Quantization and Low-Precision Number Formats: A Reference

Lothrop Gauss

Nov 6, 2026

A Zoo of Low-Precision

Those who studied Numerical Methods, or on the CS side Compilers and Programming Languages will remember the headaches of IEEE 754. That seems like child's play with the veritable *zoo* of formats we now have to contend with given the boom of LLMs in recent years. Hence, this document. First, quickly, why, then straight into details sparing as much prose as I can stand.

The zoo of small formats exists because ML tolerates low precision, and the newest accelerators have datapaths for several of them. Two refinements matter.

- **Not all of it is quantization.** BF16 and FP8 were introduced as *training* formats (mixed precision), not as post-hoc compression. Quantization in the narrow sense means mapping a trained model to a coarser code, which may or may not be a hardware type.
- **Only a handful of formats have silicon.** Tensor/matrix units natively multiply FP16, BF16, TF32, INT8, FP8 (E4M3/E5M2), and on 2025+ parts FP6, FP4 and block-scaled MX/NVFP4. Most popular community quant schemes (GGUF k-quants and i-quants, GPTQ, AWQ, NF4, EXL2) are integer or lookup-table codes plus per-block scales that a kernel unpacks into FP16/BF16 (or INT8) before the actual matmul.

The rest of this doc covers why quantization pays, each format's bit layout, how scaling and outliers work, what happens to model quality, and which hardware runs what.

1. Why quantize

Fewer bits buy three different things, and which one you get depends on where the bottleneck is.

- **Capacity.** A 70B-parameter model is 140 GB in BF16, 70 GB at 8 bits, and roughly 35–40 GB at 4 bits once per-block scales are counted (about 4.25–4.5 bits per weight).
- **Bandwidth.** Every decode step streams all active weights through the memory bus. Halving bytes per weight roughly halves time per token when you are bandwidth-bound.
- **Compute.** A tensor core that natively multiplies an 8-bit or 4-bit type does about 2× or 4× the MACs per cycle of its 16-bit mode on most NVIDIA and AMD parts (section 7 lists the exceptions). You only get this if the matmul itself runs in the low format, not if you dequantize to BF16 first.

Arithmetic intensity: why decode and prefill differ

For one layer with a $d \times d$ weight matrix stored at b bits per weight and a batch of B tokens, a matmul does $2d^2B$ FLOPs and reads $d^2 \cdot b/8$ bytes of weights. Intensity is therefore

$$I = \frac{2d^2B}{d^2b/8} = \frac{16B}{b} \text{ FLOP/byte}$$

An H100 SXM has about 989 dense BF16 TFLOPS against 3.35 TB/s of HBM, a ridge point near 295 FLOP/byte. Single-stream decode ($B = 1$, $b = 16$) sits at 1 FLOP/byte, two orders of magnitude below the ridge, so it is purely bandwidth-bound: tokens/s $\approx$ bandwidth $\div$ bytes of active weights. Weight-only 4-bit quantization helps decode a lot even with no 4-bit math units. Prefill and training run B in the thousands, are compute-bound, and only speed up when the tensor cores run the narrow type.

What gets quantized

Tensor	Typical choice	Why it is hard or easy
Weights	8, 4, sometimes 2–3 bits	Static, known ahead of time, so you can spend offline compute (GPTQ, AWQ) choosing codes
Activations	16 or 8 bits, 4 bits on newest parts	Computed on the fly, heavy-tailed, with a few huge outlier channels
KV cache	16 → 8, sometimes 4 or 2 bits	Grows with context × batch; often dominates memory in long-context serving
Gradients	BF16/FP8 in training	Wide dynamic range; E5M2 was designed for them
Optimizer state	FP32, sometimes 8-bit	Adam moments are the largest training memory cost

Notation: **WxAy** means x-bit weights and y-bit activations. W4A16 is weight-only (dequantize, then BF16 math). W8A8 and W4A4 run the matmul in the low type.

