Augmented Arithmetic Operations Proposed for IEEE-754 2018

Jason Riedy
Georgia Institute of Technology
jason.riedy@cc.gatech.edu

James Demmel
University of California, Berkeley
demmel@berkeley.edu

***Abstract*—Algorithms for extending arithmetic precision through compensated summation or arithmetics like double-double rely on operations commonly called twoSum and twoProduct. The current draft of the IEEE 754 standard specifies these operations under the names augmentedAddition and augmentedMultiplication. These operations were included after three decades of experience because of a motivating new use: bitwise reproducible arithmetic. Standardizing the operations provides a hardware acceleration target that can provide at least a 33% speed improvements in reproducible dot product, placing reproducible dot product almost within a factor of two of common dot product. This paper provides history and motivation for standardizing these operations. We also define the operations, explain the rationale for all the specific choices, and provide parameterized test cases for new boundary behaviors.**

I. Introduction and Motivation

The IEEE 754 Standards Committee for Floating Point Arithmetic is considering adding three new recommended binary floating point operations in the upcoming 2018 version of the standard. The goal of this paper is to describe these operations, and why they are being considered now, when very similar instructions were considered but not included in the last 2008 standard[1]. The first new operation is a variation of the well-known twoSum[2] operation, which takes two floating point summands x and y, and returns both their rounded sum $h = \text{round}(x+y)$, and the exact error $t = x+y-h$ (exception handling is discussed later). The letters h and t are chosen to stand for head (the leading bits of the sum) and tail (the trailing bits). For the error t to be exactly representable, the initial rounding must be to-nearest, with any tie-breaking rule and with gradual underflow. The many existing and widely used software implementations of this operation, whose goals are to simulate higher precision arithmetic efficiently, have used the standard tie-breaking rule (to nearest even), including double-double[3], quad-double[4] and compensated summation[5].

The reason for considering this operation again (as well as two analogous operations implementing subtraction and multiplication of x and y to get h), is that using a different tie-breaking rule can both preserve and accelerate all the existing uses of this operation as well as one new one: reproducible summation. Reproducible summation means getting bitwise identical sums, no matter the order in which summands appear. If ties are broken in the new operation using the right rule, then one can perform reproducible summation very efficiently by doing one read-only pass over the summands in any order. The sum is represented by a *reproducible accumulator* consisting of just a few floating point numbers (six floating point numbers are recommended to attain the usual precision or better, in double). Reproducible summation is of particular interest, for correctness and debugging reasons, on common parallel platforms where summations are not performed in a deterministic (reproducible) order. Application studies for exascale computing identify reproducibility as necessary for important applications[6].

After much consideration and input from various users, the committee chose the tie-breaking rule of rounding h toward zero (roundTiesToZero in the draft standard's parlance). This is not the only rule that would work, but it is the simplest to explain and implement. The actual reproducible summation algorithm requires quite a long proof[7], so here we will just summarize the necessary results from this work.

The new operations are named augmentedAddition, augmentedSubtraction, and augmentedMultiplication to highlight differences from existing implementations of twoSum and the analogous routines. Section II presents some of the history behind these operations along with uses in extending arithmetic precision and implementing reproducible summation. The operations are defined from a high level in Section III. We rephrase the text in the draft standard to avoid *standardese*. Note that these operations are recommended only for binary floating-point; requirements for decimal are not yet known. Reasons for the specific choices about rounding direction, exceptional behavior, and treatment of signed zeros are in Section IV. Potential performance improvements from implementing the augmented arithmetic operations as two hardware instructions range from 33% faster reproducible dot products to 2× faster double-double matrix multiplication, shown in Section V. New operations need new test cases, and we provide test cases for the new boundary behaviors in Section VI.

II. History and Related Work

These operations have a long history, and we highlight some pieces significant to the proposed operations.

A. Extending Precision

Møller[2] developed what became known as twoSum to implement *quasi double-precision* and deliver accurate results on machines that truncated rather than rounded. Kahan[5] essentially developed fastTwoSum for *compensated summation* also to alleviate truncation errors, and Dekker[8] developed

```
void
twoSum (const double x, const double y,
          double * head, double * tail)
{
   const double s = x + y;
   const double bb = s − x;
   const double err = (x − (s − bb)) + (y − bb);
   *head = s;
   *tail = err;
}

void
fastTwoSum (const double x, const double y,
             double * head, double * tail)
/* Assumes that |x| <= |y| */
{
   const double s = x + y;
   const double bb = s − x;
   const double err = y − bb;
   *head = s;
   *tail = err;
}

void
twoProduct (const double a, const double b,
             double *head, double *tail)
{
      double p = a * b;
      double t = fma (a, b, −p);
      *head = p;
      *tail = t;
}
```

Fig. 1. A "typical" C implementation of twoSum, fastTwoSum, and twoProduct. Algebraic manipulations can alter exceptional behavior and the signs of resulting zeros.

fastTwoSum to generate floating-point expansions that extend precision. Figure 1 provides C implementations of the functions. Both Kahan's and Dekker's papers cite Wolfe[9] as a more complex method that possibly is the earliest attempt. Wolfe uses binning instead of extending precision. Amusingly binning returns when we consider reproducible summation below. Higham[10], [11] provides a more complete history and a numerical analysis of different forms of compensated summation. Møller's quasi double-precision later was implemented as *double-double* arithmetic[3] and extended to *quad-double* arithmetic[4]. A double-double number consists of a pair of floating-point numbers, a *head* h and a *tail* t. The numerical value of the double-double is the exact, unevaluated sum $h+t$. Additionally, h and t do not overlap, that is $|t| \leq \text{ulp}(h)/2$ where $\text{ulp}(h)$ is the value of the last bit in h's binary representation.

