Evaluating the Numerical Stability of Posit Arithmetic

Nicholas Buoncristiani
Lawrence Berkeley National Laboratory and UC Berkeley
Berkeley, California
NickBuoncristiani@lbl.gov

Sanjana Shah
Lawrence Berkeley National Laboratory and UC Berkeley
Berkeley, California
SanjanaShah@lbl.gov

David Donofrio
Lawrence Berkeley National Laboratory and Tactical Computing Laboratories
Berkeley, California
DDonofrio@lbl.gov

John Shalf
Lawrence Berkeley National Laboratory
Berkeley, California
JShalf@lbl.gov

***Abstract*—The Posit number format has been proposed by John Gustafson as an alternative to the IEEE 754 standard floating-point format. Posits offer a unique form of tapered precision whereas IEEE floating-point numbers provide the same relative precision across most of their representational range. Posits are argued to have a variety of advantages including better numerical stability and simpler exception handling.**

The objective of this paper is to evaluate the numerical stability of Posits for solving linear systems where we evaluate Conjugate Gradient Method to demonstrate an iterative solver and Cholesky-Factorization to demonstrate a direct solver. We show that Posits do not consistently improve stability across a wide range of matrices, but we demonstrate that a simple re-scaling of the underlying matrix improves convergence rates for Conjugate Gradient Method and reduces backward error for Cholesky Factorization. We also demonstrate that 16-bit Posit outperforms Float16 for mixed precision iterative refinement – especially when used in conjunction with a recently proposed matrix re-scaling strategy proposed by Nicholas Higham.

***Index Terms*—Posit, IEEE floating-point, numerical stability, linear algebra**

I. Introduction

The IEEE 754 number formats that were standardized in 1985 have become the default approach for floating-point arithmetic in digital systems. This standard has enabled portable floating-point arithmetic across diverse computing platforms. IEEE floating-point defines a *mantissa* which stores the significant bits and an *exponent* to represent the scaling of the mantissa. The size of these two components are fixed by the IEEE standard for the 32-bit (Float32), 64-bit (Float64), and 16-bit (Float16) binary formats. The Posit format has some similarities with IEEE 754, but it allows the number of significant bits to vary depending on how close the scaling component is to zero or equivalently how close the number being represented is to one in the geometric sense. This has potentially useful properties for concentrating numerical precision where it is more likely to be used, and for extending the range of lower-precision (16-bit and 32-bit) numerical formats.

Unfortunately, many fundamental results in numerical analysis are not easily applicable to Posits because we cannot put a bound on the relative error that will arise – even for simple arithmetic operations. For this reason, we must subject Posits to a number of empirical tests to compare them with their IEEE floating-point competitors. Ultimately, the value of a new floating point format such as Posit is not just in the elegance of its implementation, but also its performance and numerical stability for real world problems.

The primary contribution of this work is to present a detailed evaluation of the stability of Posit arithmetic for numerical linear-system solvers – both for iterative and direct methods. We provide a first-order evaluation of the Posit's numerical properties so that we can assess its potential value to scientific applications. Aside from Gustafson's original work, this paper is among the first completely independent analyses of the numerical stability properties of Posit arithmetic.

Our key findings are that for matrices in their native range there is typically no substantial difference in the numerical accuracy of Posits in comparison with equivalently sized IEEE formatted numbers when applied to linear solvers. However, if a matrix is re-scaled to optimize the Posit representation, then Posit32 may offer up to 4 extra bits of precision over Float32, and Posit16 may offer 2 extra bits over Float16.

II. Background

We provide a brief overview of the IEEE and Posit floating point formats.

A. IEEE format

The IEEE 754 standard [1] defines the format of floating point variables as shown in Fig. 1. For single-precision variables the total number of bits is 32 and double-precision variables, the total number of bits is 64, including the most

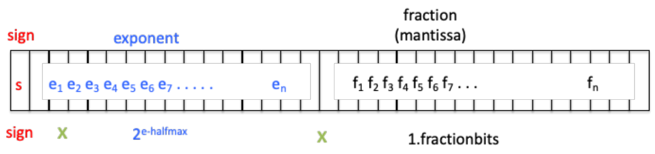


Fig. 1. Binary format for the IEEE 754 standard.

significant bit which is the sign. In both cases, the mantissa includes a silent bit at its most significant position, which is 1 if the exponent is non-zero.

If the exponent is non-zero, the number is considered normalized and a bias of -1023 is applied to the exponent value. Therefore, an exponent value of 50 represents an actual exponent of $50 - 1023 = -973$. The decimal value of a normalized double is:

$$Value = (-1)^{sign} \times [1.Mantissa] \times 2^{Exp-1023} \quad (1)$$

IEEE floating point variables with a zero exponent are considered denormalized and are used to fill the gaps in the numbers that double-precision variables can represent close to zero. A denormalized double-precision variable encodes the number:

$$DenormValue = (-1)^{sign} \times 0.Mantissa \times 2^{-1022} \quad (2)$$

Operating on numbers that have significantly different exponent values can cause bits to be dropped and thereby accumulate numerical error [2]. Such precision loss is hard to predict in many applications and can appear with just two operands, but can manifest as large numerical errors for direct solves or failure to converge to a solution for iterative solvers [3]–[5].

B. Posit format

John Gustafson developed the unum (universal number), a novel binary format for real numbers that claims to increase precision over the IEEE format for the same amount of memory. A Posit is a hardware-friendly unum representation and has four parts: sign bit, regime, exponent, and fraction. A Posit number is defined using the total number of bits, *n*, and the exponent size, *ES*. The sign bit is 0 or 1, representing positive or negative respectively. The number of regime bits is variable and is indicated by a run of identical bits after the sign bit which is terminated by the first opposite bit. The value k represents the length of this run of identical bits and is positive if these bits are 1 and negative if they are 0. If there are any bits left after the sign and regime bits, the next partition is reserved for exponent bits, and following the exponent bits are the fraction bits.

$$USEED = 2^{2^{es}} \quad (3)$$

$$RegimeValue = USEED^{k} \quad (4)$$

The fraction, *frac*, is 1 + the fraction bits following a decimal point. The absolute value of the Posit number is: regimeValue * exponentValue (if any bits) * fractionValue (if any bits).

