A Generalized Framework for Matching Arithmetic Format to Application Requirements

John L. Gustafson
School of Computing
National University of Singapore
Singapore
john.gustafson@nus.edu.sg

***Abstract*—To fit the distribution of representable computer numbers to the distribution of those needed by an application, the current tools are blunt. Floating-point format can adjust the number of significand and exponent bits to provide sufficient accuracy and dynamic range, but accuracy is flat across the entire range which is seldom the distribution needed by the application. For fixed-point, we can choose the bit size and location of the radix point, but again the uniform density of representable reals is typically not what is needed; relative accuracy is lowest near zero, We describe a new parameter that allows an adjustable amount of *tapering* for both floats and fixed-point formats. It is easy to build fast hardware for all settings of the parameter. When the parameter is set to the maximum, the format matches *posit* format, but at the minimum it matches IEEE 754 float range and accuracy. When applied to fixed-point, accuracy can be made highest near zero, which closely matches the requirements of Machine Learning. To demonstrate the high bit-efficiency possible using the new parameter, we show that 16-bit tapered representations achieve higher accuracy on Fast Fourier Transforms (FFTs) than can any 16-bit fixed-point or float representations. The accuracy is sufficiently high that 12-bit data from analog-to-digital convertors can be forward and inverse transformed back to the original signal with zero loss.**

***Index Terms*—computer arithmetic, error analysis, numerical analysis, numerical algorithms and problems, mathematics of computing, machine learning, neural networks, signal processing**

I. Introduction

Programmers of applications involving real numbers often select the largest type for which there is fast native hardware support. Since 32-bit and 64-bit floating-point hardware exists in most current processors, it is usually easy to find the data type with sufficient dynamic range and accuracy, and most of the programming effort is spent on issues other than the optimization of the numerical format to the task.

As a common-sense heuristic to maximize accuracy for a given number of bits in a format, we state the following principle: **The distribution of real numbers representable by a format should resemble the distribution that occurs in the application**. With memory bandwidth increasingly mismatched to the arithmetic speed, researchers are noticing the gains to be had by finding a better match of the format to the application so that the number of bits per value can be reduced. This is especially true in Neural Network (NN) Machine Learning (ML) and inference, but it has also long been an issue understood by the designers of Field-Programmable Gate Arrays (FPGAs) for tasks such as signal processing, where the number of bits of precision and the meaning of the bits can be highly customized.

For fixed-point arithmetic of a given precision, we can select where we place the radix point. For floating-point arithmetic (not necessarily IEEE 754 Standard [1]) we can choose the number of bits allocated to expressing the scale factor (exponent), leaving the remaining bits for the significand. Fixed-point numbers have a flat distribution of values between the most negative and most positive value. Binary floats have an equal-spaced distribution of x values within each *binade*, $2^k \leqslant x < 2^{k+1}$ for integer k within the dynamic range (see Fig. 1). Within each binade, there is the well-known "wobble" in log density of $\log_{10} 2 \approx 0.3$ [2]. We are mainly concerned with the overall envelope of the distribution and will often leave out the wobble for clarity in distribution histograms.

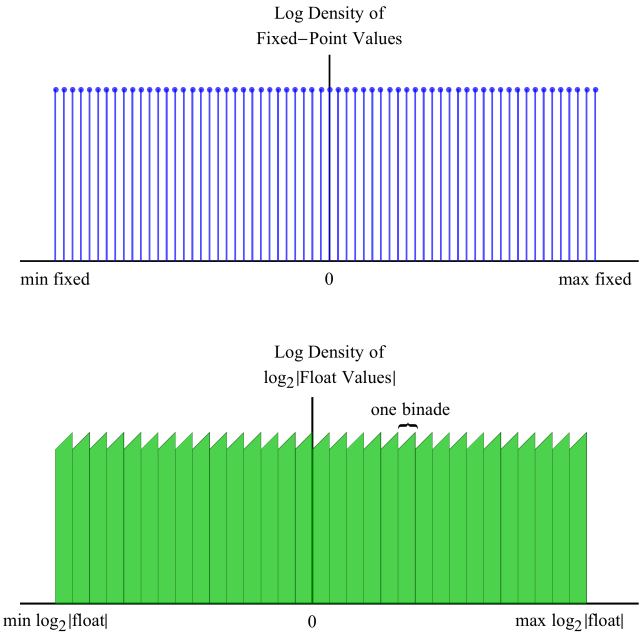

Fig. 1. Flat linear and logarithmic distributions for fixed-point and float types

We ignore *subnormal* floats here, which cause the leftmost binade to slope to zero on its left edge. Changing the location of the radix point or the exponent size does not change the

flatness of the distribution plot; it remains rectangular, but with different dimensions. We may know that an application never needs values greater than 100, say, or that most (but not all) of the values it uses are between -1 and 1, but there really are few ways to adjust the format distribution to match that of the task. Fig. 2 shows the tradeoff between accuracy and dynamic range for 16-bit floats with various exponent sizes *es*. (Wobble in the accuracy is averaged with a band, for clarity.)