These techniques found use not only in extending precision but also in improving reproducibility in large-scale parallel applications[12] and accelerating linear algebra algorithms when narrower precisions are drastically faster[13]. The proposed operations implement *error-free transformations*[14] that are the base of fast accurate summation[15] and relatively high performance extra-precise matrix and linear algebra[16]–[19].

With use came optimization. Double-double multiplication is accelerated drastically by the fusedMultiplyAdd (FMA) operation[20]. Without FMA, producing a double-double by multiplying two double operands x and y must check for overflow, possibly scale, split significands by multiplying with a constant, and accumulate the cross-multiplied pieces using fastTwoSum, up to nine operations not counting the conditional branch. FMA replaces all of this by computing $h = x \times y$ and $t = \text{FMA}(x, y, -h)$ (see twoProduct in Figure 1), which is an exact transformation so long as the result neither overflows nor has bits below the representable region. As we will see in Section IV-B, however, the exceptional values that may occur with overflow or invalid are slightly surprising. The proposed operations provide a similar optimization for addition and simplify exceptional behavior for both.

B. Reproducibility

Double-double arithmetic can improve reproducibility in parallel codes[12] but does not address the bitwise reproducibility desired in sensitive simulations[6], [21], [22]. Numerical causes of non-reproducibility on a single parallel platform include optimized parallel reductions for convergence tests receiving operands in different orders on different processors. Another is when accumulating sufficient statistics from streaming data that may arrive out of order. In these cases only a single "pass" over the data is possible. Non-reproducibility between platforms can occur from subtle implementation differences in mathematical libraries as well as differences in parallel resources. Many efforts to address reproducibility in linear algebra[7], [23]–[25] rely on the proposed error-free transformations. Many vendor libraries provide various levels of reproducibility even without support from standards, detailed in Section II-C.

We focus on one example to illustrate use of augmentedAddition, the ReproBLAS[7]. The ReproBLAS's explicit design goals are 1) to perform bitwise-reproducible floating point summation, independent of the order of summation (or shape of a reduction tree), including *handling exceptions reproducibly*; 2) to be as least as accurate as conventional summation (with tunable precision); 3) to perform just one read-only pass over the data, and/or one parallel reduction operation; 4) to use as little memory as possible, to enable high performance by tiling higher level operations like the BLAS. We sketch the core algorithm for reproducible summation; details are in [7], [23]. We take the range of floating point exponents, and divide them into fixed intervals of equal width (say 40 consecutive integers in double); we call each interval a bin. Then, for each summand (in any order) we rewrite it as the exact sum of a small number of slices, where each slice corresponds to the significant bits lying (roughly) in a bin.

This is where augmentedAddition is used: one use of augmentedAddition is enough to extract one slice and leave the exact remainder. We can then sum all the slices corresponding to the same bin exactly (and so reproducibly), because we are implicitly doing fixed point arithmetic. But we do not need to sum the slices in all the bins, only the bins corresponding to the largest few exponent ranges. The number of bins summed can be chosen based on the desired precision; in double precision the default is 3 bins, where each bin is represented by 2 doubles, so 6 doubles altogether are needed to compute a reproducible sum. Slices lying in bins with smaller exponents are discarded, or not computed in the first place. Independent of the order of summation, or parallel reduction order, we end up with the same sums of slices in the same bins, all computed exactly and so reproducibly, which we finally convert to a standard floating point number. This requires roughly $7n$ conventional floating point operations for n operands using operations in the IEEE 754-2008 standard. If augmentedAddition is implemented as two separate instructions, one for the head and one for the tail[26], this falls to roughly $4n$ instructions. When both instructions are issued simultaneously, this would have the latency of $3n$ instructions. Section V demonstrates a hardware-accelerated augmentedAddition operation's potential performance improvement for reproducible, 64-bit dot product.

C. Alternatives

There are other methods for extending precision and providing reproducible summations and dot products. Arbitrary precision software arithmetic[27], [28] greatly increases execution times but has multiple scientific applications[29]. Wide accumulators[30], [31] provide sufficient internal precision in hardware to produce correctly rounded results. These accumulators do not compose in a parallel setting unless all internal precision can be communicated between processes. Performance suffers as well, although combining error-free transformations and wide accumulators can alleviate some of the performance penalty[24]. Another method applies *high-precision anchored (HPA) accumulators*[32], a wider precision reminiscent of fixed point arithmetic but with a programmable anchor. While very efficient in hardware, the anchored accumulator requires either the programmer to specify the scaling factor or a scan across all operands to compute that factor. Correctly rounded results are another path towards reproducibility but require more computation and memory traffic[24], [33], [34]. Many commercial numerical libraries including Intel's MKL, NVIDIA's cuBLAS, and NAG's libraries provide more restricted forms of reproducibility, less strict than bitwise reproducibility, to ease debugging.

