13}$$

By (6.12) and (6.11),

$$\Big| \sum_{j=m+1}^{n-1} f_j \Big| \leq \sum_{j=m+1}^{n-1} |f_j| \leq \left(1 + 4\sqrt{\epsilon}\right) 2^{\exp(f'_m)} \leq \frac{1 + 4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}} \Big|\sum_{j=0}^{n-1} f_j\Big| \tag{6.14}$$

And by (6.12) and (6.14),

$$|S_m| \leq \Big|\sum_{j=0}^{n-1} f_j\Big| + \Big| \sum_{j=m+1}^{n-1} f_j\Big| \leq \left(1 + \frac{1 + 4\sqrt{\epsilon}}{1 - 4\sqrt{\epsilon}}\right) \Big|\sum_{j=0}^{n-1} f_j\Big| = \frac{2}{1 - 4\sqrt{\epsilon}} \Big|\sum_{j=0}^{n-1} f_j\Big| \tag{6.15}$$

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. According to [22, Equation 1.2,

2.4]

$$\left|\overline{S_m}+\sum_{j=m+1}^{m+4} f_j-\overline{S_{m+4}}\right| \le \frac{4\epsilon}{1-4\epsilon}\left|\overline{S_m}+f_{m+1}\right|+\frac{3\epsilon}{1-3\epsilon}\sum_{j=m+2}^{m+4}|f_j|$$
$$\le \frac{4\epsilon}{1-4\epsilon}\left(\left|\overline{S_m}-S_m\right|+|S_m+f_{m+1}|\right)+\frac{3\epsilon}{1-3\epsilon}\sum_{j=m+2}^{n-1}|f_j|.$$

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

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

Therefore

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

Using the triangle inequality we have

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

and by (6.15) and (6.13),

$$\left|S_{m+4}-\overline{S_{m+4}}\right| \le \frac{\epsilon}{1-4\epsilon}\left(\frac{2}{1-4\sqrt{\epsilon}}|S_{n-1}|+4|S_{n-1}|+7\frac{4\sqrt{\epsilon}}{1-4\sqrt{\epsilon}}|S_{n-1}|\right)$$
$$\le \frac{\epsilon}{1-4\epsilon}\left(\frac{6+12\sqrt{\epsilon}}{1-4\sqrt{\epsilon}}|S_{n-1}|\right)=\frac{6\epsilon}{(1-2\sqrt{\epsilon})(1-4\sqrt{\epsilon})}|S_{n-1}|$$
$$< \frac{6\epsilon}{1-6\sqrt{\epsilon}}|S_{n-1}| \qquad (6.16)$$

Notice that

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

Therefore,

$$\exp(f'_m) \geq \exp(f'_{m+5}) + p + 2 \tag{6.17}$$

Because $\exp(f'_0) > ... > \exp(f'_{n-1})$, (6.17) yields

$$\Big|\sum_{j=m+5}^{n-1} f_j\Big| \leq \sum_{j=m+5}^{n-1} |f_j| < \sum_{j=m+5}^{n-1} 2 \cdot 2^{\exp(f'_m)-p-2-(j-(m+5))} < \sum_{j=0}^{\infty} 2^{\exp(f'_m)-p-1-j} = \epsilon 2^{\exp(f'_m)} \tag{6.18}$$

Using (6.12) and (6.18),

$$\Big|\sum_{j=m+5}^{n-1} f_j\Big| \leq \sum_{j=m+5}^{n-1} |f_j| < \frac{\epsilon}{1-4\sqrt{\epsilon}} |S_{n-1}| \tag{6.19}$$

By (6.16) and (6.19)

$$\begin{aligned} \left|S_{n-1} - \overline{S_{m+4}}\right| &\leq |S_{n-1} - S_{m+4}| + \left|S_{m+4} - \overline{S_{m+4}}\right| \\ &\leq \Big|\sum_{j=m+5}^{n-1} f_j\Big| + \frac{6\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}| \\ &\leq \frac{\epsilon}{1-4\sqrt{\epsilon}} |S_{n-1}| + \frac{6\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}| \\ &< \frac{7\epsilon}{1-6\sqrt{\epsilon}} |S_{n-1}|. \end{aligned} \tag{6.20}$$

When combined with (6.12) this gives

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

which, assuming $\epsilon \ll 1$, can be simplified to

$$\left|\overline{S_{m+4}}\right| > 2^{\exp(f'_m)-1} \tag{6.21}$$

Using (6.17), for all $j \geq m+5$ we have

$$|f_j| < 2 \cdot 2^{\exp(f'_j)} \leq 2 \cdot 2^{\exp(f'_m)-p-2} = \epsilon \cdot 2^{\exp(f'_m)-1} \tag{6.22}$$

And by (6.22) and (6.21), all additions after $\underline{f_{m+4}}$ have no effect (since we are rounding to nearest) and we have $\overline{S_{n-1}} = \overline{S_{m+4}}$. This, together with (6.20), 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 6.9, all that remains is to show that it applies to Algorithms 6.11 and 6.12. Since both algorithms add the numbers according to (6.5), we must show that this ordering satisfies the assumptions of Lemma 6.9.

Lemma 6.10. *Consider the K-fold indexed sum Y of index I of finite floating point numbers $x_0, \ldots, x_{n-1}$. We denote the true value of the indexed sum as obtained using* (5.4) *by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained using an appropriate algorithm from Section 6.8 (Algorithm 6.11 or 6.12) 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 (6.5), 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 indexed type Y of index I. 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 $Y_{kP} \in (\epsilon^{-1}2^{a_{I+k}}, 2\epsilon^{-1}2^{a_{I+k}})$, $\mathcal{Y}_{kP}$ 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 $\exp(\mathcal{Y}'_{kP}) > \exp(\mathcal{Y}'_{k+1P})$ and $\exp(\mathcal{Y}'_{kC}) > \exp(\mathcal{Y}'_{k+1C})$ because $a_{I+k} > a_{I+k+1}$.

Next, note that

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

and

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

Therefore $\exp(\mathcal{Y}'_{kC}) > \exp(\mathcal{Y}'_{k-1P})$ because $W < p - 2$.

Finally, note that

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

Therefore $\exp(\mathcal{Y}'_{kC}) < \exp(\mathcal{Y}'_{k-2P})$ because $2W > p + 1$.

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:

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

Note that the above ordering is the same as that in (6.5).

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

We have just shown that

$$\exp(\gamma'_0) > ... > \exp(\gamma'_{2K-1}) \tag{6.23}$$

where γ_j denotes the normalized representation of the γ'_j. It should be noted that $\gamma_j = \gamma'_j$ as real numbers and that $\exp(\gamma_j) \leq \exp(\gamma'_j)$.

It should be noted that 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 $p \geq 8$ (4.8) we have $\exp(\gamma'_{2K-3}) = a_{I+K-1} + 2p - 3 \geq a_{I+K-1} + p + \lceil \frac{p+1}{2} \rceil - 1 = \exp(\gamma'_{2K-1}) + \lceil \frac{p+1}{2} \rceil$). Therefore, as $2W > p + 1$ and $W < p - 2$, for all $j \in \{0, ..., 2K - 3\}$

$$\exp(\gamma'_j) \geq \exp(\gamma'_{j+2}) + W \geq \exp(\gamma'_{j+2}) + \left\lceil \frac{p+1}{2} \right\rceil \tag{6.24}$$

It should be noted that the $\mathcal{Y}'_{kP}$ and the $\mathcal{Y}'_{kC}$ can be expressed exactly using floating point types 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 (5.1) and (5.2). By (6.23) and (6.24), Lemma 6.9 applies to (6.5) to yield

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

□

Now that we have Lemma 6.8 and 6.10, we have all the necessary ingredients to give the final error bounds! [5] discusses the absolute error between the indexed sum and the true sum (addressed here by Lemma 6.8), but does not give a method to compute a floating point approximation of the indexed sum. No error bound on the final floating point answer was given. Theorem 6.11 extends the error bound of [5] all the way to the final return value of the algorithm, combining the results of Lemmas 6.8 and 6.10.

Theorem 6.11. *Consider the K-fold indexed sum Y of finite floating point numbers $x_0, \ldots, x_{n-1}$. We denote the true sum $\sum_{j=0}^{n-1} x_j$ by T, the true value of the indexed sum as obtained using* (5.4) *by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained using an appropriate algorithm from Section 6.8 (Algorithm 6.11 or 6.12) by $\overline{\mathcal{Y}}$. Assuming the final answer does not overflow,*

$$\left|T - \overline{\mathcal{Y}}\right| < \left(1 + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}\right)\left(n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right)\right) + \frac{7\epsilon}{1 - 6\sqrt{\epsilon}}|T| \tag{6.25}$$

$$\approx n2^{W(1-K)} \max|x_j| + 7\epsilon|T| \tag{6.26}$$

and

$$\left|T - \overline{\mathcal{Y}}\right| < n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) + \frac{7\epsilon}{1 - 6\sqrt{\epsilon} - 7\epsilon}\left|\overline{\mathcal{Y}}\right| \tag{6.27}$$

$$\approx n2^{W(1-K)} \max|x_j| + 7\epsilon\left|\overline{\mathcal{Y}}\right| \tag{6.28}$$

Proof. Lemma 6.8 gives us

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

Lemma 6.10 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 (6.25), the absolute error of the floating point approximation of an indexed sum $|T - \overline{\mathcal{Y}}|$:

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

A perhaps more useful mathematical construction is the error expressed relative to the result $\overline{\mathcal{Y}}$, and not the theoretical sum T. Again by the triangle inequality,

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

Applying the bound on $|\mathcal{Y} - \overline{\mathcal{Y}}|$ 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 (6.27), the absolute error of the floating point approximation of an indexed sum $|T - \overline{\mathcal{Y}}|$:

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

□

(6.27) can be evaluated in ReproBLAS with the `idxd_xibound` function in `idxd.h` (see Section 8 for details).

