

facebook
Artificial Intelligence Research

Approximate floating point for AI and beyond

Jeff Johnson
Facebook AI Research
jhj@fb.com

'A new golden age for computer architecture'

Hennesy and Patterson, Turing Award lecture, ISCA 2018

- **More effective use of memory bandwidth**
 - User controlled versus caches
- **Eliminate unneeded accuracy**
 - IEEE replaced by lower precision FP
 - 32-64 bit integers to 8-16 bit integers

Underneath all the architecture is still the arithmetic.

There need to be more options than this.

A 'new golden age' for computer arithmetic too?

Are the bits you are moving meaningful? Maybe not. Can you address this without sacrificing generality? Many domain-specific accelerators might be over-tailoring their arithmetic.

- **More effective use of memory bandwidth**
 - User controlled versus caches
- **Eliminate unneeded accuracy**
 - IEEE replaced by lower precision FP
 - 32-64 bit integers to 8-16 bit integers

Do you lose too much mapping work to fixed point? Can you tolerate approximation for substantial wins?

Is accuracy always and everywhere at odds with efficiency? Attribute difference between memory (DRAM/SRAM) versus in-register (SRAM/DFF) storage can be exploited



Let's start with machine learning...



A 60 second view of neural networks

A feedforward neural network is typically composed of blocks of 3 functions (Y. LeCun):

$$f: \mathbb{R}^a \rightarrow \mathbb{R}^b \quad (a < b)$$

Filter bank: projection into a higher-dimensional space on an overcomplete basis, typically a linear function

$$g: \mathbb{R}^b \rightarrow \mathbb{R}^b$$

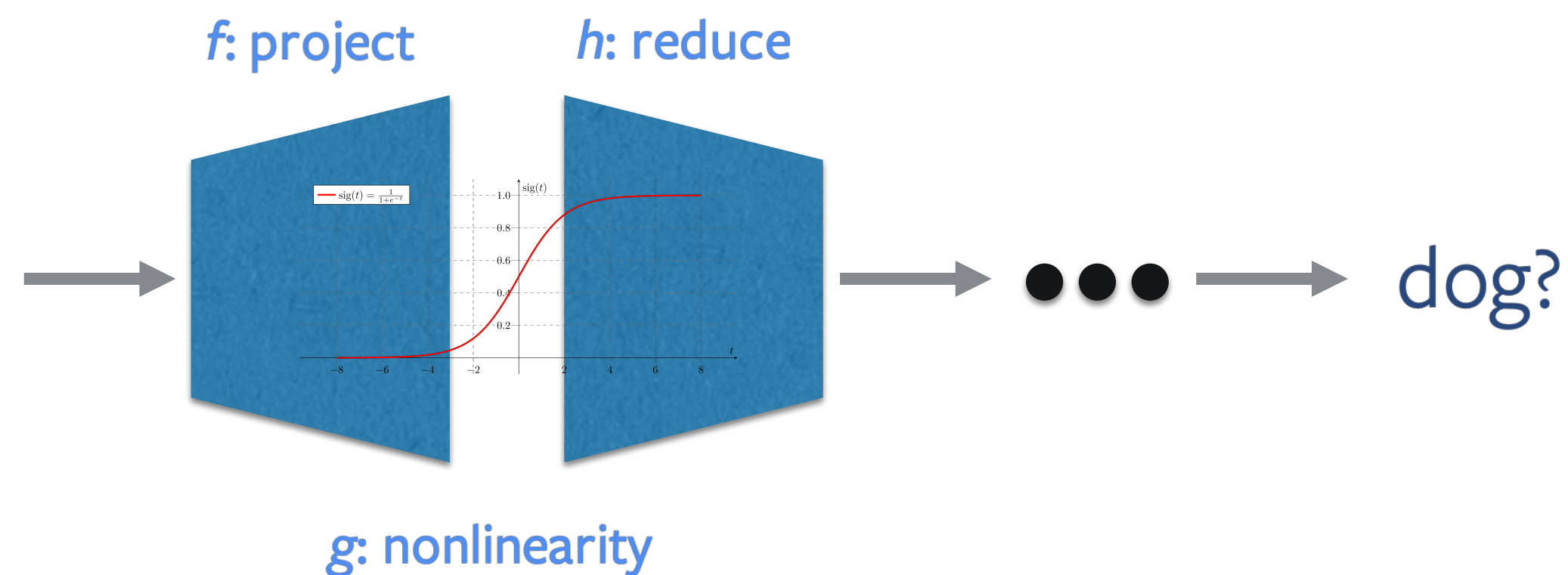
A non-linear function providing saturation and inhibition

$$h: \mathbb{R}^b \rightarrow \mathbb{R}^c \quad (b > c)$$

Dimension reduction: pooling, aggregation over features, sub-sampling

Project into a higher dimensional space, separate, reduce. The linear map f has learned parameters and is expensive to compute.

A 60 second view of (supervised) neural networks



$$\mathbb{R}^{224 \times 224} \rightarrow \dots \rightarrow \mathbb{R}^{200000} \rightarrow \dots \rightarrow \mathbb{R}^{1000}$$

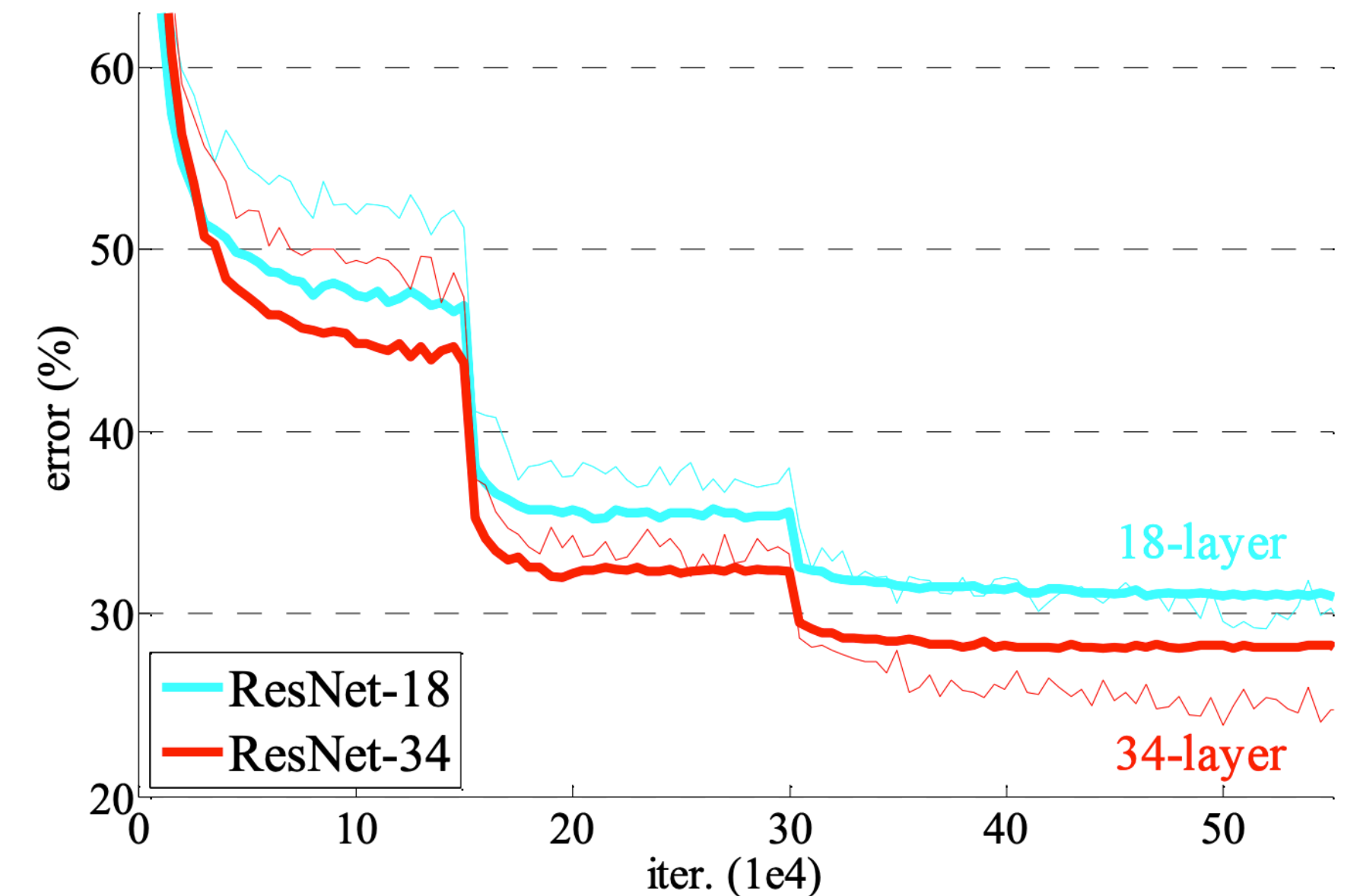
10^6 to 10^{10} parameters, 10^{15} to 10^{21} ops to learn parameters from labeled examples (supervision).

Once learned, the network is likely over parameterized (easier to train in higher dimensions) but can still hopefully generalize (avoids overfitting: hasn't simply memorized the training set).

Training

Most of the computation is straightforward linear algebra (matrix/matrix multiplication) or can be expressed as such. Such operations, especially convolution, has high **operand reuse** potential, making dedicated fixed-function hardware attractive.