III. Proposed Augmented Arithmetic Operations

At the time of writing, the draft IEEE 754-2018 standard includes three recommended (not required) operations that standardize the desired behaviors of twoSum and twoProduct. The three operations are named augmentedAddition, augmentedSubtraction, and augmentedMultiplication. augmentedSubtraction(x,y) is equivalent to augmentedAddition$(x,-y)$ and will not be discussed further. Section IV provides reasons for specific standardization decisions. This section provides an overview of the specifications. The text itself is too dependent on the standard's definitions for direct inclusion, but drafts can be found through the supporting web site[1]. The operations currently are defined in Clause 9.5 of the drafts.

[1]http://754r.ucbtest.org

The operations rely on a new rounding direction, roundTiesToZero, for reasons explained in Section IV-A. The rounding direction is required for these operations and is independent of other rounding attributes. This rounding direction delivers the floating-point number nearest to the infinitely precise result. If the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with smaller magnitude shall be delivered. roundTiesToZero carries all overflows to ∞ with the sign of the intermediate result.

Let $x+y$, $x\times y$, *etc.* be the infinitely precise result of the mathematical expression. In the following, we refer to the first returned result as the head h and the second as the tail t.

augmentedAddition delivers $h = \text{roundTiesToZero}(x+y)$ and $t = x + y - \text{roundTiesToZero}(x+y)$ when $\text{roundTiesToZero}(x+y)$ is a finite, non-zero floating-point number. In this case, h is the rounded sum, and t is the exactly representable error in that sum. If $\text{roundTiesToZero}(x+y)$ is zero, then both the head and tail have the same sign; $h = t = \text{roundTiesToZero}(x+y)$. Similarly, if $\text{roundTiesToZero}(x+y)$ overflows, both the head and tail are set to the same infinity, and in this case the operation signals inexact and overflow. If either operand is a NaN, augmentedAddition produces the same quiet NaN for both h and t. Ideally, this NaN would be one of the input NaNs or a quiet version of an input signaling NaN. If $x+y$ is invalid, that is x and y are opposite-signed infinities, then both h and t are the same NaN, and this operation signals invalid. This operation signals inexact *only* when $\text{roundTiesToZero}(x+y)$ overflows; underflows and zeros are *exact*. Table II summarizes the proposed exceptional and signed zero behavior.

augmentedMultiplication is similar to augmentedAddition but can be inexact without overflow. If some non-zero digits of the result, head or tail, lie strictly between $\pm 2^{emin-p+1}$, where *emin* is the minimum exponent and p is the bits of precision, then those bits underflow to zero. Otherwise, $h = \text{roundTiesToZero}(x\times y)$ and $t = x\times y - \text{roundTiesToZero}(x\times y)$ is exact for finite, non-zero floating-point numbers. Results for invalid operations and NaN propagation are analogous to those of augmentedAddition; see Table III. Operations on $\pm\infty$ and ± 0 produce head and tail with the same sign, the product of the signs of the operands.

Underflow signaling in augmentedMultiplication is subject to the before- or after-rounding detection of tiny, non-zero results. But any time augmentedMultiplication does not return a pair h, t such that $x\times y = h+t$, the operation signals inexact.

One important note is that the standard only specifies operations and not implementations. These operations can be implemented in software, one hardware instruction, or multiple hardware instructions. Section V's performance results assume two instructions for each operation as in [26]. One instruction produces the head h, and one instruction produces the tail t. Combined, the instructions can implement the augmented arithmetic operations.

Another note is that these operations do not make sense with some alternate exception handling attributes. Abrupt underflow, for example, breaks the exact transformation property of augmentedAddition.

IV. Rationale

The operations in Section III differ from various existing implementations of twoSum and twoProduct, examples of which are in Figure 1. This section explains the differences. Ultimately, the standard operations should define the best possible edge cases and be useful to the most possible users. The operations are recommended only for binary because the requirements for decimal are not yet known.

A. Rounding Direction

Many uses for the augmented operations can be implemented and used correctly with any faithful rounding[35] if permitted to be non-exact transformations. The bit-wise reproducible algorithms of Section II-B generally require that the tie-breaking rule be independent of the generated values. Otherwise re-ordering reduction operands could round differently. There are five relevant rounding directions defined in IEEE 754-2008: 1) roundTiesToEven (default), 2) roundTiesToAway (only required for decimal), 3) roundTowardPositive, 4) roundTowardNegative, and 5) roundTowardZero. The latter three rounding directions break the exact transformation property of augmentedAddition (barring overflow). The error in addition may not be exactly representable as a floating-point number.

An early version of the proposal specified roundTiesToAway. roundTiesToAway is sufficient for many uses of double-double arithmetic and implementations of reproducible linear algebra kernels. However, a survey of potential users turned up a few cases that cannot directly use roundTiesToAway. Some examples are the accurate geometric primitives in Triangle[36], a high-quality and high-performance Delaunay mesh generator. With roundTiesToAway, those would fail to expand precision to distinguish between nearly colinear points. Additionally, correctness proofs of some accurate summation algorithms[14], [37] would need to be modified.

