Lossless FFTs with Posit Arithmetic

Dr. Siew Hoon (Cerlane) Leong, CSCS

Prof. John L. Gustafson, ASU

March 1, 2023

Thesis

Fast Fourier Transforms (FFTs) for *signal and image processing* have format needs similar to those for Machine Learning… tent-shaped distribution bounded above but not below.

16-bit IEEE floats are **too lossy** to use for FFTs, so 32-bit is used.

16-bit posits are sufficiently accurate that FFTs followed by inverse FFTs can return the original signal *without loss*.

FFTs with **SoftPosit** and **SoftFloat** allow a fair comparison of the speed of posits with the speed of floats of the same precision.

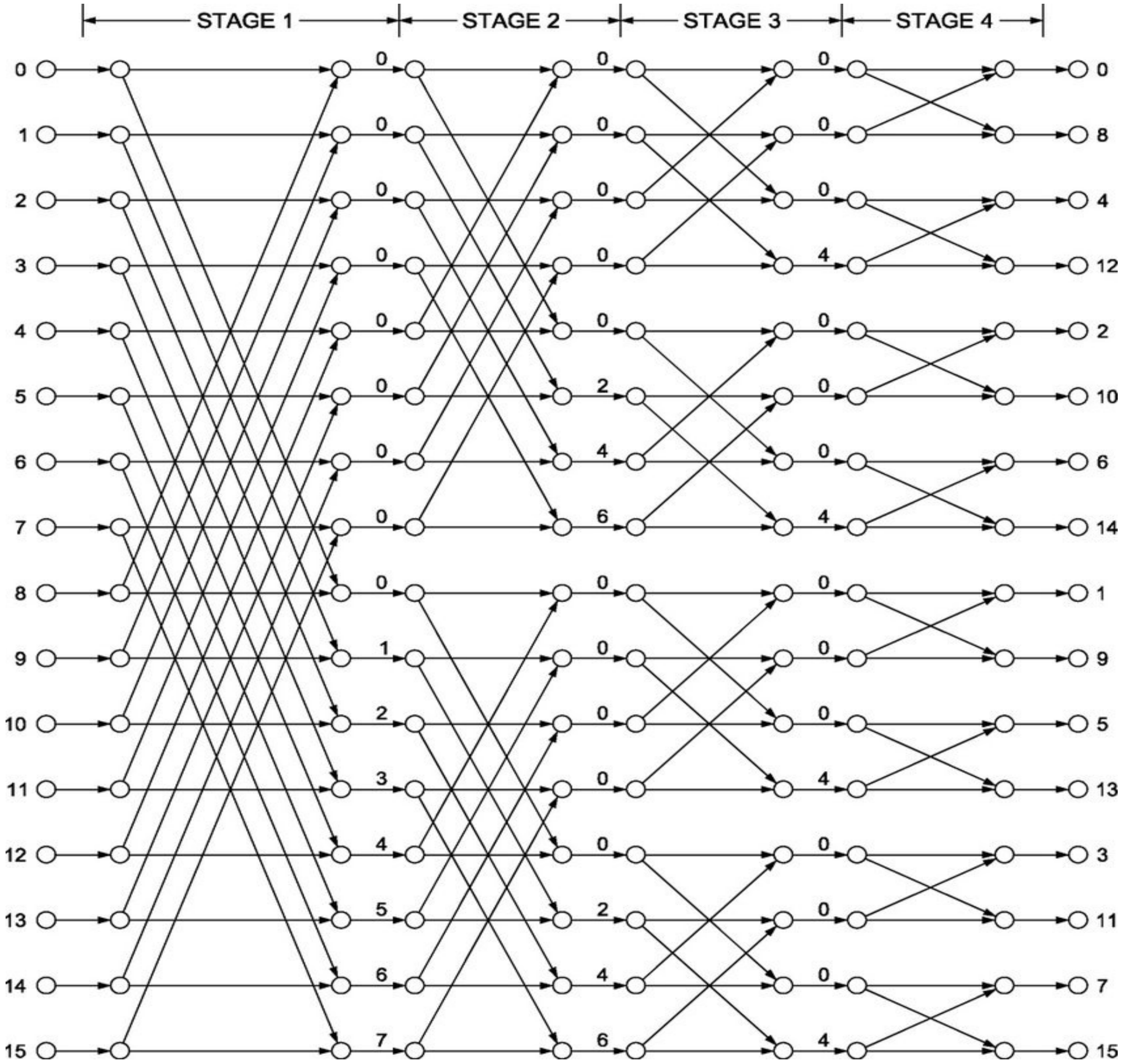
Typical Discrete Fourier Transform (DFT) Definition

Forward DFT: $$X_n = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N}$$

Inverse DFT: $$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{+2\pi i k n / N}$$

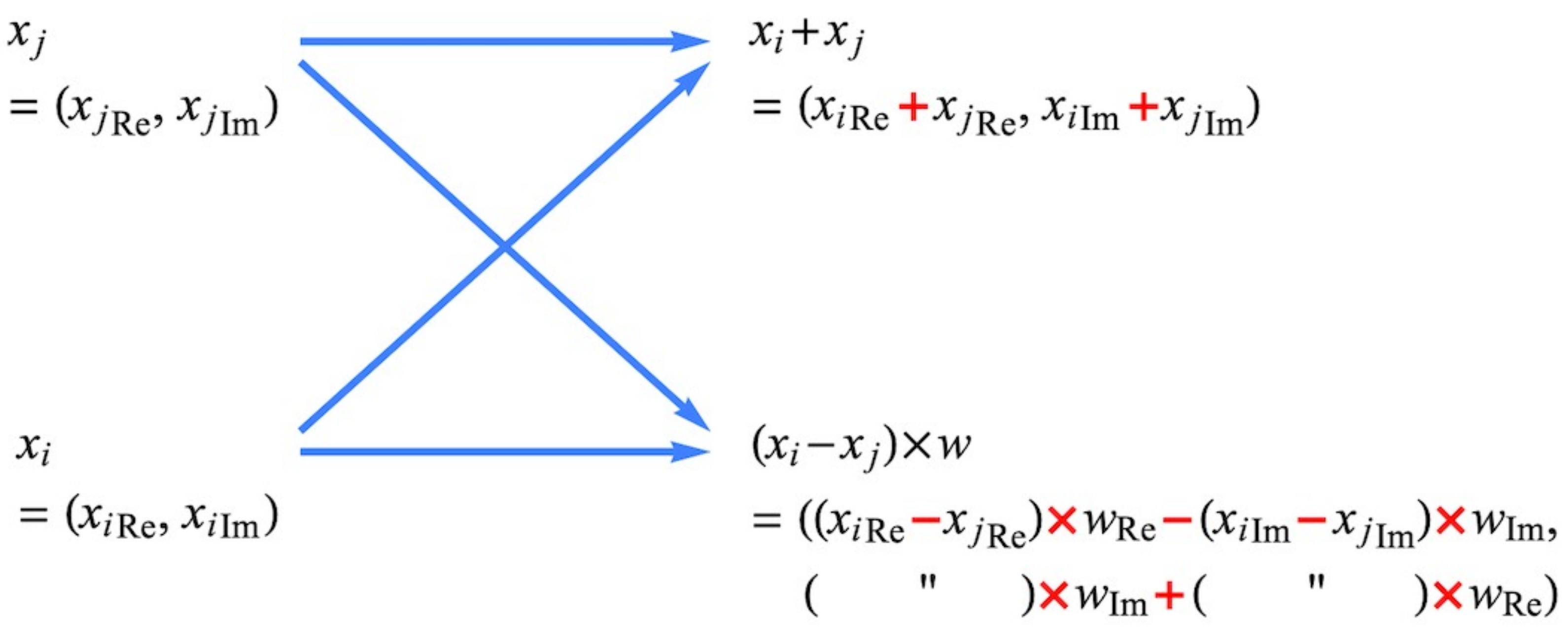
It looks like $O(N^2)$ work, but Gauss found a shortcut, the "Fast Fourier Transform."
Rediscovered by Bell Labs researchers Cooley and Tukey in the early 1960s.