We can compare (6.26) to the error bound obtained if Algorithms 6.11 and 6.12 evaluated (5.4) in some order other than (6.5). In this case, only the standard summation bound from [22] would apply and the absolute error would be bounded as in Lemma 6.12.

Lemma 6.12. *Consider the K-fold indexed sum Y of finite floating point numbers $x_0, \ldots, x_{n-1}$. We denote the true sum $\sum_{j=0}^{n-1} x_j$ by T, the true value of the indexed sum as obtained using (5.4) by $\mathcal{Y}$, and the floating point approximation of $\mathcal{Y}$ obtained by straightforward evaluation (in some fixed, arbitrary order) of (5.4) by $\overline{\mathcal{Y}}$. Assuming the final answer does not overflow,*

$$\begin{aligned}
\left|T - \overline{\mathcal{Y}}\right| &\leq n \cdot \max\left(2^{W(1-K)} \max|x_j|, 2^{e_{\min}-2}\right) \\
&\quad + \left(\frac{(2K-1)\epsilon}{1-(2K-1)\epsilon}\right)\left(\sum_{k=0}^{K-1}|\mathcal{Y}_{kP}| + \sum_{k=0}^{K-1}|\mathcal{Y}_{kC}|\right) \\
&\approx n \cdot \max|x_j|\left(2^{W(1-K)} + (2K-1)\epsilon\right) \qquad (6.29)
\end{aligned}$$

Proof. Lemma 6.8 gives us

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

The standard error bound on floating point summation (given by [22]) gives us

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

Combining with the triangle inequality, we obtain

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

□

This is not as tight a bound as (6.26), and grows linearly as the user increases K in an attempt to increase accuracy. As depicted in Figure 6.1, in cases where there is a lot of cancellation (when $|T| << \max|x_j|$), bound (6.26) can be better by over 8 orders of magnitude in default configuration for double precision, where $K = 3$ and $W = 40$.

To better understand the accuracy of indexed summation, we compare both (6.26) and (6.29) to an approximated standard error bound obtained through standard (recursive) summation of $x_0, ..., x_{n-1}$ in some arbitrary order (given by [22]).

$$n\epsilon\sum_{j=0}^{n-1}|x_j| \leq n^2\epsilon\max|x_j| \tag{6.30}$$

Figure 6.1 compares these three approximate error bounds for $K = 3$, $W = 40$ and double precision $p = 53$. The horizontal axis represents the condition number, which is a measure of how much cancellation occurs in the sum. Note that the term $7\epsilon|\sum_{j=0}^{n-1} x_j|$ is only 7 times larger than the smallest possible error from rounding the exact sum of the x_j to the nearest floating point value. To compare the other terms, bound (6.26) grows like $2^{-80}n \cdot \max|x_j|$, whereas bound (6.29) grows like $5\epsilon n \cdot \max|x_j| = 5 \cdot 2^{-53}n \cdot \max|x_j|$, which is approximately 8 orders of magnitude larger. Note that in this comparison we bound $\sum_{j=0}^{n-1}|x_j|$ by $n \cdot \max|x_j|$, which is the case where the input data are almost equal in magnitude, so the error bound of

the standard summation (6.30) can grow like n^2 instead of n which is much worse than both bounds (6.26) and (6.29) when the number of input values n is great. In cases where $\sum_{j=0}^{n-1} |x_j| \approx \max |x_j|$, for example when there are just a few large values and the others are small, then bounds (6.30) and (6.29) are almost of the same order of magnitude, which is still worse than bound (6.26) by 8 orders of magnitude when the true sum is tiny.

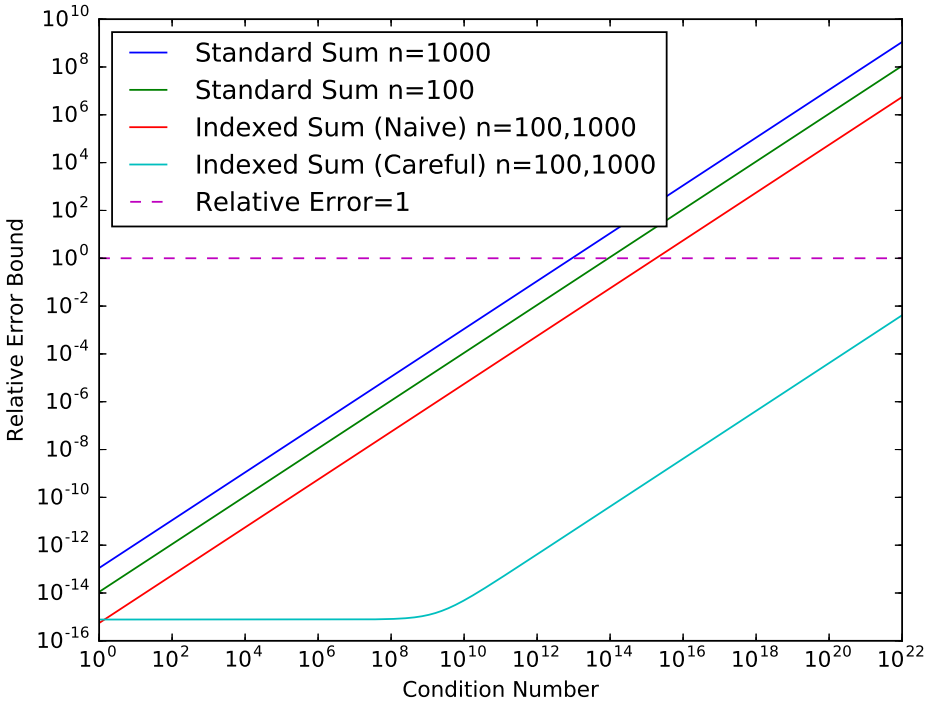

Figure 6.1: Relative error bounds ($\frac{\text{absolute error bound}}{|\sum_{j=0}^{n-1} x_j|}$) in calculating $|\sum_{j=0}^{n-1} x_j|$ for different condition numbers (which we define as $\frac{n \cdot \max |x_j|}{|\sum_{j=0}^{n-1} x_j|}$) of the sum. It is assumed that we sum using `double`, $K = 3$, and $W = 40$. "Indexed Sum (Careful)" corresponds to (6.26) divided by the true sum. "Indexed Sum (Naive)" corresponds to (6.29) divided by the true sum. "Standard Sum" corresponds to (6.30) divided by the true sum and due to a dependence on n multiple error bounds are shown.

6.10 Restrictions

Here we summarize the restrictions on indexed summation presented thus far. The restrictions are summarized in Table 3. All macros are defined in header files discussed in Section 8.

As discussed previously, for a K-fold indexed type the minimum K accepted by ReproBLAS is 2. The maximum useful K is $K_{max} = \lfloor (e_{\max} - e_{\min} + p - 1)/W \rfloor$, 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 6.9).

As discussed in [5], $W < p-2$. As discussed in Section 5.1.1, $2W > p+1$. The combination of these two implies $p \geq 8$.

By (6.25), we see that increasing W and K will increase the accuracy of reproducible summation. However, increases to K increase the number of flops necessary to compute an indexed sum and increases to W decrease the number of deposits that can be performed between renormalizations (discussed in Section 6.4). We have chosen default values for W and K that give good performance while maintaining better accuracy than standard recursive summation. ReproBLAS uses the values $W = 40$ for indexed `double` and $W = 13$ for indexed `float`. W is configurable as the `XIWIDTH` macro in `config.h`.

As absolute values of individual quantities added to Y_{kP} are not in excess of $2^{b_{I+k}}$, a maximum of $0.25\epsilon^{-1}2^{-W}$ elements may be deposited into Y_{kP} between renormalizations, as discussed in Section 6.4. For indexed `double` this number is 2^{11}, whereas for indexed `float` this number is 2^9. This number is referred to as the **endurance** of an indexed type and is supplied in ReproBLAS using the `XIENDURANCE` macro in `idxd.h`.

By (6.2), an indexed type is capable of representing the sum of at least $0.25\epsilon^{-1}2^{-W}\epsilon^{-1} = 2^{2p-W-2}$ floating point numbers. For indexed `double` this number is 2^{64}, whereas for indexed `float` this number is 2^{33}. This number is referred to as the **capacity** of the indexed type and supplied in ReproBLAS using the `XICAPACITY` macro in `idxd.h`.

The indexed types provided by ReproBLAS will, when used correctly, avoid intermediate overflow assuming the restrictions in the previous paragraph.

As discussed in Section 5.1.2, we assume gradual underflow and ReproBLAS will round the sum at best to the nearest $2^{e_{\min}-1}$. However, Sections 5.1.2 and 5.1.3 discuss ways that reproducible summation could be implemented and improved in the presence of both gradual and abrupt underflow.

Table 3: ReproBLAS parameter limits

Data Type	`float`	`double`	`quad`
W	13	40	100
K_{min}	2	2	2
K_{max}	21	52	328
Endurance	2^9	2^{11}	2^{11}
Capacity	2^{33}	2^{64}	2^{124}

7 Composite Operations

The ultimate goal of the ReproBLAS library is to support reproducible linear algebra functions on many widely-used architectures.

We must account for any combination of several sources of nonreproducibility. These sources include arbitrary data permutation or layout, differing numbers of processors, and arbitrary reduction tree shape. The methods we will describe can be used to account for any or all of these sources of nonreproducibility. However, if there is only one potential source of nonreproducibility then it may be more efficient to use a method that deals specifically with that source. For example, if there is an arbitrary reduction tree shape (due to the underlying MPI implementation) but not arbitrary data layout (because this is fixed by the application) then it is probably cheaper to just to use a method that deals with arbitrary reduction trees shapes.

