Fractional-Exponent Floating-Point Numbers

Tianyun Zhang
tianyun2@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Franz Franchetti
franzf@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

Using fewer bits can reduce storage and data movement at the cost of numerical accuracy. Binary floating-point formats trade dynamic range against precision by reallocating whole bits between exponent and fraction fields. We introduce fractional-exponent floating-point numbers, allowing the base-two exponent to take fractional values. Changing the exponent step adjusts where representable values lie and how closely they are spaced, while keeping the exponent and fraction fields the same size. We characterize the range–precision tradeoffs and demonstrate their numerical benefit in a weighted-sum example. Fractional exponents open a continuum of range–precision choices at any fixed bit allocation, letting a format be tuned to the values it must represent.

1 Introduction

Representing neural-network weights and activations with fewer bits can reduce storage and data movement, but introduces a tradeoff between these savings and numerical accuracy. Smaller floating-point formats offer different ways to allocate the available bits to range and precision. Which of these choices is useful depends on the computation: a format needs enough range for the values that occur and sufficiently fine spacing to distinguish changes that matter.

The base raised to the power of the exponent determines the factor by which the significand is scaled. In standard binary floating point, the base is two. For a normal binary value, the significand lies between one and two, and the fraction bits select among evenly spaced significand values in that interval. For each exponent, this gives evenly spaced represented values between successive powers of two. Floating point therefore provides finer spacing at smaller magnitudes and coarser spacing at larger magnitudes.

For a fixed bit budget, the number of bits allocated to the exponent and fraction determines how many such intervals are available and how finely each is subdivided. Adding an exponent bit extends the exponent range, but removes a fraction bit, doubling the spacing within each interval. At low bit budgets, this whole-bit reallocation provides only coarse control over the range-precision tradeoff. A computation may need less range than one allocation provides, yet more than the next allocation can cover with normal values. Moving a whole bit does not let us choose how much of that excess range to exchange for greater precision.

We would like to trade range for precision as though we could use 3.5 exponent bits, instead of having to choose three or four. We explore this idea with fractional-exponent floating-point numbers. In binary floating point, each exponent increment doubles the multiplier applied to the significand. We instead increase the exponent by a positive fraction α, multiplying by 2^{α} at each increment. For example, $\alpha = 1/2$ gives a multiplier of $\sqrt{2}$ rather than two. The fraction field subdivides the smaller intervals between these exponent

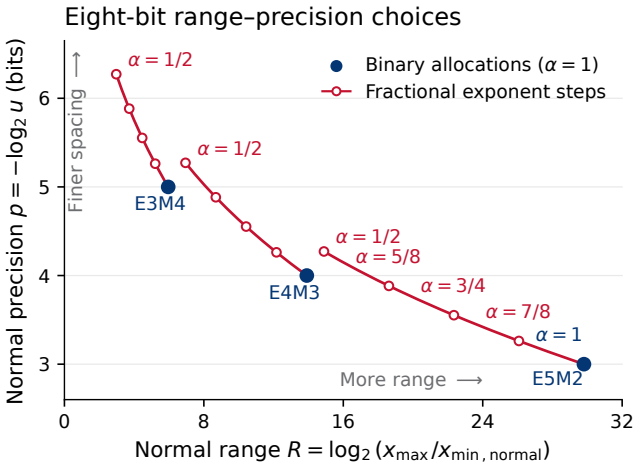


Figure 1: Eight-bit range-precision choices. Changing the exponent step α lets us trade range for finer spacing while keeping the numbers of exponent and fraction bits fixed. Filled navy points show binary formats ($\alpha = 1$); each red curve shows the choices available within one bit allocation. Marker labels on E5M2 apply to corresponding points on all three curves. The horizontal axis shows dynamic range R, and the vertical axis shows precision $p = -\log_2 u$, where u bounds relative rounding error for normal values.

levels, providing finer spacing over a smaller range. Figure 1 traces the choices obtained by varying α while keeping the numbers of exponent and fraction bits fixed.

The opportunity is to change where the representable values lie on the number line: how closely they are spaced near zero, through intermediate magnitudes, and towards the upper end of the range. A whole-bit reallocation changes this balance in one large step. Fractional exponents provide finer adjustments, giving us more choice in where to spend the available values.