The KV cache scales as 2 × layers × KV heads × head dim per token. For Llama-3-70B (80 layers, 8 KV heads, head dim 128) that is 320 KiB per token in BF16, or about 40 GiB for a single 128K-token sequence. Mixed-precision Adam training costs about 16 bytes per parameter before activations (BF16 weights and grads, FP32 master weights and two FP32 moments).

PTQ vs QAT

- **Post-training quantization (PTQ)** takes a trained model, maybe a few hundred calibration samples, and picks codes and scales. Cheap, minutes to hours. GPTQ, AWQ and the GGUF schemes in sections 4 and 5 are PTQ.
- **Quantization-aware training (QAT)** simulates quantization in the forward pass during training or fine-tuning, with a straight-through estimator for the gradient of the rounding step, so the weights adapt to the grid. More expensive, but it recovers most of the loss at 4 bits and below.
- **Low-precision training** is a third thing: the training itself runs in FP8 or FP4 with master weights kept wider. The model is then natively in that format.

2. The formats

Every float format here is a sign bit, E exponent bits with bias b, and M mantissa bits. Exponent bits buy dynamic range; mantissa bits buy relative precision (machine epsilon 2^{-M}). With exponent field e and mantissa field m:

$$x = (-1)^s\, 2^{e-b}\left(1 + \frac{m}{2^M}\right) \quad (e \neq 0), \qquad x = (-1)^s\, 2^{1-b}\,\frac{m}{2^M} \quad (e = 0,\ \text{subnormal})$$

IEEE-style formats reserve the all-ones exponent for $\pm\infty$ and NaN. The small ML formats mostly do not: they spend that code on finite values and keep at most one NaN pattern, because a 3- or 4-bit exponent cannot afford to waste a binade. The notation EeMm (E4M3, E2M1) names the field widths.

Wide formats (16–32 bits)

- **FP32 (E8M23, bias 127).** Max $\approx 3.4 \times 10^{38}$, epsilon $2^{-23} \approx 1.2 \times 10^{-7}$. The reference format; still used for master weights, optimizer state, reductions and accumulators.
- **TF32 (E8M10).** Not a storage format. NVIDIA tensor cores (Ampere onward) read FP32 operands, keep FP32's 8-bit exponent and truncate to a 10-bit mantissa, then accumulate in FP32. FP32 range with FP16 precision, transparently, for "FP32" matmuls.
- **FP16 (IEEE binary16, E5M10, bias 15).** Max 65504, min normal $2^{-14} \approx 6.1 \times 10^{-5}$, epsilon 2^{-10}. Good precision, narrow range: training needs loss scaling to keep small gradients from flushing to zero, and activations above 65504 overflow.
- **BF16 (E8M7, bias 127).** Google Brain's format: the top 16 bits of an FP32. Same range as FP32, epsilon $2^{-7} \approx 0.0078$ (about 2.4 decimal digits). Conversion from FP32 is a truncate-and-round, and no loss scaling is needed. The default training and serving format since about 2020.

FP8 (the OCP 8-bit float spec)

Two encodings, standardized by Arm, Intel and NVIDIA (later OCP), used together: E4M3 where precision matters, E5M2 where range matters.

Format	Bias	Max	Min normal	Min subnormal	∞?	NaN
E4M3 (OCP, also "E4M3FN")	7	448	2^{-6}	2^{-9}	No	S.1111.111 only
E5M2	15	57344	2^{-14}	2^{-16}	Yes	IEEE-style
E4M3FNUZ	8	240	2^{-7}	2^{-10}	No	single code at -0
E5M2FNUZ	16	57344	2^{-15}	2^{-17}	No	single code at -0

- **E4M3** gives up infinities and all but one NaN so the top binade holds values 256–448. NVIDIA's FP8 E4M3 is this OCP encoding. Typical use: weights and activations in forward passes.
- **E5M2** is literally FP16 with the low 8 mantissa bits dropped, so it keeps IEEE ∞/NaN. Typical use: gradients in training.

- **FNUZ** ("finite, no negative zero") variants come from Graphcore and were the only FP8 on AMD's MI300 (CDNA3); CDNA4 (MI350) switched to OCP FP8 only. Bias is one higher and -0 becomes the NaN, so E4M3FNUZ tops out at 240. Bit-identical codes mean different values across the two families, which matters when moving FP8 checkpoints between vendors.