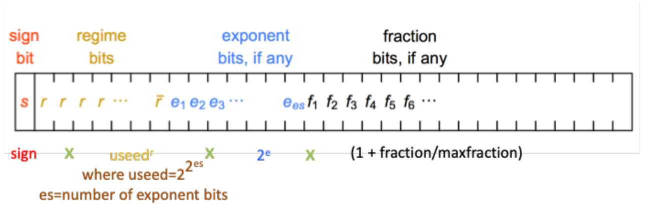

sign bit
regime bits
exponent bits, if any
fraction bits, if any
s r r r r ··· r̄ e1 e2 e3 ··· e_es f1 f2 f3 f4 f5 f6 ···
sign
x
useed^r
x
2^e
x
(1 + fraction/maxfraction)
where useed=2^(2^es)
es=number of exponent bits

Fig. 2. Binary format for the Posit arithmetic standard as described by its inventor, John Gustafson.

There are two special cases for Posit, 0 which is represented by all 0 bits and $\pm\infty$ which is represented by a 1 in the sign bit followed by n-1 0 bits. The latter is sometimes referred to as NaR or “Not a Real”.

For the rest of this paper, we will identify the Posit datatypes succinctly according to their size and the ES parameter by Posit(size, ES). For example, 32-bit Posit with ES set to 2 will be identified by Posit(32, 2).

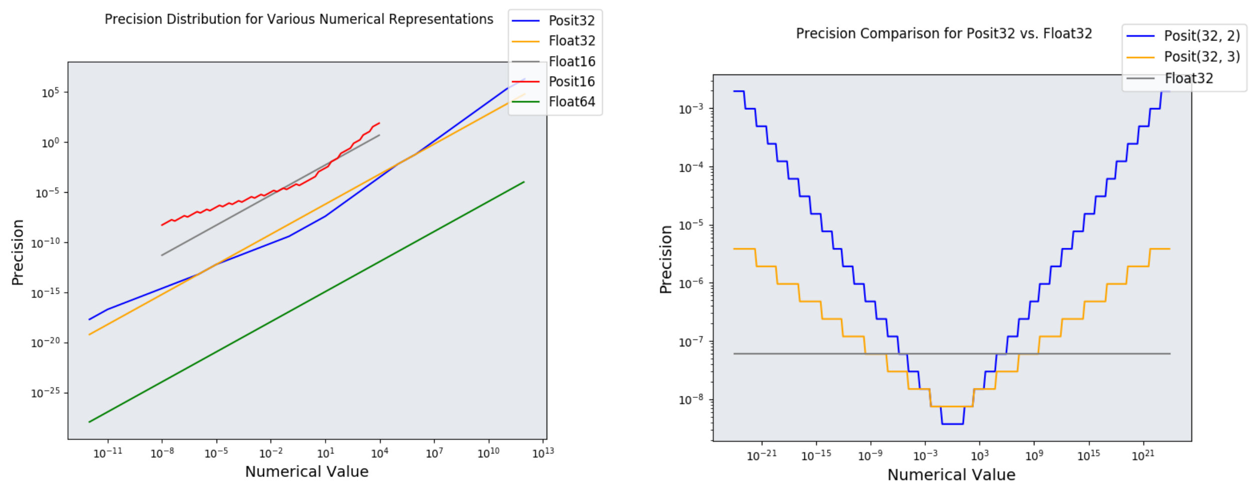
Numerical error is incurred whenever an operation requires a number to be rounded to the nearest representable value. For Float, this error will always be bounded relative to the magnitude of the result. What this means precisely is as follows. Define $F \subset \mathbb{R}$ to be the set of representable floating point values and let $f : \mathbb{R} \to F$ denote the conversion from $\mathbb{R}$ to the nearest value in F. Then for any $x \in \mathbb{R}$ (that is within the range of normalized Float), $f(x) = x(1+\epsilon)$ where ϵ is a constant value. In the case of Float32, for example, $\epsilon = 5.96 * 10^{-8}$ and for Float64, $\epsilon = 1.00 * 10^{-16}$.

For Posits this axiom no longer holds for a fixed ϵ. For Posit(32, 2), for example, ϵ ranges from $3.73*10^{-9}$ to USEED at the largest. The assumption which motivates the Posit format is that numbers close to one in the geometric sense are the most commonly occurring in computation, so to take advantage of this design numbers close to one are represented such that the relative error bound ϵ is as small as possible. A visualization of this interpretation is shown in Fig. 3. Taking inspiration from de Dinechin [6], we will refer to the area where Posit has improved precision over Float as the “golden-zone” for convenience.

Fig. 3 compares the absolute precision and relative precision of the floating point and Posit number encodings. In particular Fig. 3(b) shows more clearly how the Posit precision is enhanced for numbers in the vicinity of 1.0 (and -1.0), and that decreasing the number of ES bits improves maximum precision but precision tapers off faster.

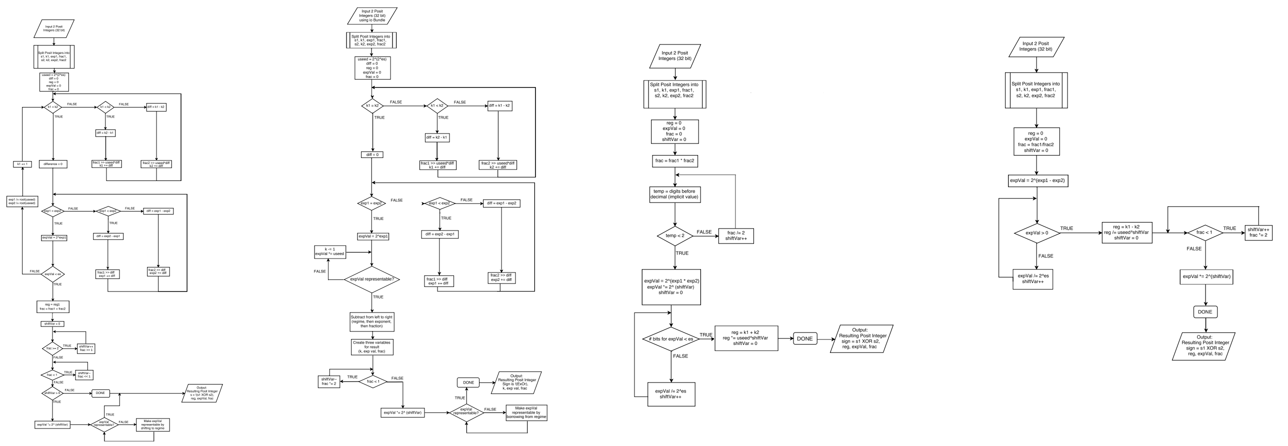
C. Posit Arithmetic

The algorithm for Posit addition shown in Fig. 4(a) is very similar to that of subtraction, so we will only walk through the addition example. Addition consists of comparing regimes and exponents for both numbers and shifting each up by the difference in number of bits to make them equal. Once the regimes and exponents are equal, addition can be performed on the fraction bits. Overflow would occur if the fraction was greater than or equal to 2 and underflow would occur if the fraction was less than 1. After adding, if any overflow exists, the overflow gets carried into the exponent or all the way to



(a) Absolute Precision (b) Relative Precision

Fig. 3. The absolute precision distributions in $[10^{-12}, 10^{12}]$ of the different Posit and IEEE number formats are shown in (a). Relative precision distributions (or simply "digits of precision") are shown in (b) for Posit32 and Float32. Numbers close to 10^0 have higher precision in Posit format and have better relative precision until roughly 10^{-5} for Posit(32, 2).