People pretend they are operating on the reals, but it is unclear to what degree people depend upon the regularization or other effects of floating point quantization noise in their computation. Reasonably high precision (and dynamic range) floating point is seemingly necessary, especially in open-ended research where the end result is not predictable.



He et al., CVPR 2016

Training alternatives

Ignoring algorithmic or GD/optimization changes (which should be looked to first), can the required exa/zetta-ops of compute for training be achieved more efficiently in hardware? We're limited today by memory overheads and by MAC efficiency.

Precision reduction alone might not cut it, especially in cases where we don't know the gold standard outcome.

Task trained	Total compute (ops)
Original AlexNet	$4.7 \times 10^{17} = 0.47$ exaflops
DeepSpeech 2	$2.6 \times 10^{19} = 26.0$ exaflops
Neural Machine Translation	$6.9 \times 10^{21} = 6.9$ zettaflops

<https://blog.openai.com/ai-and-compute/>

Inference

Inference is using the trained network to perform classification or other non-training tasks on new data. We prefer to use cheaper arithmetic for this. Quantization down to 1-3 uniform bits can be achieved for many use cases, but might be “overfitting” the problem, and may be unsuitable for general-purpose accelerators.

Network generalization makes it resistant to noise (and thus quantization error), though mapping $\mathbb{R}^{224 \times 224}$ to, say, \mathbb{R}^{1000} maps much junk to the same output value. This can be taken advantage of by an adversary, but quantization noise will likely be uncorrelated and will not “follow the gradient”.

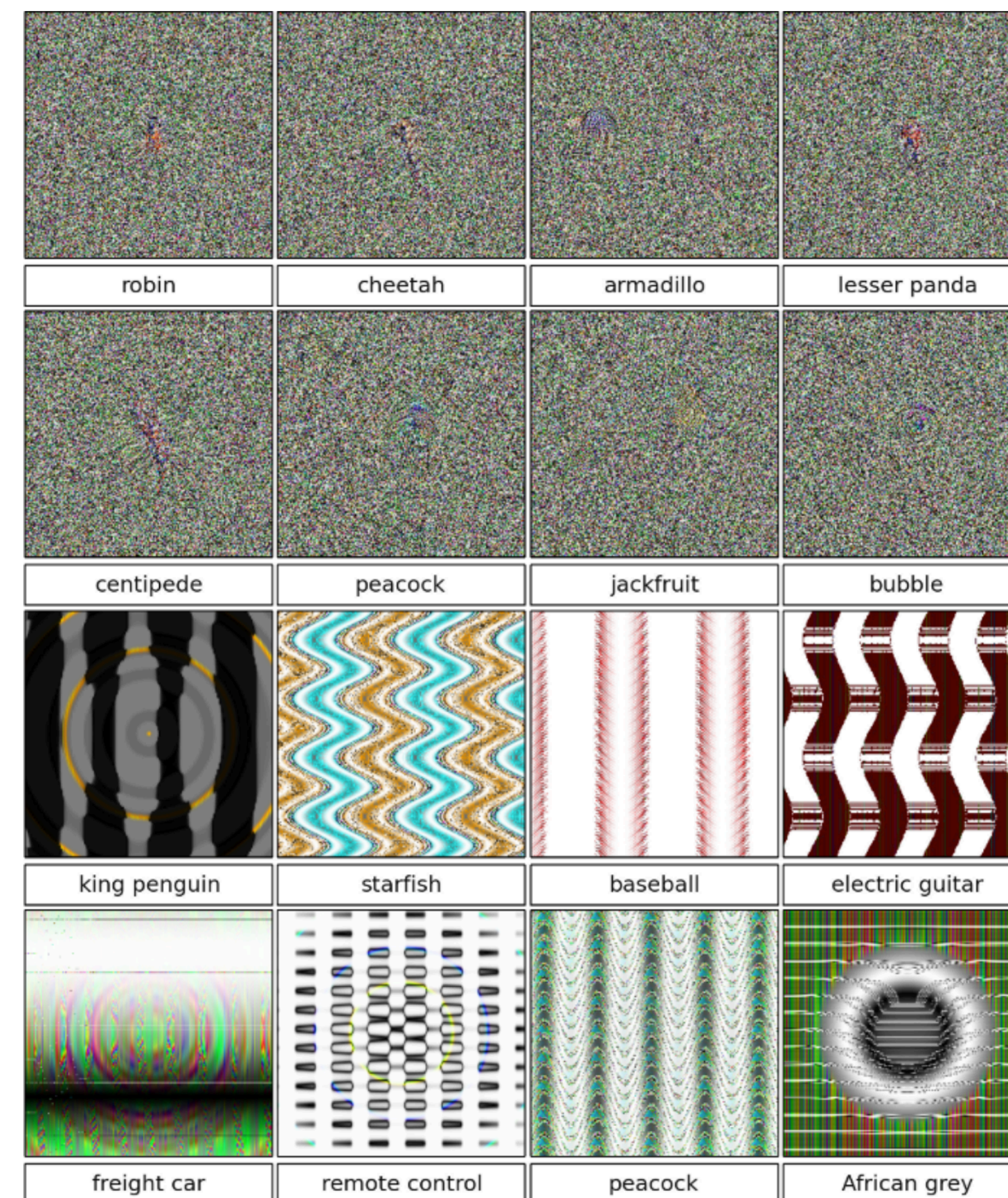


Figure 1. Evolved images that are unrecognizable to humans, but that state-of-the-art DNNs trained on ImageNet believe with $\geq 99.6\%$ certainty to be a familiar object. This result highlights differences between how DNNs and humans recognize objects. Images are either directly (*top*) or indirectly (*bottom*) encoded.

Inference alternatives

My 2018 NeurIPS Workshop for ML paper *Rethinking floating point for deep learning* shows that tiny posit and a log alternative (we'll consider in a bit) are useful as a drop-in replacement for float32 parameters and activations for inference without learned quantization, but the utility of this by itself is perhaps limited.

Extend this to training and beyond?

Table 2: ResNet-50 ImageNet validation set accuracy per math type

Math type	Multiply-add type	top-1 acc (%)	top-5 acc (%)
float32	FMA	76.130	92.862
(8, 1, 5, 5, 7) log	ELMA	-0.90	-0.20
(7, 1) posit	EMA	-4.63	-2.28
(8, 0) posit	EMA	-76.03	-92.36
(8, 1) posit	EMA	-0.87	-0.19
(8, 2) posit	EMA	-2.20	-0.85
(9, 1) posit	EMA	-0.30	-0.09
Jacob et al. [15]:			
float32	FMA	76.400	n/a
int8/32	MAC	-1.50	n/a
Migacz [23]:			
float32	FMA	73.230	91.180
int8/32	MAC	-0.20	-0.03



The search for alternatives



Look to the past (and present) for ideas

Binary stochastic numbers

von Neumann 1952, Gaines 1969

Non-integer and multiple base number systems (β -expansions, MDLNS)

Rényi 1957, Dimitrov et al. 1996

Non-linear maps: log (special case of above), reciprocal, ...

Kingsbury and Rayner 1971, Gustafson 2015

Floating point tapering and entropy coding

Morris 1971, Gustafson 2016, Lindstrom et al. 2018

Learned representations using ROMs/LUTs or RAMs (k-means, VQ, PQ, ...)

Steinhaus 1956, Jégou 2011

An approximate hybrid log-linear computer arithmetic

Improve upon LNS

Unify attributes of logarithmic number systems (LNS) with typical floating point arithmetic. Simplify hardware while substantially improving accuracy in many cases over LNS and energy efficiency over floating point, up to 3x versus bfloat16 FMA.

However, unlike LNS or FP, the arithmetic is approximate with addition ULP error occasionally > 0.5

Rounding and entropy coding

#1 energy cost is moving data (DRAM, SRAM, DFF), especially for low arithmetic intensity workloads where overhead can't be amortized via 'chunky operations'.

Maximize bit information and potentially reduce word size

Dealing with approximation

There is a long literature on floating point error (error-free transforms etc.), but "sloppy" arithmetic provides few guarantees, or perhaps difficult to evaluate and prove guarantees.

The degree of approximation is configurable, but experiment is unfortunately leading theory here.

Traditional Logarithmic Number System (LNS)

Floating point is already hybrid log/linear (exponent/significand) but has wobbling accuracy and doesn't exploit all the log benefits (avoiding HW multipliers, dividers, square root). Sign/magnitude logarithmic number systems (LNS) are fully log domain, typically log base 2.

Example: an 8-bit (3, 4)* log base 2 value: 8 ' b11110101

1: sign bit (-)

111.0101: two's complement fixed point exponent (-000.1011, or -0.6875)

Thus 8 ' b11110101 = $-2^{-0.6875} = -0.6209\dots$

Biased exponent or IEEE-style encodings (or posit-style, or whatever) can also be used. A sign representation and zero encoding are specified, which are outside the log domain.

*(e, f) notation: e bits of integer exponent
 Linear domain: f bits of significand fraction ($\pm 2^e (1 + f)$)
 Log domain: f fractional bits of exponent ($\pm 2^{(e+f)}$)
 e.g., IEEE 754 binary32 FP is (8, 23), ignoring subnormals.

LNS concerns

Multiplication and division are easy and exact.