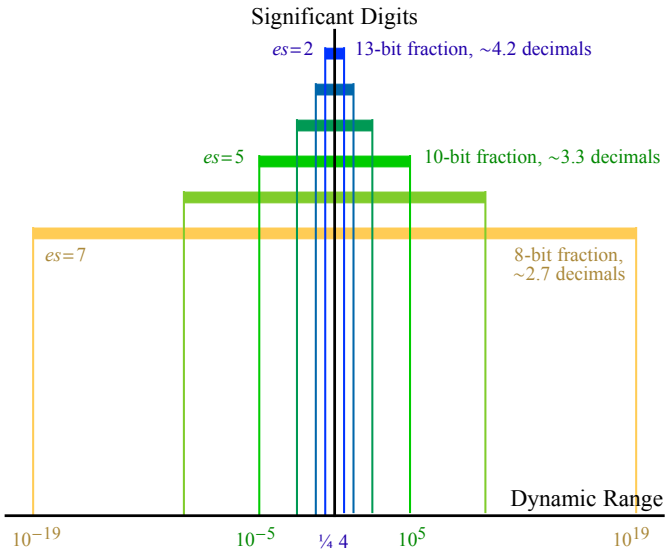

Fig. 2. 16-Bit Floats, Varying Exponent Size *es*

The IEEE 754 Standard Committee chose $es = 5$ for 16-bit floats; it is natively supported in Nvidia GPUs [3]. The low dynamic range has proved inadequate for Deep Learning (DL), leading to variations such as Google's "brain float" or "bfloat" [4] ($es = 8$, one step below the lowest accuracy format shown in Fig. 2), and IBM's DLFloat with $es = 6$ [5]. The low dynamic range of IEEE 754 16-bit floats is aggravated by the use of the top binade for infinity and NaN values instead of real numbers. Fig. 2 assumes the simpler (non-IEEE) form of floats with no subnormals and no NaN values.

What is the distribution of values needed for DL? A typical histogram is shown in Fig. 3 for AlexNet [6]. Notice that the vertical scale is logarithmic and that there are no values outside the range -0.5 to 0.5, and the distribution is tent-shaped rather than flat. Fixed-point approaches can be specified to cover the range from -0.5 to 0.5 but then show too much quantization error near 0.0, where the density of values needed is almost 10^4 greater than near -0.5 and 0.5.

For many technical applications, the density of values forms a tent shape. One reason a flat distribution is unlikely is that overflow is usually catastrophic and algorithms are designed to avoid numbers near the largest magnitudes. (To a lesser extent, underflow can also result in total loss of information.) Fig. 3 shows why flat distributions like those shown in fig. 2 are generally a poor match to what is needed.

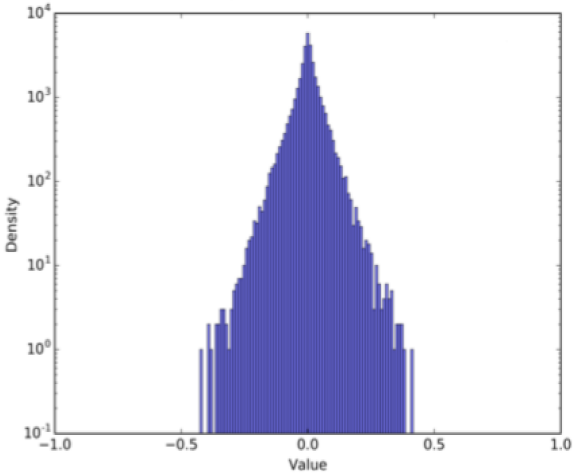

Fig. 3. Distribution of Real Values needed for Deep Learning (AlexNet)

II. An Attempt at Creating a Tapered Distribution: The Morris Floats

In 1981, Morris proposed that a bit field within a number format indicate the number of bits in the exponent field, with an offset [7]. For example, two bits suffice to indicate if the exponent field is 5, 6, 7, or 8 bits. Actual hardware to support this idea was built in 1989. The approach never received acceptance, and it is not difficult to discover the reasons it was unsuccessful.

First, the Morris approach leads to many redundant ways to express the same real number. As the exponent bit field size is changed, there are overlaps in the integer sets represented by the exponent and fraction fields. This reduces information per bit in the Shannon sense. Instead of creating a smoothly tapered distribution, the Morris floats create a rather fractal-like distribution of numbers created by the redundant values.

Second, no matter where the extra field is placed within the word, the simple task of *comparing* two values is made complicated. Redundant representations have to be converted to the maximum exponent and maximum fraction size just to compare them for $=$, $<$, or $>$ tests. At least with IEEE 754 floats, the values behave monotonically if we ignore the sign bit; that is, the value represented as a real is monotone in the value those bits represent as an unsigned integer.

After hardware for Morris floats was developed and tested in 1989, there was no adoption, probably because of the two disadvantages just described.

III. An Efficient Tapered Format: Posits

Posits were introduced at Stanford in 2017 [8]. They are a monotone increasing map of 2's complement integers to real numbers, with an efficient method of tapering that produces no redundant values. The method involves the concept of a *regime* field that is a run of identical bits terminated by either the opposite bit or the end of the format. The following shows how the regime represents an

integer k using a varying number of bits up to some maximum:

regime	run length	k
00000	5	-5
00001	4	-4
0001x	3	-3
001xx	2	-2
01xxx	1	-1
10xxx	1	0
110xx	2	1
1110x	3	2
11110	4	3
11111	5	4

The **x** bits are "don't care" bits that can be used to encode other information, and we follow the color-coding convention of [8] that the repeated bit is gold and the terminating opposite bit, if any, is brown. Notice that the regime bits in the above table are in lexical order, and thus monotone increasing with the k value. The ten integers represented have the same asymmetry as 2's complement integers, with the most negative value (-5 here) lacking a corresponding positive value but otherwise centered about 0. While the regime has been described as a form of Golomb-Rice encoding by Lindstrom et al. [9], it is neither Golomb-Rice nor a form of Huffman encoding since it is in a fixed number of bits known in advance. It resembles *unary* notation, except that it is able to represent signed integers.