To support known uses, we introduce a new mode defined only in the recommended augmented arithmetic operations clause, roundTiesToZero, which is defined in Section III. This mode suffices for the known uses and potential uses of augmentedAddition.

B. Exceptional Behavior

The behavior of exceptional situations, exceptional values, and signed zeros differs between "typical" implementations of twoSum and twoProduct. Table I shows some behaviors from the implementations in Figure 1. Double-double operations with a NaN tail produce a NaN head, so producing a NaN tail on overflow will leave head and tail as NaN after any subsequent operations. On overflow, multiplication produces an invalid pair; adding the head and tail is invalid and would deliver NaN. So "typical" implementations of double-double and quad-double do not have IEEE 754-like overflow semantics. This rarely

TABLE I
UNFORTUNATE "TYPICAL" IMPLEMENTATION BEHAVIOR OF AUGMENTEDADDITION. SIMILAR BEHAVIOR OCCURS IN THE IMPLEMENTED AUGMENTEDMULTIPLICATION. HERE x AND y ARE POSITIVE OPERANDS.

x		y		head	tail	signal
∞	$+$	∞	$\Rightarrow$	∞	NaN	invalid
$-\infty$	$+$	$-\infty$	$\Rightarrow$	$-\infty$	NaN	invalid
x	$+$	y	$\Rightarrow$	∞	NaN	invalid, overflow, inexact ($x+y$ overflows)
$-x$	$+$	$-y$	$\Rightarrow$	$-\infty$	NaN	invalid, overflow, inexact ($-x-y$ overflows)
-0	$+$	-0	$\Rightarrow$	-0	$+0$	(none)
x	$\times$	y	$\Rightarrow$	∞	$-\infty$	overflow, inexact ($x\times y$ overflows)
$-x$	$\times$	y	$\Rightarrow$	$-\infty$	∞	overflow, inexact ($-x\times y$ overflows)
-0	$\times$	0	$\Rightarrow$	-0	0	(none)

TABLE II
PROPOSED EXCEPTIONAL AND SIGNED ZERO BEHAVIOR FOR AUGMENTEDADDITION. HERE x AND y ARE POSITIVE OPERANDS.

x	y	head	tail	signal
NaN	NaN	NaN	NaN	invalid on sNaN
$\pm\infty$	NaN	NaN	NaN	invalid on sNaN
NaN	$\pm\infty$	NaN	NaN	invalid on sNaN
∞	∞	∞	∞	(none)
∞	$-\infty$	NaN	NaN	invalid
$-\infty$	∞	NaN	NaN	invalid
$-\infty$	$-\infty$	$-\infty$	$-\infty$	(none)
x	y	∞	∞	overrflow, inexact ($x+y$ overflows)
$-x$	$-y$	$-\infty$	$-\infty$	overrflow, inexact ($-x-y$ overflows)
$+0$	$+0$	$+0$	$+0$	(none)
$+0$	-0	$+0$	$+0$	(none)
-0	$+0$	$+0$	$+0$	(none)
-0	-0	-0	-0	(none)

TABLE III
PROPOSED EXCEPTIONAL AND SIGNED ZERO BEHAVIOR FOR AUGMENTEDMULTIPLICATION. BEHAVIOR ON UNDERFLOW IS DESCRIBED IN THE TEXT. HERE x AND y ARE POSITIVE OPERANDS.

x	y	head	tail	signal
NaN	NaN	NaN	NaN	invalid on sNaN
$\pm\infty$	NaN	NaN	NaN	invalid on sNaN
NaN	$\pm\infty$	NaN	NaN	invalid on sNaN
∞	∞	∞	∞	(none)
∞	$-\infty$	$-\infty$	$-\infty$	(none)
$-\infty$	∞	$-\infty$	$-\infty$	(none)
$-\infty$	$-\infty$	∞	∞	(none)
x	y	∞	∞	overflow, inexact ($x\times y$ overflows)
$-x$	y	$-\infty$	$-\infty$	overflow, inexact ($-x\times y$ overflows)
x	$-y$	$-\infty$	$-\infty$	overflow, inexact ($x\times -y$ overflows)
$-x$	$-y$	∞	∞	overflow, inexact ($-x\times -y$ overflows)
$+0$	$+0$	$+0$	$+0$	(none)
$+0$	-0	-0	-0	(none)
-0	$+0$	-0	-0	(none)
-0	-0	$+0$	$+0$	(none)

TABLE IV
DOUBLE-DOUBLE ARITHMETIC LATENCY AND THROUGHPUT IMPROVEMENTS WITH TWO-INSTRUCTION AUGMENTEDADDITION AND AUGMENTEDMULTIPLICATION OPERATIONS[26]. LATENCY DECREASES AND THROUGHPUT INCREASES BOTH ARE IMPROVEMENTS.