Reproducibility is a concern for several parallel architectures. Apart from the typical sequential environment, linear algebra packages can be implemented on shared-memory systems (where multiple processors operate independently and share the same memory space), distributed memory systems (where each processor has its own private memory), or even cloud computing systems (remote internet-based computing where resources are provided to users on-demand).

The Basic Linear Algebra Subprograms (BLAS) [24] are widely used as an efficient, portable, and reliable sequential linear algebra library. The BLAS are divided into three categories.

1. The Level 1 BLAS (BLAS1) is a set of vector operations on strided arrays. Several of these operations are already reproducible, such as

the `xscal` operation, which scales a vector by a scalar value or the `xsaxpy` operation, which adds a scaled copy of one vector to another. These operations are reproducible provided they are implemented in the same way. For example `xsaxpy` might not be reproducible if one implementation uses a fused multiply-add instruction and the other does not. Other operations, such as the dot product or vector norm, are not reproducible with respect to data permutation, and must be modified to have this quality. These operations are `xasum`, `xnrm2`, and `xdot`.

2. The Level 2 BLAS (BLAS2) is a set of matrix-vector operations including matrix-vector multiplication and a solver for $\vec{x}$ in $T\vec{x} = \vec{y}$ where $\vec{x}$ and $\vec{y}$ are vectors and T is a triangular matrix. These operations can all be made reproducible, but for brevity we discuss only the representative operation `xgemv`, matrix-vector multiplication.

3. The Level 3 BLAS (BLAS3) is a set of matrix-matrix operations including matrix-matrix multiplication and routines to perform $B = \alpha T^{-1} B$ where α is a scalar, B is a matrix, and T is a triangular matrix. These operations can all be made reproducible, but for brevity we discuss only the representative operation `xgemm`, matrix-matrix multiplication.

The Linear Algebra PACKage (LAPACK) [25], is a set of higher-level routines for linear algebra, providing routines for operations like solving linear equations, linear least squares, eigenvalue problems, and singular value decomposition. LAPACK does most floating point computation using the BLAS.

The BLAS and LAPACK have been extended to distributed-memory systems in the PBLAS [26] and SCALAPACK [27] libraries. While most BLAS and PBLAS operations can clearly be made reproducible, it is an open question as to which LAPACK and SCALAPACK routines can be made reproducible simply by using reproducible BLAS, or whether there are other sources of nonreproducibility to eliminate. This is future work.

Out of the large design space outlined above, we describe in this paper and implement in ReproBLAS sequential versions of key operations from the BLAS, and a distributed-memory reduction operation which can be used to parallelize these sequential operations in a reproducible way. We described the BLAS1 operations `xasum` and `xdot` at the end of Section 6.6 previously. The BLAS1 operation `xnrm2` is discussed in Section 7.2. The BLAS2 and

BLAS3 operations `xgemv` and `xgemm` are discussed in Sections 7.3 and 7.4 respectively. Eventually, we intend to extend ReproBLAS to contain reproducible versions of all BLAS and PBLAS routines.

7.1 Reduce

Perhaps the most important function to make reproducible, the parallel reduction computes the sum of P floating point numbers, each number residing on a separate processor. The numbers are added pairwise in a tree. As different processors may have their number available at slightly different times, the shape of the tree may change from run to run. If a standard reduction is used, the results may differ depending on the shape of the reduction tree.

A parallel reduction can be accomplished reproducibly by converting the floating point numbers to indexed types with the ConvertFloatToIndexed procedure (Algorithm 6.8), reducing the indexed types reproducibly using the AddIndexedToIndexed procedure (Algorithm 6.10) at each node of the reduction tree, and then converting the resulting indexed sum to a floating point number with the ConvertIndexedToFloat procedure (Algorithm 6.12).

If there are several summands on every processor, then each processor should first compute the indexed sum of the local summands using the Sum procedure (Algorithm 6.9). The P resultant indexed sums can be reduced and converted to a floating point number as described above.

It is easy to see that the previously described reduction operations produce the indexed sum of their summands by inductively applying the "Ensure" claim of Algorithm 6.10.

An MPI data type that holds an indexed type can be created (creation is only performed once, subsequent calls return the same copy) and returned using the `idxdMPI_DOUBLE_COMPLEX_INDEXED`, etc. function of `idxdMPI.h`. `idxdMPI.h` also contains the `XIXIADD` method, an MPI reduction operator that can reduce the data types in parallel. See Section 8 for details.

7.2 Euclidean Norm

The Euclidean norm of a vector $\vec{x}$ (sometimes referred to as $\|\vec{x}\|$) is defined as

$$\|\vec{x}\| = \sqrt{\sum_{j=0}^{n-1} x_j^2}, \quad x_j \in \mathbb{F}. \tag{7.1}$$

Since the summands in the summation of (7.1) are all non-negative, there is no cancellation, so the Euclidean norm operation is usually of high accuracy. It can be implemented simply using a dot product operation on two copies of the same input vector, or by using the sum operation on the squares of the input vector, both of which can be made reproducible using corresponding reproducible dot product and summation operations. However, both of these methods are prone to unnecessary overflow and underflow. On one hand, if some input values were too big ($\geq \sqrt{2^{e_{max}+1}}$), the partial sum would overflow. The computed result would be `Inf` when the real Euclidean norm would be much smaller than the overflow threshold. On the other hand, if all input values were too small ($< \sqrt{2^{e_{min}}}$), their squared value would be smaller than the smallest representable floating point number and underflow would occur, causing the returned result to be 0.

Scaling techniques can be used to handle the underflow/overflow issues. For example, the function `xnrm2` from the BLAS library [24] scales input data by the intermediate maximum absolute value to avoid both overflow and underflow. However, in our indexed format scheme, such a scaling would alter the mantissae of the input data and the splitting of the mantissae, and therefore cannot guarantee reproducibility. In order to maintain reproducibility, first, scaling factors must be powers of two so that the scaling and rescaling won't change any mantissae of input values. Second, the scaling factors' exponents must differ only by multiples of W, so that the slicing processes using different scaling factors are identical. Algorithm 7.1 summarizes the algorithm for sum of squares, which will be used to compute the Euclidean norm. Note that the handling of special input values `Inf`, `-Inf`, and `NaN` is similar to the sum operation, which is not included here for simplicity.

Algorithm 7.1. If Y is the K-fold indexed sum of some $(x_0/s)^2, ..., (x_{n-1}/s)^2$ where $x_0, ..., x_n \in \mathbb{F}$, $s \in 2^{W\mathbb{Z}}$, produce the K-fold indexed sum of $(x_0/t)^2, ..., (x_n/t)^2$ where $t \in 2^{W\mathbb{Z}}$.

Require:

$s = 0$ if $\max_{j\in\{0,...,n-1\}} |x_j| = 0$. Otherwise $2^{-p-W-1}s < \max_{j\in\{0,...,n-1\}} |x_j| < 2^{W+2}s$ and $s \in 2^{W\mathbb{Z}}$.

Y is the indexed sum of $(x_0/s)^2, ..., (x_{n-1}/s)^2$.

1: **function** ADD FLOAT TO INDEXED NORM(K, x_n, Y, s)
2: $e = W\lfloor \max(\exp(x_n) - 1, e_{min} + W)/W \rfloor$
3: $t = 2^e$
4: **if** $s < t$ **then**
5: **for** $k = 0$ to $K - 1$ **do**
6: $Y_{kP} = (s/t)^2 Y_{kP}$
7: **end for**
8: $s = t$
9: **end if**
10: DEPOSIT(K, $(x_n/s)^2$, Y)
11: **end function**

Ensure:

$s = 0$ if $\max_{j\in\{0,...,n\}} |x_j| = 0$. Otherwise $2^{-p-W-1}s < \max_{j\in\{0,...,n\}} |x_j| < 2^{W+2}s$ and $s \in 2^{W\mathbb{Z}}$.

Y is the indexed sum of $(x_0/s)^2, ..., (x_n/s)^2$.

A method to add the scaled indexed sum of squares of a vector to a scaled indexed type (using an Algorithm similar to 7.1) is available in ReproBLAS as `idxdBLAS_xixssq` in `idxdBLAS.h`. A method to return the reproducible Euclidean norm of a vector is available as `reproBLAS_rxnrm2` in `reproBLAS.h`. The function `idxd_xixiaddsq` in `idxd.h` can be used to add two scaled indexed sums of squares. An MPI data type that holds a scaled indexed type can be created (creation is only performed once, subsequent calls return the same copy) and returned using the `idxdMPI_DOUBLE_INDEXED_SCALED`, etc. function of `idxdMPI.h`. `idxdMPI.h` also contains the `XIXIADDSQ` method, an MPI reduction operator that can reduce the scaled sums of squares in parallel (see Section 8 for details).

Lemma 7.1. *Let s and Y be the output of Algorithm 7.1, then the updated scaling factor s is either* 0 *or it satisfies*

$$2^{-p-W-1}s \leq \max_{j\in\{0,\ldots,n\}} |x_j| < 2^{W+2}s \tag{7.2}$$

Proof. From line 2 of Algorithm 7.1, it is to see that e is a multiple of W and

$$\begin{aligned} e &\geq W\lfloor (e_{\min}+W)/W \rfloor > e_{\min} \\ e &\leq \max(\exp(x_n)-1, e_{\min}+W) < e_{\max}. \end{aligned}$$

Therefore both $t = 2^e$ and $1/t$ are representable. It also means that if there are no input exceptional values, the scaling factor s is always representable.