While hardware for handling integers is obviously part of any hardware for a scaled number format like an IEEE float, less obvious is that float hardware must already support "Count Leading Zeros" (CLZ) operations, to renormalize (align the radix point) of the result of any addition or subtraction that cancels bits in the most significant bits of the fraction. A fast circuit for CLZ was published by Oklobzidja [10] with a logic delay that increases as $\lceil \log_2(n) \rceil$ where n is the number of bits, much like carry-lookahead and barrel shifters. Thus, regime decoding does not stray outside of the set of circuits that have been in standard use for decades.

If the **x** bits are used to encode an integer using the usual positional notation, this system provides a way to place two integers into a single fixed-bit field without requiring an extra bit field like that of the Morris floats. To provide a wider integer range, posits use the first *es* bits of the "don't care" bits as the *exponent* field, where any bits not visible are treated as **0** bits. The regime value is scaled by 2^{es}, and the exponent bits represent an unsigned integer e in positional notation that is added to the scaled regime value, as shown in the example below for $es = 2$. The tapered distribution is already evident; the integers are spaced farther apart at the most negative and positive values than near zero.

regime-exponent	k	e	$2^{es}k + e$
00000	-5	0	-20
00001	-4	0	-16
00010	-3	0	-12
00011	-3	2	-10
00100	-2	0	-8
00101	-2	1	-7
00110	-2	2	-6
00111	-2	3	-5
0100x	-1	0	-4
0101x	-1	1	-3
0110x	-1	2	-2
0111x	-1	3	-1
1000x	0	0	0
1001x	0	1	1
1010x	0	2	2
1011x	0	3	3
11000	1	0	4
11001	1	1	5
11010	1	2	6
11011	1	3	7
11100	2	0	8
11101	2	2	10
11110	3	4	12
11111	4	5	16

For posit notation, the "don't care" bits are the fraction, after a hidden bit that is always **1** even when no exponent or fraction bits are visible. A sign bit precedes the regime bits with the usual meaning, **0** if positive, **1** if negative. There are two exception values: all **0** bits for (unsigned) zero, and **1** followed by all **0** bits for Not-a-Real (NaR) exceptions. Fig 4 shows how posits have a tent-like accuracy envelope, in contrast to the flat envelope shown for the floats in Fig. 1.

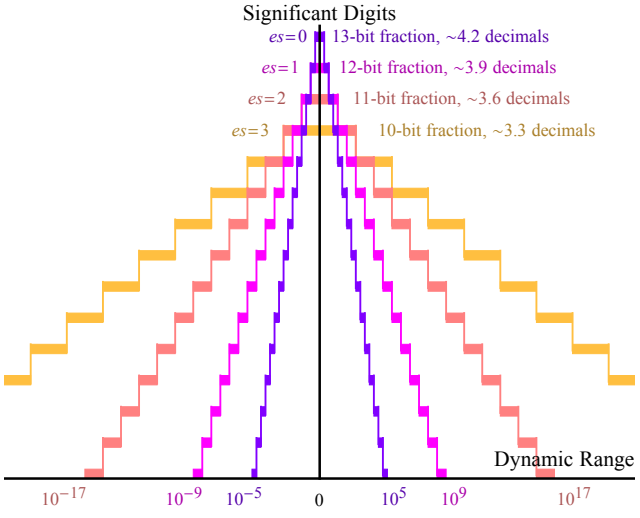

Fig. 4. 16-bit posits, varying exponent size *es*

Studies have shown that posits can achieve more accurate answers than floats with the same number of bits [6], [8],

[9], [11]; the advantage of more significant bits in the center of the magnitude range appears to dominate the disadvantage of lower accuracy at the extremes of the magnitude range. The tented shape of a posit distribution appears to more closely resemble the distribution of numbers used in many applications. Given the total number of bits, there is still only a single degree of freedom that we can adjust, the *es* value. For machine learning tasks like the one in Fig. 3, both floats and posits waste bit patterns on large magnitude real numbers, when we clearly need only to cover the range -0.5 to 0.5. Yet, fixed-point approaches that cover just that range of values do a poor job for that workload because of insufficient accuracy near 0. Is there another parameter that would allow us to craft even better distributions to match workload needs, without giving up the simplicity and hardware-suitability of those formats?

IV. The Regime Size Parameter

Posits allow the regime to be as large as the entire word after the sign bit, leaving only the hidden bit as a significant bit. We can restrict the regime size of an n-bit format to some maximum number of bits, *rs*, where $1 \leqslant rs \leqslant n-1$. When $rs = 1$ or $rs = 2$, there is **no variability in the number of regime bits.** Imagine a 16-bit format that is posit-like but with $rs = 2$ and $es = 3$. The smallest representable positive number is `0000000000000001`. Its regime and exponent bits represent $2^{es}k + e = 2^3 \cdot (-2) + 0 = -16$, and the hidden bit and fraction bits represent $1 + 2^{-10}$. The largest representable positive number is `0111111111111111`. Its regime and exponent bits represent $2^{es}k + e = 2^3 \cdot 1 + 3 = 15$, and the hidden bit and fraction bits represent $2 - 2^{-10}$. That is, the dynamic range is from $0.0000153\cdots$ to 65504, extremes which are almost exact reciprocals so the representation is symmetric about 1 (that is, centered at magnitude 0).

The restriction to $rs = 2$ *creates a float format*. If we allow the regime to be as large as everything past the sign bit, $rs = n - 1$, the format becomes the posit format as described in the Draft Posit Standard [12]. The *rs* parameter allows us to dial the distribution from flat to tent-like. While posits lose all but one bit of significance at extreme magnitudes, reducing *rs* assures there are always visible fraction bits, raising the lower bound on the accuracy across the entire range. Since theorems about floats often depend on such a bound, such an intermediate form addresses some of the criticisms of posit format [13]. We can call the adjustable *rs* variation of posits the *generalized posit format*.