(a) Add (b) Subtract (c) Multiply (d) Divide

Fig. 4. **Flowcharts for Posit arithmetic algorithms.**

the regime if necessary, while the fraction shifts right by one bit. If carried into the regime, the exponent will be brought down to 0. Conversely, if any underflow exists, the scaling component will be shifted down by one bit while the fraction shifts up one bit. The regime of the output takes on the value of one of the regimes, as they are both equal. The same process goes for the exponent, where the output takes on the value of one of the exponents, as they are both equal. The sign of the output is taken by computing the sign of the first input value XOR the sign of the second input value and then is negated.

The algorithm for Posit multiplication shown in Fig. 4(c) is similar to the division algorithm, but unlike addition it does not consist of any comparisons made between the regimes and exponents of the two values. Instead, the fraction bits are simply multiplied, as well as the exponent bits, and overflow is dealt with by checking if the resulting exponent value can be represented using the number of ES bits. If it cannot be, the exponent value is divided by two as the overflow gets carried into the regime. The value of the regime is obtained by adding the k values of the regimes of the two input values and multiplying by $useed^{shiftVar}$, where shiftVar represents the value the regime must shift by. This is done to include the carried overflow in the regime. The sign of the output is taken by computing the sign of the first input value XOR the sign of the second input value.

Posit arithmetic conventions state that collective operations such as the dot-product should be carried out in a scratchpad register called a *quire* such that rounding is deferred until the end of the calculation. Initial comparisons between Posits and floats [6] rely on the assumption that Posits have access to the *quire* whereas floats are forced to round after every intermediate computation or they are limited to FMA as their only fused operation. Such fused operations like extended dot-product are entirely plausible for floats as well as Posits; Michelogiannakis [7] demonstrates that a rounding-deferred accumulate for Float32 and Float64 can be implemented using relatively simple hardware. Because these fused operations may improve performance substantially for both IEEE Floats and for Posits, we do not believe it is illuminating to compare the two formats this way because this does not highlight advantages that may be derived from the Posit format itself. Therefore, we offer our experiments operate without this assumption of deferred rounding.

III. Related Work

Very little work has been published evaluating the properties of Posits aside from the publications of its inventor. In his seminal work describing Posits, John Gustafson [6] uses some simple experiments to demonstrate that Posit may perform very well for solving linear systems. The initial work shows examples where a single precision (32-bit) Posit may achieve a better quality numerical result than an IEEE double precision float for Gaussian elimination if it is allowed a step of iterative refinement where the residual is computed using the *quire* to fuse the dot-product operation. The matrix used in this experimented was generated with pseudo-random entries evenly distributed over the interval [0, 1) which naturally gives Posit an advantage over Float since most of these entries will lie close to 0 on a log-scale. To level the playing field, our experiments use a wide variety of scientific matrices taken from Matrix Market some of which may be poorly scaled for Posits, and we avoid the use of the *quire* for either format.

Improved hardware support for low precision floats targeting artificial intelligence applications has inspired work on mixed precision linear algebra as a way to improve performance by exploiting fast low-precision arithmetic. Recent work by Haider, Carson, and Higham [8]–[10] have demonstrated the effectiveness of using Float16 to perform the factorization stage of an iterative refinement (IR) procedure. The motivating idea is to perform the $O(n^3)$ work (i.e LU factorization) in a lower precision to capitalize on the faster arithmetic, and to refine the solution in a second stage by $O(n^2)$ refinement iterations. For this paper we use Cholesky-Factorization for the work intensive $O(n^3)$ stage of the algorithm and we use classic iterative-refinement to achieve a solution that is accurate to Float64 precision. We use Cholesky Factorization instead of LU because it does not depend on row-pivoting and we are working with symmetric positive-definite matrices in this paper.

Higham [10] proposes strategies for scaling a matrix so that the limited dynamic range of Float16 is less problematic for mixed-precision IR. In this paper, we demonstrate that using Posit(16, 2) instead of Float16 may also help mitigate this issue due to its substantially increased dynamic range. We also demonstrate that if we apply Higham's scaling method to Posits then by tweaking a single step we can easily fit the matrix into the Posit "golden-zone" and achieve faster convergence then is possible with Float16. This points to a potential advantageous use of Posits for mixed precision arithmetic.

de Dinechin [6] analyzes some high level numerical properties of Posits and makes the observation that effective use of the Posit format requires the programmer to be aware of the scale of their problem similar to how fixed point arithmetic once required this before the days of floating point arithmetic. Based on this observation, we can think of Posit as an intermediate between scale-agnostic IEEE floating-point, which sacrifices some precision for the sake of convenience and fixed-point, which offers the maximum possible precision at the expense of high effort to the programmer. In this paper we demonstrate via experimental results how re-scaling affects the numerical stability of algorithms for solving linear systems.

IV. Experimental Methodology

The primary applications we will emphasize in this paper are numerical methods for solving systems of linear equations. We test CG, Cholesky-factorization, and mixed-precision iterative refinement. For each application of interest, we offer a simple re-scaling strategy to exploit Posit tapered precision, and we demonstrate results with and without this manual re-scaling. We believe re-scaling strategies for Posits may be an interesting avenue of research in their own right.

Our experiments for evaluating the numerical properties of Posits and IEEE Floats for solving classic $Ax = b$ linear systems are summarized below.

1) Classic conjugate gradient method (CG)
2) CG with matrix re-scaling
3) Cholesky factorization solver
4) Cholesky factorization with matrix re-scaling
5) Mixed precision iterative refinement
6) Mixed precision iterative refinement using Higham's scaling

A. Implementation of Posit Arithmetic Library

We implemented the Posit arithmetic library using simple C++ operator overloading to enable the key operators (+, -, /, *, etc...) to implement a variety of Posit formats, and allow us to use one algorithm specification to test each different arithmetic format. Although there are a number of Posit library implementations available, most did not allow for varying the number of ES bits. Lastly, we created differential validation tests against GNU GMP (GNU Multi-Precision Arithmetic library) that operates at effectively unlimited precision and offers a reliable ground truth. We were able to exhaustively test the results of our library against existing implementations to validate our approach.