Operation		Skylake	Haswell
Addition	latency	−55%	−45%
	throughput	+36%	+18%
Multiplication	latency	−3%	0%
	throughput	+11%	+16%

TABLE V
PERFORMANCE (IN MFLOPS) OF GENERAL DOUBLE-DOUBLE MATRIX-MATRIX MULTIPLICATION[26]

Operation	Intel Skylake	Intel Haswell
"Typical" implementation	1732 ($\approx 1/37$ DP)	1199 ($\approx 1/45$ DP)
Two-insn augmentedAddition	3344 ($\approx 1/19$ DP)	2283 ($\approx 1/24$ DP)
Native DGEMM	63603 (MKL)	51409 (MKL)

matters in practice; twoSum and fastTwoSum implementations are remarkably robust to overflow[38].

Signed zeros also produce unexpected results, leaving head and tail with opposite signs in many cases. The head of Figure 1's functions always has the same sign as the non-augmented operation, but the sum of head and tail will be $+0$ even when the sign could be preserved. A single twoSum or twoProduct can make sense, but an arbitrary mix of double-double operations based on twoSum and twoProduct could produce unexpected signs.

Standards must either define all results or leave the results explicitly as implementation-defined. The latter reduces portability and so reproducibility. We have chosen to define these exceptional and signed zero results so the head and tail always match. This maintains the IEEE 754 semantics for double-double, quad-double, and similar arithmetics during sequences of operations as well as when collapsing double-double and quad-double by adding the components.

V. POTENTIAL PERFORMANCE IMPROVEMENTS

Reducing twoSum from Figure 1 to two instructions from six is one obvious benefit to a two-instruction implementation of augmentedAddition. Here we refer to hardware operations as instructions and IEEE 754 operations as operations. Many of the instructions in twoSum have serializing data dependencies as well. A two-instruction augmentedAddition has no data dependencies, and both could be dispatched simultaneously if there are two appropriate execution units and provide nearly the performance of a single-operation augmentedAddition.

The operation twoSum only creates a double-double number from two input doubles. Adding two double-double numbers with an IEEE 754-style error bound requires two calls to twoSum, two additions, and two calls to fastTwoSum[3], [4] for a total of 20 instructions. An implementation using two-instruction augmentedAddition requires 10 instructions. Multiplying two double-double numbers shows only a tiny improvement on platforms with fusedMultiplyAdd in hardware, from 9 instructions to eight instructions. Reduced data dependencies could permit more of the instructions to be issued simultaneously.

To emulate two-instruction augmentedAddition and augmentedMultiplication, [26] replaces double-double addition and multiplication functions with two arithmetic instructions, $+/\times$ and min. This produces incorrect numerical results but models the expected performance benefits. Experiments with two Intel processors, a 4 GHz i7-6700K (Skylake) and a 3.5 GHz i7-4700K (Haswell), and full AVX2 vectorization demonstrate the potential improvements for double-double arithmetic. The two microarchitectures differ in the supported number of simultaneous additions; Skylake has two addition issue ports while Haswell has one.

Table IV shows around a 50% improvement in latency for double-double addition. The throughput improvement is lower with 36% for Skylake (two issue ports) and 18% for Haswell (one issue port). Double-double multiplication shows no gain in latency, as expected, but still over 10% gain in throughput. Table V considers general dense matrix multiplication (DGEMM). A slightly optimized double-double DGEMM kernel shows a $2\times$ performance improvement using the emulated two-instruction augmented arithmetic operations. The kernels still are $20\times$ slower than the vendor-tuned DGEMM (Intel's MKL). The reduced data dependencies permit further optimizations which *may* bring double-double matrix multiplication within a factor of ten in performance compared to vendor-tuned kernels.

For improvements possible in reproducible linear algebra, we count instructions in the ReproBLAS function that splits input operands and deposits them into bins. We expect a two-instruction augmentedAddition to improve performance of reproducible dot product (rddot) by around 33% from the instruction count. To validate, we modify the ReproBLAS source as above, replacing the fastTwoSum operations with augmentedAddition. Figure 2 shows performance of the modified rddot relative to the fastTwoSum-based rddot on an Intel E5-2650 (Haswell) at 2.2 GHz with 2.4 GHz DDR4 memory. Both rddot implementations are sequential but vectorized using AVX2. The performance improves by almost 33% for sufficiently long vectors (over 2048 elements). This brings the run time of reproducible dot products to around $2\times$ the OpenBLAS 0.2.18 dot product.

VI. GENERATING TESTS

Generating tests for the exact cases without ties is straightforward, as are cases for the overflow exceptional case, signed zeros, and NaN propagation. Additionally, tests verifying augmentedAddition remains exact on underflow are straightforward. Tests that exercise the data paths selectively are very efficient[39] but particular to the implementation. Verifying exact transformation property also verifies that the rounding mode is not directed (to $\pm\infty$, to zero); cases that violate exact transformation in those rounding modes are well-known. Here we consider test cases to verify use of roundTiesToZero as well as augmentedMultiplication's underflow behavior. In