The proof is trivial for the case $x_n = 0$. We suppose that $x_n \neq 0$. Hence $x_n \geq 2^{e_{\min}-p}$, or $e_{\min} \leq \exp(x_n)+p$, where p is the precision of input floating-point numbers. Therefore $e \leq \max(\exp(x_n), \exp(x_n)+p+W)$, which means

$$t = 2^e \leq 2^{\exp(x_n)}2^{p+W} < |x_n|2^{p+W+1}. \tag{7.3}$$

Moreover, we have $e \geq W((\exp(x_n)-1)/W - 1) = \exp(x_n) - W - 1$. So

$$s = 2^e \geq 2^{\exp(x_n)}2^{-W-1} > |x_n|2^{-W-2} \tag{7.4}$$

The proof can be deduced by combining the Algorithm's requirement with (7.3) and (7.4). □

Lemma 7.1 means that for non-trivial input values, the maximum absolute value of the scaled inputs will always be kept in range $(2^{-p-W-1}, 2^{W+2})$, which guarantees that there will be no overflow or underflow in computing the sum of squares.

Since the scaling factors in Algorithm 7.1 are always representable and $s \in 2^{W\mathbb{Z}}$, the binning process is not affected by the scaling as well as the rescaling. Therefore the reproducibility of Algorithm 7.1 is guaranteed by the reproducibility of the indexed sum (see Section 5.2). Finally, the Euclidean norm can be simply computed as (using (5.4))

$$s\sqrt{\mathcal{Y}} \tag{7.5}$$

It is worth noting that the same scaling technique can be used to avoid overflow and underflow for the summation, regardless of the index of the partial sum. It is possible to scale input data by a factor in $2^{W\mathbb{Z}}$ so that the maximum absolute value is always in range $[1, 2^W)$. This will help to avoid computing the index of the partial sum at the cost of having to store and communicate the scaling factor along the computation. It is therefore left as a tuning parameter for future work.

7.3 Matrix-Vector Product

The matrix-vector product $y \in \mathbb{R}^m$ of an $m \times n$ matrix A and a vector $x \in \mathbb{R}^n$ where $\alpha, \beta \in \mathbb{R}$ (denoted by $y = \alpha Ax + \beta y$ and computed in the BLAS by `xgemv` [24]) is defined as follows (where $A_{i,j}$ is the entry in the i^{th} row and j^{th} column, indexing from zero for consistency):

$$y_i = \beta y_i + \sum_{j=0}^{n-1} \alpha A_{i,j} x_j$$

It is clear that computation of the matrix vector product is easily distributed among different values of i, as rows of A and copies of x can be distributed among P processors so that entries of y may be calculated independently. However, in the case where n is quite large compared to m/P, it may be necessary to parallelize calculation of the sums of each y_i so that the entirety of x does not need to be communicated to each processor. As the local blocks would then be computed separately and then combined, different blocking patterns or reduction tree shapes could lead to different results. Although the reproducible reduction discussed in Section 7.1 would guarantee reproducibility with respect to reduction tree shape, it would not guarantee reproducibility with respect to blocking pattern. This stronger guarantee can be obtained if each sum y_i is calculated as an indexed sum. It is for this reason that we need a local version of `xgemv` that returns a vector of indexed sums, the matrix-vector product with all sums calculated using indexed summation.

A consequence of calculating the indexed matrix-vector product using indexed sums is that it will be reproducible with respect to permutation of the columns of A together with entries of x. Let $\sigma_0, ..., \sigma_{n-1}$ be some permutation of the first n nonnegative integers such that $\{\sigma_0, ..., \sigma_{n-1}\} = \{0, ..., n-1\}$ as sets. Then we have

$$y_i = \sum_{j=0}^{n-1} A_{i,j} x_j = \sum_{j=0}^{n-1} A_{i,\sigma_j} x_{\sigma_j} \tag{7.6}$$

More importantly, the matrix-vector product should remain reproducible under any reduction tree shape. If $A = [A_{(0)}, A_{(1)}]$ where $A_{(0)}$ and $A_{(1)}$ are submatrices of size $m \times n_{(0)}$ and $m \times n_{(1)}$ and if $x = [x_{(0)}, x_{(1)}]$ where $x_{(0)}$ and $x_{(1)}$ are subvectors of size $n_{(0)}$ and $n_{(1)}$ then we have

$$Ax = A_{(0)} x_{(0)} + A_{(1)} x_{(1)} \tag{7.7}$$

It is clear from Theorems 5.4 and the "Ensure" claim of Algorithm 6.10 that if the matrix-vector product is computed using indexed summation, the result is reproducible.

Computing the matrix-vector product using indexed summation is not difficult given the primitive operations of Section 6. In ReproBLAS, we mirror the function definition of `xgemv` in the BLAS as closely as possible, adding two additional parameters α and β so that the entire operation performs the update $y \leftarrow \alpha Ax + \beta y$ (we also add the standard parameters for transposing the matrix and selecting row-major or column-major ordering for the matrix).

At the core, ReproBLAS provides the function `idxdBLAS_xixgemv` in `idxdBLAS.h` (see Section 8 for details). This version of the function adds to the vector of indexed sums y the previously mentioned indexed matrix vector product of the floating point A and x, where x is first scaled by α as is done in the reference BLAS implementation. To be clear, `idxdBLAS_xixgemv` assumes that y is a vector of indexed types and that all other inputs are floating point. A version (`reproBLAS_rxgemv`) of the matrix vector product routine that assumes y to be a floating point vector is discussed later. It is important to notice that the parameter β is excluded when y is composed of indexed types, as we do not yet know how to scale indexed types by different values (other than 0, 1, or -1) while maintaining reproducibility. β can be included if y is composed of floating point numbers (a case we will discuss below), as they can be scaled before they are converted to an indexed type.

Because the reproducible dot product is so compute-heavy, we can get good performance implementing the matrix-vector product using the `idxdBLAS_diddot` routine at the core. We must use a rectangular blocking strategy to ensure good caching behavior, and because we are making calls to the dot product routine, it is sometimes (depending on the transpose and row-major vs. column-major parameters) necessary to transpose each block of A before computing the dot product. These considerations ensure that the dot product can compute quickly on contiguous sequences of cached data.

Built on top of `idxdBLAS_xixgemv` is the routine `reproBLAS_rxgemv` in `reproBLAS.h` (see Section 8 for details), which takes as input floating point A, x, y, α, and β. y is scaled by β, then converted to a vector of indexed types. The matrix-vector product is computed using indexed summation with a user specified number of accumulators, and the output is then converted back to floating point and returned. The routine `reproBLAS_xgemv` uses the default number of accumulators.

7.4 Matrix-Matrix Product

The $m \times n$ matrix-matrix product C of an $m \times k$ matrix A and a $k \times n$ matrix B where $\alpha, \beta \in \mathbb{R}$ (referred to as $C = AB$ and computed in the BLAS by `xgemm`) is defined as follows (where $A_{i,j}$ is the entry in the i^{th} row and j^{th} column of A, indexing from zero for consistency):

$$C_{i,j} = \beta C_{i,j} + \sum_{l=0}^{k-1} \alpha A_{i,l} B_{l,j}$$

The matrix-matrix product is a similar construction to the matrix-vector product, and discussion of reproducibility proceeds similarly. Computation of the matrix-matrix product can be distributed in a myriad of ways. SUMMA (Scalable Universal Matrix Multiply Algorithm) [28] is used in the PBLAS, the most popular parallel BLAS implementation. In SUMMA and almost all other parallel matrix-matrix algorithms, the computation is broken up into rectangular blocks of A, B, and C. Although SUMMA has a static reduction tree (assuming a fixed number of processors), this cannot be assumed for all parallel matrix multiply algorithms [29] [30] [31], so it may be necessary to use a reproducible reduction step (discussed in Section 7.1). Even if the reduction tree is reproducible for a particular blocking pattern, the blocking pattern may not be reproducible (for instance, if more or fewer processors are used). Therefore, we must compute the matrix-matrix product using indexed summation.

It is for this reason that we need a local version of `xgemm` that returns a matrix of indexed sums, the matrix-matrix product with all sums calculated using indexed summation.

A consequence of calculating the indexed matrix-matrix product using indexed sums is that it will be reproducible with respect to permutation of the columns of A together with rows of B. Let $\sigma_0, ..., \sigma_{k-1}$ be some permutation of the first k nonnegative integers such that $\{\sigma_0, ..., \sigma_{k-1}\} = \{0, ..., n-1\}$ as sets. Then we have

$$C_{i,j} = \sum_{l=0}^{k-1} A_{i,l} B_{l,j} = \sum_{l=0}^{k-1} A_{i,\sigma_l} B_{\sigma_l,j} \tag{7.8}$$

More importantly, the matrix-vector product should remain reproducible under any reduction tree shape. If $A = [A_{(0)}, A_{(1)}]$ where $A_{(0)}$ and $A_{(1)}$ are

submatrices of size $m \times k_{(0)}$ and $m \times k_{(1)}$ and if $B = \begin{bmatrix} B_{(0)} \\ B_{(1)} \end{bmatrix}$ where $B_{(0)}$ and $B_{(1)}$ are submatrices of size $k_{(0)} \times n$ and $k_{(1)} \times n$ then we have

$$AB = A_{(0)}B_{(0)} + A_{(1)}B_{(1)} \tag{7.9}$$

It is clear from Theorems 5.4 and the "Ensure" claim of Algorithm 6.10 that if the matrix-matrix product is computed using indexed summation, the result is reproducible.

Like the matrix-vector product, we can compute the matrix-matrix product using indexed summation with some function calls to `idxdBLAS_xixdot`. In ReproBLAS, we mirror the function definition of `xgemm` in the BLAS as closely as possible, adding two additional parameters α and β so that the entire operation performs the update $C \leftarrow \alpha AB + \beta C$ (we also add the standard parameters for transposing the matrices and selecting row-major or column-major ordering of all matrices).