Note that while the float format has exactly the same fraction size and almost exactly the same dynamic range as an IEEE 754 Standard binary16 float, the largest binade is not used for NaN and infinities and the smallest binade is not used for subnormals. This is similar to IBM's 16-bit DLFloat [5] and some of the ARM variants of 16-bit floats. The lack of subnormals in a float environment means it is possible for $|x - y|$ to underflow to zero even when $x \neq y$; this cannot happen in posit environments because too-small results are rounded to the smallest nonzero real instead of to 0.

Fig. 5 shows the transition of distributions from posits to floats with the *rs* and *es* parameters adjusted so that all four formats have approximately the same dynamic range. Notice that the vertical scale is logarithmic as in previous figures; since the bin size is a binade, there is no jaggedness and the stepwise nature of the density of values is easier to see.

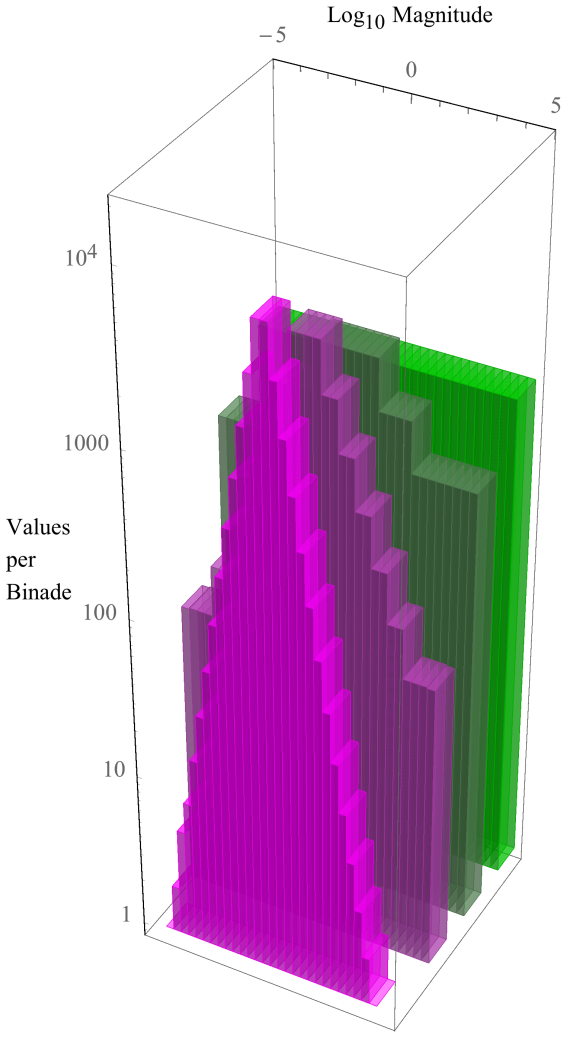

Fig. 5. 16-bit generalized posits, from posit-like to float-like

rs	*es*	Range	Accuracy Profile	color
15	0	8.4 decades	0.3–4.2–0.3 decimals	
8	1	9.6 decades	1.8–3.9–1.8 decimals	
4	2	9.6 decades	2.7–3.6–2.7 decimals	
2	3	9.6 decades	3.3 decimals (flat)	

The tent-like posit distribution (in front, magenta) peaks at 8192 values per binade, because there are 13 significant bits for numbers between $1/2$ and 2 in magnitude. The density steps down to 4096 values per binade for the bins next to the peak, for numbers between $1/4$ and $1/2$ or 2 and 4 in magnitude, and so on, dwindling to just one value per binade at the smallest posit ($1/16384$) and largest posit (16384). Notice that the pattern in the preceding table suggests $rs = 16$ in the

first entry, but this is not possible without sacrificing the sign bit or using a 17-bit word; hence, the dynamic range for the first entry is slightly smaller than the others.

The second entry already looks quite a bit more like the distribution needed for Machine Learning, and in the extreme magnitudes the wobbling accuracy is from 1.8 to 2.1 decimals, safer than what might happen if the calculation wanders into the very low accuracy extremes of the pure posit format. The step is twice as wide as the other steps for those extreme magnitudes because the limit on *rs* causes the bit patterns to contribute to accuracy instead of range. The same step widening is visible in the third case with $rs = 4$.

V. The Scale Parameter

The formats of the previous section have magnitudes centered about 0 (that is, values x such that $|x| \approx 1$). It is quite common for an application to center its distribution about a different number, as we see in Fig. 3. We can include an integer scaling parameter to accomplish this; call it *ebias* for "exponent bias" since it is the effective bias in the power-of-2 scaling expressed by the regime and exponent bits. An *ebias* of -3, say, would scale all values by $2^{-3} = 1/8$. The dynamic range is the same number of decades, but shifted toward smaller values. The *ebias* plays the same role for generalized posits as the parameter in a fixed-point number that says where the radix point is. It is easy to build hardware for such scaling because it needs only to shift significands left or right, or offset the exponent values with an add or subtract.

Since the earliest days of computer floating-point (going back as far as the ingenious designs of Konrad Zuse of the late 1930s) formats tend to center the exponent ranges near 0, perhaps motivated by a desire for better closure under reciprocation. The center of the IEEE 754 range for normal floats is 2, not 1. As long as an application does not approach overflow or underflow, the center for a floating-point format can be changed with *ebias* without changing the answers. For the tapered accuracy types, on the other hand, it can be beneficial to use *ebias* to move the peak of the format density (where accuracy is highest) closer to the peak of the distribution of numbers needed by the application.