FFTs are the "Achilles Heel" of HPC



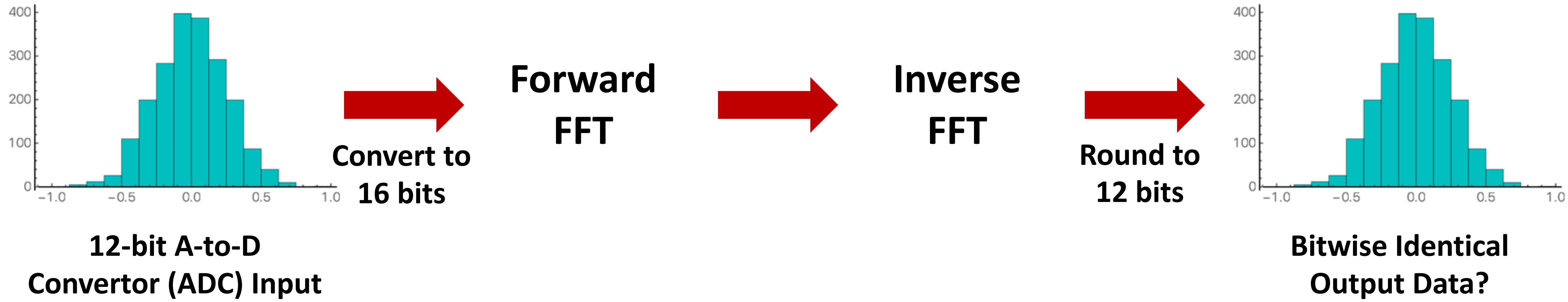
- Memory wall: $O(N \log N)$ operations, but $O\left(N^{4/3}\right)$ data motion
- TOP500 supercomputers are typically *thousands of times slower* at FFTs than at LINPACK.
- Solution: more info per bit in the data format!

The kernel FFT operation is a “butterfly.”



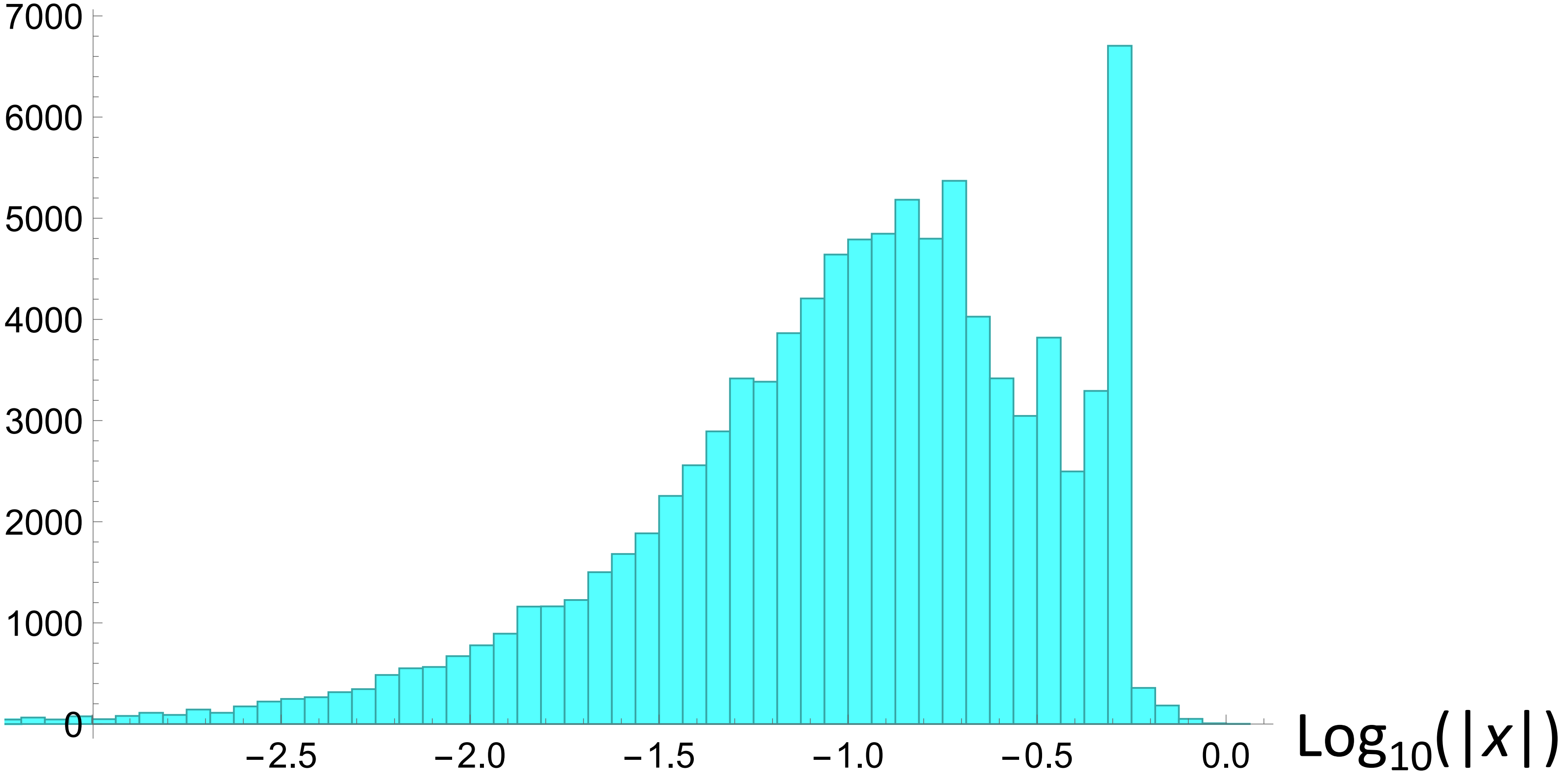
- ”Twiddle factors” $e^{\pm 2\pi i k n / N} = \cos\left(\frac{2\pi n}{N}\right) + i\,\sin(\frac{2\pi n}{N})$ are often written as w for short.
- A radix 2 FFT butterfly takes four multiplies, six add/subtracts in general.
- The radix 4 FFT butterfly is more complicated but uses 10% fewer operations.

Are *Lossless* FFTs possible with 16-bit formats?



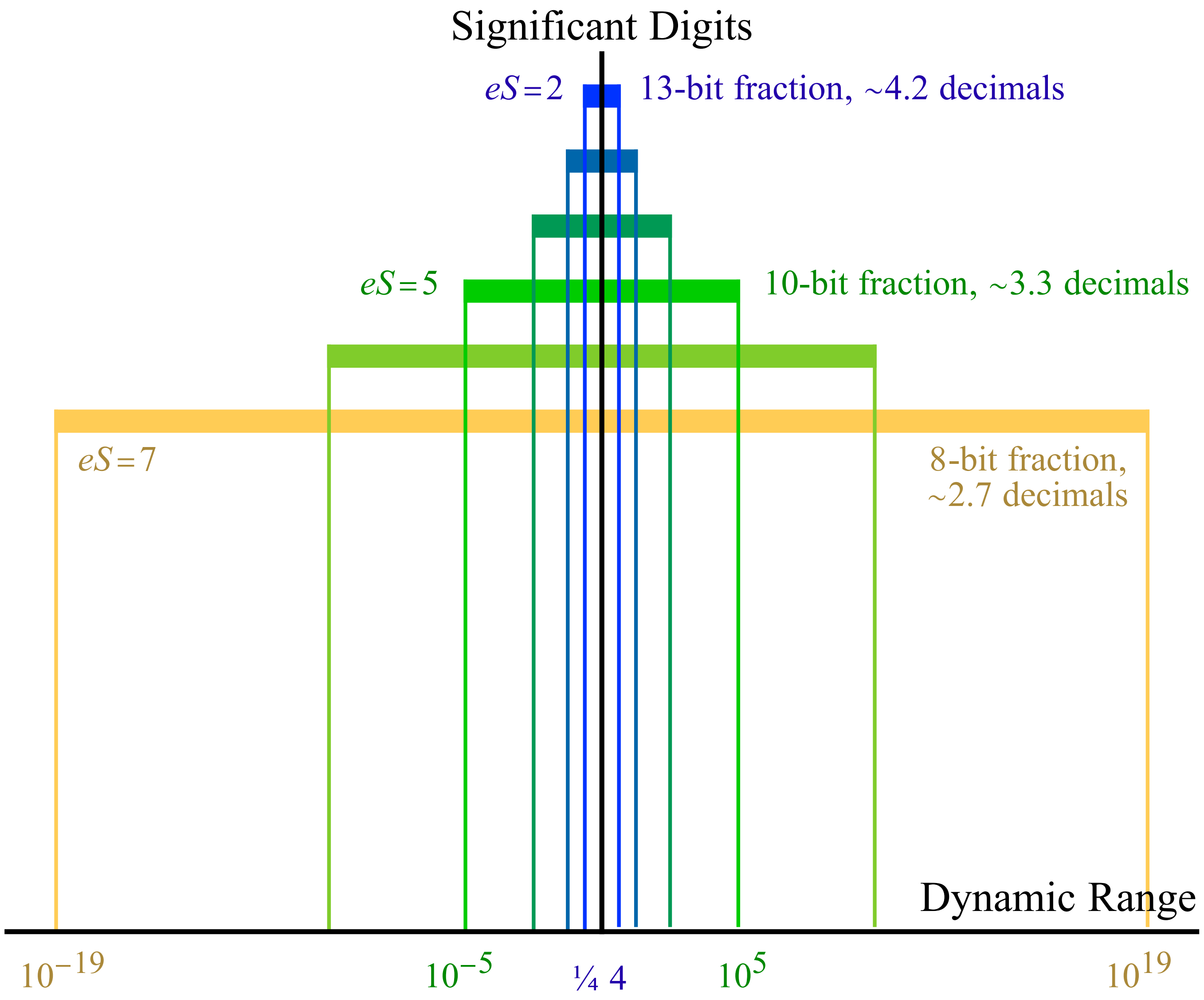
- We will show that 16-bit IEEE floats cannot do this.
- An "idealized" 16-bit float cannot, either. More on this later.
- Since the 1970s, image and signal processing have had to use 32-bit floats to prevent severe accuracy loss.