Given $i = \log_2 x, j = \log_2 y$:

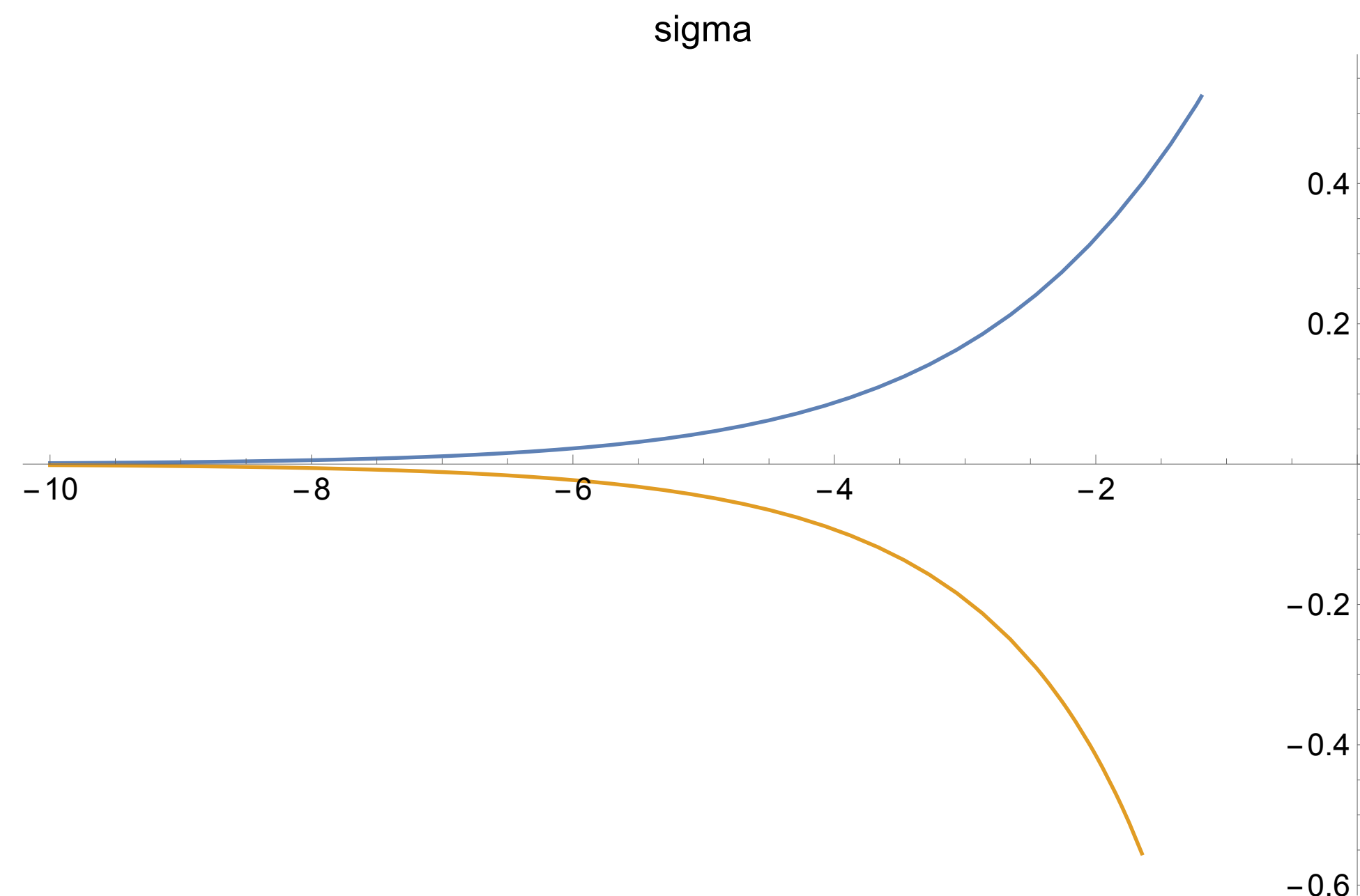
$$\log_2 xy = i + j \quad \log_2 \sqrt{x} = i/2$$

$$\log_2 x/y = i - j$$

Addition and subtraction are not easy or exact.

$$\log_2(x \pm y) = i + \sigma_{\pm}(j - i)$$

$$\sigma_{\pm}(x) = \log_2(1 \pm 2^x)$$



Integers not powers of the base cannot be exactly represented, nor can binary radix floating point values. For example, exact addition by 3 or division by 3 cannot be done.

Similar issues hold with usual FP too; many decimal radix FP values (e.g., 0.1) cannot be exactly represented in binary radix FP. Representation is sensitive to domain and radix.

Why not a traditional LNS?

- Error of LNS basic operations is less than traditional FP (0 ulp for mul/div, 0.5 ulp avg for add/sub vs 0.5 ulp avg for add/sub/mul/div in FP), but only if $\sigma_{\pm}(x)$ is calculated to 0.5 ulp average.
- $\sigma_{\pm}(x)$ requires all range of differences if rounding is to be accurate. This means that the function evaluator as an exact naive LUT has a $\mathcal{O}(n2^n)$ cost for a N-bit word arithmetic, double this for both add and subtract. Long literature of “co-transformation” and approximation to reduce this.
- Hardware resources for high-precision yet accurate add/sub are large!

TABLE 2
Storage for 32-bit and 20-Bit LNS Addition and Subtraction

Table	32-bit		20-bit	
	Organisation	Bits	Organisation	Bits
F, D, E Add	256 words x 6	80,384	32 words x 6	4,328
F, D, E Sub	256 words x 6	84,480	32 words x 6	4,928
P	1 kword	27,648	32 words	480
F1	2 kwords	63,488	32 words	576
F2	2 kwords	65,536	32 words	608
Total		321,536		10,920

Arithmetic on the European Logarithmic Microprocessor, Coleman et al. (2000)

No analogue to high precision accumulator in LNS

Reduced precision LNS that sums into a higher precision accumulator (like accumulating float16 in float32, or a Kulisch accumulator) is not possible, unless the hardware machinery for $\sigma_{\pm}(x)$ encompasses the range of the higher precision type.







Linear algebra (many sums of products) is thus a worry, especially in reduced precision.

Note that the concept of “fused multiply-add” $c' = r(c + ab)$ is also no longer necessary; the multiplication loses no precision with log domain rounding.

“Fused square root-add” $c' = r(c + \sqrt{a})$ can be a thing though.

A new hybrid system

We want the best of both worlds:

- no hardware multipliers/dividers (LNS , floating point )
- accurate multiplication/division (LNS , floating point )
- reasonably accurate and efficient addition/subtraction (LNS , floating point )

For linear algebra with long sums of products, accumulating in a more precise type than the argument, as seen in e.g., Nvidia GPU mixed-precision `wmma` instructions, or exact fixed-point Kulisch accumulators (Kulisch's "long adder"), would be useful.

A domain mismatch between LNS and floating point would ensure that additional error would be introduced, though.

Hybrid log/linear systems

Log values + linear add or linear values + log multiply?

Floating point is hybrid log/linear, but significant operations are always linear (mul/div/add). Mismatch introduces LZ counters, shifters, loss of mul/div precision, ...

One can operate on linear values, approximate linear domain mul via log domain.

(Mitchell, *Computer Multiplication and Division Using Binary Logarithms*, 1962)

LNS has issues with a fully log domain summation, as summarized earlier.

But we are typically evaluating sums of products, not products of sums.

A primary log representation (thus, log values + linear add) makes more sense to me if you're going to mix the two, as the approximate linear add can be made arbitrarily exact.

Hybrid log/linear systems

Log values + linear add or linear values + log multiply?

Conversion from a log value to a (linear) floating point value and vice versa requires evaluating the functions $p(x) = 2^x$ and $q(x) = \log_2(1 + x)$ for $x \in [0, 1)$. The precision of the function evaluation can be adjusted.

Thus log to (linear) floating point is $2^{\lfloor y \rfloor} \times p(y - \lfloor y \rfloor)$, sign and zero handled separately.

$p(x)$ and $q(x)$ only need to operate on the fractional portion of the value. So, brute force table sizes go from $2 \times 2^{e+f} \times (e + f)$ for LNS to $2 \times 2^m \times m$ for a e -bit exponent and f -bit fractional significand. We will show several strategies that can reduce this cost.

We can then apply usual floating point methods on summation.

Log arithmetic with linear sums: FLMA/ELMA

ELMA: log multiply, approximate linear accumulation in fixed point (exact Kulisch accumulator), convert to log domain when done with sums

FLMA: log multiply, approximate linear accumulation in floating point (with possibly greater precision), convert to log domain when done with sums

Accumulation accuracy is controllable by three parameters: α, β, γ .

$p(x)$ takes f log domain fraction bits as input and produces α linear domain fraction as output. $q(x)$ takes β linear domain fraction bits as input and produces γ log domain fraction bits of output. Except for posit-type codes with an additional rounding, $\gamma = f$.

To ensure that $q(p(x)) = x$, we need $a \geq f + 1$ and $b \geq f + 1$.

Log arithmetic with linear sums: FLMA/ELMA

ELMA is cheaper than FLMA for low dynamic range (≤ 6 bits of exponent or so)

For FLMA, mul/div in this arithmetic is 8-20x+ less energy than addition!

(8, 7) log bfloat16 with FLMA (8, 14), mul is **~9x lower energy** than add!

(8, 7) log bfloat16 with FLMA (8, 23), mul is **~15x lower energy** than add!

Addition is about the same as normal float FMA, so overall energy is way lower.