We can use the *ebias* parameter and the *es* parameter in combination to increase accuracy at the expense of reduced dynamic range. For example, the IEEE-type 16-bit float can be modified from the standard $es = 5$ exponent bits to $es = 4$, reducing the dynamic range to $(minreal, maxreal) = (0.015625, 255.9375)$, but we can bias the exponent with $ebias = -9$ so that all values are scaled by $1/512$. The dynamic range then becomes approximately 0.00003 to 0.5, which covers the values needed in Fig. 3. Doing so increases the accuracy of the float by one bit, or about 0.3 decimals. We will later see that Fast Fourier Transform (FFT) calculations on bounded data can use formats capable of largest numbers far smaller than the largest numbers of floats (or posits). Signal processing in general can benefit from the range-accuracy tradeoff provided by lowering *ebias* and *es* in tandem to the point where the calculation just barely avoids overflow.

VI. A New Format: The *Taper*

Instead of making an even more complicated format, we can test one that is simpler and that applies some of the previous ideas to the traditional *fixed-point* format. Consider a fixed-point format; for consistency, we can again assume 16 bits, and we will put the radix point 9 positions from the end of the number (see Fig. 6):

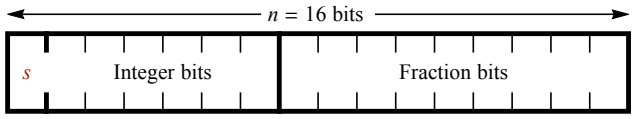

Fig. 6. 16-bit fixed-point example, with a 9-bit fraction

The entire number is 2's complement, not to be confused with sign-magnitude where the sign value s is independent of the interpretation of the other bits. The range of values expressed by this fixed-point format is -32768×2^{-9} to 32767×2^{-9}, or -64 to 63.998046875. The smallest positive value expressible is $1/512 = 0.001953125$.

If a number system can approximate real values x with approximate values x_a that are at most half of a Unit in the Last Place (ULP) away, then the maximum relative error is $0.5\ \mathrm{ULP}(x)/x$. The accuracy can be defined as the reciprocal of the relative error, and the number of decimals of accuracy is the log base 10 of that accuracy. Fig. 7 shows a plot the decimals of accuracy for any fixed-point number system.

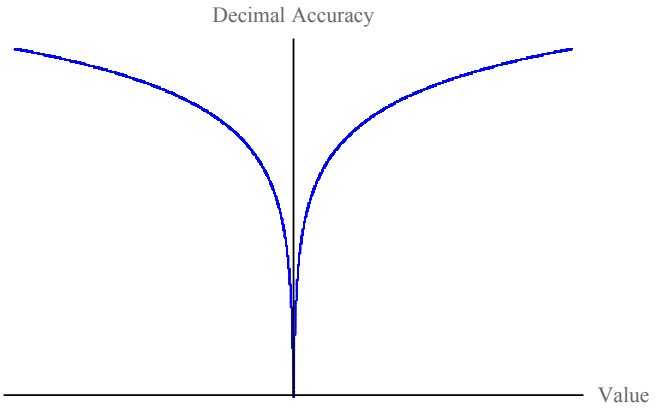

Fig. 7. Decimal accuracy envelope for fixed-point formats

The accuracy plummets near 0, which is often the opposite of what is needed in scientific computations; hence the need for scaled formats. However, there is another way to achieve higher density of values near 0 than to scale a significand by a power of 2. We can use the regime idea to divide a fixed-length bit string into two variable-size bit fields efficiently, but instead of using the first integer as the exponent and the rest as hidden bit plus fraction, we can simply treat the first part as an integer and the second part as a fraction to be *added*, not scaled. Call this the *taper* format. Consider the following example of the 32 values expressible with a 5-bit taper format:

Bit String	i	f	$i+f$
10000	-5	$0/2^0$	-5
10001	-4	$0/2^0$	-4
10010	-3	$0/2^1$	-3.0
10011	-3	$1/2^1$	-2.5
10100	-2	$0/2^2$	-2.00
10101	-2	$1/2^2$	-1.75
10110	-2	$2/2^2$	-1.50
10100	-2	$3/2^2$	-1.25
11000	-1	$0/2^3$	-1.000
11001	-1	$1/2^3$	-0.875
⋮	⋮	⋮	⋮
11111	-1	$7/2^3$	-0.125
00000	0	$0/2^3$	0.000
00001	0	$1/2^3$	0.125
⋮	⋮	⋮	⋮
00111	0	$7/2^3$	0.875
01000	1	$0/2^2$	1.00
01001	1	$1/2^2$	1.25
01010	1	$2/2^2$	1.50
01011	1	$2/2^2$	1.75
01100	2	$0/2^1$	2.0
01101	2	$1/2^1$	2.5
01110	3	$0/2^0$	3
01111	4	$0/2^0$	4

The decoding of the regime is as before except that the sign bit is flipped and regarded as the first bit of the regime. In this example, the regime size is the maximum possible, $rs = n = 5$ (including the flipped sign bit as the first bit of the regime). The extreme values have 3 more regime bits than the values near 0, which means the density of values is 2^3 higher near 0 then at the extreme values. See Fig. 8.

The example represents the maximum amount of tapering. As with generalized posits, we can place a regime size limit *rs*, and if we choose a limit as low as $rs = 2$, the regime is always the same length and the format becomes a fixed-point format with no tapering in the density plot. We also parameterize an integer *ebias* that scales the represented values by 2^{ebias}. It serves a purpose similar to the number that describes the position of the radix point in a fixed-point representation. Another way we could describe taper format is *generalized fixed-point format*, but this seems like an oxymoron since the location of the radix point is only fixed when $rs = 2$.