B. Sample Matrices

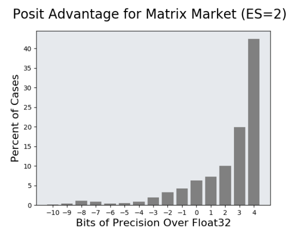
The matrices that are used throughout the rest of our experiments in this paper are shown in Table I and are listed in increasing order of their 2-Norm, $\|A\|_2$. The reason for this ordering will become apparent later in this article.

To evaluate Gustafson's claim that most arithmetic occurs close to 10^0 where Posit has improved precision, we created histograms which represent the number of additional bits of precision offered by Posit32 relative to the Float32 format when representing matrices taken from the Matrix Market repository. We obtained these results by loading non zero

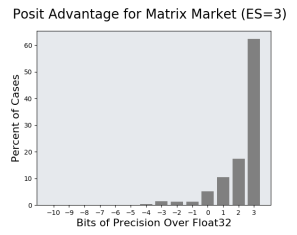
TABLE I

MATRICES FROM MATRIX MARKET REPOSITORY LISTED IN INCREASING ORDER OF THEIR 2-NORM, $\|\cdot\|_2$.

Matrix	$k(A)$	N	$\|A\|_2$	NNZ
plat362	2.2e11	362	7.7e-01	5786
mhd416b	5.1e9	416	2.2e0	2312
662_bus	7.9e5	662	4.0e3	2474
lund_b	3e4	147	7.4e3	2441
bcsstk02	4.3e3	66	1.8e4	4356
685_bus	4.2e5	685	2.6e4	3249
1138_bus	8.6e6	1138	3.0e4	4054
494_bus	2.4e6	494	3.0e4	1666
nos5	1.1e4	468	5.8e5	5172
bcsstk22	1.1e5	138	5.9e6	696
nos6	7.7e6	685	7.7e6	3255
bcsstk09	9.5e3	1083	6.8e7	18437
lund_a	2.8e6	147	2.2e8	2449
nos1	2e7	237	2.5e9	1017
bcsstk01	8.8e5	48	3.0e9	400
bcsstk06	7.6e6	420	3.5e9	7860
msc00726	4.2e5	726	4.2e9	34518
bcsstk08	2.6e7	1074	7.7e10	12960
nos2	5.1e9	957	1.57e11	4137



(a) Precision Posit(32,2)



(b) Precision Posit(32,3)

Fig. 5. Average distribution of non-zero entries for matrices in Matrix Market repository in terms of number of bits of precision compared with Float32. Most matrices seem to fit nicely within the golden-zone for Posits.

entries from these matrices into Posit and counting how many fraction bits were available, each matrix was weighted equally in obtaining these plots so that huge matrices would not dominate the results. Fig. 5(a) and Fig. 5(b) show these results for Posit(32, 2) and Posit(32,3).

C. Conjugate Gradient Method

Conjugate Gradient Method (CG) is an iterative method for solving linear systems, meaning that we should obtain a more precise solution as we increase the number of iterations. More precise arithmetic should be able to allow for faster convergence which makes this a good test for comparing the practical precision of the Posit and Float format. CG is a Krylov-subspace method, so it is driven by matrix-vector multiplications rather than by a factorization as are most direct methods. This latter fact becomes relevant when we want to re-scale to improve Posit performance.

As shown in line 5, the residual is computed by a mathematically equivalent recurrence relation rather than directly from the definition, $b - Ax_i$. If CG takes too many iterations to converge, it may introduce a significant discrepancy between

Algorithm 1 CG
```
1: $x_0 = 0, r_0 = b, p_0 = r_0$
2: for $i = 0, 1, 2, ...$ do:
3:     $\alpha_i \leftarrow \frac{\langle r_i, r_i \rangle}{\langle p_i, Ap_i \rangle}$
4:     $x_i \leftarrow x_{i-1} + \alpha p_i$
5:     $r_i \leftarrow r_{i-1} - \alpha_i Ap_i$
6:     $\beta_i \leftarrow \frac{\langle r_i, r_i \rangle}{\langle r_{i-1}, r_{i-1} \rangle}$
7:     $p_i \leftarrow r + \beta_i p_{i-1}$
```

the computed residual r_i and the true residual. This can in some cases lead to a slightly premature convergence since CG uses r_i as its convergence test. We noticed that this was the case in a handful of our experiments however we felt that it was not enough of a differentiating factor to be worth including in our results.

D. Cholesky Factorization

A direct solution to $Ax = b$ typically involves a factorization of A, followed by a solution stage where the factors of A are used to find a solution. Typically the factorization stage requires much more work than the solution stage.

Our direct-solve will use the Cholesky factorization $A = R^T R$ where R is upper-triangular and R^T is lower-triangular.

Given A, b the algorithm we use in our experiments is defined in Algorithm 7.

Algorithm 2 Cholesky Factorization
```
1: $R^T R = A$
2: $x_0 = 0$
3: for $i = 1, 2, 3, ...$ do:
4:     $r_i \leftarrow b - Ax_{i-1}$
5:     solve $R^T y = r_i$ for $y$
6:     solve $Rd = y$ for $d$
7:     $x_i \leftarrow x_{i-1} + d$
```

The presence of the for loop is so that we may generalize this algorithm to IR for our final set of experiments. For our single precision experiments with Cholesky Factorization we run a single iteration and examine the relative backward error which is is defined as $\frac{\|b - Ax\|_2}{\|b\|_2}$.

E. Mixed Precision

Our mixed precision experiments are based on traditional IR where the cholesky factorization is performed in 16-bit arithmetic and the factorization is cast into Float64 after line 1. of Algorithm 7 and is used to obtain a solution that is accurate to Float64 precision by several refinement iterations. We measure the number of iterations it takes to achieve an accurate solution after performing the factorization in Float16 and Posit16 respectively.

Float16 has been shown to be successful for this task as shown by Carson, Haider, and Higham. However Higham explains that the limited reach of Float16 may make it challenging to load a matrix into a Float16 approximation in some cases, or limit its ability to factorize the matrix without experiencing an overflow midway. We may be able to reduce the chance of overflow by using Posit(16, 2) instead of Float16 due to its wider representational range however we will show that Posits are also particularly well suited to the strategies proposed by Higham to mitigate overflow concerns for Float16 since with little extra effort we can squeeze the matrix into a range where Posit16 has superior accuracy.