What values actually *occur* in a signal FFT?



A skewed tent-shaped distribution, provably bounded on the right.

Idealized 16-bit floats do not fit FFT distribution.

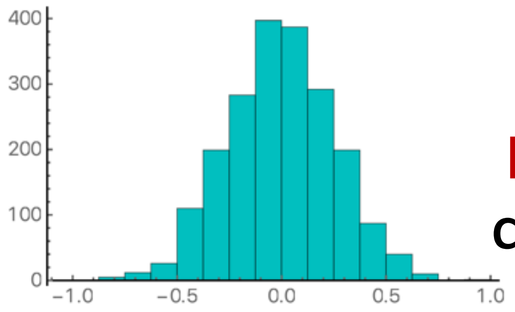


"Idealized" means

- only one NaN value
- only one zero
- largest and smallest exponent cases are treated as normal

We tried them all, and they are all very lossy.

New Approach Using 16-Bit Posits



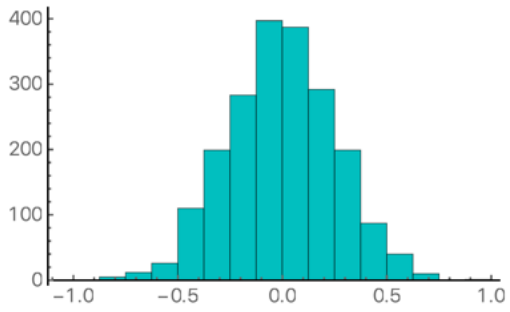
12-bit A-to-D Convertor Input

Convert to 16 bits

Forward FFT

Inverse FFT

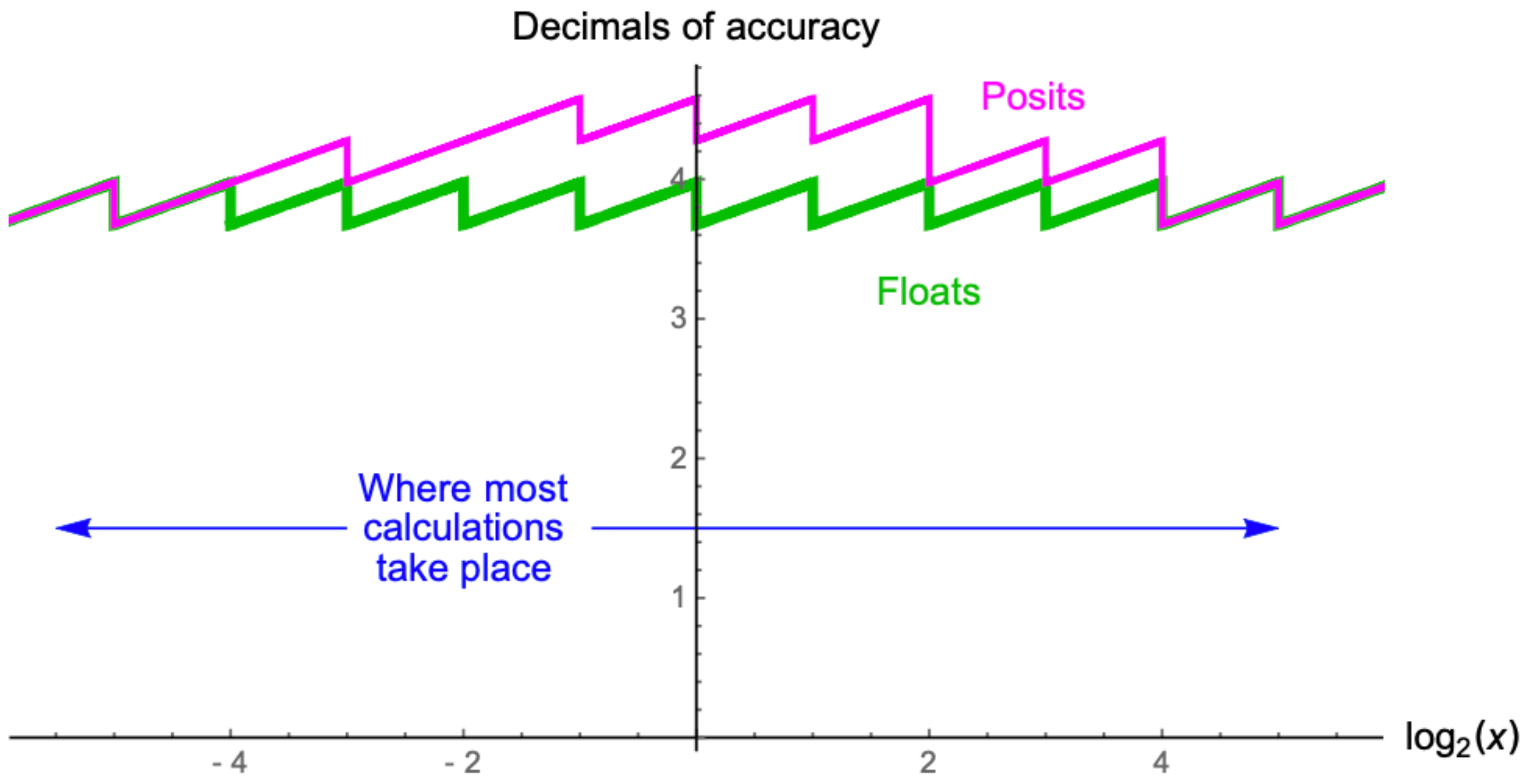
Round to 12 bits



Bitwise Identical Output Data?

- Tested N = 1024 and 4096 points
- Decimation-in-Time (*slightly* more accurate)
- Radix 4 (five or six “butterfly” passes)
- Value magnitudes cannot exceed $\sqrt{N}$ = 32 or 64.

Remember the posit "sweet spot"



With *eS* = 1, 16-bit posits have accuracy ≥ 16-bit IEEE floats for magnitudes 2^{-6} to 2^{6} (1/64 to 64).