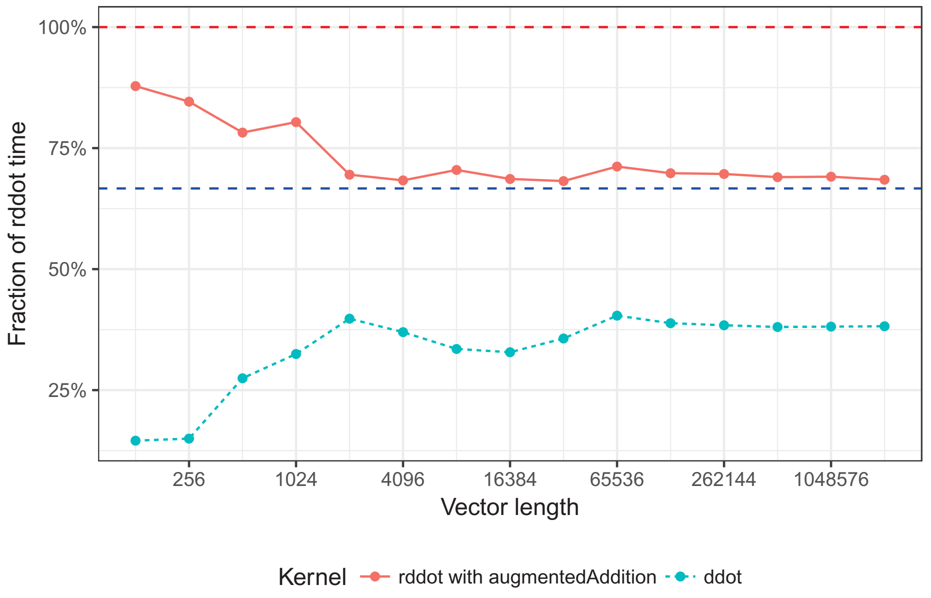

Fig. 2. Fraction of ReproBLAS rddot performance showing improvement with emulated augmentedAddition. Lower numbers are better improvement over rddot. The dashed horizontal line at 66% is the expected limit of improvement from the proposed operations. The bottom curve is the native BLAS optimized dot product for a lower bound.

the following, we call the least significant bit of a floating-point number x's significand the least significant bit of x. The minimum and maximum exponents in floating point formats are denoted *emin* and *emax*, respectively. In IEEE 754, $emin = 1 - emax$. Expressing the numbers as signed integers times powers of the radix (binary) is convenient for describing these test cases.

Assuming that augmentedAddition(x, y) produces a result where $x + y = h + t$ exactly, a simple case can verify that an implementation uses roundTiesToZero instead of another rounding mode. Let x be a normal, finite floating-point number where $x = (-1)^{1-s} \cdot T \cdot 2^E$ with T odd. Let $y = (-1)^{1-s} \cdot 2^{E-p-1}$, which has the same sign as x and which lines up just after the last bit of x when shifted to the same exponent for the intermediate infinitely precise result. Then augmentedAddition$(x, y) = (x, y)$ if the rounding is roundTiesToZero but $(x + 2y, -y)$ for roundTiesToEven or roundTiesToAway. With T even, y must have the opposite sign as x. These are the only cases where the tie-breaking direction matters, as also noticed in [38].

To test augmentedMultiplication, we need two x_m and y_m whose product becomes the x and y above. Correctness of the sign can be tested separately, so here we only consider positive x_m and y_m. For a precision of p bits, any two such x_m and y_m such that $x_m \times y_m$ requires $p + 1$ bits to represent and that have 11 as the last two bits suffice for the odd T case above. Requiring $p + 1$ bits in this case sets the first bit as well. So $x_m \times y_m = (4M + 3 + 2^{p+1}) \cdot 2^E$ with $0 \le M < 2^{p-1}$ and $emin \le E + p - 1 < emax$. Test cases can sample M, factor the significand $4M + 3 + 2^{p+1}$, and sample random exponent pairs E_{x_m} and E_{y_m} whose product satisfies the bounds on E.

augmentedMultiplication also can be inexact when the infinitely precise result's least significant bit is strictly between the least representable non-zero floating-point numbers, $\pm 2^{(emin-p+1)}$. If x_m and y_m both have magnitudes less than the square root of the least representable non-zero floating-point number, their product rounds to zero. So let at least one of x_m and y_m be a normal floating-point number with magnitude at least $2^{(emin-p+1)}$. Now we generate cases where the product $x_m \times y_m$ loses bits when rounded to a limited exponent range. Because the product of two p-bit significands can be exactly represented in $2p$ bits, the head and tail of augmentedMultiplication have adjacent binary representations. With roundTiesToZero, both h_m and t_m have the same sign.

The product $x_m \times y_m$ again needs the bits at each end of the $2p$-long significand product set to 1. The least bit corresponds to a number at most 2^{emin-p} to fall below the least subnormal number and be rounded to a representable exponent. Because they lack the implicit leading bit in IEEE 754 formats, subnormal numbers have $p - 1$ bits for a significand. Let $0 \le k < p - 1$ be the number of additional bits we want below the hard underflow threshold. We form the product from three sampled integers: $0 \le M < 2^{p-1} - 1$, $0 \le N < 2^{p-k-1}$, and $0 \le T < 2^k$. The integer T will be the portion below the hard underflow threshold and will be rounded. The upper bound on M ensures that one bit is free to absorb any carry from rounding. The desired products are

$$x_m \times y_m = \left(\underbrace{(2^{p-1} + M) \cdot 2^p}_{\text{head}} + \underbrace{N \cdot 2^k}_{\text{tail}} + \underbrace{2T + 1}_{\substack{\text{below}\\\text{underflow}}} \right) \cdot 2^{emin-p-k}$$

Because the last bit is set to one, there are no ties when $k > 0$. The $k = 0$ case can be tested separately to ensure roundTiesToZero. The results will accumulate rounding in the bit positions occupied by M and N above when $T > 2^{k-1}$. In that case, either the result's N is incremented by one or, when $N = 2^{p-k-1} - 1$, the result's $N = 0$ and M is incremented by one.