V. Results

A. Conjugate Gradient Method Results

1) Introduction: In this section we compare the rate of convergence of CG for Float32, Posit(32, 2), and Posit(32, 3). Following the CG experiments in [11], we choose $\hat{x} = (\frac{1}{\sqrt{n}}, \ldots, \frac{1}{\sqrt{n}})^T$ such that $\|\hat{x}\| = 1$ and we assume convergence only when the size of our computed residual $\|b - A\hat{x}\|$, drops below $\|b\| * 10^{-5}$. i.e the relative backward error goes below 10^{-5}. It is worth mentioning that this convergence criteria is fairly strict since it was originally selected for double precision CG convergence tests, however we see it fit to exercise these numerical formats to their limits.

Experiments are performed on a wide variety of matrices with different condition numbers and matrix-norms that were obtained from the Matrix-Market collection. We load these matrices into an extended precision format before casting into finite precision. Float64 results are shown for reference.

2) Analysis: From our results shown in Fig. 6, we notice similar convergence results between Float32 and Posit(32, 3). On the other hand, Posit(32, 2) which is the conventional configuration for Posit32 [6], [12], diverges when applied to a handful of matrices and has very poor convergence for others.

As seen in Fig. 6 where matrices are sorted from left to right with low matrix norms on the left and high matrix norms to the right, matrices with large norms present a challenge for Posits. Convergence issues begin to emerge starting at nos1 and for all matrices to the right of it where Posit(32, 2) fails to converge and Posit(32, 3) shows poor convergence.

B. CG after re-scaling

CG is a Krylov-subspace method, so it takes advantage of frequent matrix multiplication to converge to a solution as opposed to operating directly on matrix itself. Since $\|\cdot\|_2$ describes how much a matrix is capable of scaling a vector through multiplication, it follows that the magnitudes of the iterates and hence the quality of their representations are intrinsically related to this matrix characteristic. This suggests a simple strategy for stabilizing performance where we try to scale the matrix such that Posit can work within the golden-zone as much as possible by choosing a matrix norm which is more likely to be advantageous for Posit. We decided somewhat arbitrarily to scale such that $\|\cdot\|_\infty$ is close to 2^{10} for these experiments. This will fit our matrices such that they fall somewhere between 662_bus and 685_bus in scale. We scale according to $\|\cdot\|_\infty$ as opposed to $\|\cdot\|_2$ because it is much easier to compute. We scale by a power of two so that Float32 results should remain almost the same if not exactly the same.

It is worth mentioning that scaling by a power of two is not necessarily loss-free for Posit [6] so we sacrifice some precision before the algorithm starts unless we start in a higher precision, which we assume for the sake of convenience in our CG and Cholesky-Factorization results. Note that this issue does not apply to our mixed precision results which are shown in Table III since this application assumes a higher precision to start with.

As shown in Fig. 7, this re-scaling has improved Posit convergence rates such that Posit(32, 3) converges faster for all matrices. Furthermore, we may safely leave Posit32 in its default configuration with 2 ES-bits without worrying about convergence issues.

C. Cholesky Factorization Results

1) Overview: In this section we compare Float32 against Posit32 when applied to Algorithm 7. We use the same matrices and our method for computing b is the same as in our CG experiments. Using Cholesky Factorization instead of LU has little effect on the results, but it offers a simpler algorithm which does not depend on row-pivoting to achieve optimal performance. Our performance metric is relative backward error which is defined as $\frac{\|b-Ax\|_2}{\|b\|_2}$ where x is our computed solution.

Posit(32, 2) does not give better results than Float32. Posit(32, 3) does offer some benefit although the advantage that either format offers degrades when matrix-norm is increased. This relationship is shown explicitly in Fig. 8(b).

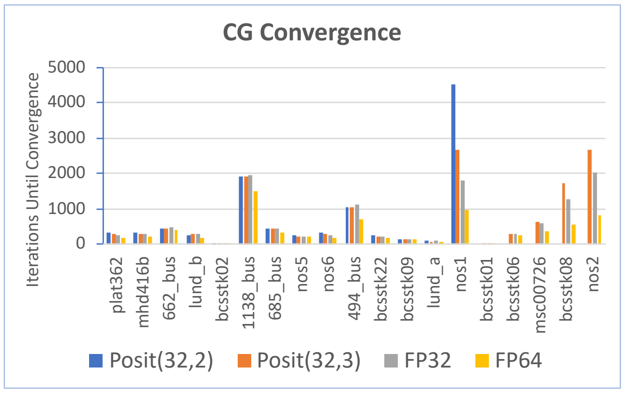
2) Re-scaling Cholesky: Unlike CG, which is built out of matrix-vector multiplications, a factorization based direct-solver operates directly on the entries of the matrix. For this reason, we suspect the performance of Posit depends largely on how well the entries of A fit into the golden-zone – meaning that a matrix with entries close to 1 should perform best for Posit.

Bringing entries closer to one by multiplying the matrix by the inverse average of all nonzero entries showed little performance gain for Posit. Observing that precise representation of diagonal entries might have a greater influence on numerical stability since these values are used as pivots, we tried scaling the matrix by the reciprocal of the average absolute value of all diagonal entries taken to the nearest power of two. Pseudo-code for implementing this strategy is shown in Algorithm 4.

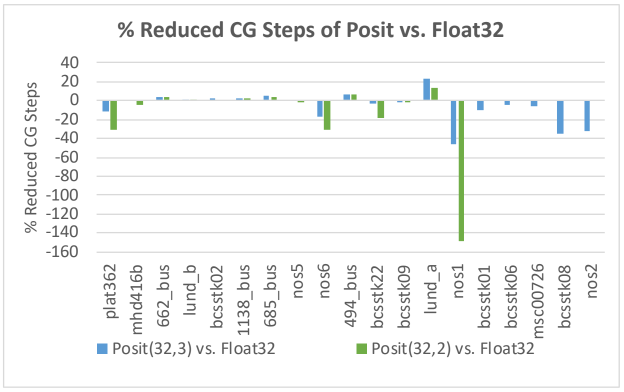
Algorithm 3 Matrix re-scaling for Cholesky

```
1: $s \leftarrow$ nearestPowerOfTwo(average($|A_{kk}|$))
2: $A^{'} \leftarrow \frac{1}{s}A$
3: $b^{'} \leftarrow \frac{1}{s}b$
4: Solve $A^{'}x = b^{'}$
```

Fig. 9 shows our results after scaling according to this strategy. Posit(32, 2) and Posit(32, 3) both perform better than Float32 in every experiment. Posit(32, 2) consistently achieves at least one extra digit of precision over Float32. Recall that Posit(32, 2) should theoretically offer an extra 1.2 digits or equivalently 4 bits of precision compared with Float32 if we