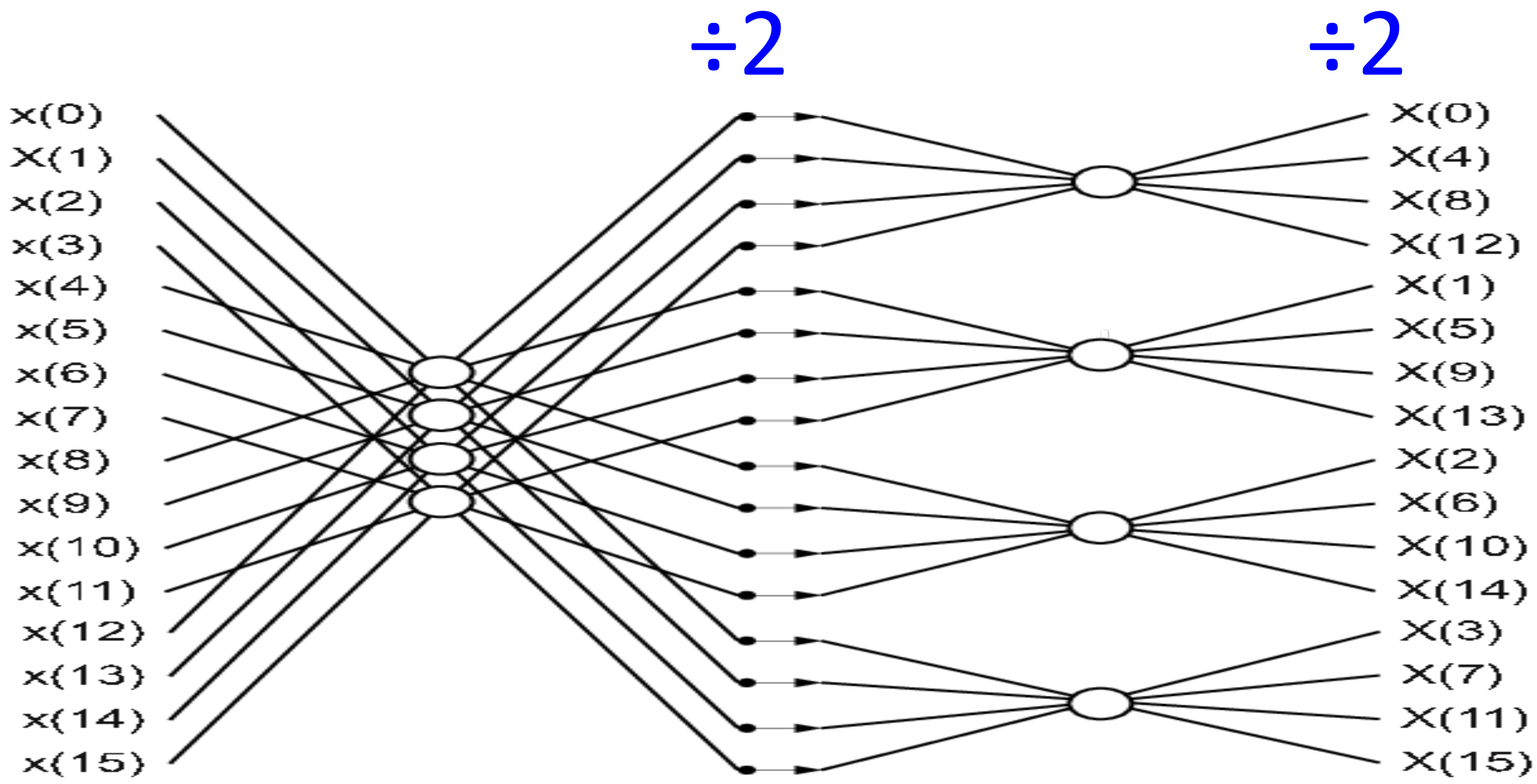
Trick #1: Symmetric DFT stays in posit "sweet spot"

Forward DFT: $$X_n = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N}$$

Inverse DFT: $$x_n = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} X_n e^{+2\pi i k n / N}$$

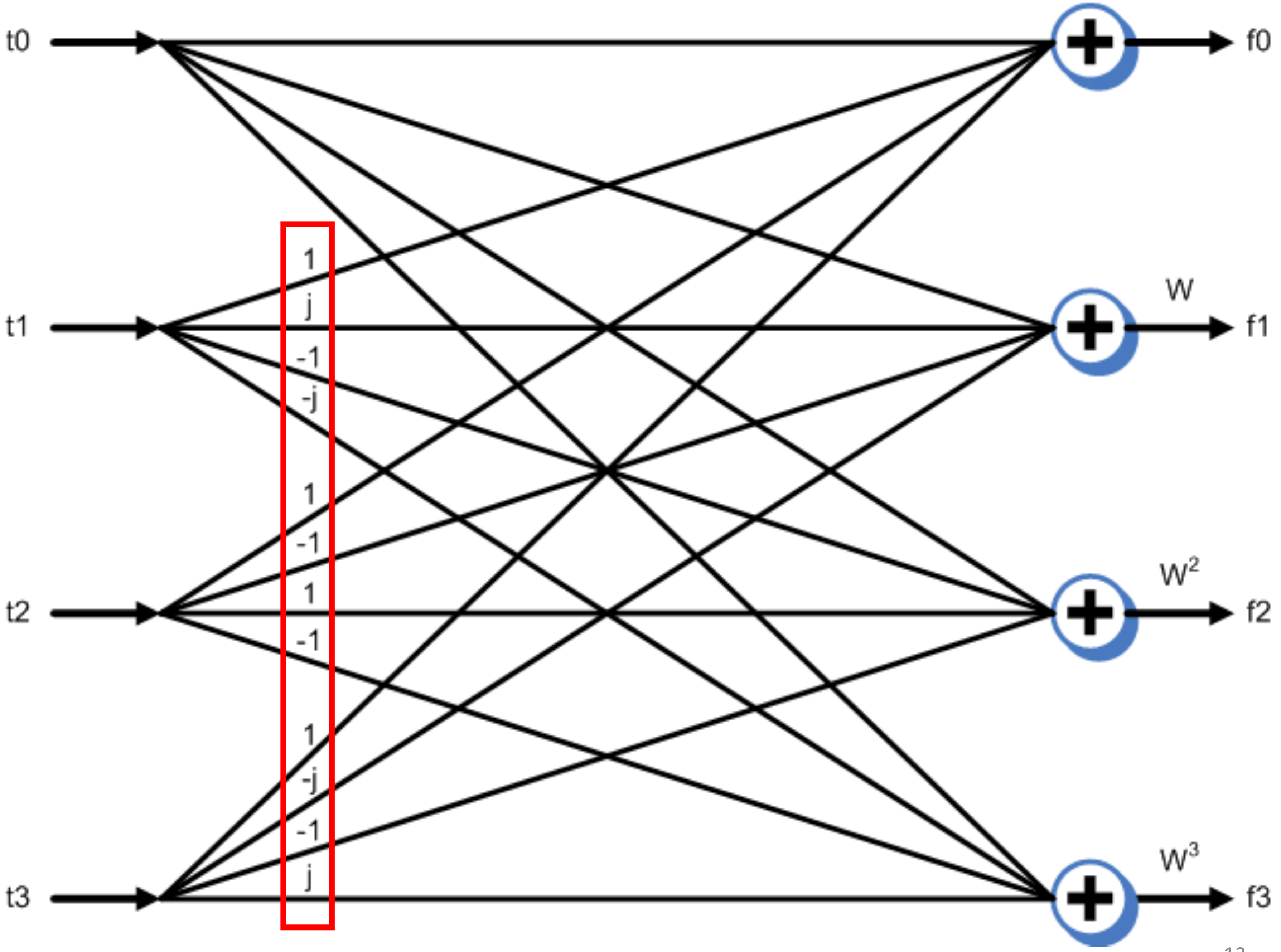
If inputs are in $[-1, 1]$, outputs are in $\left[-\sqrt{N},\ \sqrt{N}\right]$.

Trick #2: Use radix 4, normalize *on each pass*.



This keeps accuracy in the “sweet spot” and normalizes by $1/\sqrt{N}$.
Division by 2 is zero cost if you apply it in the “twiddle factor” table.

One of the passes involves *no* rounding from multiplication; w is just $\sqrt{-1}$.



Trick #3: Use the quire for the kernel operation

Operations grouped by overbar are exact dot products, then rounded once.

$$gg_{[[iA+1,1]]} = \overline{xA_{[[1]]} + xB_{[[1]]} + xC_{[[1]]} + xD_{[[1]]}};\quad \text{(* Multiply by } (iflg*i)^{0} \text{ *)}$$

$$gg_{[[iA+1,2]]} = \overline{xA_{[[2]]} + xB_{[[2]]} + xC_{[[2]]} + xD_{[[2]]}};$$

$$gg_{[[iB+1,1]]} = \overline{xA_{[[1]]} - \mathit{iflg} * xB_{[[2]]} + -xC_{[[1]]} + \mathit{iflg} * xD_{[[2]]}};\quad \text{(* Multiply by } (iflg*i)^{1} \text{ *)}$$

$$gg_{[[iB+1,2]]} = \overline{xA_{[[2]]} + \mathit{iflg} * xB_{[[1]]} + -xC_{[[2]]} - \mathit{iflg} * xD_{[[1]]}};$$

$$gg_{[[iC+1,1]]} = \overline{xA_{[[1]]} - xB_{[[1]]} + xC_{[[1]]} - xD_{[[1]]}};\quad \text{(* Multiply by } (iflg*i)^{2} \text{ *)}$$