At the core, ReproBLAS provides the function `idxdBLAS_xixgemm` in `idxdBLAS.h` (see Section 8 for details). This version of the function adds to the matrix of indexed sums C the previously mentioned indexed matrix-matrix product of the floating point A and B, where x is first scaled by α as is done in the reference BLAS implementation. To be clear, `idxdBLAS_xixgemm` assumes that C is a matrix of indexed types and that all other inputs are floating point. A version (`reproBLAS_rxgemm`) of the matrix matrix product routine that assumes C to be a floating point matrix is discussed later. Again the parameter β is excluded when C is composed of indexed types (but not when C is composed of floating point numbers which will be discussed below), as we do not yet know how to scale indexed types by different values (other than 0, 1, or -1) while maintaining reproducibility.

Again because the reproducible dot product is so compute-heavy, we can get good performance implementing the matrix-matrix product using the `idxdBLAS_diddot` routine at the core. The blocking strategy is more complicated this time, however, as computation can proceed under several loop orderings. Because the matrices A and B are composed of single floating point entries and C is composed of the much larger indexed types (each indexed type usually contains at least $6 = 2*K$ of its constituent floating-point types), we chose to completely compute blocks of C by iterating over the matrices A and B. This strategy avoids having to perform multiple iterations over the matrix C composed of much larger data types. Again, to keep the

dot product running smoothly we first transpose blocks of A and/or B (depending on the transpose and row-major or column-major ordering options) when it is necessary to obtain contiguous sequences of cached data.

Built on top of `idxdBLAS_xixgemm` is the routine `reproBLAS_rxgemm` in `reproBLAS.h` (see Section 8 for details), which takes as input floating point A, B, C, α, and β. C is scaled by β, then converted to a vector of indexed types. The matrix-matrix product is computed using indexed summation with a user specified number of accumulators, and the output is then converted back to floating point and returned. The routine `reproBLAS_xgemm` uses the default number of accumulators.

8 ReproBLAS

ReproBLAS is the name given to our library of implementations of algorithms described in this paper. The code is available online at `http://bebop.cs.berkeley.edu/reproblas`. To be useful to the greatest number of performance-conscious scientific software developers, ReproBLAS is written in C (conforming to the C99 Standard [21], with calling conventions that are compatible with the data types of the C89 Standard [17]). The choice of C allows for intrepid Fortran and C++ users to take advantage of the library as well. Code generation and testing utilities are implemented in the more productive language Python [32]. A few distributed memory functions are supplied using MPI [33], the industry standard for distributed memory programming.

We leave the specifics of the library to the documentation included with the library itself, and here offer only a summary of some of the design decisions made in ReproBLAS.

Several of the functions in ReproBLAS are optionally vectorized using Intel AVX or SSE intrinsics [34], depending on what is available on the system. With AVX, vectorization allows for 256-bit registers (4 `double` or 8 `float`) to be operated on in one instruction. Because so many routines were vectorized and due to the complicated nature of the operations, a Python suite was implemented to generate code that is generic to the particular vectorization in question. By simply augmenting this suite of code generation functions, this allows for future modifications of ReproBLAS to use newer intrinsics (such as AVX-512) when they become widely available. Another benefit of code generation is that it allows us to programatically restructure

code to take advantage of loop unrolling and instruction-level parallelism.

To handle the complex build processes involved in code generation without increasing the software requirements of the library, we created a custom build system written entirely in GNU Make. We adopted the build system from a non-recursive makefile template called nonrec-make [35]. The build system handles some of the complexity of code generation with the help of the Python package Cog [36], which allows the programmer to write Python code inline with C code.

Code generation and cache blocking add several parameters to ReproBLAS that must be tuned. OpenTuner [37], an extensible Python autotuner, was used to search the parameter space to find optimal values for ReproBLAS parameters.

ReproBLAS is divided into several modules, each one with a separate header file and static library. `idxd.h` contains the primitive operations discussed in Section 6 and the utility functions regarding the indexed type discussed in Section 5. `idxd.h` also contains several basic functions not mentioned that relate to the core reproducible summation algorithm. `idxdBLAS.h` contains the indexed versions of the BLAS functions discussed in Section 7. These functions are optimized composite operations built from functions in `idxd.h`. `reproBLAS.h` contains versions of functions in `idxdBLAS.h` that do not require the user to use indexed types. Each function has the same name and signature as its BLAS counterpart, and behaves similarly (except with added guarantees of reproducibility). Functions in `reproBLAS.h` with an "r" prepended to their name allow the user to specify the number of accumulators (K, where the internal indexed type used is K-fold) used to compute the result, allowing for a user-specified level of accuracy. `idxdMPI.h` contains MPI data types used to communicate indexed types, and also contains an MPI reduction operator allowing the user to reproducibly reduce the MPI indexed types.

8.1 Timing

In addition to giving a count of floating point operations in our algorithms, we show that they can be implemented as a BLAS library comparable in speed to commercially available optimized non-reproducible versions.

All timings are performed with an Intel®Core i7-2600 CPU operating at 3.4 GHz with 32 KB L1 Cache, 256 KB L2 Cache, and 8192 KB L3 Cache. Every test is run at least 100 times successively to amortize the data loading

time and warm up the cache. The largest BLAS1 problems (sum and dot product) were resident in the L2 cache. The largest BLAS2 and BLAS3 problems exceeded even the L3 cache size. With the intention that these results may be reproduced, we chose to use the widely available open-source compiler `gcc`. The code was compiled with `gcc` version 4.8.4 using the highest level "`-O3`" of optimization (and no other optimization flags). We compare our BLAS routines against the sequential Intel®Math Kernel Library (MKL) BLAS Version 11.0.5 routines [2]. All matrices are represented in column-major order. Denormalized floating point arithmetic is known to be much slower than normalized floating point arithmetic. It should be noted that for all tests, we do not measure with denormal values, as this is the most common case in practice.

The theoretical peak time is calculated as the idealized theoretical time in which the CPU could complete the given operations (in any order). We include the or-bit operation as a FLOP along with multiplication and addition. We multiply the peak processing rate by the number of floating point types that can be processed in a single vectorized instruction (using SSE or AVX intrinsics) [34]. Because instructions of differing type (addition, multiplication, or or-bit operation) can be completed in parallel on our particular CPU, we assume that the peak processing time depends only on the maximum number of instructions of a single type. The theoretical peak time t is therefore calculated according to (8.1).

$a =$ number of additions
$m =$ number of multiplications
$o =$ number of or-bit operations
$f =$ CPU frequency
$v =$ number of floating-point types that fit into largest supported vector register

$$t = \frac{\max(a, m, o)}{vf} \tag{8.1}$$

8.1.1 Difficult Input

Because the reproducible summation algorithm needs to perform additional operations (Algorithm 6.3) if the maximum absolute value of the data increases during summation, the run time depends (up to a constant factor) upon the input. To show the differences in run time, we show in Figure 8.1

the time it takes to reproducibly sum 2^{13} `double`s and `float`s from two different data sets. The first data set is an easy case, the uniform distribution from 0 to 1. The second data set is the most difficult possible case, numbers (alternating in sign to avoid infinities) increasing in absolute value exponentially starting at 1 and ending with the largest positive finite floating point value. We chose to start this increase at 1 rather than the minimum positive floating point value to avoid denormalized floating point values. This data set is difficult because the exponent of the maximum absolute value increases linearly from 0 to its maximum possible value. Therefore, the UPDATE operation (Algorithm 6.3) must be performed more frequently to adjust the index of the indexed type. We can compare this to the uniform distribution in $[0, 1)$, which very quickly achieves a number (greater than 0.5) that has the largest floating point exponent possible from the distribution. After such a number is seen, no more updates need to be performed. Note that these problems were resident in the L2 cache. We calculate the peak performance of `rxsum` using only the core operations in the DEPOSITRESTRICTED operation (Algorithm 6.4). `rdsum` achieved performance of $1.20 \cdot 10^{10}$ FLOPS (61.9% of peak) on the difficult dataset. `rssum` achieved performance of $1.91 \cdot 10^{10}$ FLOPS (49.3% of peak) on the difficult dataset. It is clear from the figure that the additional cost of the UPDATE operation is modest, and the performance of the summation algorithm has only a weak dependence on the data.

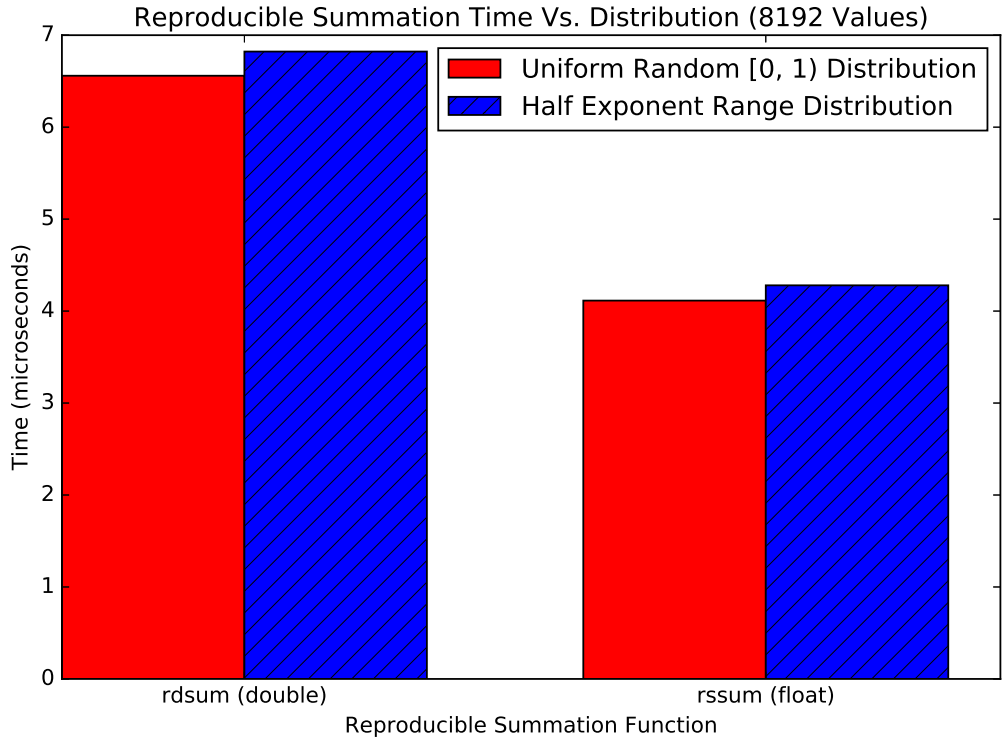

Figure 8.1: Time taken to sum 2^{13} floating point numbers of varying difficulty.