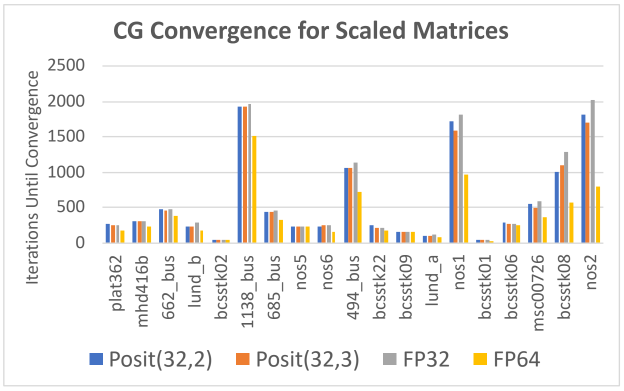


(a) Convergence

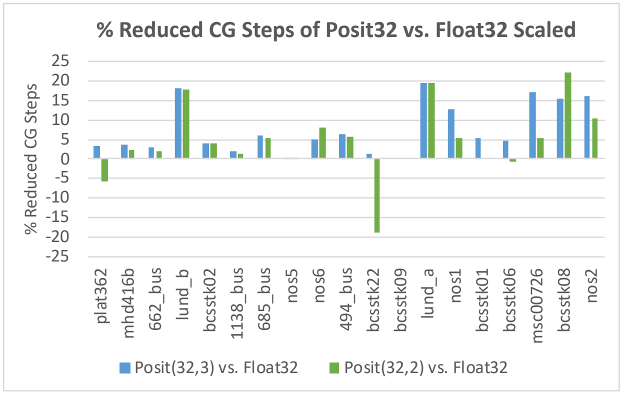


(b) %Improvement

Fig. 6. Number of iterations before CG convergence for 32-bit IEEE Floats vs. 32-bit Posit with 2 and 3 ES bits are shown in (a). Percent improvement of Posit32 over Float32 is shown in (b), with negative values indicating that Posit32 did worse than the Float32. The matrices to the right of bcsstk01 do not converge for Posit(32, 2).



(a) Convergence



(b) %Improvement

Fig. 7. Number of iterations to convergence (a) and percent improvement (b) on scaled matrices

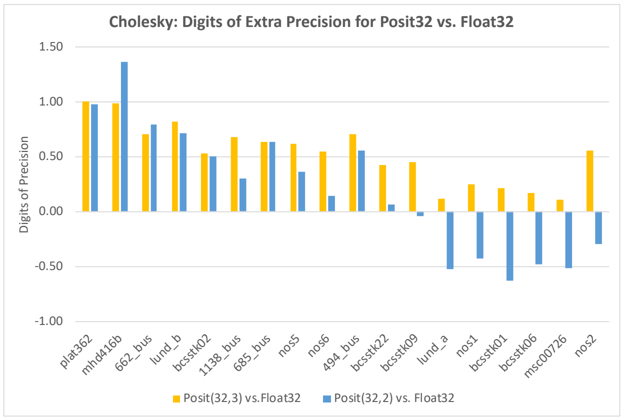
are able to take full advantage of its tapered precision (for reference Float32 offers about 7 digits of precision in the absence of round-off error). We observe that across all of our experiments Posit(32, 2) performs near this optimal mark, thus by prioritizing that the diagonal entries of our matrix fall within the golden-zone, we can achieve a final backward-error which is correct to golden-zone precision irrespective of whether or not our entire matrix is represented with this same degree of precision.

D. *Mixed Precision Iterative-Refinement*

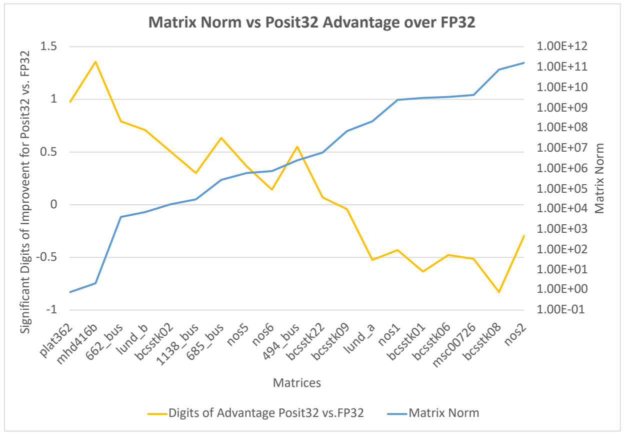
1) Introduction: Mixed precision has attracted interest in recent years owing to improved hardware support for Float16 and the discovery that we can achieve a high quality solution using iterative refinement even if the factorization is computed using low precision arithmetic. One difficulty with this approach is that Float16 has a very narrow dynamic range which greatly increases the potential for overflow and underflow. Indeed, a number of matrices in the Matrix-Market repository, including nos1, have a majority of entries which are completely out of range of Float16. Higham proposes a strategy to mitigate this issue by multiplying the matrix on either side by a diagonal matrix and then scaling by a constant value such the entries fit neatly within the range of Float16. In this section, we show that this additional step may be avoided in several cases if we use Posit(16, 2) in place of Float16 because Posit(16, 2) simply offers much greater reach. More importantly, we show that a minor modification of Higham's refinement algorithm allows us to fit the matrix comfortably into the golden-zone for Posit16 and take nearly full advantage of golden-zone precision.

2) Results: For our first set of experiments, we test the performance for naive mixed precision iterative refinement using Float16 and Posit16. Using the sample matrices, we run Algorithm 7 such that the factorization stage is performed using the low precision format. If an entry in the matrix is larger then the maximum representable value of Float16 or Posit16 then we round down to this value [10]. Note that for both Posit16 and Float16 we use Float64 as our working precision in order to isolate the effects that the factorization precision has on convergence rate.

As shown in Table II, Posit(16, 2) is capable of solving more matrices out of the box due to its wider dynamic range but it is still appears to be a very challenging task. An entry



(a) Precision Advantage



(b) Error vs. Norm

Fig. 8. (a) shows Posit32 improvement over Float32 in terms of number of extra digits of precision which is computed as $log_{10}(\frac{FloatResidual}{PositResidual})$. Positive values indicate that Posit has an advantage over Float32. b) plots the digits of precision advantage for Posit(32, 2) against the norm of the matrix.