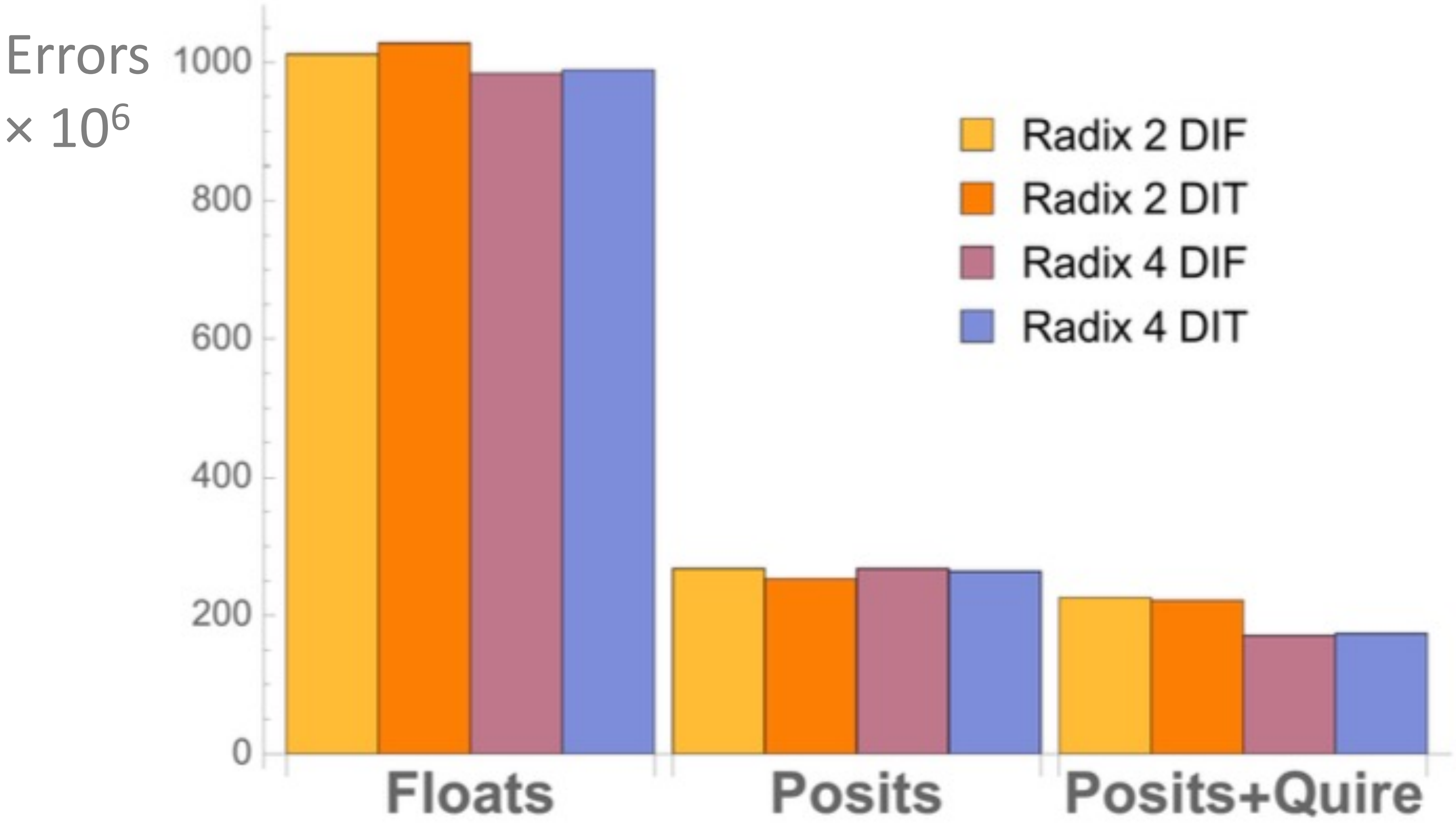
$$gg_{[[iC+1,2]]} = \overline{xA_{[[2]]} - xB_{[[2]]} + xC_{[[2]]} - xD_{[[2]]}};$$

$$gg_{[[iD+1,1]]} = \overline{xA_{[[1]]} + \mathit{iflg} * xB_{[[2]]} + -xC_{[[1]]} - \mathit{iflg} * xD_{[[2]]}};\quad \text{(* Multiply by } (iflg*i)^{3} \text{ *)}$$

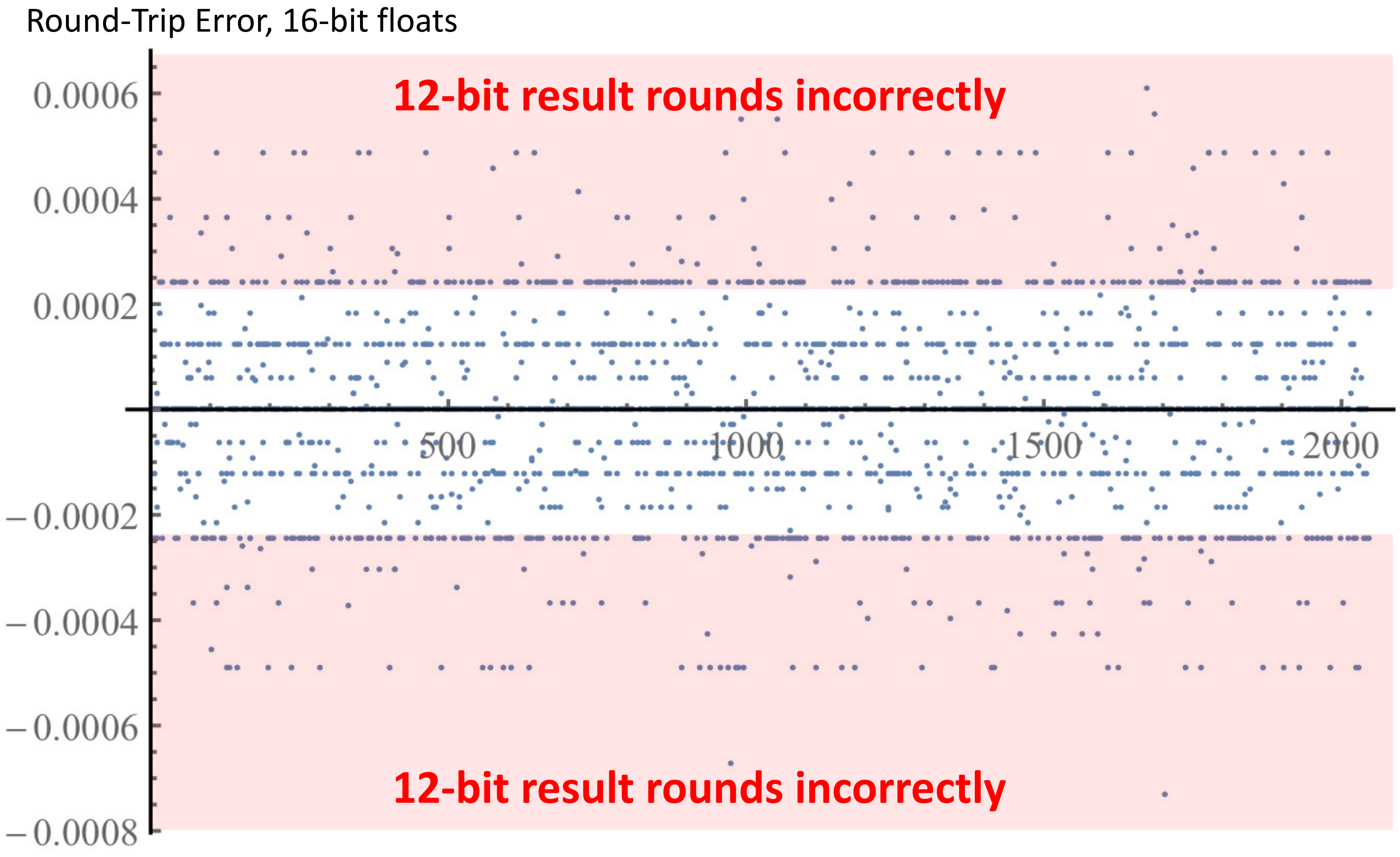
$$gg_{[[iD+1,2]]} = \overline{xA_{[[2]]} - \mathit{iflg} * xB_{[[1]]} + -xC_{[[2]]} + \mathit{iflg} * xD_{[[1]]}};$$

Results of a 1024-point FFT accumulate only **four** rounding errors from beginning to end of the five “butterfly” passes!