FP8 has about 18 binades of range for E4M3, far too little to cover a whole tensor of an LLM without a scale factor. FP8 is always used as scaled FP8: $x \approx s \cdot q$ with s stored per tensor, per channel or per block in higher precision (section 4).

FP6 and FP4

These only exist as elements of a block-scaled format; standalone they are nearly useless. None has ∞ or NaN.

- **FP6 E2M3 (bias 1).** Max 7.5, min subnormal 0.125. More mantissa; usually the better FP6 for inference.
- **FP6 E3M2 (bias 3).** Max 28, min subnormal 0.0625. More range.
- **FP4 E2M1 (bias 1).** Sixteen codes, fifteen distinct values: $\pm\{0, 0.5, 1, 1.5, 2, 3, 4, 6\}$. The grid is denser near zero, which suits bell-shaped weight distributions better than INT4's uniform grid.

Block-scaled formats: OCP Microscaling (MX)

The OCP MX spec (v1.0, 2023; AMD, Arm, Intel, Meta, Microsoft, NVIDIA, Qualcomm) defines a block of $k = 32$ elements sharing one scale X:

$$v_i = X \cdot P_i, \qquad X = 2^{E_{\text{shared}}} \in \text{E8M0}, \quad i = 1, \ldots, 32$$

- **E8M0** (renamed UE8M0 in MX spec v1.1, October 2026) is an unsigned 8-bit exponent with bias 127: it encodes 2^{-127} through 2^{127}, with 0xFF reserved for NaN. A power-of-two scale means rescaling is an exponent add, not a multiply.
- **Element types:** MXFP8 (E4M3 or E5M2), MXFP6 (E2M3 or E3M2), MXFP4 (E2M1), and MXINT8 (8-bit two's complement with an implicit 2^{-6}, so values in $[-2, 1.984375]$).
- **Cost:** 8 scale bits per 32 elements is 0.25 bit per element, so MXFP8 = 8.25, MXFP6 = 6.25, MXFP4 = 4.25 bits per value.
- **Scale choice:** the reference recipe sets the shared exponent to $\lfloor \log_2(\max |x|) \rfloor$ minus the element format's max exponent, then rounds elements. Because the scale is a power of two, the block max lands anywhere in the top binade, so on average about half a binade of the element range is wasted.

NVFP4 (NVIDIA, Blackwell)

NVIDIA's 4-bit format fixes MXFP4's two weaknesses with a two-level scale:

- Elements are FP4 E2M1, in blocks of **16** instead of 32, so each scale tracks a smaller neighborhood.
- The block scale is **FP8 E4M3** instead of E8M0, so it carries 3 mantissa bits and can match the block max closely instead of only to the nearest power of two.
- E4M3's 18-binade range cannot span a whole tensor, so a **per-tensor FP32 scale** sits on top and maps the block scales into E4M3's range.

Cost is $4 + 8/16 = 4.5$ bits per value (plus a negligible per-tensor word). NVIDIA's own illustration puts scale-encoding MSE at 0.08 for an E4M3 scale vs 0.72 for E8M0. For DeepSeek-R1-0528, NVFP4 PTQ stayed within about 1 point of FP8 on every reported benchmark (MMLU-Pro 85 to 84, GPQA-Diamond 81 to 80, AIME24 89 to 91).

Integers

- **INT8** (two's complement, $[-128, 127]$, usually used symmetric $[-127, 127]$). Uniform grid, 256 levels. With a zero-point it can be asymmetric (affine): $x \approx s \cdot (q - z)$. The workhorse of CNN-era inference and of CPU/NPU inference.
- **INT4** ($[-8, 7]$ or unsigned $[0, 15]$ with a zero-point). Nearly always per-group scaled (group of 32–128). Mostly a storage format for weights; native INT4 tensor-core math has been dropped on recent NVIDIA datacenter parts in favor of FP4 (section 7).
- **INT2, ternary, binary.** Ternary $\{-1, 0, +1\}$ carries $\log_2 3 \approx 1.58$ bits. BitNet b1.58 trains ternary weights from scratch (absmean scaling) with 8-bit activations; matmuls collapse to additions and subtractions. These are training-time choices: post-hoc quantization of an ordinary model to 1–2 bits destroys it.