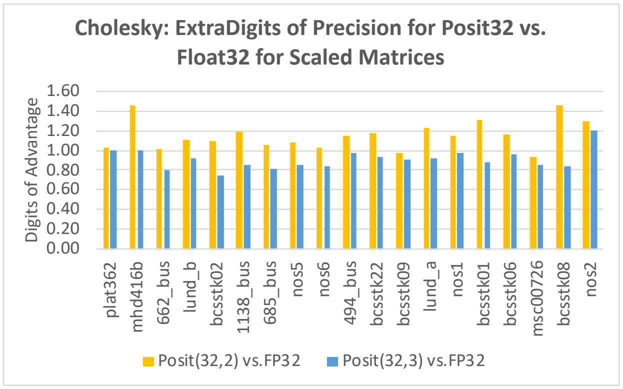


Fig. 9. Shows Posit32 advantage in terms of number of extra digits of precision over Float32 for scaled matrices.

TABLE II
OUT OF BOX PERFORMANCE OF MIXED PRECISION ITERATIVE REFINEMENT WITH FLOAT16 AND POSIT16. POSIT(16, 2) CAN SOLVE MORE PROBLEMS THAN FLOAT16.

Matrix	Float16	Posit(16; 1)	Posit(16; 2)
mhd416b	-	-	8
662_bus	52	187	90
lund_b	7	12	6
bcsstk02	13	51	23
685_bus	17	160	45
nos6	-	1000+	1000+
494_bus	-	-	991
bcsstk09	-	-	872
lund_a	-	-	35
bcsstk01	-	-	60
nos2	-	-	1000+

of '-' denotes that the matrix was unable to converge due to poor factorization or arithmetic error encountered during factorization. Matrices which are not included in this table but are included in Table I were not able to converge for any formats. 1000+ iterations indicates that convergence was not achieved within 1000 iterations but the factorization itself was successful. Note that if we were to use a more sophisticated approach such as GMRES for solving the correction equation these scenarios would be less likely to occur.

Now we turn our attention to a re-scaling strategy proposed by Higham to better utilize Float16. The strategy is summarized in Algorithm4 including some simplifications which are allowed when we only consider symmetric matrices as we do in this paper.

Algorithm 4 Higham Rescaling

```
1: obtain diagonal matrix $R$ such that $RAR$ has maximum element of each row and each column equal to one.
2: choose $\mu$ according to some strategy to shift the entries of $A$ such that we can take better advantage of the range of our half-precision format.
3: $A^{(h)} \leftarrow fl_h(\mu(RAR))$ where $fl_h(\cdot)$ denotes conversion to half-precision.
```

The strategy for choosing R is shown in Algorithm 6. The choice for μ is not portable across Float16 and Posit16, so we made sure to pick μ so that we would not bias either format. With Float16 in mind, Higham explains that μ should be large enough to take full advantage of its limited dynamic range, however it should be small enough to prevent subsequent operations from overflowing. Higham chooses μ arbitrarily to be $.1 * FP16max$ where $FP16max$ is the largest representable value for Float16. However because Posit works with tapered precision, it would be undesirable to push its entries so close to its maximum representable value. Experimentation has shown us that the best choice for μ for Posit16 is simply USEED, which is defined in Equation 3. Recall that we lose one fraction bit every time our quantity exceeds a power of USEED, and this will safely ensure that each row and column have maximum entry equal to USEED.

We observe that scaling the matrix by a power of 4 gave the best results – likely because it is both a power of two and a perfect square (Cholesky factorization, unlike LU, makes

TABLE III
MIXED PRECISION ITERATIVE REFINEMENT AFTER SCALING BY HIGHAM'S METHOD. POSIT(16, 1) OUTPERFORMS FLOAT16 IN EVERY EXPERIMENT.

Matrix	Float16	Posit(16, 1)	Posit(16, 2)	% diff
mhd416b	6	5	5	16.7
662_bus	71	31	17	56.3
lund_b	6	5	6	16.7
bcsstk02	13	8	10	38.5
685_bus	18	2	16	88.9
nos5	11	10	11	9.1
nos6	1000+	151	241	84.9
bcsstk22	17	9	11	47.1
bcsstk09	62	11	16	82.3
lund_a	23	9	17	60.9
nos1	1000+	822	1000+	17.8
bcsstk01	11	8	9	27.3
bcsstk06	41	25	25	39.0
msc00726	17	7	10	58.8
bcsstk08	18	15	11	16.7
nos2	1000+	1000+	1000+	0

use of the square-root operator). For this reason we round the scaling factor suggested by Higham to the nearest power of 4 to even the playing field since USEED is already a power of 4 for positive ES values.

Algorithm 5 Strategy for finding R matrix

```
1: $R = I$
2: while $max_i|r_{ii} - 1| \leq tolerance$ do
3:     for i = 1,2,...,n do:
4:         $r_i \leftarrow \|A(i,:)\|_\infty^{-1/2}$
5:     $A \leftarrow diag(r)Adiag(r)$
6:     $R \leftarrow diag(r)R$
```

Our convergence criterion are taken from Higham's mixed precision experiments which requires that the solution be accurate to Float64 precision, this criteria is somewhat strict because Higham uses the analysis done by Carson and himself [9] which suggests that the residual should be computed in twice the working precision in order to reliably obtain an accurate solution. However to avoid unnecessary complication since our objective is merely to compare Posit against Float, all operations after the factorization are performed in Float64. Furthermore, we use simple IR as shown in Algorithm 7 instead of the more powerful GMRES strategy for computing the correction equation. Similar to our CG tests, our convergence criteria being fairly strict helps to accentuate the differences between the two formats.

If we analyze the relative error involved in the factorization shown in Figure 10(a) we notice that Posit16 is performing consistently better than Float16 given this scaling strategy. Posit(16, 1) will offer an extra two bits of precision (or .6 digits of precision) to the result in the best case scenario. Figure 10(b) shows that Posit consistently achieves close to this mark.

Table III shows that Posit16 requires fewer refinement iterations than Float16 as expected by the reduced backward error in the factorization.

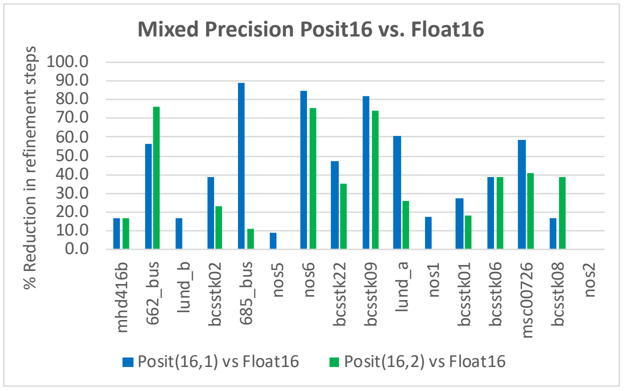
VI. ANALYSIS

The results of our experiments show that Posit performs slightly worse when applied to the CG algorithm and it may perform marginally better or worse with Cholesky Factorization depending on scaling. Our results show that performance of Posit relative to Float is highly dependent on scaling, so we took advantage of this fact in our secondary experiments which show that we can easily improve Posit performance by a simple scalar multiplication.