The usefulness of these additional choices depends on the values being represented and how errors affect the computation. For example, sigmoid is more sensitive to input changes near zero than in its saturated tails. Finer spacing in that region may therefore be useful. Choosing a scale and a conventional format can already concentrate values near the center. The fuller requirement is to maintain useful spacing through the transition while still distinguishing values as the function approaches saturation. Once the spacing very near zero is fine enough for the required output accuracy, additional values there may be less useful than values farther along the transition or into the tails. Fine-tuning a representation to this balance is a potential application of fractional exponents.

The representation also retains the exponent-and-significand structure used in floating-point arithmetic. Multiplication adds the exponents and multiplies the significands, while division subtracts the exponents and divides the significands. Addition and subtraction require aligning the significands, which can involve multiplication by a fractional power of two, such as $1/\sqrt{2}$, as well as binary shifts. With a few fractional bits in the exponent, only a small set of such constants is needed. At low precision, these constant multiplications can be implemented with small lookup tables. For example, an eight-bit fixed-point magnitude can be split into two four-bit pieces, their products with a fixed-point approximation to the constant looked up separately, and the results combined with a shift and an addition. The partial products retain their full precision until they are combined. This gives a concrete approach to implementing fractional alignment.

These additional range-precision choices could support compact value storage in sparse computation. Sparse representations store both the positions and values of their nonzero entries. Previous work compresses the positions by recording distances between consecutive ordered entries, using fewer bits for short distances while preserving the positions exactly [10]. A complementary opportunity is to reduce the bits used for numerical values while meeting the computation's accuracy requirements.

Contributions. We make the following contributions:

- A fractional-exponent floating-point format redistributes values along the number line without changing the bit allocation.
- An analysis of the exponent step and bit allocation characterizes normal range and precision, identifying choices beyond whole-bit allocations.
- We demonstrate, through a weighted-sum example, how exchanging excess normal range for finer spacing over the input interval can improve numerical accuracy without increasing the bit width.

2 Background and related work

In standard binary floating point, the base is two, and a normal significand lies in $[1, 2)$. Because its leading 1 is implicit, only the fraction bits are stored.

Therefore, the encoding has three fields: sign, exponent, and fraction. With one sign bit, e exponent bits, and m fraction bits, the total width is $T = 1 + e + m$. Writing the sign bit as s, the fraction bits as the unsigned integer q, and the unbiased exponent as k, a normal binary value is

$$x = (-1)^s \left(1 + \frac{q}{2^m}\right) 2^k, \qquad q \in \{0, \ldots, 2^m - 1\}.$$

For each exponent k, there are 2^m evenly spaced positive normal values in $[2^k, 2^{k+1})$. For example, with three fraction bits, the values between 1 and 2 are spaced $1/8$ apart, while values between 2 and 4 are spaced $1/4$ apart. The absolute spacing is constant within each normal exponent interval and doubles from one interval to the next. Therefore, floating point provides finer spacing at smaller magnitudes and coarser spacing at larger magnitudes.

This arrangement combines features of fixed-point and logarithmic representations. The distinction is in how the spacing changes with magnitude. For adjacent positive values $x < x'$, the absolute spacing is the gap $x' - x$, and the relative spacing is this gap divided by x. Fixed point uses a constant absolute gap between represented values. For example, a fixed-point grid with spacing $1/8$ keeps that gap both between 1 and 2 and between 2 and 4. Its relative spacing decreases as the values grow, e.g., a gap of $1/8$ is one-eighth of 1 but only one-sixteenth of 2.

A uniformly spaced logarithmic number system represents positive values that are equally spaced on a logarithmic scale. For example, the values 1, 2, 4, and 8 have base-two logarithms 0, 1, 2, and 3. Each step doubles the value, so the absolute gaps grow from 1 to 2 to 4. Each gap is equal to the smaller value in its pair, so the relative spacing stays constant. Smaller steps in the stored logarithm give finer relative spacing.

Standard floating point uses constant absolute spacing locally within each exponent interval and places interval boundaries at successive powers of two. Within one interval, the fraction bits give a uniformly spaced grid, as in fixed point. From one interval to the next, the exponent scales that grid by a factor of two. Its relative spacing decreases within each interval and rises again at the next boundary.

Floating point combines these two spacing patterns. Values within an exponent interval are equally spaced, as in fixed point, while the interval boundaries are equally spaced on a logarithmic scale. The two fields determine how much range these intervals cover and how many values lie within each one. This is the structure behind the whole-bit range-precision choices described in the introduction.