By trial and error, we adjust the word size $n = 12$, maximum regime size $rs = 9$, and power-of-two scaling $ebias = -4$ to create the distribution of expressible values shown in Fig. 9. It is a nearly perfect match to the distribution of values needed for Machine Learning as shown in Fig. 3.

VII. An Error Type for Integers and Fixed Point

While tapers are a superset of 2's complement signed integer and fixed-point format, we would not use them for tasks such as addressing. They are for approximating real numbers in a smaller range than a scaled format. For that reason, it may

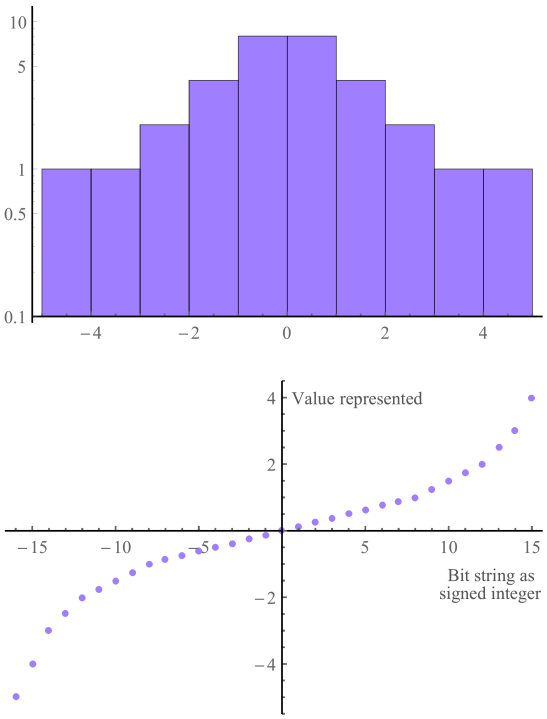

Fig. 8. Histogram and represented values for a 5-bit taper

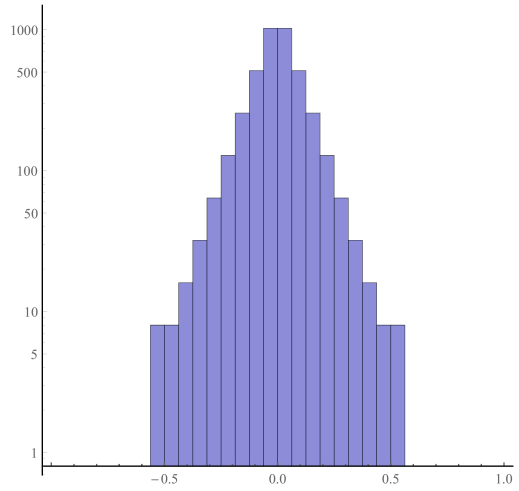

Fig. 9. Histogram of a taper format adjusted for Machine Learning

be prudent to reserve the most negative value, 1000···000, to indicate an error, similar to NaN for floats and NaR for posits. Call the taper exception value *Err*. An Err value is produced when dividing by 0, computing a value outside of range, and performing any calculation with Err as an input, thus propagating the exception through the calculation to the end. Having an error value, as opposed to processor flags for error conditions, has several benefits.

First, it makes the system closed under negation. The negative of Err is also Err, and that happens automatically when using the usual rule for negating a 2's complement number (flip all bits and add 1 to the least significant bit, ignoring any

overflow). It also makes the absolute value function closed. The set of all representable values is perfectly symmetrical about 0.

Second, it provides deterministic preservation of a NaN or NaR value when converting a number to an integer. In the current (2019) version of the IEEE 754, conversions of a NaN to an integer throws an exception, but does not produce an integer with a deterministic value [1]. This can lead to irreproducible results and unportable software, particularly when a programmer opts to suppress exception-handling. A NaN should always propagate to the final answer to alert the programmer to a failure of some kind, but converting to an integer and back to a float erases the NaN and turns it into some finite value (albeit unspecified in the Standard).

Third, it gives generalized posits and tapers the same bit patterns for exceptions. Like posits, the error value is equal to itself (unlike NaN) and in conditional ordering tests, tests as less than any other value. This allows the conditional tests for generalized posits, tapers, and signed integers to use the same hardware instructions for all conditional tests.

Finally, it prevents a defective calculation from masquerading as a correct one in the final output. With integer and traditional fixed-point, it is very easy to perform a calculation that goes outside the range, but it wraps to another integer value and continues. We still need traditional integers to do this for modulo-type calculations like pseudo-random number generation, but if we are using tapers to approximate a range of real numbers, we need to communicate any intermediate error to the final output. Whether to stop at the first Err value or continue can be a run-time option, providing some of the functionality of signaling NaN and quiet NaN in IEEE 754, respectively.

VIII. Case Study: The Fast Fourier Transform

The Discrete Fourier Transform (DFT) has a multitude of uses in technical computing, including signal and image processing, and solution of partial differential equations (PDEs). For PDEs, we typically want at least five or six decimals of accuracy. For signal processing, input data frequently is fixed-point since it comes from an analog-to-digital convertor (ADC) with 12 or fewer significant bits. Let's focus on the signal processing application here, where we would like to make a 16-bit format suffice; we will show that standard 16-bit floats, for example, do not suffice to preserve the information of a 12-bit DAC signal. The preceding sections may allow us to craft a format with a value distribution that more closely matches what is needed.

Suppose the signal has a mean of 0 and a Gaussian distribution with a standard deviation of $\sigma = 0.25$; assume that values x outside the range $-1 < x < 1$ are clipped to fit that range, where the ADC digitizes values to $-\frac{2047}{2048}, -\frac{2046}{2048}, \ldots, -\frac{1}{2048}, 0, \frac{1}{2048}, \ldots, \frac{2046}{2048}, \frac{2047}{2048}$. We generate 1024 real and imaginary values with that distribution and in this case do not need to clip any of the rare values beyond 4σ (see Fig. 10).