Add/subtract accuracy: effect of α/β

All unique $x + y$ via ELMA (or FLMA with sufficient accuracy) for (5, 4) log (*i.e.*, $f = 4$)

Compare $q_\beta(p_\alpha(x) + p_\alpha(y))$ ulp error versus exact answer

$\alpha = \beta =$	[0, 0.5] ulp	(0.5, 1.5] ulp	(1.5, 2.5] ulp	(3.5, 4.5] ulp	(4.5, 5.5] ulp	> 5.5 ulp
f+1	90.65%	8.89%	0.24%	0.07%	0.06%	0.08%
f+2	96.08%	3.74%	0.13%	0.04%	0.01%	-
f+3	98.15%	1.83%	0.02%	-	-	-
f+4	98.82%	1.17%	0.01%	-	-	-
f+5	99.58%	0.42%	-	-	-	-
f+6	99.82%	0.18%	-	-	-	-
f+7	99.90%	0.10%	-	-	-	-
f+8	100.00%	-	-	-	-	-

FLMA addition accuracy versus traditional LNS

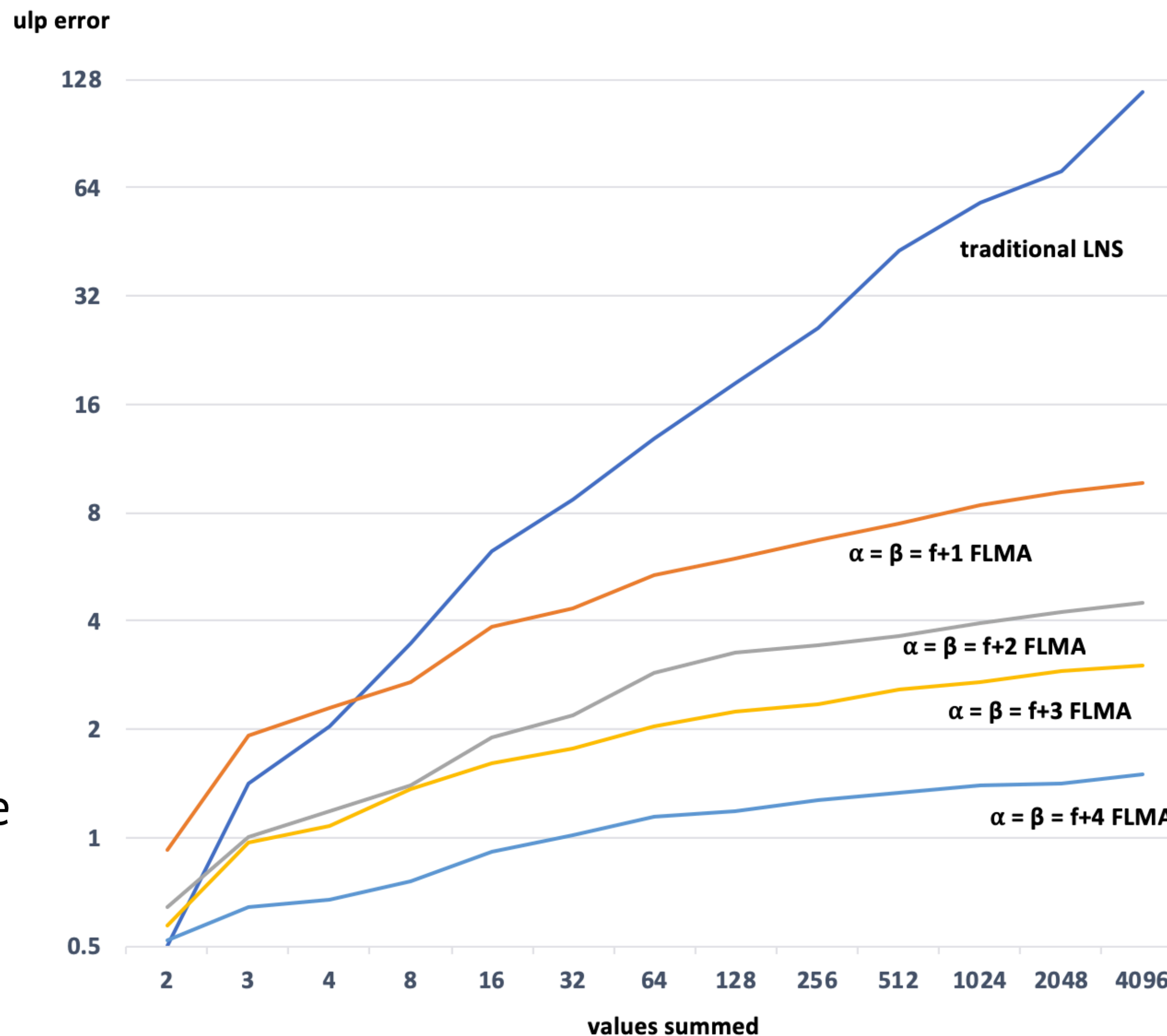
(5, 10) log arithmetic, (5, 20) adder

100 trials of $\sum_{i=1}^n x_i, x_i \sim N(0,1)$

for $n \in \{2, 3, 4, 8, 16, \dots\}$

LNS summation via σ_{\pm} is more accurate only for ≤ 4 sequential sums.

Note that LNS summation is as accurate as normal FP summation (0.5 ulp average error in their respective domains)

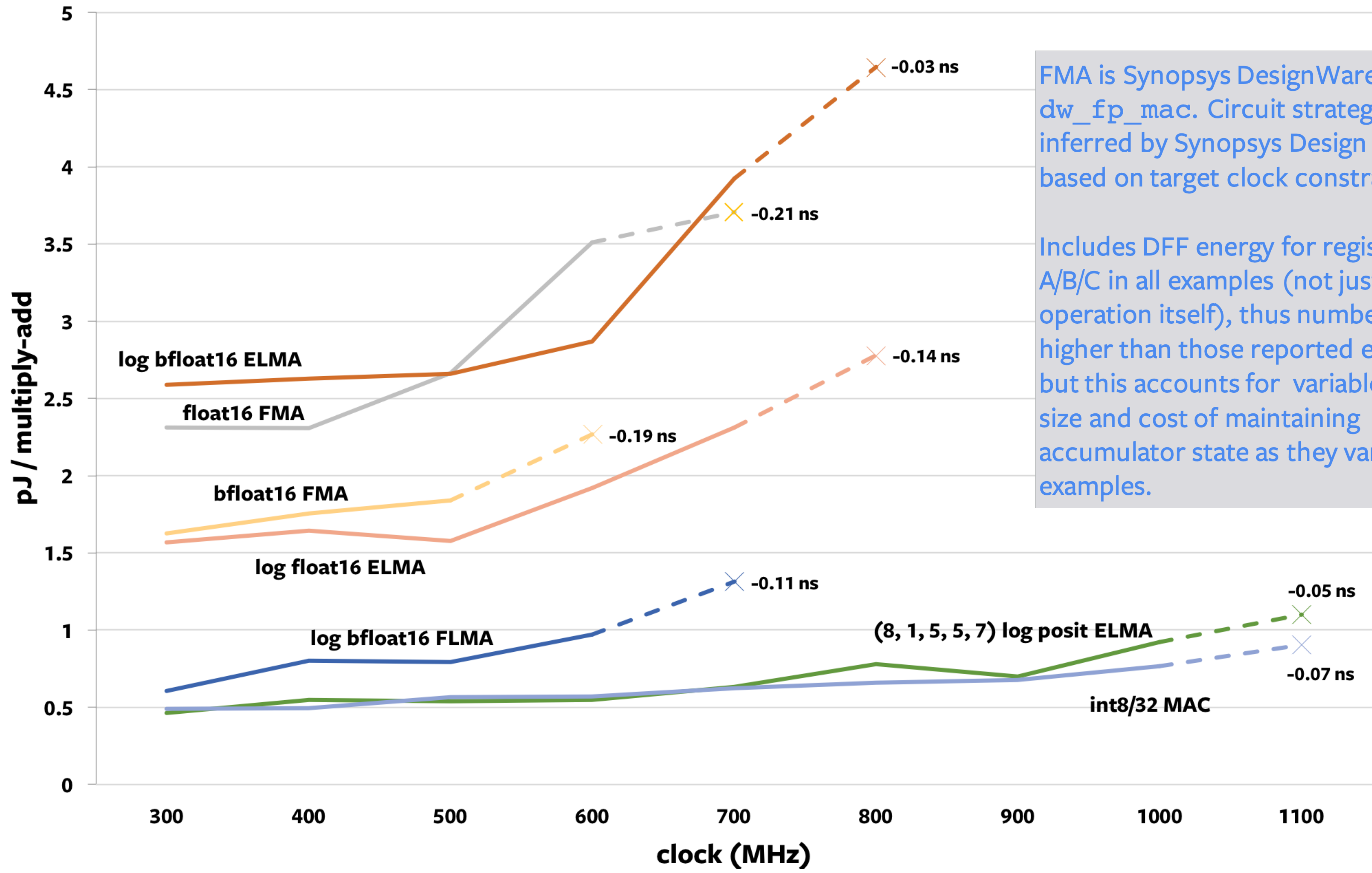


Benefits of ELMA/FLMA

- Savings from avoiding multipliers can be spent on more accurate linear accumulation (e.g., larger float accumulator or fixed point Kulisch). Such a mechanism can be used for floating point as well, but you are starting from a more expensive baseline. We have no hardware dividers or multipliers whatsoever!
- Sum order independence with ELMA (Kulisch), not possible in typical LNS.
- For an adder reduction tree (to reduce data movement), n copies of the $p(x)$ evaluator and $\frac{n}{2} + \frac{n}{4} + \dots + 1 = n - 1$ floating point or Kulisch adders is cheap compared to stamping out $n - 1$ copies of the $\sigma_{\pm}(x)$ evaluator (LNS adder).