8.1.2 BLAS1

We show how our reproducible summation compares to a simple C `for`-loop in Figure 8.2. We compiled the `for`-loop using the command "`gcc -O3`". The `for`-loop is as follows (where `X` is the input vector and `res` is the resulting sum):

```
res = 0;
for(j = 0; j < N; j++){
   res += X[j];
}
```

Time (measured relative to the peak theoretical time for recursive summation) is shown for each method. The double-precision floating point num-

bers to be summed are normally distributed with mean 0 and variance 1. The values of N shown are powers of two from 2^6 to 2^{12}. All of the problems shown on this plot were resident in L1 cache. The peak performance of `rdsum` is $1.94 \cdot 10^{10}$ FLOPS. The peak performance of the `for`-loop is $1.36 \cdot 10^{10}$ FLOPS. The peak performance of the `for`-loop is 7 times that of `rdsum`, as the `for`-loop requires 1 addition per element, but `rdsum` requires $3K - 2 = 7$ additions and $K = 3$ or-bit operations per element which can be done in parallel. This also explains why the theoretical peak time required for `rdsum` is 7 times larger than the `for`-loop. Compiling with the `gcc` flag `-fopt-info-vec-optimized` tells us that the `for`-loop is not vectorized by the compiler, and therefore not running at peak. With this in mind, reproducible summation is competitive with a simple `for`-loop.

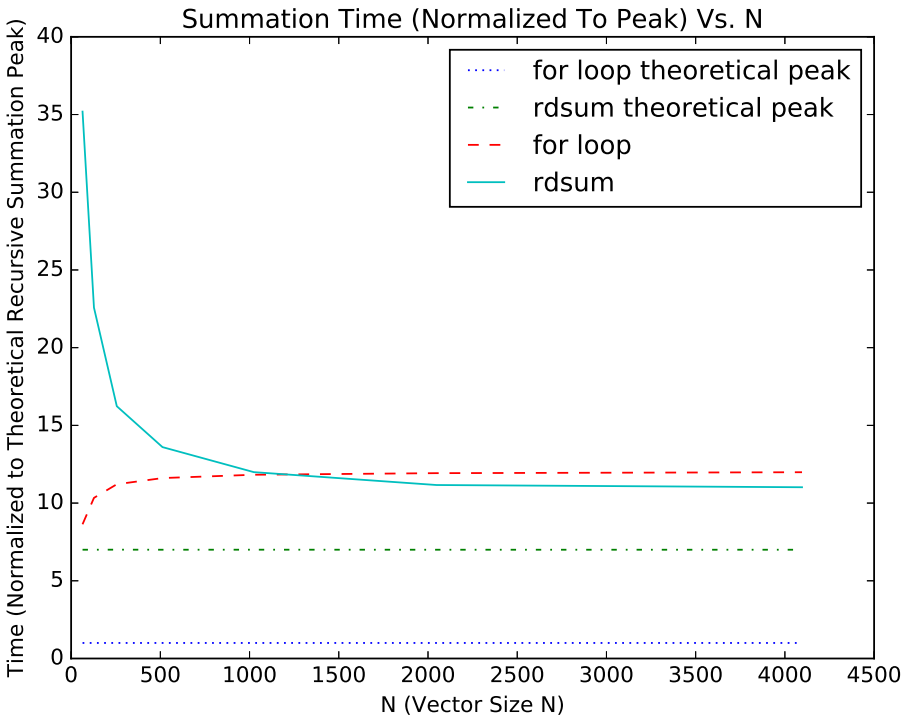

Figure 8.2: Relative floating point summation time

To give a comparison to a BLAS1 function, we show in Figure 8.3 timings of the reproducible dot product versus the MKL BLAS dot product. Again, the double-precision floating point data is distributed such that the elements of each input vector are normally distributed with mean 0 and variance 1. The values of N shown are powers of two from 2^6 to 2^{12}. All problems shown here were L1 resident except for the largest problem, which was resident in the L2 cache. We calculate peak performance of `rddot` using only the core operations in the DepositRestricted operation (Algorithm 6.4) and the additional pointwise multiplication. The peak performance of `rddot` is $2.14 \cdot 10^{10}$ FLOPS. The peak performance of `ddot` is $2.72 \cdot 10^{10}$ FLOPS. The peak performance of `ddot` is 1.27 times that of `rddot`. This is because `ddot` requires 1 addition and 1 multiplication per element, while `rddot` requires $3K - 2 = 7$ additions, 1 multiplication, and $K = 3$ or-bit operations per element. This also explains why the theoretical peak time required for `rddot` is 7 times larger than that of `ddot`. For vectors of size 2^{10} and larger, the slowdown of `rddot` as compared to `ddot` ranges from 4.15 to 3.33.

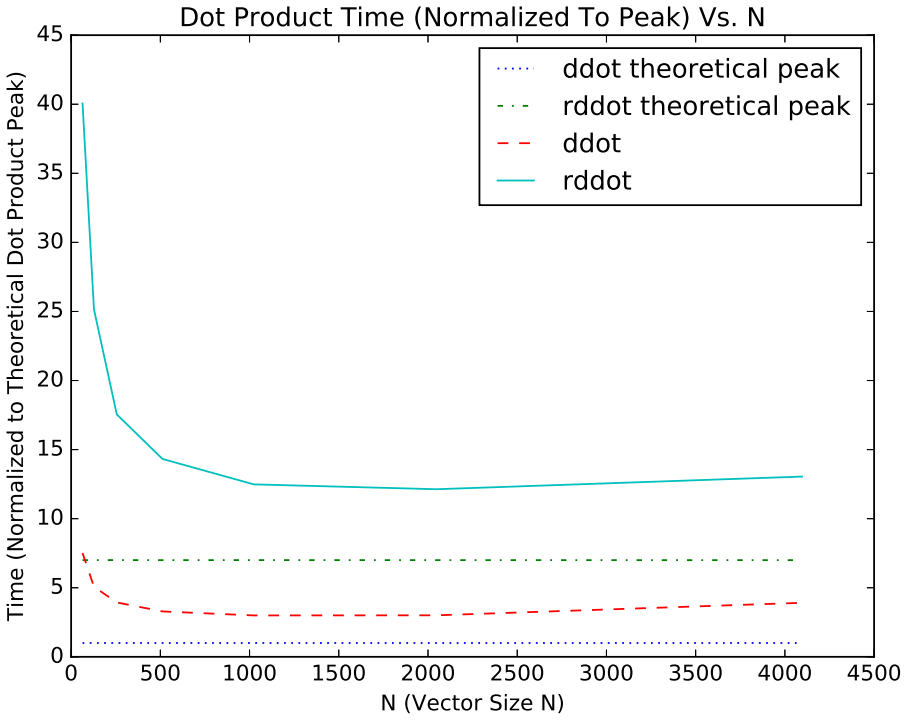

Figure 8.3: Relative dot product time

8.1.3 BLAS2

We show in Figure 8.4 timings of the reproducible matrix-vector product versus the MKL BLAS matrix-vector product. Each double-precision floating point input matrix or vector element is normally distributed with mean 0 and variance 1. The values of N shown are powers of two from 2^6 to 2^{12}. Problem size 2^6 was resident in L1 cache, problem size 2^7 was resident in L2 cache, problem sizes 2^8 to 2^{10} were resident in L3 cache, and problem sizes 2^{11} and 2^{12} exceeded the L3 cache size. The peak performances of `rdgemv` and `dgemv` are the same as that of `rddot` and `ddot`. For problem sizes 2^{10} and larger, the slowdown of `rdgemv` as compared to `dgemv` ranges from 7.70 to 5.71.