Round-trip Error Measured with RMS



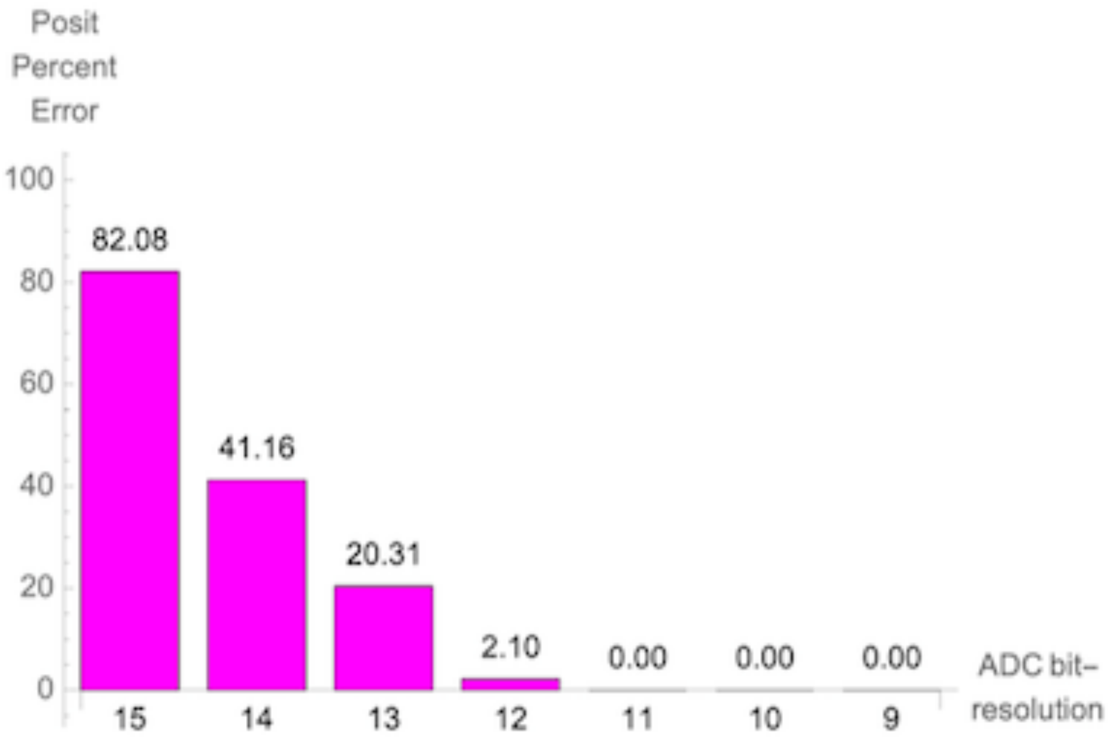
16-bit IEEE floats lose far too much information



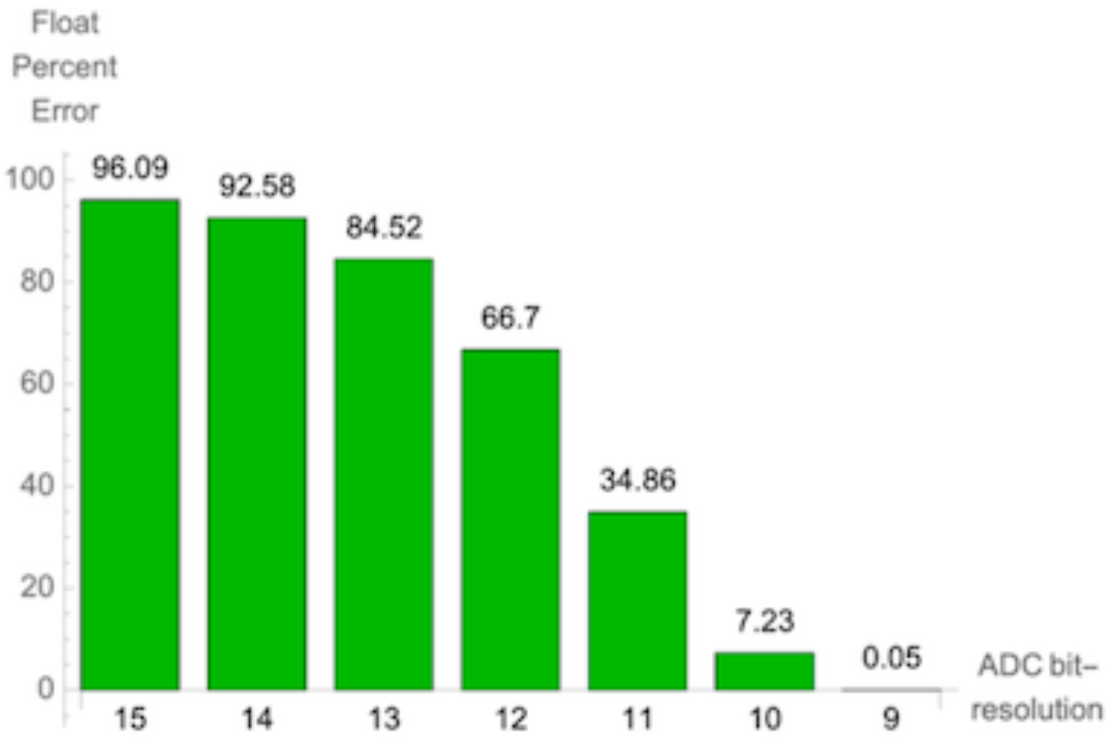
Scatter plot of errors of real and imaginary data (2048 points).

Losses force use of **32-bit** floats for signal processing.

Round-Trip Errors *After Rounding to ADC Accuracy*



(a) Posits

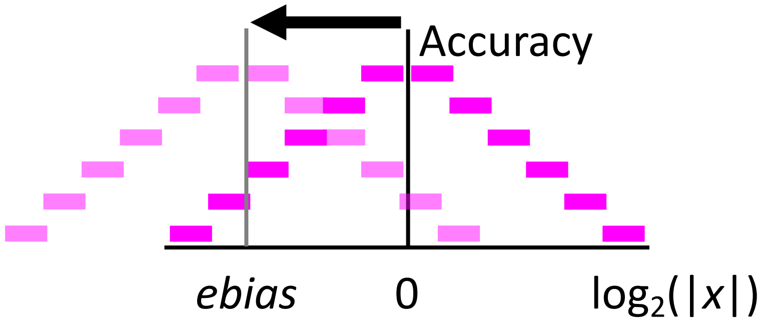


(b) Floats

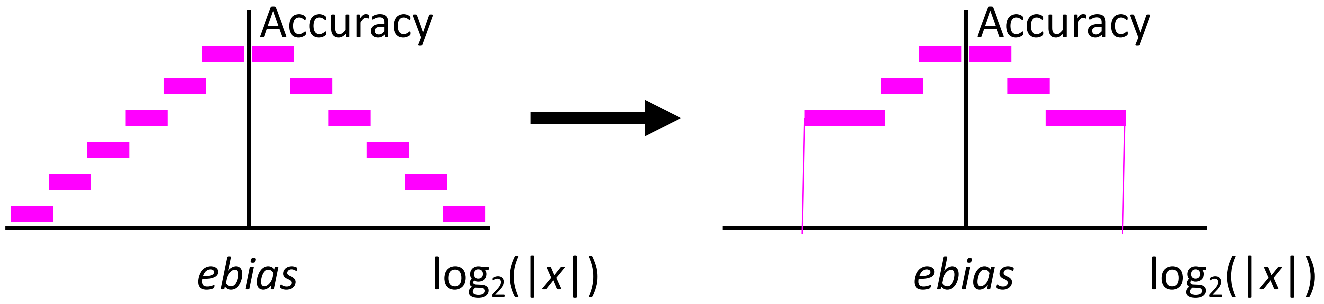
For 12-bit ADC signals, 16-bit posits with *eS* = 1 are off by 1 Unit in Last Place (ULP) for 2.1% of the values, versus 66.7% for floats. **But we can do even better.**

Generalized Posits: New Parameters

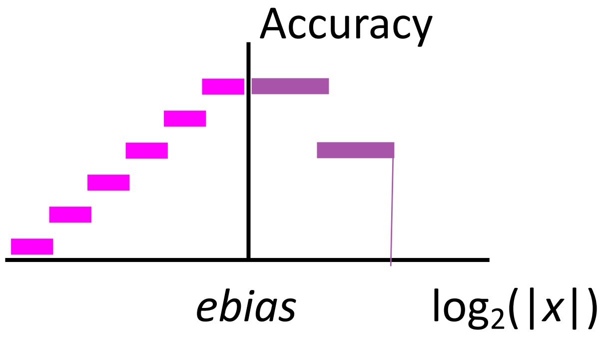
- Move center of exponent range with *eBias.*



- Blunt the tapering by limiting the maximum regime to *rS* bits.