28 nm energy efficiency for 1 cycle multiply-add

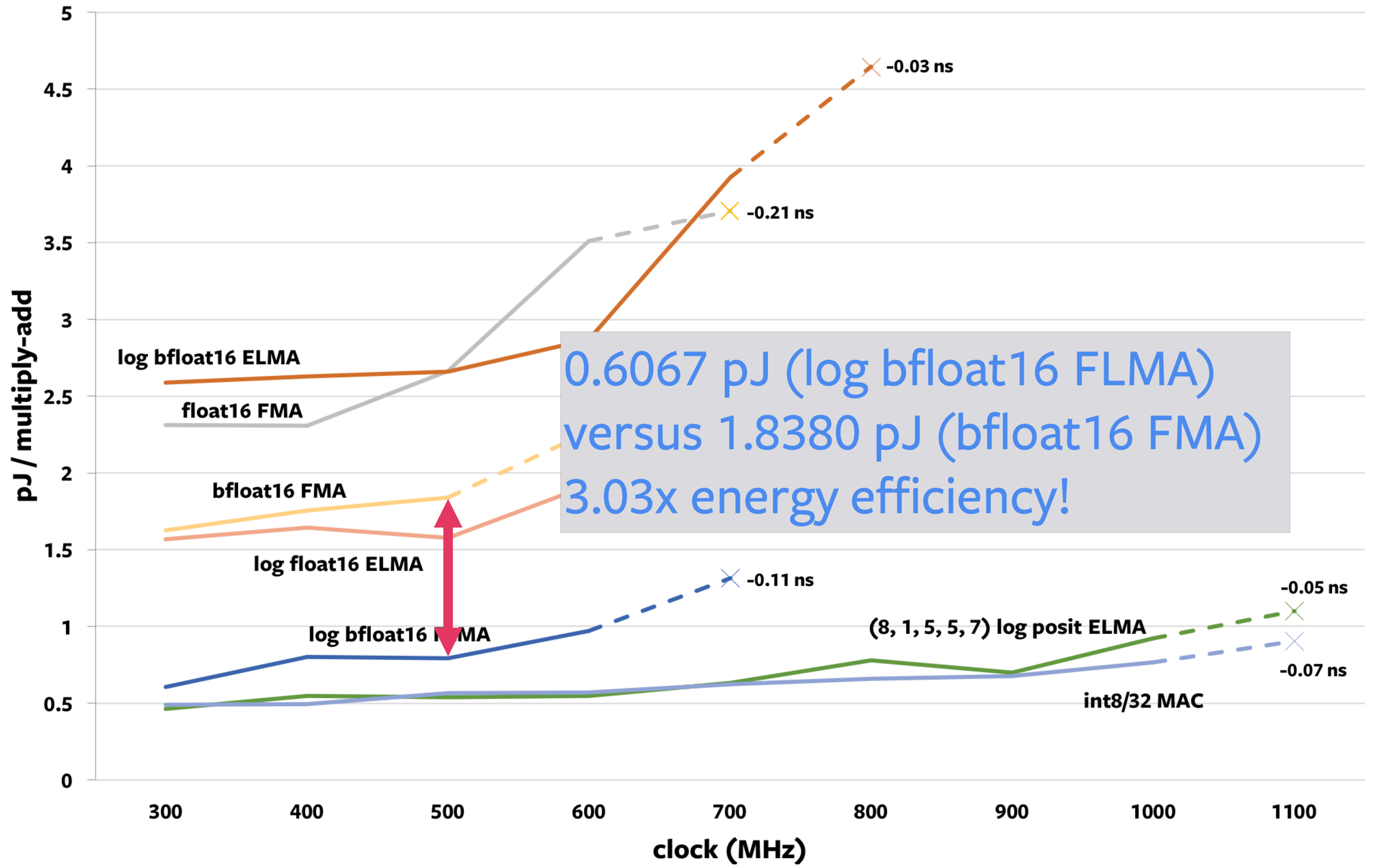
(solid line meets timing, dashed line fails closure at given clock, with negative slack reported)



FMA is Synopsys DesignWare dw_fp_mac. Circuit strategy is inferred by Synopsys Design Compiler based on target clock constraints.

Includes DFF energy for registered A/B/C in all examples (not just the operation itself), thus numbers are higher than those reported elsewhere but this accounts for variable word size and cost of maintaining accumulator state as they vary across examples.

28 nm energy efficiency for 1 cycle multiply-add
 (solid line meets timing, dashed line fails closure at given clock, with negative slack reported)



0.6067 pJ (log bfloat16 FLMA)
 versus 1.8380 pJ (bfloat16 FMA)
 3.03x energy efficiency!

Increasing precision

Against floating point fused multiply-add, simple $p(x)$ LUTs (combinational logic or process-specific compiled ROMs, evaluated on a 28 nm technology) are good on a power/area basis up to about 13 bits of fractional precision (cost of the LUT is substantially overridden by the omission of HW multipliers).

This LUT is expensive on an FPGA though (either in BRAM or as combinational logic). Not sure why you wouldn't use anything but hardened DSP-friendly math on an FPGA, though.

What about higher precision?

Increasing precision: $p(x)$ (exp2)

Naive LUT: Evaluted to infinite precision and rounded; $(2^f \alpha)$ -bit LUT

LUT encoding Mitchell error $(2^m - m)$: $(2^{(f-2)} \alpha)$ -bit LUT + adder

Piecewise linear/poly approximation: reintroduces multipliers we got rid of

Hyperbolic CORDIC: sequential, slow for high f

High-radix CORDIC ($r=8, 16, \dots$): sequential but faster by factor $\log_2 r$, more HW

Note that with ELMA (or FLMA if 1-cycle FP add), sequential algorithms might be ok since they can be fully pipelined, but copies must be instantiated (high power, area)

Increasing precision: $q(x)$ (\log_2)

Naive LUT: Evaluted to infinite precision and rounded; $(2^\beta \gamma)$ -bit LUT

LUT encoding Mitchell error ($m - \log_2(1 + m)$): $(2^{(\beta-2)} \gamma)$ -bit LUT + adder

Piecewise linear/poly approximation: reintroduces multipliers we got rid of

Hyperbolic CORDIC: sequential, slow for high β

High-radix CORDIC ($r=8, 16, \dots$): sequential but faster by factor $\log_2 r$, more HW

Note that with ELMA (or FLMA if 1-cycle FP add), sequential algorithms might be ok since they can be fully pipelined, but copies must be instantiated (high power, area)

Higher precision: hybrid approaches

Novel quasi-symmetrical approach for efficient logarithmic and anti-logarithmic converters, Hoang and Pham, 2012

On-line high-radix exponential with selection by rounding, Piñeiro et al., 2003

A fast hardware approach for approximate, efficient logarithm and antilogarithm computation, Paul et al., 2009

The Paul et al. approach is almost a hybrid of log domain approximation for linear interpolation via multiplication. ~20 bits of precision is achievable fairly cheaply.

For linear algebra, $p(x)$ is the thing that needs the most optimization.



Using ELMA/FLMA



We want more multiplications, fewer additions!

Tricks that substitute multiplication with addition no longer make as much sense; e.g.,

Winograd's original matrix multiplication algorithm:

Consider $\sum_{i=1}^n x_i y_i$ (assuming n even):

$$\text{Let } \xi = \sum_{j=1}^{n/2} x_{(2j-1)} x_{2j}, \quad \eta = \sum_{j=1}^{n/2} y_{(2j-1)} y_{2j};$$

$$\text{then } \sum_{i=1}^n x_i y_i = \sum_{j=1}^{n/2} (x_{(2j-1)} + y_{2j})(x_{2j} + y_{(2j-1)}) - \xi - \eta.$$

A New Algorithm for Inner Product, S. Winograd, IEEE Trans. on Computers (1968)

For $(m \times k) \times (k \times n)$ matrix multiplications, this trick replaces mnk multiplications and additions with $\mathcal{O}(\frac{1}{2}mnk)$ multiplications and $\mathcal{O}(\frac{3}{2}mnk)$ additions.

Winograd $F(m \times m, r \times r)$ convolution

The Winograd minimal filtering algorithms similarly substitute addition for multiplication and are not a win over direct convolution for a single channel convolution, but it is a tile-based algorithm and there is reuse of the data possible for both data and filter.

2x2 output tile:

Winograd $F(2 \times 2, 3 \times 3)$: has 16 mul, 36 add

Direct convolution: is 36 mul, 32 add

Thus, Winograd $F(2 \times 2, 3 \times 3)$ per tile is **>1.05x more energy*** than direct convolution in this arithmetic! It is still useful due to the preprocessing/reuse nature.