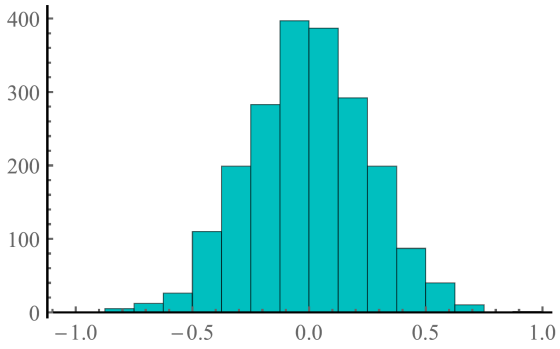

Fig. 10. 12-bit signal input data

The radix 4 *Fast Fourier Transform* (FFT) factors the DFT such that the number of multiplication and addition operations is approximately $4.5N\log(N)$ for a vector of N complex numbers, 10 percent fewer operations than a radix 2 FFT. We use the symmetrical definition for the forward and inverse DFT:

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-i2\pi}{N} kn} \tag{1}$$

$$x_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} X_n \cdot e^{\frac{i2\pi}{N} kn} \tag{2}$$

By normalizing both the forward and inverse transforms by $1/\sqrt{N}$, the two equations differ only in the sign of the exponent of e and the interchange of x and X. The radix-4 (decimation in time) factorization creates an FFT with $\log_4 N$ passes. Each pass performs a set of dot products of four values with four trigonometric values from a precomputed table, sometimes called "twiddle factors."; by incorporating a factor of $\frac{1}{2}$ into the twiddle factors, the normalization by $1/\sqrt{N}$ happens automatically. Besides saving the cost of the multiplication at the end of the calculation, this tends to keep intermediate calculations from growing in magnitude unnecessarily, which helps exploit number formats that have more accuracy at low magnitudes.

Fig. 11 shows the distribution of every value that occurs in performing a radix-4 FFT on the signal input data distributed as shown in Fig. 10. The theoretical maximum possible magnitude is $\sqrt{N}$, or 32 for the case here of $N = 1024$. This happens when all of the signal energy is concentrated into a single frequency; the distribution suggests we may be able to tolerate clipping of values above a threshold smaller than $\sqrt{N}$ in practice. It is obvious that any format that uses most of its bit patterns to represent values with large magnitudes, such as conventional floats, is inefficient at expressing the set of values shown in Fig. 11.

We summarize the error as the vector distance (Euclidean norm) between the original signal data and the result of a forward and inverse FFT of that data. We can also plot the error of each of the 2048 points (real and imaginary parts). Fig. 12 shows such a plot for a standard 16-bit IEEE floats. The error

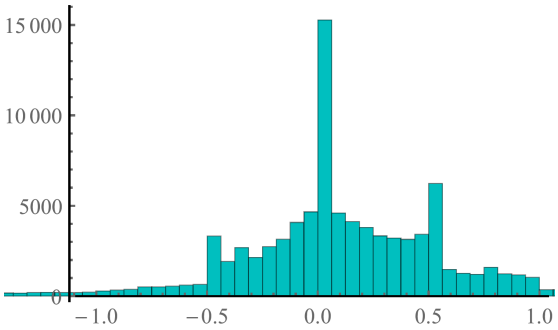

Fig. 11. Distribution of *all values used* in radix-4 FFT

is 0.0604. The horizontal banding reflects the discrete nature of the format. Recall that the original data has an ULP spacing of $\frac{1}{2048}$; if the results are rounded to the nearest 12-bit DAC value, it will differ if the error is greater than 0.5 ULP, or about 0.000244. This means the round-trip calculation is quite lossy, with hundreds of points differing from their original values. FFTs are generally done with 32-bit floats, which have enough accuracy that a round-trip forward-inverse transform is *lossless* when rounded to the accuracy of the DAC.

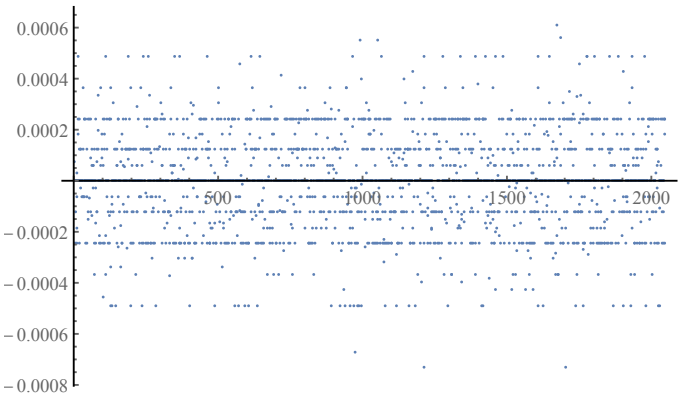

Fig. 12. Errors for 16-bit IEEE floats, FFT forward-inverse

Now that we have many parameters that can be adjusted, we can try them all and find the optimal error for 16-bit generalized posits. The best parameters are $rs = 14$, $es = 0$, and $ebias = -2$. The *rs* value of 14 is one less than the maximum, and with no exponent bits ($es = 0$, there is room for one more bit in the significand. Scaling all values by $2^{ebias} = 1/4$ centers the most accurate posits where they are most needed, reducing the waste of bit patterns on too-large values. Of the 65535 possible bit patterns for real values, only 254 of them represent values outside the range $|x| < 32$ that we need for the FFT. The error is reduced by almost an order of magnitude to 0.000029. Fig. 13 shows the point-wise errors using a vertical scale similar to that of Fig. 12. The worst-case error is well within the 0.5 ULP resolution of the 12-bit DAC, so this represents a lossless round-trip accuracy; there is no need for higher accuracy, since every bit of the original information is preserved.