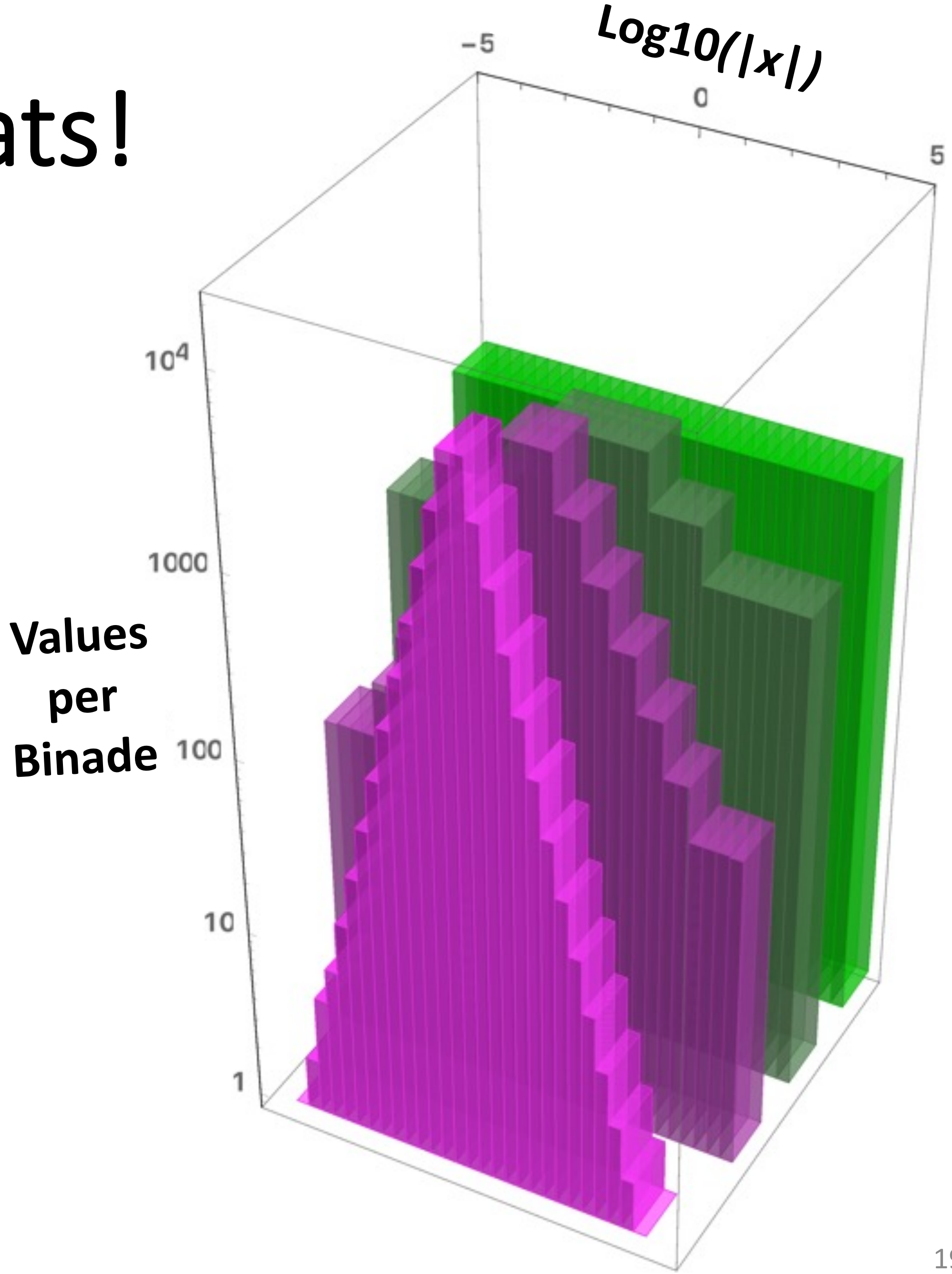


- Can allow different *rS* and *eS* values for left and right halves of the tent.

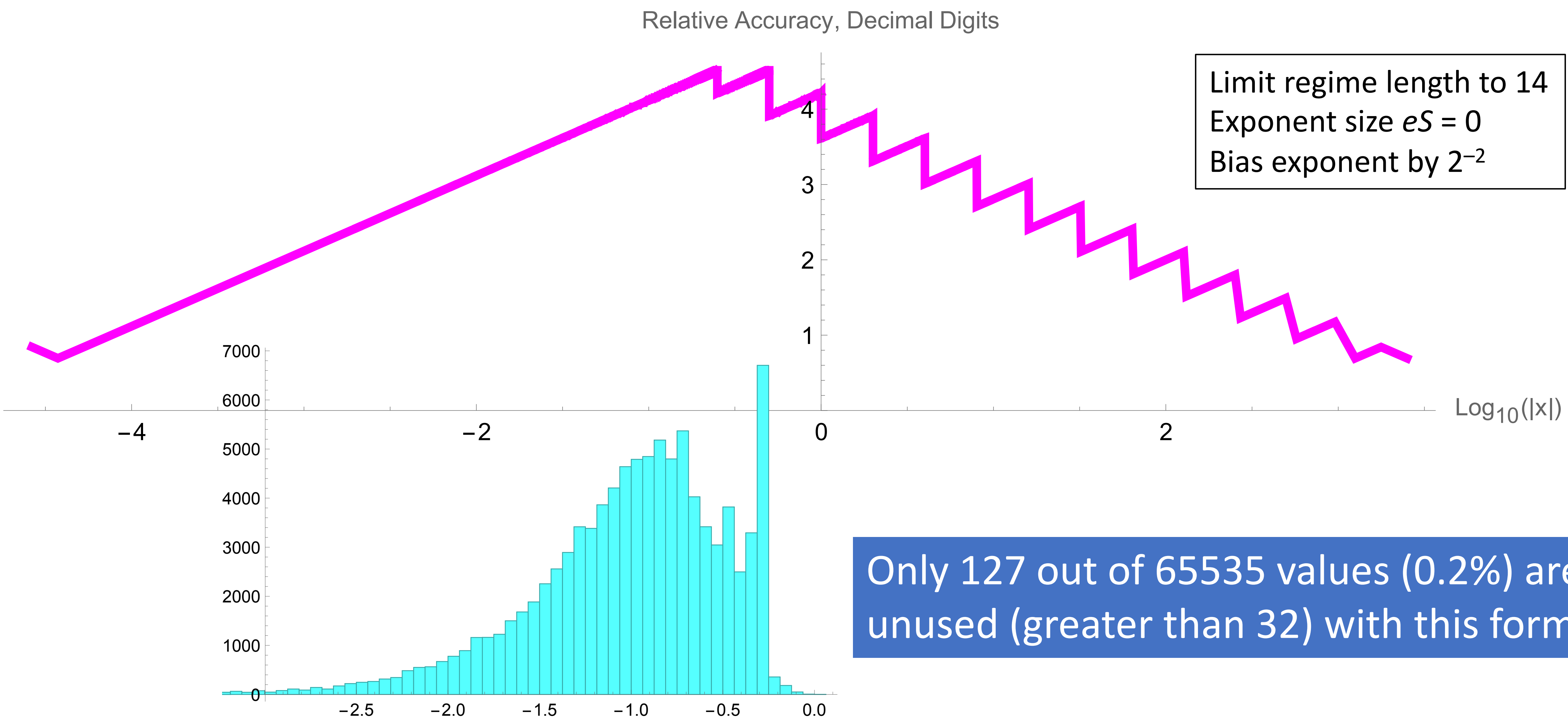


Can dial from posits to floats!

- Adjust *eS* and *rS* in tandem to keep dynamic range similar.
- When *rS* becomes 2, you get idealized floats (green block).
- Ideal *rS* for a particular application is often the original posit definition (magenta triangle), but not always.
- Asymmetric option useful when maximum $|x|$ is known but minimum $|x|$ could be anything.