Semi-logarithmic representations explore arithmetic tradeoffs between floating-point and logarithmic number systems. They combine an approximation to the base-two logarithm with a residual significand, rounded to a binary fractional grid. Keeping the residual small simplifies multiplication and division, while retaining it reduces the table sizes needed for addition and subtraction compared with a fully logarithmic representation [6].

The choice of base provides another way to control value spacing. Work on low-precision logarithmic representations examines how this choice affects conversion error, arithmetic error, and circuit cost. Its comparisons include different input distributions and matched maximum magnitudes [1]. The positive values in these formats are equally spaced on a logarithmic scale. Choosing the base changes the ratio between every adjacent pair.

Scaling, bit allocation, and codebook design provide other ways to adjust the range and spacing of representable values. AdaptivFloat selects an exponent bias for each neural-network layer, adapting the range to its values without changing the exponent-fraction allocation [11]. MX formats also adapt the scale to the data, with scaling shared within small blocks [2]. In standardized MX, each block has a common scale, allowing different blocks to cover different ranges while sharing the scale's storage cost [9]. Posits vary the number of fraction bits with magnitude, providing more fraction bits near magnitude one and fewer for much smaller or larger magnitudes [4]. Codebook quantization represents each weight by an index into a table of quantization levels. Deep Compression constructs this table by clustering trained weights and then fine-tunes its entries to reduce the network loss [5]. QLoRA's NF4 instead uses a fixed codebook designed for normally distributed weights, combined with block scaling [3].

We aim to preserve familiar floating-point spacing while allowing fractional exponent steps. This builds on SLNS's representation, with a different choice for how the fraction bits place values between consecutive exponent levels [6]. Changing the exponent step changes both the interval boundaries and the spacing within them. This lets us exchange excess normal range for finer spacing without moving a whole bit between fields. Our representation also retains a separate significand, so addition and subtraction can align significands as in floating point. With a few binary fractional bits in the exponent, alignment combines binary shifts with multiplication by a small set of fractional powers of two.

3 Number representation

We use fractional values of the base-two exponent to adjust the placement of representable values. The fraction field continues to select equally spaced values within each normal exponent interval, with the spacing adjusted to the interval's width. This lets us vary the interval boundaries and the spacing inside them without moving bits between the stored fields.

Let α be a positive fraction specifying the step between consecutive base-two exponents. Writing k for the integer index of an exponent level, the significand is multiplied by $2^{\alpha k}$. For compact notation, we write

$$2^{\alpha k} = (2^{\alpha})^k = b^k, \qquad b = 2^{\alpha},$$

so b is the factor between successive exponent levels. The representation allows any positive real α, but we focus on rational steps.

The word width remains $T = 1 + e + m$. The exponent field stores an unsigned integer E, and the fraction field stores an unsigned integer $q \in \{0, \ldots, 2^m - 1\}$. We use the exponent bias $B = 2^{e-1} - 1$, so the integer index is $k = E - B$ and the base-two exponent is αk. We consider $e \geq 2$ and $m \geq 0$.

For normal values, $1 \leq E \leq 2^e - 2$, the significand is $1 + (b-1)\frac{q}{2^m}$. The factor $b - 1$ adjusts the fraction spacing to the interval $[1, b)$. The normal value is therefore

$$x = (-1)^s \left[1 + (b-1)\frac{q}{2^m}\right] b^{E-B}.$$

The stored fraction remains a binary integer. Changing the base changes how this integer maps to a significand. It does not reinterpret the fraction bits as digits in base b.

For example, consider two fraction bits and the normal interval starting at 1. In binary floating point, the next interval starts at 2, so the four values are spaced $(2-1)/4 = 1/4$ apart. With $\alpha = 1/2$, the next interval starts at $2^{1/2}$, so the spacing becomes $(2^{1/2} - 1)/4$. In both cases, the fraction bits select values at the start, one-quarter, one-half, and three-quarters of the way through the interval. The upper endpoint starts the next interval. The same two fraction bits now place four values in a smaller interval, giving finer spacing.

Within the normal exponent interval $[b^k, b^{k+1})$, the 2^m values are equally spaced, with absolute spacing

$$\Delta_k = \frac{(b-1)b^k}{2^m}.$$

The spacing increases by a factor of b between successive intervals. At $\alpha = 1$, both the significand and exponent mappings reduce to those of normal binary floating point. When $m = 0$, the significand is one and the positive normal values are equally spaced on a logarithmic scale, with a ratio of b between adjacent values.