*: naive sum of per-operand cost

F(2x2, 3x3) Winograd Convolution (for 2D CNNs)

4x4 float64 data d_{ref} , 3x3 filter f_{ref} (all $x_i \sim N(0, 1)$)

$$R_{ref} = d_{ref} * f_{ref} \text{ (direct conv)}$$

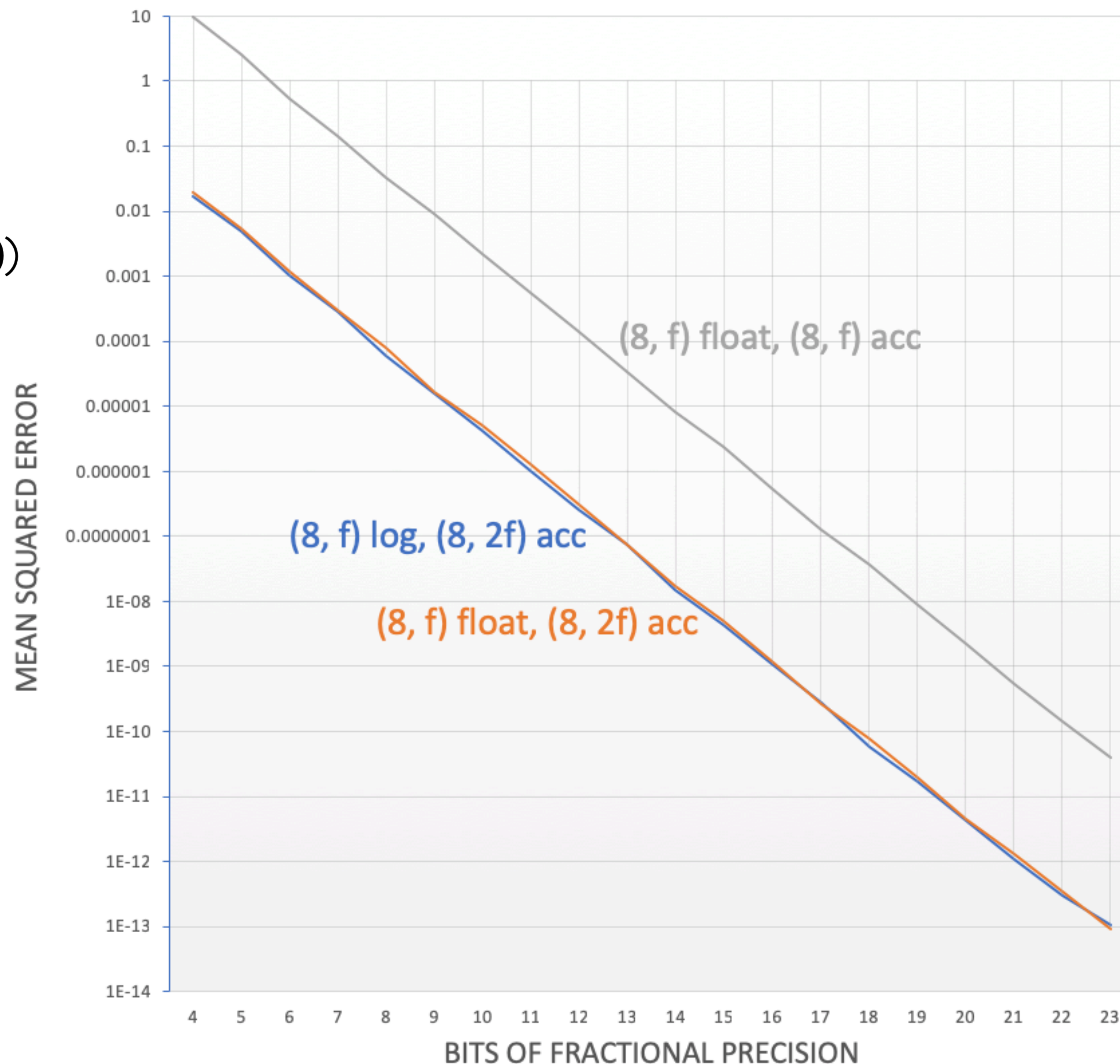
$$R_t = A^T ((G f_t G^T) \circ (B^T d_t B)) A$$

MSE of $(R_t - R_{ref})$

log FLMA is more accurate than both

Dynamic range less of an issue; sum-heavy problem

Type	log FLMA is better by # of precision bits
(8, f) float, (8, f) acc	+4.60
(8, f) float, (8, 2f) acc	+0.09



LU matrix decomposition and recovery

32x32 float64 matrix M_{ref}

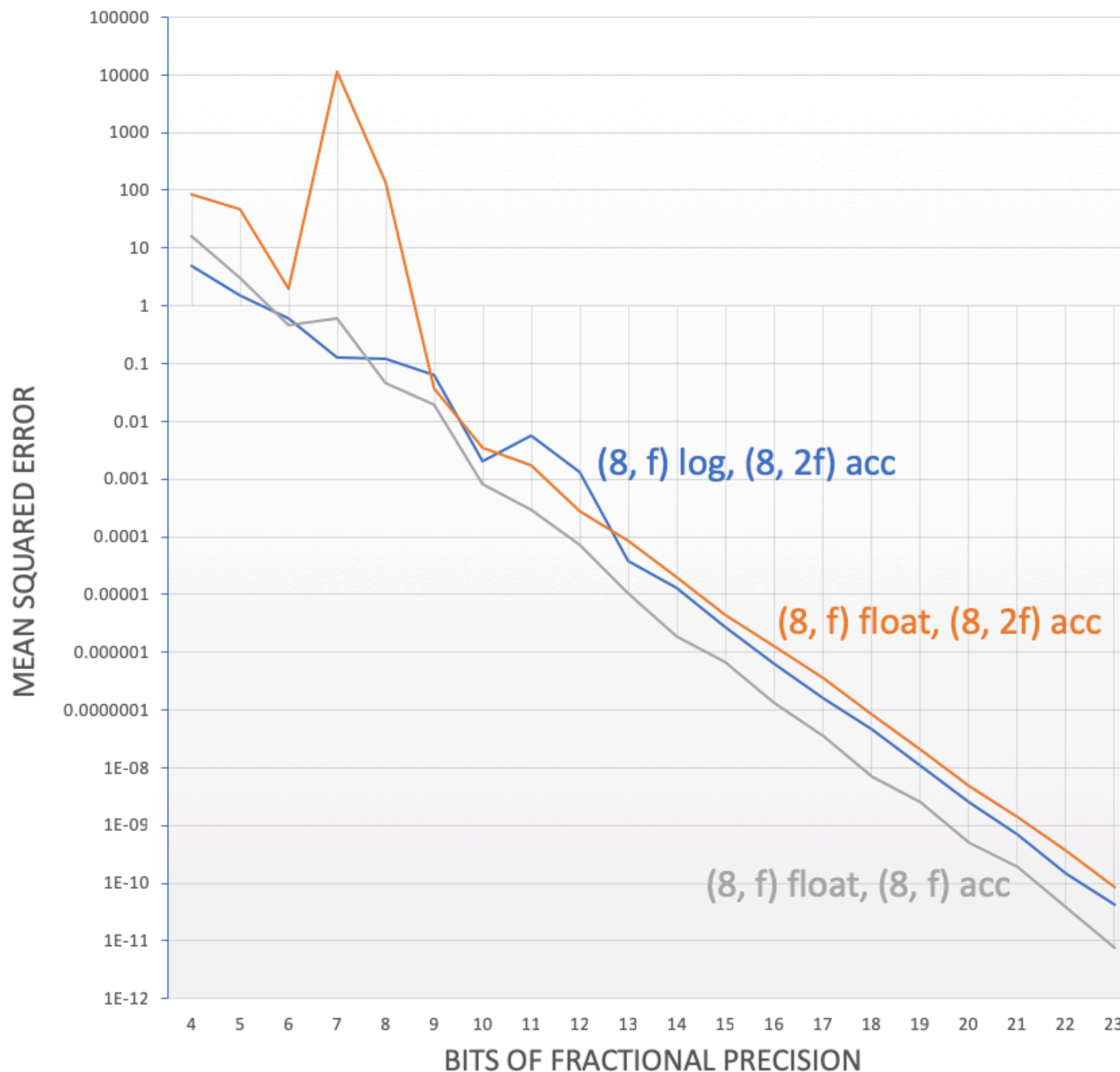
$$M_t = r(M_{ref}), M_t' = L_t U_t$$

MSE of $(M_t' - M_{ref})$

Ill-conditioning is a problem (random matrices, no pre-conditioning)

Why is the 2f accumulator worse?