Floats vs integers at equal width

For a fixed bit budget, an integer grid is uniform and a float grid is logarithmic. Given a good scale per small block, INT8 has more precision than FP8 near the block max and FP8 handles heavy tails better. At 4 bits the same tradeoff holds between INT4 and FP4. Block scaling shrinks the gap: with 16- or 32-element blocks the tail problem mostly disappears, which is one reason MXINT8 and INT4-with-groups remain competitive.

3. Side-by-side

Range is $\log_2$(max ÷ smallest positive value) in binades. Bits per value includes amortized block scales. Element ranges for block formats are before scaling.

Format	Bits/value	Layout	Max (element)	Epsilon	Range (binades)	Typical role
FP32	32	E8M23	3.4e38	1.2e−7	277	Master weights, optimizer, accumulators
TF32	19 (in 32)	E8M10	3.4e38	9.8e−4	264	Tensor-core input for FP32 matmuls
FP16	16	E5M10	65504	9.8e−4	40	Inference, older mixed-precision training
BF16	16	E8M7	3.4e38	7.8e−3	261	Default training and serving
FP8 E4M3	8 (+scale)	E4M3	448	0.125	17.8	Weights/activations, FP8 training fwd
FP8 E5M2	8 (+scale)	E5M2	57344	0.25	31.8	Gradients

Format	Bits/value	Layout	Max (element)	Epsilon	Range (binades)	Typical role
MXFP8	8.25	E4M3/E5M2 × 32, E8M0 scale	448 / 57344	0.125 / 0.25	per block	Training and inference on Blackwell/CDNA4
MXFP6	6.25	E2M3/E3M2 × 32, E8M0 scale	7.5 / 28	0.125 / 0.25	5.9 / 8.8	Inference middle ground
MXFP4	4.25	E2M1 × 32, E8M0 scale	6	0.5	3.6	Weight storage, some W4A4
NVFP4	4.5	E2M1 × 16, E4M3 scale, FP32 tensor scale	6	0.5	3.6	W4A4 inference, 4-bit training on NVIDIA
MXINT8	8.25	INT8 × 32, E8M0 scale	1.98	uniform, 2^{-6}	n/a (uniform)	Inference
INT8	8 (+scale)	two's complement	127	uniform	n/a	W8A8 inference, CPUs/NPUs
INT4	4 (+group scale)	two's complement or + zero-point	7	uniform	n/a	Weight-only (W4A16)
Ternary	1.58	$\{-1, 0, 1\}$	1	n/a	n/a	Trained-from-scratch models (BitNet)

Two patterns stand out. FP8 and smaller formats cannot represent an LLM tensor without scales, so the scale scheme matters as much as the element type. And the jump from 8 to 4 bits is not just "half the bits": FP4 has 15 distinct values, so its quality depends almost entirely on how small and how precise the blocks' scales are.

4. Scaling granularity and outliers

The element format sets how many levels you have; the scale scheme decides where those levels sit. Finer scales cost bits and kernel complexity but are what made 4-bit floats workable.

Granularity

Take $y = Wx$ with W of shape $N \times K$ (K is the reduction dimension).

Granularity	One scale per	Can be factored out of the dot product?	Typical use
Per-tensor	whole tensor	Yes, one multiply in the epilogue	FP8 with delayed scaling (Hopper Transformer Engine)
Per-channel (weights)	output row of W	Yes, a per-column multiply on y	INT8 weights

Granularity	One scale per	Can be factored out of the dot product?	Typical use
Per-token (activations)	row of X	Yes, a per-row multiply on y	Dynamic INT8/FP8 activations
Per-group / per-block	g elements along K (g = 16–128)	**No**, the scale varies inside the sum	INT4 weight-only, MX, NVFP4, DeepSeek FP8

Per-channel and per-token scales commute with the matmul: $Y = (s_x s_w^\top) \odot (Q_x Q_w^\top)$, so the tensor core runs pure integer or FP8 math and a cheap epilogue rescales. Block scaling along K breaks that, because the dot product becomes a sum of differently scaled partial sums. Software handles it by dequantizing to BF16 before the matmul (fine for weight-only, bandwidth-bound decode) or by accumulating per-block partial sums and rescaling them in a higher-precision accumulator. DeepSeek-V3's FP8 training uses 1×128 activation tiles and 128×128 weight blocks with exactly this kind of promoted accumulation.