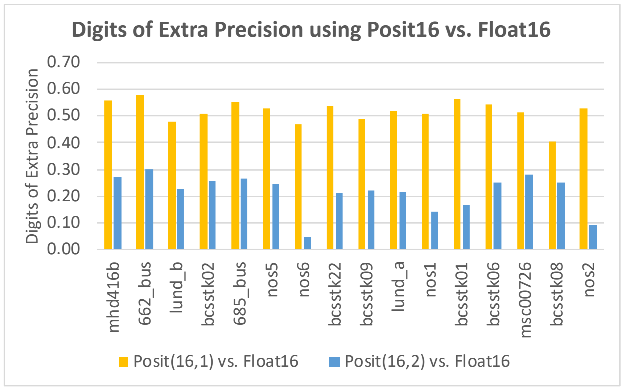
We hypothesize that certain procedures such as Bi-CG which have been observed to produce even larger iterates than traditional CG [10] may limit the potential for re-scaling as a means to stabilize Posit since the working dynamic range is very high.

Conversely, other procedures such as direct methods for solving linear systems may require a narrower working dynamic range which should make them amenable to re-scaling since the actual values used in computation are unlikely to drift to dramatic highs and lows if we center around 1. For example, LU factorization is observed to produce factors which are scaled similarly to the initial matrix and for Cholesky and QR factorization we know for certain that the norm of the original matrix will not fluctuate too much compared with that of its factors: $\|R\| = \|A\|$ for QR factorization and $\|R\| = \|R^T\| = \sqrt{\|A\|}$ for Cholesky Factorization. This may suggest that if the entries in A are within the golden-zone, then subsequent arithmetic is likely to remain near the golden-zone as well. In this paper we demonstrate that Cholesky Factorization in particular is amenable to scaling approaches, but we hypothesize that direct methods for solving linear systems in general may be more suited to re-scaling strategies and to the Posit format than iterative ones.

Since the advantage for Posit in terms of numerical stability is at most 1 to 2 extra digits of precision. A more compelling advantage for Posit may arise when we consider application where dynamic range is also a concern. Since Float16 has such limited representational range, we hypothesized that Posit16 could enable a more reliable alternative for the task of mixed precision iterative refinement. We found that this was indeed the case; however, in our experiments we found that issues still arose as a result of the use of low precision where there was too much error in the factorization to reliably derive an accurate solution in double precision arithmetic or an arithmetic error was encountered mid-way. So switching to Posit16 alone did not show to be a sufficient action to enable reliable low precision factorization. However we found that Higham's proposed re-scaling method for matrices helped mitigate these issues considerably for both Float16 and Posit16 and it also provides an easy way to reap additional benefit from the tapered precision of Posit16 and achieve a more accurate factorization requiring fewer refinement iterations.



(a) Percent Reduction of Refinement Steps



(b) Improvement in Digits of Precision

Fig. 10. (a) shows percent reduction of number of refinement steps required for convergence when switching from Float16 to Posit16 using Higham's scaling (b) shows the number of additional digits of precision offered by Posit16 over Float16 for Cholesky-Factorization backward error which is computed as $\frac{\|R^T R - A\|_F}{\|A\|_F}$, where R is the approximate factor.

VII. Conclusion

In this paper we explored the potential for the Posit format in terms of its numerical stability and offered a few strategies for improving stability by scaling the problem to a more appropriate working interval. We found that Posit did not improve performance considerably for CG or Cholesky Factorization results without this preliminary scaling however Cholesky-Factorization appears to be particularly well suited to deriving benefit from a re-scaling of the matrix. We found that while both Float16 and Posit16 formats suffered from overflow or arithmetic errors during naive iterative-refinement, Posit16 with 2 ES bits was able to achieve convergence in more cases as a result of its superior reach. We found that if we re-scaled the matrix as proposed by Higham then we could easily fit the matrix into an optimal range for Posit16 and reduce backward error by a factor which is consistent with the maximum possible precision supported by Float16.

In future works we will explore other scientific algorithms such as FFT, Bi-CG, and Sod's Shock tube for CFD. In particular we are interested in finding applications which may naturally work well with Posit and those which be difficult to reconcile even with re-scaling. We suspect that FFT may be a good application for Posit because its narrow working range makes it easy to squeeze into the Posit golden-zone.

Acknowledgment

We would like to thank Sherry Li, Phil Colella, and John Gustafson for their feedback and valuable advice in pursuing this paper. This research was carried out at LBNL, which is operated for the U.S. Department of Energy Office of Science under Contract No. DE-AC02-05CH11231.

References

[1] "IEEE standard for floating-point arithmetic," *ANSI/IEEE Std 754-2008*.
[2] K. R. Ghazi, V. Lefevre, P. Theveny, and P. Zimmermann, "Why and how to use arbitrary precision," *Computing in Science and Engineering*, vol. 12, no. 3, p. 5, 2010.
[3] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, 2012.
[4] E. Allen, J. Burns, D. Gilliam, J. Hill, and V. Shubov, "The impact of finite precision arithmetic and sensitivity on the numerical solution of partial differential equations," *Mathematical and Computer Modelling*, vol. 35, no. 11-12, 2002.
[5] J. M. Chesneaux, S. Graillat, and F. Jézéquel, "Rounding errors," in *Wiley Encyclopedia of Computer Science and Engineering*, 2008.
[6] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ser. CoNGA'19. New York, NY, USA: ACM, 2019, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/3316279.3316285
[7] G. Michelogiannakis, X. S. Li, D. H. Bailey, and J. Shalf, "Extending summation precision for network reduction operations," *International Journal of Parallel Programming*, vol. 43, no. 6, pp. 1218–1243, 2015. [Online]. Available: https://doi.org/10.1007/s10766-014-0326-5
[8] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11. [Online]. Available: https://doi.org/10.1109/SC.2018.00050
[9] E. Carson and N. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions," *S I A M Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, 2018.
[10] N. J. Higham, S. Pranesh, and M. Zounon, "Squeezing a matrix into half precision, with an application to solving linear systems," *SIAM J. Scientific Computing*, vol. 41, no. 4, pp. A2536–A2551, 2019. [Online]. Available: https://doi.org/10.1137/18M1229511
[11] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014. [Online]. Available: https://doi.org/10.1016/j.parco.2013.06.001
[12] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206