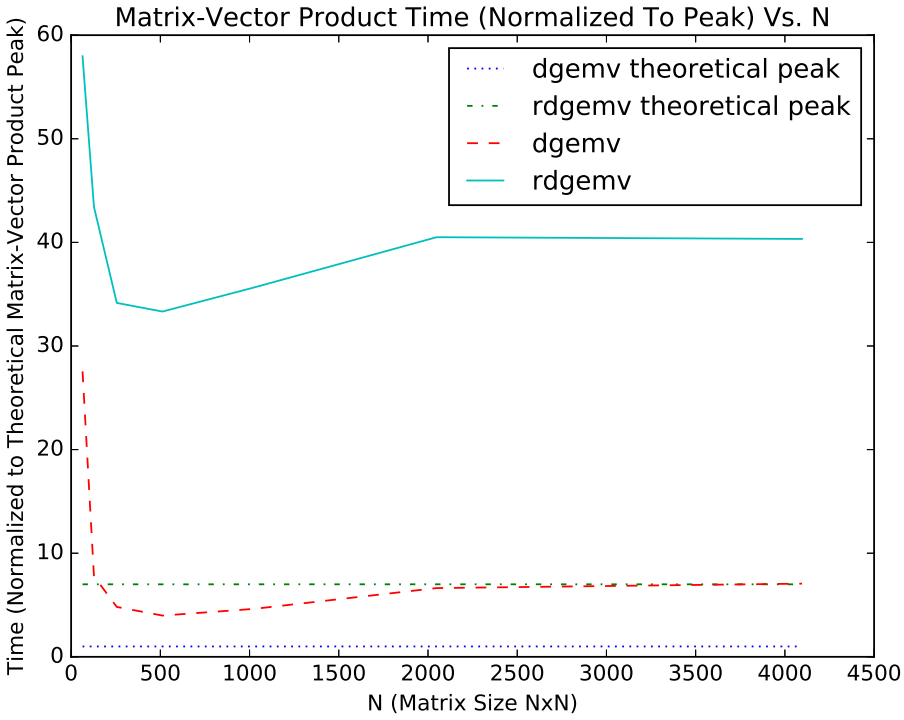

Figure 8.4: Relative matrix-vector product time

The non-transposed matrix-vector product is the only case where composition of BLAS2 and BLAS3 functions did not achieve results within a factor of 2 of theoretical peak. Because the reproducible dot product operates most efficiently on contiguous sections of memory, the matrix must be transposed so that memory can be read contiguously. This causes the reproducible matrix-vector product to perform poorly due to the extra cost of matrix transposition in an already memory-bound routine. In future versions of the library, this method could be improved by writing a custom inner-loop kernel that does not make calls to `xixdot`. The kernel would operate on the primary fields of several indexed types at the same time using vectorized operations.

The transposed matrix-vector product performs better than the non-transposed case. Timings are shown in Figure 8.5. The reader should notice that in this case, the reproducible routine is only a factor of 2.43 times slower

than the optimized BLAS routine.

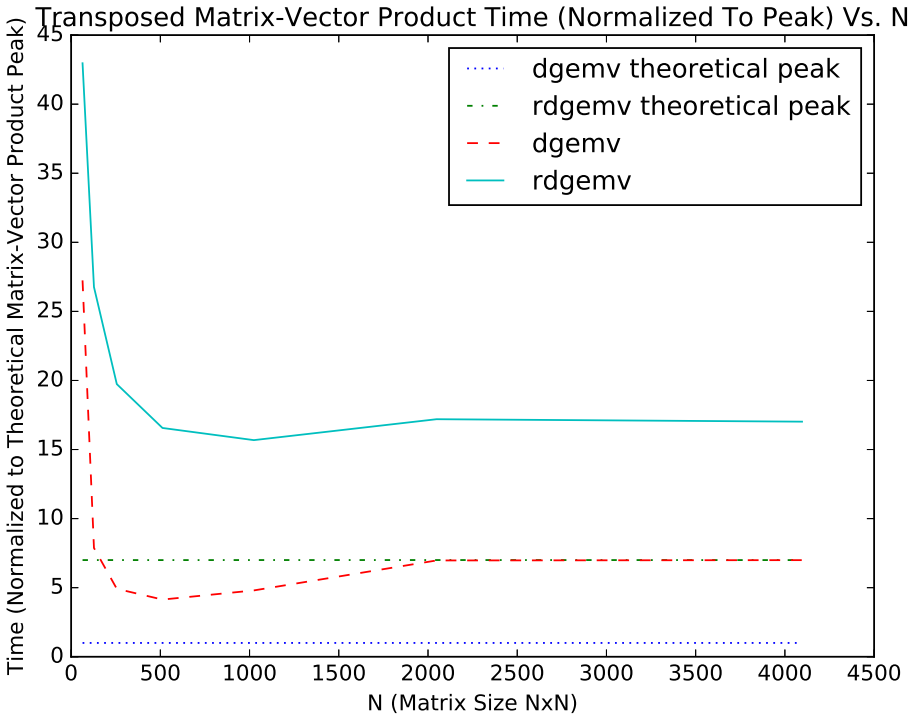

Figure 8.5: Relative transposed matrix-vector product time

8.1.4 BLAS3

We show in Figure 8.6 timings of the reproducible matrix-matrix product versus the MKL BLAS matrix-matrix product. Because the timings for each transposition case (transposing or not transposing A or B) are similar, we show only the standard case for brevity. Each double-precision floating point input matrix element is normally distributed with mean 0 and variance 1. The values of N shown are powers of two from 2^6 to 2^{12}. Problem size 2^6 was resident in L2 cache, problem sizes 2^7 to 2^9 were resident in L3 cache and problem sizes 2^{10} to 2^{12} exceeded the L3 cache size. The peak performances of `rdgemm` and `dgemm` are the same as that of `rddot` and `ddot`. In this case, since `dgemm` is running close to peak and `rdgemm` is running at 47.1% of peak,

the reproducible routine is a factor of 12.6 times slower than the optimized BLAS routine.

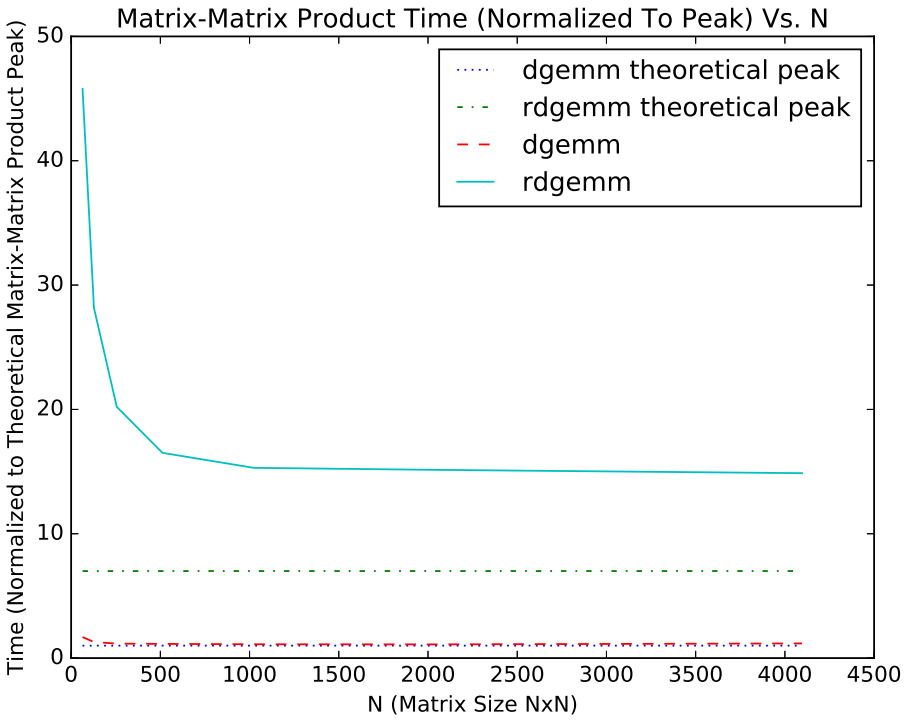

Figure 8.6: Relative matrix-matrix product time

8.2 Testing

In understanding the testing methodology behind ReproBLAS, is important to distinguish between the metrics of reproducibility and accuracy. Although high accuracy can sometimes result in reproducibility, it is not a guarantee. For this reason, we test the accuracy and the reproducibility of the ReproBLAS methods separately.

Testing in ReproBLAS starts with the BLAS1 methods (`xsum`, `xasum`, `xnrm2`, and `xdot`). These methods are first checked to see that their results are accurate, then checked to see if the results are reproducible. All tests are repeated several times with parameters changed. The data is scaled by 1 and

-1. The data is permuted by reversing, sorting (in ascending or descending order of value or absolute value), or random shuffling. To check that the result is independent of blocking, the data is grouped into several blocks and each block is operated on separately, then the results for each block are combined using the appropriate function. Several different block sizes are tested for each permutation of the data.

The accuracy of the reproducible BLAS1 methods is validated by checking to see if the results of each method are within the error margin specified by (6.27) from the true results. Because the true summation result must be known, these tests are performed on distributions with known sums. To ensure that our algorithms can handle sums with large amounts of cancellation, we tested on the XBLAS dot product test generator [38]. Because XBLAS can perform floating point multiplication exactly, we split the floating point outputs of the XBLAS test generator into two floating point numbers each with half of the original mantissa bits. This ensures that the multiplications occur exactly and the only error seen in the reproducible routine comes from summation. The input vectors used are shown in Table 4. Note that random() is a floating point number uniformly distributed between 0 and 1.

Because the reproducible summation methods operate on bins, we repeat the accuracy tests on scaled versions of the input data set W times, each time increasing the scale by a factor of two. This allows us to test all splits of the sum across bin boundaries. This is performed on data very close to overflow to test cases where the data is split between bin 0 and bin 1. This is also performed on data very close to underflow to test cases where some data is lost due to underflow.

To validate the accuracy of the methods in the presence of `Inf`, `-Inf`, and `NaN`, the reproducible BLAS1 methods are tested on the vectors (scaled by 1 and -1) shown in Table 5.

After validating the accuracy of the reproducible BLAS1 methods, their reproducibility is verified. Each method is checked to see if it produces the same result under the list of permutations given above (reversing, sorting, or random shuffling). Because we do not need to know the true sum of the data, we test using a uniform random and normal random distribution (in addition to those mentioned in Tables 4 and 5).