The Blackwell-generation change is that **block-scaled MMA is in the tensor core**: MX and NVFP4 instructions take the element tiles and their scale vectors and apply scales inside the operation. That removes the dequantization tax and is the main reason FP4 compute is usable rather than a curiosity.

Why block scaling rescued 4-bit floats

E2M1 spans only 3.6 binades. A whole weight matrix spans many more, and activations span more still. With 16 or 32 elements per scale, the local dynamic range is usually small enough to fit, and an outlier ruins only its own block instead of the whole tensor. NVFP4's E4M3 scale lets the block max hit the top of the FP4 grid closely, while MXFP4's power-of-two scale wastes up to a binade, which is a large fraction of 3.6.

The outlier problem

Transformer activations have a few hidden dimensions whose magnitudes are tens to hundreds of times the rest, consistently across tokens. With per-tensor or per-token activation scaling, these channels stretch the grid and everything else collapses onto a few levels. Weights are comparatively well-behaved. That asymmetry drives most PTQ methods:

- **LLM.int8()** (Dettmers et al., 2022). Splits the matmul: the few emergent outlier feature dimensions (under 0.1% of values) run in FP16, the rest in INT8 with vector-wise scales. Dettmers traces a phase shift near 6.7B parameters where the same outlier dimensions appear in every layer. Near-lossless to 175B, but the decomposition is slow.
- **SmoothQuant** (Xiao et al., 2022). Migrates difficulty from activations to weights with an exact per-channel rescale: $X' = X\ \mathrm{diag}(s)^{-1}$, $W' = \mathrm{diag}(s)\ W$, with $s_j = \max|X_j|^\alpha / \max|W_j|^{1-\alpha}$, where the migration strength α is tuned per model (the reference repo uses 0.85 for Llama-2-7B and Llama-3, 0.9 for Llama-2-70B). Enables plain W8A8 INT8.
- **AWQ** (Lin et al., 2023). Weight-only. Finds the small fraction of weight channels that matter (by activation magnitude, not weight magnitude) and scales them up before quantization so they lose less precision. No backprop, little calibration data, robust across domains.
- **GPTQ** (Frantar et al., 2022). Layer-wise second-order rounding from Optimal Brain Quantization: quantize W column by column and update the not-yet-quantized columns to cancel

the error, using the inverse Hessian $H = 2XX^\top$ from calibration activations. 3–4 bit weights for 100B+ models in hours on one GPU.

- **Rotations: QuaRot, SpinQuant.** Multiply by an orthogonal matrix R and its inverse on either side of a linear map: $(WR)(R^\top x) = Wx$. A randomized Hadamard R spreads an outlier's energy across all channels (incoherence), leaving near-Gaussian vectors that quantize well. With RMSNorm-based transformers, most rotations can be folded into the weights for free. QuaRot uses fixed Hadamards for W4A4 with a 4-bit KV cache; SpinQuant learns R. NVIDIA's NVFP4 training recipe also uses random Hadamard transforms.
- **Vector quantization: QuIP#, QTIP, AQLM.** Quantize groups of weights jointly to codebook or lattice points (QuIP# uses the E8 lattice, QTIP a trellis code). State of the art at 2 bits, decode-only (dequantize in the kernel); no native hardware.

These stack. A modern recipe might rotate, then smooth, then GPTQ the weights onto an NVFP4 or INT4 grid with an FP8 KV cache.