A shared scale $S > 0$ can multiply every represented value. This multiplies all absolute spacings and range endpoints by S, while leaving relative spacing and the ratio between range endpoints unchanged. The unscaled representation uses $S = 1$. The parameters α and S are shared settings, not fields stored in each value, so their storage cost is separate from T.

The exponent-zero field represents zero. With fraction bits, it also provides equally spaced subnormal values below the smallest normal magnitude, allowing small values to approach zero gradually. Zero has both positive and negative encodings, following the conventional sign-bit interpretation. Subnormal spacing can differ from that of the first normal interval when the exponent step changes. In the numerical examples, finite inputs round to the nearest represented value, with exact ties toward zero. Inputs outside the finite range saturate to the signed maximum normal value.

Low-precision formats make different choices about these encodings. OCP E5M2 supports infinities and NaNs, whereas E4M3 omits infinities and uses most encodings with an all-ones exponent for finite values [8]. The FNUZ variants omit infinities and use a single zero, assigning the negative-zero bit pattern to NaN [7].

For this first study, we reserve the all-ones exponent field for special values and keep these conventions fixed as the exponent step changes. Our E/M labels specify field widths, rather than full compatibility with these FP8 formats. Alternative uses of special encodings, their behavior during arithmetic, and the subnormal-to-normal transition remain design choices to examine as the representation is developed further.

With a common maximum, the difference between changing the exponent step and moving a whole bit becomes visible on the number line (Figure 2). Starting from binary E3M2, we can move one exponent bit to the fraction to obtain E2M3, or keep E3M2 and reduce the exponent step. The fractional adjustment gives smaller gaps at the upper end of the range than binary E3M2, while retaining finer spacing near zero than E2M3. All three formats use six bits. This provides an intermediate choice in near-zero spacing while changing how values are distributed farther along the number line.

To characterize the normal range-precision tradeoff, we use

$$R = \log_2 \frac{x_{\max}}{x_{\min,\text{normal}}}, \qquad u = \frac{b-1}{2^{m+1}}, \qquad p = -\log_2 u.$$

Here R is the base-two logarithm of the ratio between the largest and smallest positive normal values. The quantity u is the conventional relative-rounding-error bound obtained from half the spacing at the start of a normal interval. The precision measure p expresses this bound in bits: increasing p by one halves u. This bound describes rounding within the normal range and does not cover underflow or saturation.

At a fixed allocation, reducing the exponent step leaves the number of representable values unchanged. With a common maximum, the smaller step concentrates the normal values into a smaller range below that maximum. This gives the tradeoff shown in Figure 1:

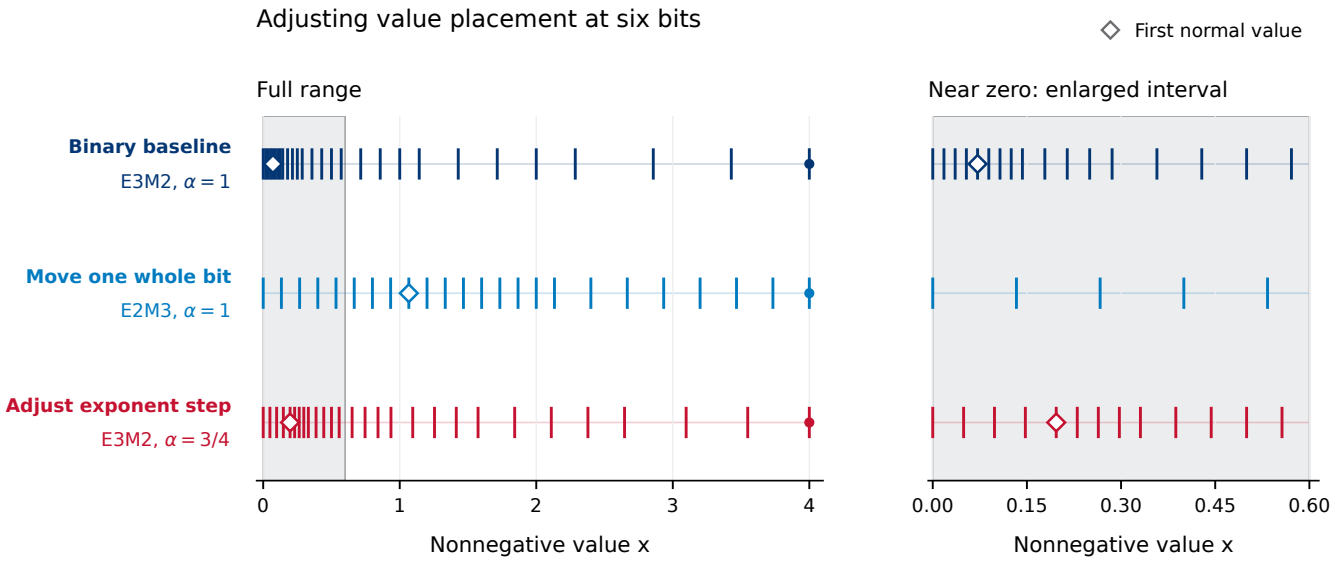


Figure 2: Nonnegative representable values for three six-bit formats, each rescaled to maximum magnitude 4. The right panel enlarges the shaded interval. Diamonds mark the first normal value. Positive subnormals lie to its left.

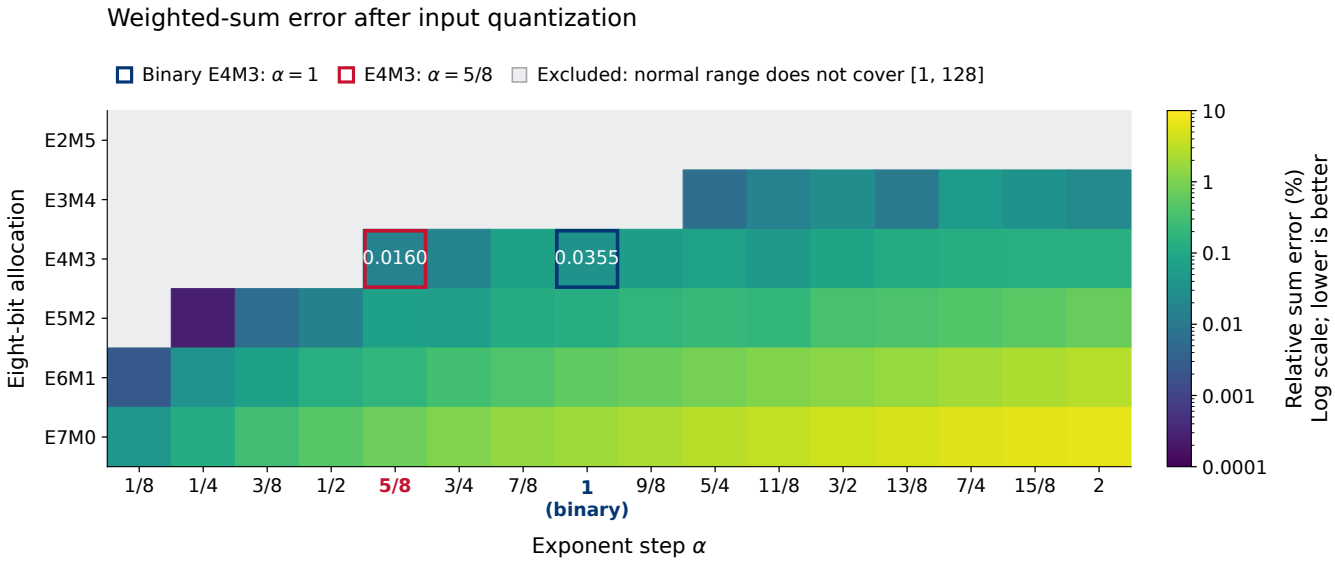


Figure 3: Weighted-sum error after quantizing inputs to eight-bit formats. The highlighted E4M3 pair compares binary $\alpha = 1$ with $\alpha = 5/8$, the smallest tested step providing normal coverage of $[1, 128]$. Gray cells lack this coverage.

normal range decreases while the bound on relative rounding error improves. Each curve varies the step within one allocation.

4 Numerical evaluation

Weighted sum. The number-line example shows how fractional exponents redistribute the available values. We now examine how this affects a weighted sum whose inputs span $[1, 128]$, using eight-bit input formats. We round the inputs to each format and evaluate the same weighted sum at high precision, so the comparison measures the effect of input representation.

We use $N = 2{,}048$ logarithmically spaced inputs, $x_i = 128^{i/(N-1)}$ for $i = 0, \ldots, N-1$, with weights $1/2$ at the endpoints and 1 inside. Each format is scaled to maximum magnitude 128. Weights and sums use binary64. We report the relative error against the unquantized-input sum.

We restrict this comparison to configurations whose normal range covers all inputs, since the fractional-exponent subnormal

mapping and its transition to normal values remain under investigation. This restriction excludes some formats that could achieve lower sum error by using subnormals.