Type	Avg bits of additional fractional precision versus log FLMA
(8, f) float, (8, f) acc	-0.85
(8, f) float, (8, 2f) acc	+1.14

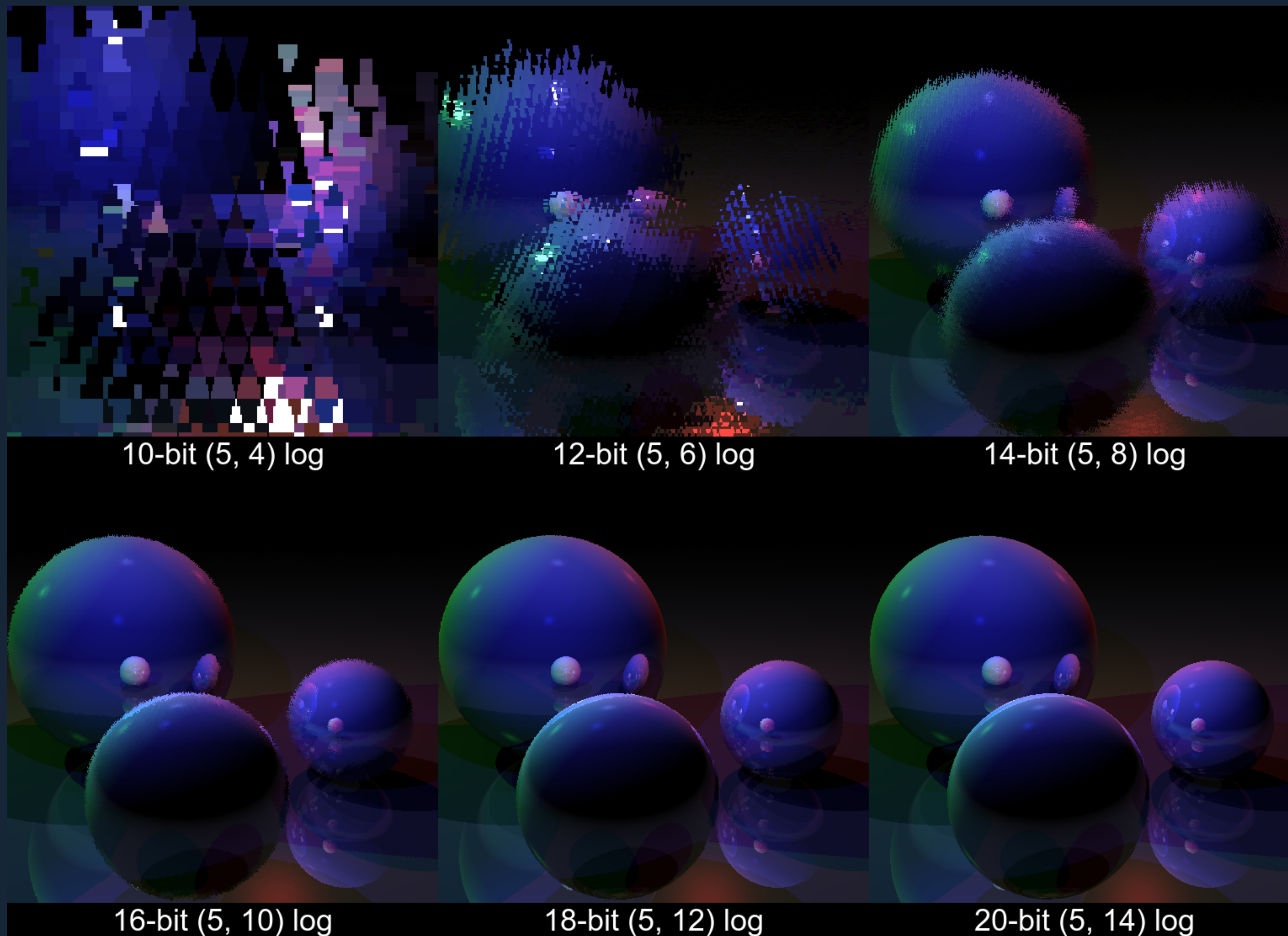


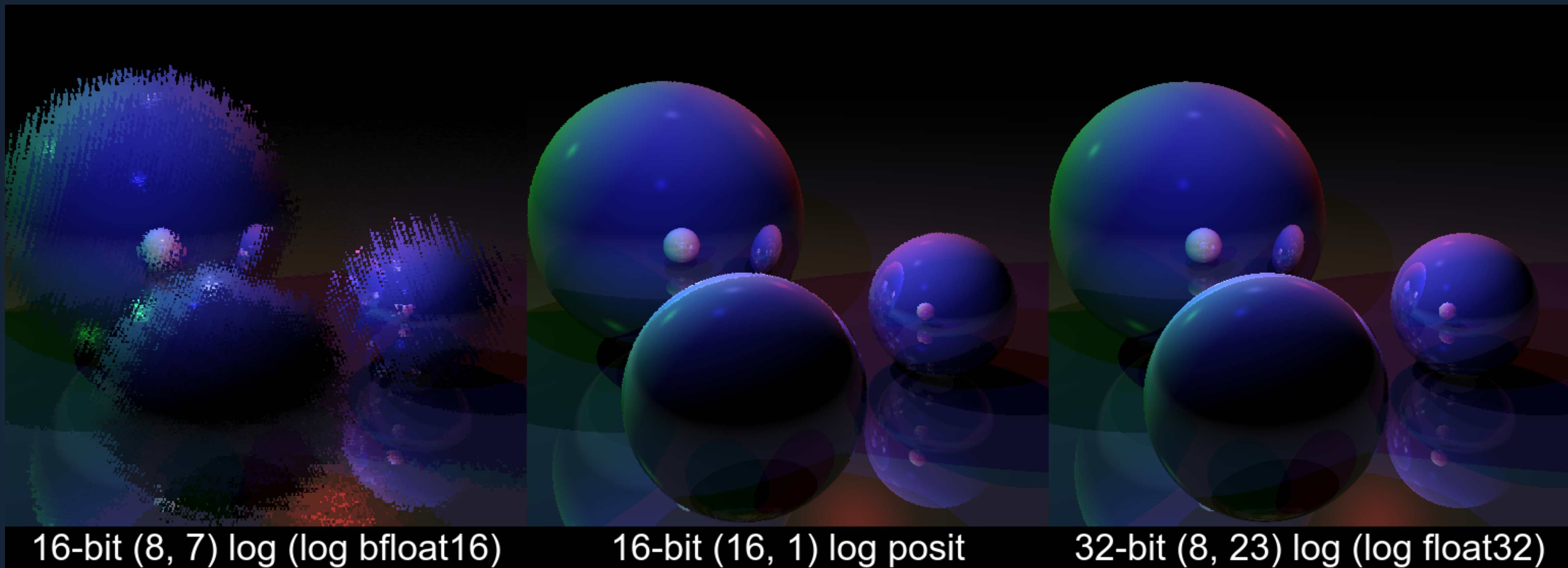
Raytracing with only adders, shifters and LZ counters!

FLMA

$2f$ acc precision

$$\alpha = \beta = f + 1$$





Expression order

How to reorder add/sub relative to mul/div (*i.e.*, where the log/linear conversions happen):

Precision: mul/div before add or after add?

1. $\alpha x + \alpha y + \alpha z$ or 2. $\alpha(x + y + z)$?

3. $\alpha x + \alpha y + z$ or 4. $\alpha(x + y) + z$?

Based on my tests, I suspect #2 but also #3 if this is the entire expression; minimize domain conversions. How should this heuristic be weighted for more complicated expression trees?

Efficiency: preserve sub-expressions in the linear domain where possible:

$$\alpha(x + y + z) + \beta(w - x - y)$$

$(x + y)$ is a common factor to both, duplicate the accumulator register.



Memory overhead and quantization



40

Representation efficiency

Efficiency is dominated by word size and the bits that you are moving (DRAM to SRAM, SRAM to DFF, DFF to DFF).

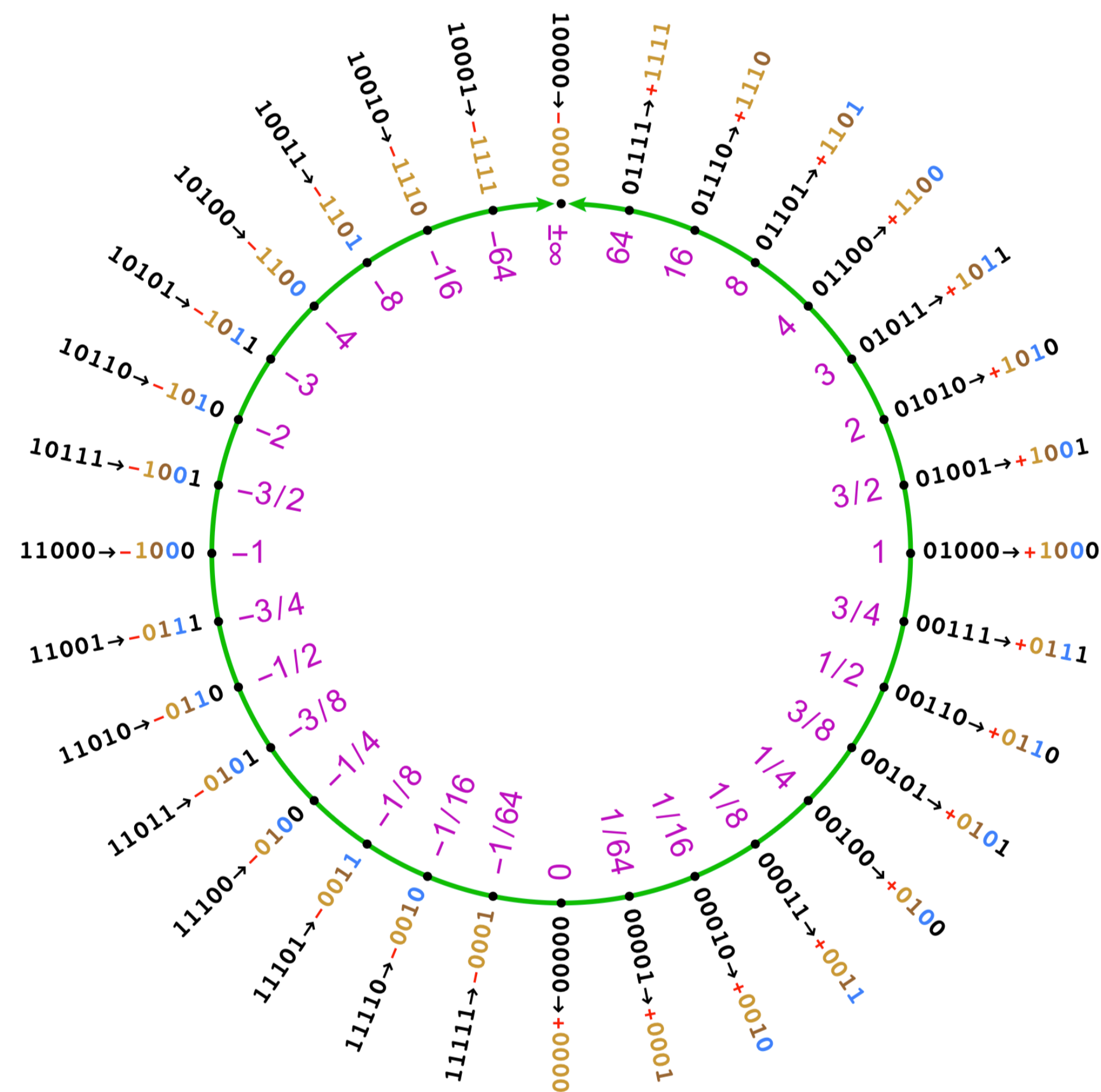
Hardware algorithmic operand reuse (what Bill Dally calls amortization of the memory overhead by “chunky operations”) provides significant savings. This only works if the algorithm in a *roofline model* sense has high compute intensity (ops per byte loaded/stored).