Once the BLAS1 methods are tested, the results of BLAS2 and BLAS3 (`xgemv` and `xgemm`) methods are tested against "reference" reproducible versions. These reference versions are simply naive implementations of `xgemv` and `xgemm` in terms of the reproducible dot product. They use no blocking

Name	Value	Sum
sine	$X_j = \begin{cases} \sin(2\pi j/n) \text{ if } j < \lfloor n/2 \rfloor \\ 0 \text{ if } j = (n-1)/2 \\ -X_{n-j-1} \text{ otherwise} \end{cases}$	0
mountain	$X_j = \begin{cases} \text{random() if } n = 0 \\ X_{j-1} * 2^{-\lfloor p/2 \rfloor - 1} \text{ if } j < \lfloor n/2 \rfloor \\ 0 \text{ if } j = (n-1)/2 \\ -X_{n-j-1} \text{ otherwise} \end{cases}$	0
$small + big$	$X_j = \begin{cases} 2^{\lfloor p/2 \rfloor + 1} \text{ if } j = 0 \\ 2^{-\lfloor p/2 \rfloor - 1} \text{ otherwise} \end{cases}$	$2^{\lfloor p/2 \rfloor + 1} + (n-1)2^{-\lfloor p/2 \rfloor - 1}$
$small + big + big$	$X_j = \begin{cases} 2^{\lfloor p/2 \rfloor + 1} \text{ if } j = 0 \text{ or } j = n-1 \\ 2^{-\lfloor p/2 \rfloor - 1} \text{ otherwise} \end{cases}$	$2^{\lfloor p/2 \rfloor + 2} + (n-2)2^{-\lfloor p/2 \rfloor - 1}$
$small + big - big$	$X_j = \begin{cases} 2^{\lfloor p/2 \rfloor + 1} \text{ if } j = 0 \\ -2^{\lfloor p/2 \rfloor + 1} \text{ if } j = n-1 \\ 2^{-\lfloor p/2 \rfloor - 1} \text{ otherwise} \end{cases}$	$(n-2)2^{-\lfloor p/2 \rfloor - 1}$
`XBLAS`	We used the `BLAS_xdot_testgen` routines to generate test data, and then split the floating point numbers (as in [20]) such that the true sum of the pairwise floating point products of the new vectors would be equal to the true dot product of the original vectors [38].	

Table 4: ReproBLAS Accuracy Validation Vectors $X \in \mathbb{F}^n$

Name	Value	Sum
`Inf`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \\ 0 \text{ otherwise} \end{cases}$	`Inf`
`Inf` + `Inf`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \text{ or } j = n-1 \\ 0 \text{ otherwise} \end{cases}$	`Inf`
`Inf` − `Inf`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \\ -\texttt{Inf} \text{ if } j = n-1 \\ 0 \text{ otherwise} \end{cases}$	`NaN`
`NaN`	$X_j = \begin{cases} \texttt{NaN} \text{ if } j = 0 \\ 0 \text{ otherwise} \end{cases}$	`NaN`
`Inf` + `NaN`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \\ \texttt{NaN} \text{ if } j = n-1 \\ 0 \text{ otherwise} \end{cases}$	`NaN`
`Inf` + `NaN` + `Inf`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \\ \texttt{NaN} \text{ if } j = \lfloor n/2 \rfloor \\ \texttt{Inf} \text{ if } j = n-1 \\ 0 \text{ otherwise} \end{cases}$	`NaN`
`Inf` + `NaN` − `Inf`	$X_j = \begin{cases} \texttt{Inf} \text{ if } j = 0 \\ \texttt{NaN} \text{ if } j = \lfloor n/2 \rfloor \\ -\texttt{Inf} \text{ if } j = n-1 \\ 0 \text{ otherwise} \end{cases}$	`NaN`

Table 5: ReproBLAS Exception Validation Vectors $X \in \mathbb{F}^n$

and are simple to understand and code. Because `xgemv` and `xgemm` compute vectors and matrices of dot products, we can use the same data that was used for the dot product. For `xgemv`, we permute the columns of the input matrix together with the entries of the input vector as shown in (7.6). We break the computation into column-blocks of various sizes as shown in (7.7) and combine them to ensure that the operation is reproducible with respect to blocking of computation. For `xgemm`, we permute the columns of one input matrix together with the rows the other input matrix as shown in (7.8). We break the computation into column-blocks and row-blocks of various sizes as shown in (7.9) and combine them to ensure that the operation is reproducible with respect to blocking of computation.

Each test described above must be performed with different values for several parameters, including K (where the indexed types used are K-fold) (we tested $K = 2, 3, 4$), the increment between elements of a vector, row or column major ordering of matrices, whether or not to transpose matrices, and scaling factors on vectors and matrices. To handle this software complexity, a Python test harness was created to exhaustively test each combination of values for these parameters.

9 Conclusions And Future Work

The algorithms we have presented have been shown to sum binary IEEE 754-2008 floating point numbers accurately and reproducibly. The algorithms behave sensibly in overflow and underflow situations and work on exceptional cases such as `Inf`, `-Inf`, and `NaN`. Reproducibility is independent of the ordering of data, blocking schemes, or reduction tree shape. Our algorithms remain reproducible on heterogeneous systems independent of the number of processors assuming only a subset of IEEE Standard 754-2008 [16].

We have specified all of the necessary steps to carry out reproducible summation in practice, including a conversion from the intermediate indexed type to a floating point result. Our method allows for user-specified accuracy in the result, improving significantly on the accuracy of [5]. We have also specified methods for reproducible absolute sums, dot products, norms, matrix-vector products, and matrix-matrix products. We have created an optimized library for reproducible summation, tested it, and shown that the performance is comparable to industry standards. We have included in our library methods for distributed memory reductions so that reproducible par-

allel routines may be built from our library.

Allowing the user to adjust accuracy yields an interesting tradeoff between performance and accuracy. Using only the existing ReproBLAS interface, a basic long accumulator [7] can be built by setting K to its maximum value so that the entire exponent range is summed. A careful examination of the error bound (6.25) shows that this would give almost exact results regardless of the dynamic range of the sum. However, because ReproBLAS was not designed with such a use case in mind, this would be very slow. Future work could also involve optimizing some of the routines for a high-accuracy use case. In such a scheme, when a number is deposited (as in Algorithm 6.5), it would only be added to the 3 bins its slices fit in. Only accumulators with nonzero values would need to be stored, allowing for another possible optimization.

In the future, we plan to add PBLAS routines to our library so that users may benefit from tested reproducible parallel BLAS routines without having to code them themselves. We will extend the existing interface to ReproBLAS with parallel BLAS function signatures.

10 Acknowledgments

This research is supported in part by NSF grants NSF ACI-1339676, DOE grants DOE DE-SC0010200, DOE DE-SC0008699, DOE DE-SC0008700, DOE AC02-05CH11231, and DARPA grant HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and Samsung. Other industrial sponsors include Mathworks and Cray. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] SC'15 BoF on "Reproducibility of high performance codes and simulations: tools, techniques, debugging", 2015. `https://gcl.cis.udel.edu/sc15bof.php`.

[2] Intel®. Intel® Math Kernel Library reference manual. Technical report, 630813-051US, 2011. `http://software.intel.com/sites/`

products/documentation/hpc/mkl/mklman/mklman.pdf.

[3] NVIDIA®. NVIDIA® cuBLAS. 2016. http://docs.nvidia.com/cuda/cublas.

[4] A. Arteaga, O. Fuhrer, and T. Hoefler. Designing bit-reproducible portable high-performance applications. In *IEEE Intern. Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE, 2014.

[5] J. Demmel and Hong Diep Nguyen. Parallel reproducible summation. *Computers, IEEE Transactions on*, 64(7):2060–2070, July 2015.

[6] Sharan Chetlur. private communication.

[7] U. Kulisch. *Computer Arithmetic and Validity - Theory, Implementation and Applications*. de Gruyter, 2nd edition, 2013.

[8] S. Collange, D. Defour, Graillat S., and R. Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015.

[9] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat. ExBLAS: Reproducible and Accurate BLAS Library. hal.archives-ouvertes.fr, 2015.

[10] S. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comp.*, 31(5):3466–3502, 2009.

[11] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, 1993.

[12] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[13] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Mat. Anal. Appl.*, 32(3):866–901, 2011.

[14] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[15] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays - part 1. Tech Report UCB/EECS-2013-61, UC Berkeley Computer Science Division, May 2013.

[16] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[17] *ANSI/ISO 9899-1990 American National Standard for Programming Languages - C*. 1990.

[18] D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 1969.

[19] T. Dekker. A floating point technique for extending the available precision. *Num. Math.*, 18:224–242, 1971.

[20] Yozo Hida, Xiaoye Li, and David Bailey. Algorithms for quad-double precision floating point arithmetic. Lawrence Berkeley National Laboratory, 2000.

[21] ISO Standard. C99-ISO, 1999.

[22] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, July 1993.

[23] J. Demmel and Y. Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comp.*, 25(4):1214–1248, 2003.

[24] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[25] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[26] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 107–114. Springer, 1996.

[27] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK Users' guide*, volume 4. SIAM, 1997.

[28] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[29] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. Technical Report UCB/EECS-2011-72, EECS Department, University of California, Berkeley, Jun 2011.

[30] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *Euro-Par 2011 Parallel Processing*, pages 90–109. Springer, 2011.

[31] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Ofer Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 261–272. IEEE, 2013.

[32] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.

[33] David W Walker and Jack J Dongarra. MPI: A Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[34] Intel Intel. Advanced vector extensions programming reference. *Intel Corporation*, 2011.

[35] nonrec-make - Non-recursive make template for GNU make. `http://github.com/aostruszka/nonrec-make`.

[36] Cog - Generate code with inlined Python code. `http://nedbatchelder.com/code/cog/`.

[37] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[38] XS Li, JW Demmel, DH Bailey, Y Hida, J Iskandar, A Kapur, MC Martin, B Thompson, T Tung, and DJ Yoo. Xblas–extra precise basic linear algebra subroutines.