With the maximum fixed at 128, binary E3M4 has insufficient normal range to cover the inputs, while E4M3 extends far below the smallest input. Changing the exponent step lets us reduce this excess range while retaining the coverage we need. Within E4M3, $\alpha = 5/8$ is the smallest tested eighth-step setting that meets this requirement. Reducing α from 1 to 5/8 increases the number of normal values inside the input interval from 57 to 90, without changing the bit allocation or maximum. For this sum, the relative error after input quantization falls from 0.0355% to 0.0160%.

Figure 3 shows the error across exponent-fraction allocations and exponent steps. We highlight the E4M3 pair to show what changing the exponent step provides within a fixed allocation. The heatmap reports the combined effect of input rounding on the sum. It also contains lower-error choices under other allocations. Rounding errors can offset one another in a sum, so finer spacing does not guarantee a smaller sum error. Here, the additional choice improves accuracy while preserving the required normal coverage and the same bit width.

5 Conclusion and future work

Fractional-exponent floating point lets us adjust where a fixed number of values lie on the number line, giving finer range-precision choices without moving a whole bit between fields. The normal-range analysis describes these choices, and the weighted-sum example shows that the best setting for a computation can lie between the whole-bit allocations.

If finer control over the representation lets a computation maintain necessary accuracy with fewer bits per value, it could reduce storage and data movement. A next step is to choose the exponent step, allocation, and scale together, and test whether choices made from representative data remain useful as the data change. Reductions and nonlinear functions can lead to tests of neural-network weights and activations, comparing application accuracy at equal total storage with conventional wider computation.

One direction is to share the format description across a vector of values. For example, ten six-bit values fit in a 64-bit word, leaving four bits to select among sixteen predefined configurations. Packed inputs could feed wider accumulators, with results converted back to the storage format when written out. Evaluating this organization requires counting any additional scale information and conversion costs along with the stored values.

An accompanying direction is a bit-accurate model of arithmetic in the fractional-exponent format. Such a model would show how much intermediate precision is needed for fractional alignment, normalization, and rounding, and whether the numerical benefit survives the full computation. It would also provide a basis for evaluating packed vector operations and FPGA or other hardware implementations. These studies would test when improved value placement can translate into fewer stored and transferred bits at the accuracy required by the application.

References

[1] Syed Asad Alam, James Garland, and David Gregg. 2021. Low-precision Logarithmic Number Systems: Beyond Base-2. *ACM Transactions on Architecture and Code Optimization* 18, 4, Article 47 (2021), 25 pages. https://doi.org/10.1145/3461699

[2] Bita Darvish Rouhani, Ritchie Zhao, Venmugil Elango, Rasoul Shafipour, Mathew Hall, Maral Mesmakhosroshahi, Ankit More, Levi Melnick, Maximilian Golub, Girish Varatkar, Lai Shao, Gaurav Kolhe, Dimitry Melts, Jasmine Klar, Renee L'Heureux, Matt Perry, Doug Burger, Eric Chung, Zhaoxia (Summer) Deng, Sam Naghshineh, Jongsoo Park, and Maxim Naumov. 2023. With Shared Microexponents, A Little Shifting Goes a Long Way. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 83, 13 pages. https://doi.org/10.1145/3579371.3589351

[3] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., Red Hook, NY, USA, 10088–10115. https://doi.org/10.52202/075280-0441

[4] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017), 71–86. https://doi.org/10.14529/jsfi170206

[5] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations (ICLR)*. 14 pages. https://arxiv.org/abs/1510.00149

[6] Jean-Michel Muller, Alexandre Scherbyna, and Arnaud Tisserand. 1998. Semi-Logarithmic Number Systems. *IEEE Trans. Comput.* 47, 2 (Feb. 1998), 145–151. https://perso.ens-lyon.fr/jean-michel.muller/IEEETC-Fev98.pdf

[7] ONNX. 2025. Float stored in 8 bits. ONNX documentation. Retrieved October 20, 2026 from https://github.com/onnx/onnx/blob/a69e5a227c66/docs/docsgen/source/technical/float8.md Source revision October 1, 2025.

[8] Open Compute Project. 2023. *OCP 8-bit Floating Point Specification (OFP8), Revision 1.0.* Open Compute Project. https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

[9] Open Compute Project. 2023. *OCP Microscaling Formats (MX) Specification, Version 1.0.* Open Compute Project. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf

[10] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/3352460.3358330

[11] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 2020. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, Piscataway, NJ, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218516