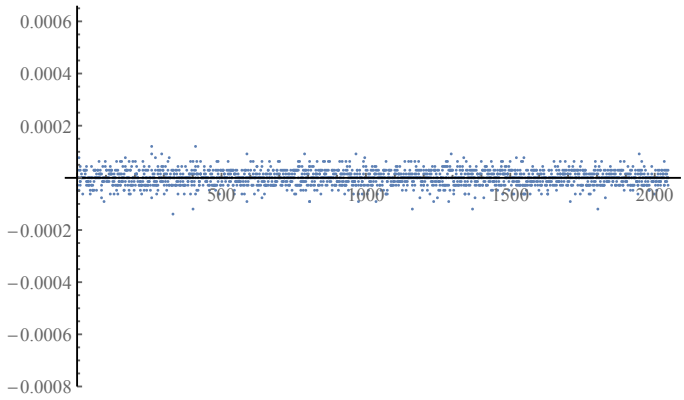

Fig. 13. Errors for 16-bit generalized posits, FFT forward-inverse

What about the *taper* format introduced earlier? If we are willing to risk some clipping for cases when most of the energy is in a single frequency, then a taper with $rs = 5$ and $es = -2$ does slightly better than the generalized posit, even though limited to x in $(-1.25, 1.25)$. It is much more accurate than a simple fixed-point format with the same number of bits. See Fig.14; the error is 0.000025. Since taper format appears simpler to build in hardware than the generalized posit format, it could have significant advantages for signal processing. We look forward to performing experiments with neural networks to see if tapers excel there as well.

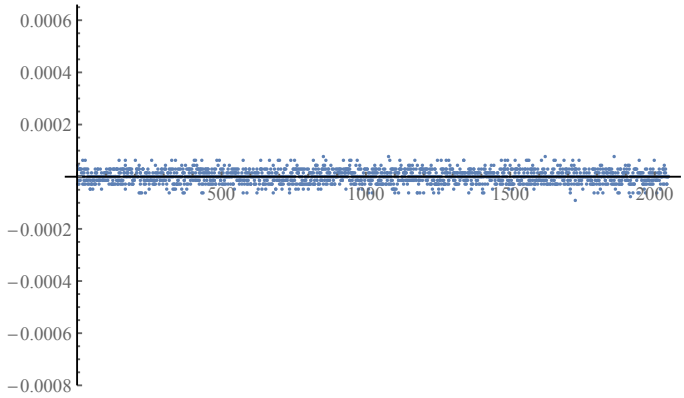

Fig. 14. Errors for 16-bit tapers, FFT forward-inverse

IX. Summary

We have introduced two frameworks for arithmetic formats. One generalizes float and posit format, and another generalizes fixed point and integer format. They allow adjustable amounts of tapering in their distribution of real values and adjustable bias in the mean magnitude. Together with the adjustable tradeoff between dynamic range and fraction precision, we can craft distributions that are an excellent match to the needs of particular applications. The resulting improved accuracy has been demonstrated for the FFT, and we plan to research this approach to improve accuracy on a wide range of scientific applications. □

REFERENCES

[1] 754-2019: IEEE Standard for Floating-Point Arithmetic. Online. Available: https://standards.ieee.org/standard/754-2019.html.

[2] Muller, J.-M. et al.: Handbook for Floating-Point Arithmetic, Second Edition. Springer, Cham Switzerland (2018).

[3] Nvidia: Nvidia Half Precision, 2020. Online. Available: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.

[4] Kalamkar, D. et al.: A Study of BFLOAT16 for Deep Learning Training. arXiv preprint arXiv:1905.12322 (2019).

[5] Agrawal, A. et al.: DLFloat: a 16-bit floating point format designed for deep learning training and inference. 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 92–95, IEEE (2019).

[6] Carmichael, Z. et al.: Deep positron: A deep neural network using the posit number system. Design, Automation and Test in Europe 2019. arXiv1812.01762v2.

[7] Morris, Sr., R.H.: Tapered floating point: A new floating-point representation. IEEE Transactions on Computers, Vol. C-20, Issue 12. (December 1971).

[8] Gustafson, J.L.: Stanford Seminar: Beyond floating point: Next generation computer arithmetic Online. Available: https://www.youtube.com/watch?v=aP0Y1uAA-2Y (February 2017).

[9] Lindstrom, P., Lloyd, S., Hittinger, J.: Universal coding of the reals: alternatives to IEEE floating point. CoNGA'18: Proceedings of the Conference for Next Generation Arithmetic, pp. 1–14. Online. Available: https://doi.org/10.1145/3190339.3190344 (March 2018).

[10] Oklobzidja, V.: An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis. IEEE Trans. VLSI Syst. DOI: 10.1109/92.273153 (1994).

[11] Klower, M., Duben, P.D., Palmer, T.N.: Posits as an alternative to floats for weather and climate models. CoNGA'19: Proceedings of the Conference for Next Generation Arithmetic, pp. 1–8. Online. Available: https://doi.org/10.1145/3316279.3316281 (March 2019).

[12] Posit Working Group: Posit Standard Documentation Release 3.2-draft June 23, 2018. Online. Available: http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libultim.3.

[13] de Dinechin, F., Forget, L., Muller, J.-M., Uguen, Y.: Posits: the good, the bad, and the ugly. hal-01959581v3 (2019).