The significands of x_m and y_m can be generated by factoring the above $2p$-long integer significand. Another option is to generate two of the integer parameters and then iterate over the third, likely T The exponents of x_m and y_m can be generated by any pair that add to $emin - p - k$ such that at least one is greater than $(emin - p + 1)/2$.

VII. Conclusions

Previous uses extending precision and new uses providing bitwise reproducible summation motivate including *recommended* (not required) augmented arithmetic operations in the next revision of IEEE 754. Inclusion provides an opportunity to define exceptional situations carefully and provide IEEE 754-like semantics to extended arithmetics like double-double and quad-double. Even if implemented using two hardware instructions, augmentedAddition provides a 50% improvement in double-double addition latency and nearly $2\times$ improvement in double-double matrix multiplication.

We have not discussed alternate exception handling (*e.g.* trapping) beyond abrupt underflow (flush to zero). Other alternate exception handling methods require a wider system perspective to encompass many implementation possibilities. We encourage platforms to consider how to provide alternate exception handling for multi-instruction operations in an efficient manner, whether through hardware, compiler, or library support. Decimal users could benefit from reproducible summation, but details of the proof[7] require careful modification to convert from bits to decimal digits.

VIII. Acknowledgments

This work partially was supported by NSF Grant ACI-1339745 (XScala). The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." We also thank the other members of the IEEE 754 revision committee for their attention to proposal details, as well as Jonathan Shewchuck and Siegfried Rump for considering the effects of roundTiesToAway. Greg Henry, Peter P.T. Tang, and Xiaoye S. Li discussions on simplifying and optimizing the XBLAS highlighted the optimization possibilities. And we thank the reviewers for their detailed comments.

References

[1] "IEEE standard for floating-point arithmetic," Microprocessor Standards Committee of the IEEE Computer Society, New York, NY, IEEE Std 754-2008, Aug. 2008, pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[2] O. Møller, "Quasi double-precision in floating point addition," *BIT*, vol. 5, no. 1, pp. 37–50, Mar. 1965, ISSN: 1572-9125. DOI: 10.1007/bf01975722.

[3] K. Briggs, *Doubledouble floating point arithmetic*, World-Wide Web document. Downing Street, Cambridge CB2 3EA, UK, 1998.

[4] Y. Hida, X. Li, and D. Bailey, "Algorithms for quad-double precision floating point arithmetic," *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, DOI: 10.1109/arith.2001.930115.

[5] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, Jan. 1965, ISSN: 0001-0782. DOI: 10.1145/363707.363723.

[6] S. Habib and *et al.*, "ASCR/HEP exascale requirements review report," *CoRR*, 2016. arXiv: 1603.09303 [physics.comp-ph].

[7] J. Demmel, P. Ahrens, and H. D. Nguyen, "Efficient reproducible floating point summation and BLAS," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-121, Jun. 2016.

[8] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, Jun. 1971, ISSN: 0945-3245. DOI: 10.1007/bf01397083.

[9] J. M. Wolfe, "Reducing truncation errors by programming," *Communications of the ACM*, vol. 7, no. 6, Jun. 1964, ISSN: 0001-0782. DOI: 10.1145/512274.512287.

[10] N. J. Higham, "Accuracy and stability of numerical algorithms," Jan. 2002. DOI: 10.1137/1.9780898718027.

[11] ——, "The accuracy of floating point summation," *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, Jul. 1993, ISSN: 1095-7197. DOI: 10.1137/0914050.

[12] Y. He and C. H. Q. Ding, "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications," *Proceedings of the 14th international conference on Supercomputing - ICS '00*, 2000. DOI: 10.1145/335231.335253.

[13] I. Yamazaki, S. Tomov, and J. Dongarra, "Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C307–C330, Jan. 2015, ISSN: 1095-7197. DOI: 10.1137/14m0973773.

[14] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, Jan. 2005, ISSN: 1095-7197. DOI: 10.1137/030601818.

[15] S. M. Rump, "Ultimately fast accurate summation," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3466–3502, Jan. 2009, ISSN: 1095-7197. DOI: 10.1137/080738490.

[16] X. S. Li, M. C. Martin, B. J. Thompson, T. Tung, D. J. Yoo, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, and et al., "Design, implementation and testing of extended and mixed precision BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2,

pp. 152–205, Jun. 2002, ISSN: 0098-3500. DOI: 10.1145/567806.567808.

[17] J. W. Demmel, Y. Hida, X. S. Li, and E. J. Riedy, “Extra-precise iterative refinement for overdetermined least squares problems,” *ACM Transactions on Mathematical Software*, vol. 35, no. 4, pp. 1–32, Feb. 2009, ISSN: 0098-3500. DOI: 10.1145/1462173.1462177.