A Generalized 16-Bit Posit Matched to FFT Needs:

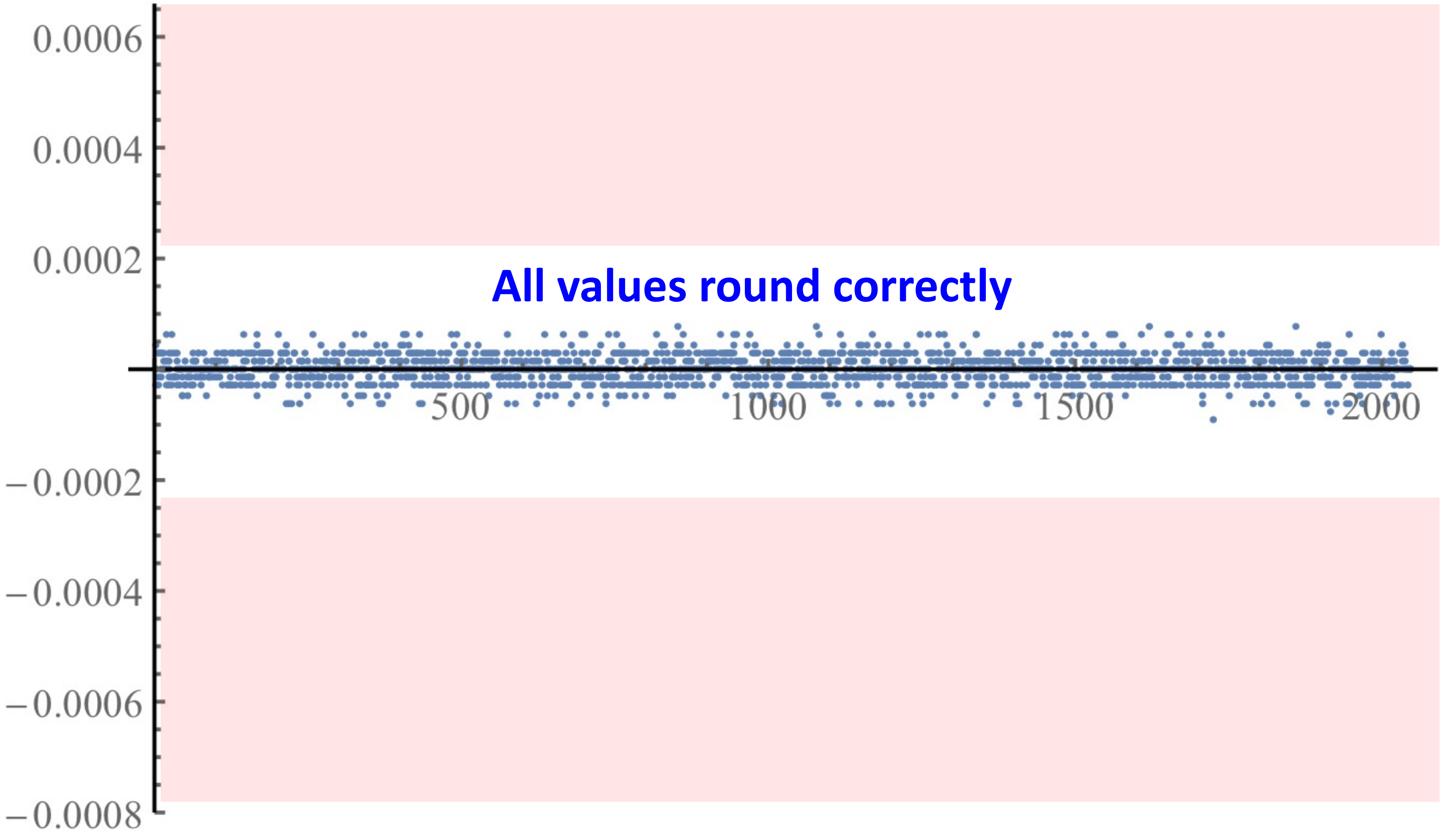


Limit regime length to 14
Exponent size *eS* = 0
Bias exponent by 2^{-2}

Only 127 out of 65535 values (0.2%) are unused (greater than 32) with this format.

16-bit generalized posits can easily do a *lossless FFT*

Round-Trip Error, 16-bit floats

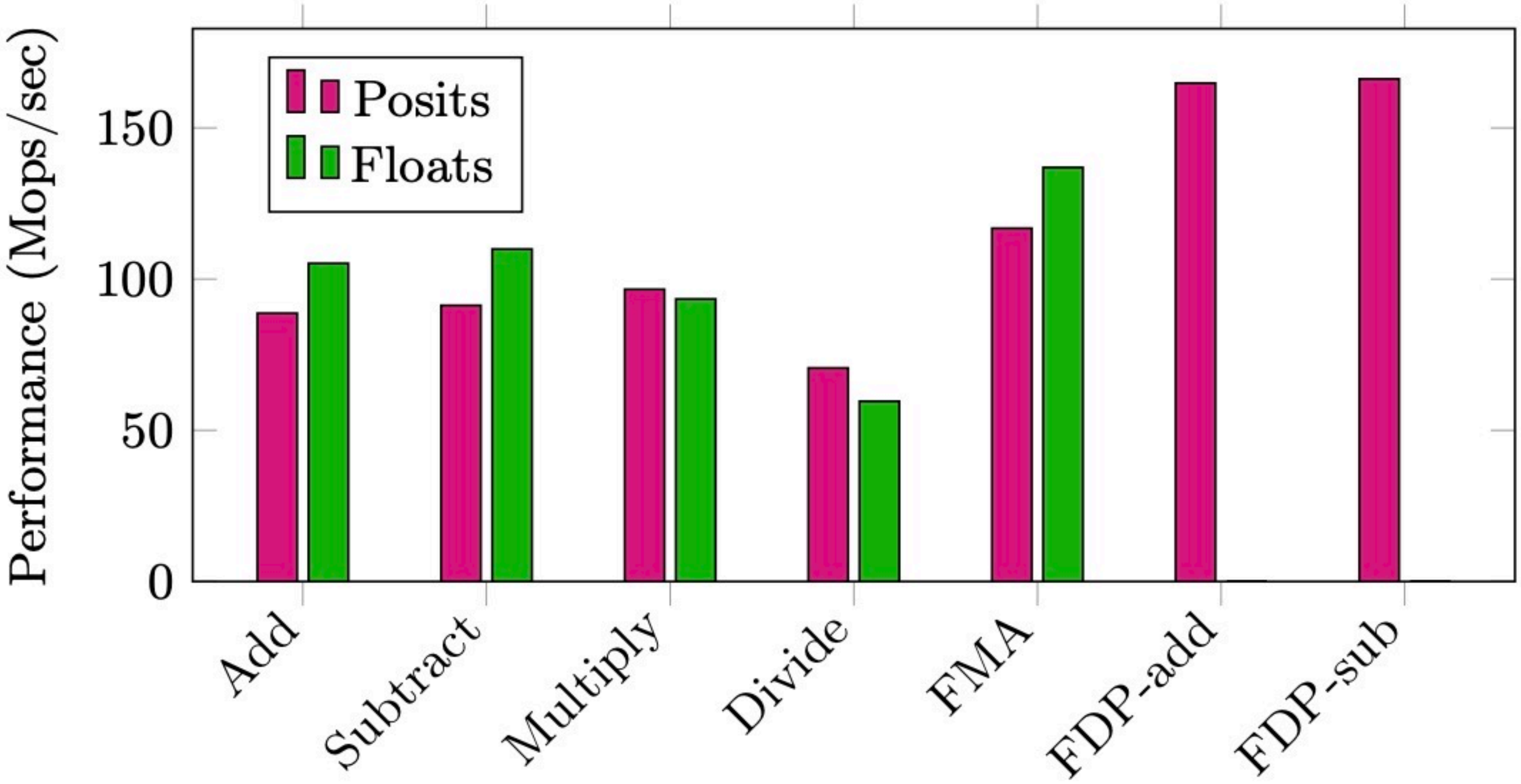


Quire is only 64-bit. Fast.

Signal noise reduced by 10 dB.

16 bits suffice.

SoftPosit vs. SoftFloat speed for FFT data



Measured on Intel® Xeon® CPU E5-2699 v4; base frequency 2.2 GHz, max Turbo frequency 3.6 GHz

OS is OpenSUSE 42.2 (x_86_64)

Compiled with Gnu gcc 4.8.5, -O2, architecture ”core-avx2”

Corroborates Kulisch: Exact dot product is *faster* than a series of fused multiply-adds.

16-bit posits vs *32-bit* floats is a clear win

- With 32-bit floats, 1024-point FFT might *not fit in cache*
- Speed increases by **>2×** (half the data motion)
    - Cache effects (especially for 2D and 3D FFTs)
    - Quire is inherently faster than rounded float multiply-adds
- Power decreases by **>2×** (data motion dominates the power cost)
- Energy cost (power × time) therefore decreases by **>4×.**
- Aside: since Finite Impulse Response (FIR) filtering also can use quire, the same advantage for posits applies.

Summary

- 16-bit posits suffice for signal processing FFTs.
- They can replace 32-bit floats now in use.
- Workload is very similar to Machine Learning.
- More than 2× power savings, 4× energy savings
- Tweaking 16-bit standard posits can yield *lossless* FFTs for 12-bit A-to-D convertors.
- The key tricks are to use radix 4, normalize by ½ on each pass, and use the quire.
- Benefits radio astronomy, MRI scans, X-ray crystallography, 5G networking, etc.