Maybe the reason people have been jumping into AI hardware is because convolution has high algorithmic reuse.

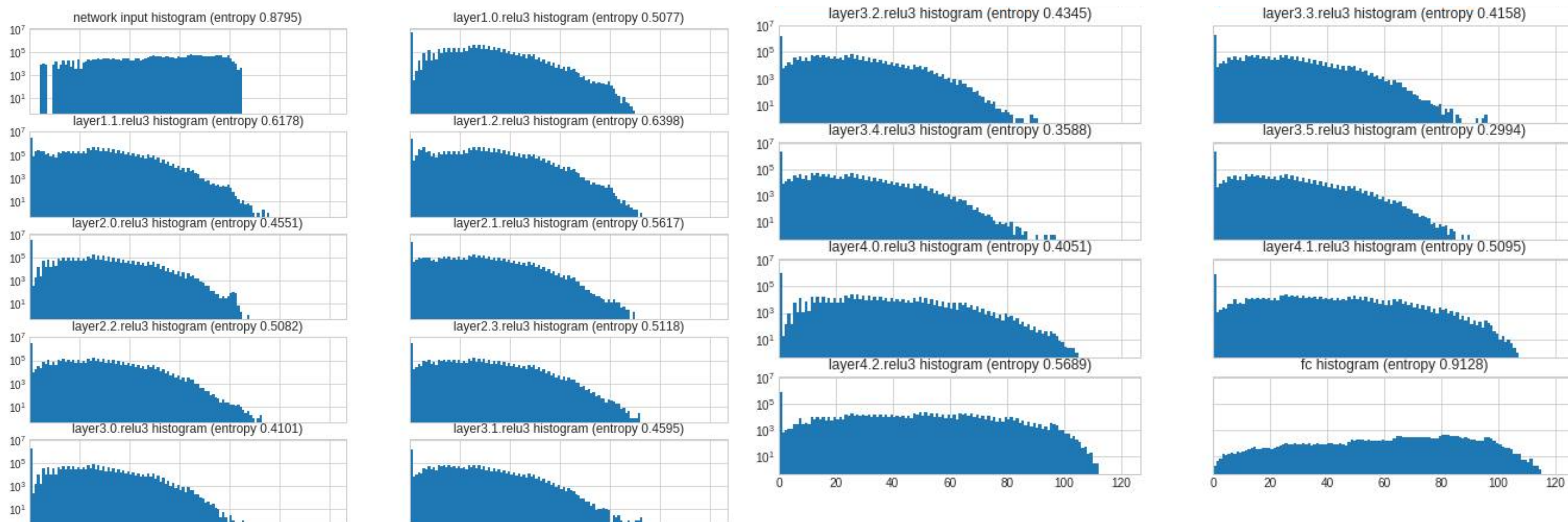
Quantization via entropy coding

Gustafson's posit is a good solution to decreasing quantization reproduction error for common FP data distributions. It encodes the floating point exponent in a prefix-free code, leaving the remainder to encode the significand.

It still assumes symmetry around ± 1.0 which is still not appropriate for many seen distributions.

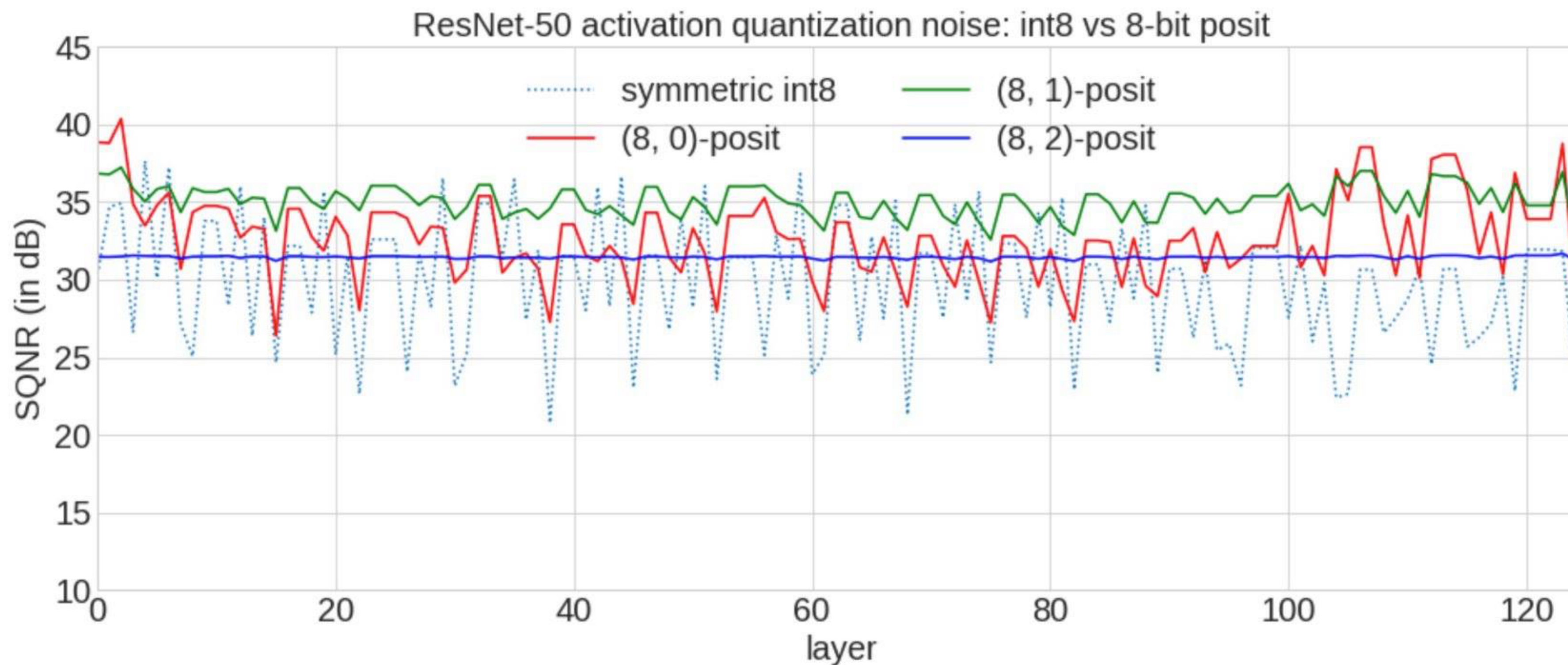


ResNet-50 v1 (8, 1) linear posit activation distribution



Quantization noise for activation encodings

The default posit representation is quite suitable for NNs, but I believe the real savings would be for training situations.



Combining the techniques

Posit-type encodings can be applied to log values as well (to the integer and fractional part of the fixed point exponent).

Intermediate results of computations that must be written out to memory can be quantized by a posit-type encoding, but operations on the value in-register or in a fixed function unit (e.g., convolver) can operate on the higher-precision value.

For model training, parameter update $w' = w + \alpha \Delta w$ is a single sum that will be inaccurate with ELMA/FLMA. However, since this is a memory bandwidth bound operation, a (lower throughput) unit performing 0.5 ulp LNS-style addition can be used.

Conclusion

- Many techniques to consider for “alternative math” representations in hardware, with a long history but little practical consideration in designs.
 - Judicious use of approximation can result in substantial increases in energy efficiency, and may even produce more accurate results for many use cases. Approximation for log domain arithmetic is very useful in avoiding log overheads.
 - Codecs (in the guise of quantization) can substantially reduce memory overheads.
- 3x+ energy efficiency in compute and 2-4x in memory from these techniques can enable more powerful AI model training and broader gains on a wide variety of non-high precision HPC-type workloads that still use half or single precision FP.

Rethinking floating point for deep learning

2018 NeurIPS Systems for ML Workshop paper

arXiv: 1811.01721

github.com/facebookresearch/deepfloat

facebook

Artificial Intelligence Research