[18] J. W. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, “Error bounds from extra-precise iterative refinement,” *ACM Transactions on Mathematical Software*, vol. 32, no. 2, pp. 325–351, Jun. 2006, ISSN: 0098-3500. DOI: 10.1145/1141885.1141894.

[19] K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, “Generalization of error-free transformation for matrix multiplication and its application,” *Nonlinear Theory and Its Applications, IEICE*, vol. 4, no. 1, pp. 2–11, 2013, ISSN: 2185-4106. DOI: 10.1587/nolta.4.2.

[20] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, “Handbook of floating-point arithmetic,” 2010. DOI: 10.1007/978-0-8176-4705-6.

[21] L. Liu, S. Peng, C. Zhang, R. Li, B. Wang, C. Sun, Q. Liu, L. Dong, L. Li, Y. Shi, Y. He, W. Zhao, and G. Yang, “Importance of bitwise identical reproducibility in earth system modeling and status report,” *Geoscientific Model Development Discussions*, vol. 8, pp. 4375–4400, 2015. DOI: 10.5194/gmdd-8-4375-2015.

[22] C. Charpilloz, A. Arteaga, O. Fuhrer, C. Harrop, and M. Thind, “Reproducible climate and weather simulations: An application to the cosmo model,” in *Platform for Advanced Scientific Computing (PASC) Conference*, Jun. 2017.

[23] J. Demmel and H. D. Nguyen, “Numerical reproducibility and accuracy at exascale,” *2013 IEEE 21st Symposium on Computer Arithmetic*, Apr. 2013. DOI: 10.1109/arith.2013.43.

[24] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, “ExBLAS: Reproducible and Accurate BLAS Library,” in *NRE: Numerical Reproducibility at Exascale*, Austin, TX, United States, Nov. 2015.

[25] A. Arteaga, O. Fuhrer, and T. Hoefler, “Designing bit-reproducible portable high-performance applications,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1235–1244. DOI: 10.1109/IPDPS.2014.127.

[26] M. Dukhan, R. Vuduc, and J. Riedy, “Wanted: Floating-point add round-off error instruction,” in *The 2nd International Workshop on Performance Modeling: Methods and Applications (PMMA16)*, Frankfurt, Germany, Jun. 2016.

[27] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, “ARPREC: An arbitrary precision computation package,” Sep. 2002. DOI: 10.2172/817634.

[28] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, “MPFR: A multiple-precision binary floating-point library with correct rounding,” *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007, ISSN: 0098-3500. DOI: 10.1145/1236463.1236468.

[29] D. Bailey, “High-precision floating-point arithmetic in scientific computation,” *Computing in Science and Engineering*, vol. 7, no. 3, pp. 54–61, May 2005, ISSN: 1521-9615. DOI: 10.1109/mcse.2005.52.

[30] U. W. Kulisch and W. L. Miranker, “The arithmetic of the digital computer: A new approach,” *SIAM Review*, vol. 28, no. 1, pp. 1–40, Mar. 1986, ISSN: 1095-7200. DOI: 10.1137/1028001.

[31] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanović, “A hardware accelerator for computing an exact dot product,” *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, Jul. 2017. DOI: 10.1109/arith.2017.38.

[32] D. R. Lutz and C. N. Hinds, “High-precision anchored accumulators for reproducible floating-point summation,” *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, Jul. 2017. DOI: 10.1109/arith.2017.20.

[33] C. Chohra, P. Langlois, and D. Parello, “Reproducible, accurately rounded and efficient blas,” in *Euro-Par 2016: Parallel Processing Workshops*, F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds., Cham: Springer International Publishing, 2017, pp. 609–620, ISBN: 978-3-319-58943-5.

[34] T. Ogita, S. M. Rump, and S. Oishi, “Accurate sum and dot product,” *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005. DOI: 10.1137/030601818.

[35] D. M. Priest, “On properties of floating point arithmetics: Numerical stability and the cost of accurate computations,” UMI Order No. GAX93-30692, PhD thesis, Berkeley, CA, USA, 1992.

[36] J. Richard Shewchuk, “Adaptive precision floating-point arithmetic and fast robust geometric predicates,” *Discrete and Computational Geometry*, vol. 18, no. 3, pp. 305–363, Oct. 1997, ISSN: 0179-5376. DOI: 10.1007/pl00009321.

[37] S. M. Rump, T. Ogita, and S. Oishi, “Fast high precision summation,” *Nonlinear Theory and Its Applications, IEICE*, vol. 1, no. 1, pp. 2–24, 2010, ISSN: 2185-4106. DOI: 10.1587/nolta.1.2.

[38] S. Boldo, S. Graillat, and J.-M. Muller, “On the robustness of the 2Sum and Fast2Sum algorithms,” *ACM Transactions on Mathematical Software*, vol. 44, no. 1, pp. 1–14, Jul. 2017, ISSN: 0098-3500. DOI: 10.1145/3054947.

[39] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, “FPgen - a test generation framework for datapath floating-point verification,” *Eighth IEEE International High-Level Design Validation and Test Workshop*, DOI: 10.1109/hldvt.2003.1252469.