5. Software schemes vs native formats

Most quantized checkpoints you download are not a hardware number format. They are packed integer (or codebook) indices plus per-block scales, and a kernel unpacks them to FP16/BF16 or INT8 in registers right before the matmul. They win on bandwidth-bound decode and lose nothing on hardware support, but they never use FP4/FP8 math units. The exceptions are FP8, MXFP4/NVFP4 checkpoints (gpt-oss, NVIDIA's ModelOpt exports), which do map onto tensor cores.

Scheme	What is stored	Bits/weight	Calibration	Native silicon?
GGUF k-quants (Q2_K–Q6_K)	Integer codes in 256-weight super-blocks of 16–32-weight sub-blocks, with quantized sub-block scales (and mins)	3.2–6.6 file average	Optional importance matrix	No; dequant in kernel
GGUF i-quants (IQ1–IQ4)	Groups of 8 magnitudes mapped to E8-lattice codebook points, signs with a parity trick	2.0–4.5	Importance matrix effectively required	No
GPTQ	INT4/INT3 with group scales (often g = 128), Hessian-corrected rounding	~4.1–4.25	Yes, a few hundred samples	Dequant to FP16, or INT4×FP16 kernels (Marlin, Machete)
AWQ	INT4 with group scales, salient channels pre-scaled	~4.1–4.25	Yes, small	Same as GPTQ
bitsandbytes NF4	4-bit index into 16 quantiles of N(0,1), block of 64, double-quantized FP8 constants	~4.13	No	No; lookup table
HQQ	INT2–8 with group scales and zero-points fit by half-quadratic optimization	2–8 + scales	No	No
EXL2	GPTQ-style, 2–8 bit mixed within and across layers to hit a target average	any 2–8 average	Yes, measures per-layer error	No

Scheme	What is stored	Bits/weight	Calibration	Native silicon?
EXL3	QTIP-style trellis code with procedural codebook	2–8	Yes	No
FP8 checkpoints	E4M3 weights (and often activations) with per-tensor/channel/block scales	8 + scales	Light	Yes: Ada, Hopper, Blackwell, CDNA3/4, Gaudi
MXFP4 / NVFP4 checkpoints	E2M1 blocks with E8M0 or E4M3 scales	4.25 / 4.5	Light (PTQ) or QAT	Yes: Blackwell, CDNA4

Details worth knowing:

- **k-quant internals.** Q4_K and Q5_K use 8 sub-blocks of 32 with an affine $x = a \cdot q + b$ and 6-bit quantized scales and mins (4.5 and 5.5 bpw nominal). Q3_K, Q2_K and Q6_K use 16 sub-blocks of 16. The _S/_M/_L suffix decides which tensors (attention V, FFN down) get bumped up a type, which is why files average higher than the nominal rate: Q4_K_M is 4.89 bpw on Llama-3.1-8B (llama.cpp quantize README).
- **i-quants** come from the QuIP# line of work: incoherence plus lattice codebooks, with an importance matrix computed from calibration text deciding which weights get the precision. IQ2_XXS is 2.06 bpw nominal; its author's note is that it is garbage without the importance matrix (PR #4773).
- **NF4** levels are quantiles of a standard normal rescaled to $[-1, 1]$, so each code is equally likely for Gaussian weights; double quantization stores the block constants in FP8 with block 256, cutting scale overhead from 0.5 to about 0.127 bit/param (QLoRA).
- **HQQ** is calibration-free and fast (Llama-2-70B in under 5 minutes) with quality close to GPTQ/AWQ at 4 bits (HQQ).
- **MXFP4 in the wild.** OpenAI's gpt-oss shipped its MoE weights (over 90% of parameters) in MXFP4 at 4.25 bits, letting the 117B-parameter model fit on one 80 GB GPU (model card). llama.cpp added a native MXFP4 GGUF type for it.

The practical difference: a native format speeds up prefill and large-batch serving (compute-bound) because the tensor core consumes it directly. A software format speeds up only memory-bound decode and costs some ALU work to unpack; at large batch, unpacking can make it slower than plain FP8.

6. What it does to model behavior

8-bit is close to free, 4-bit weight-only costs a little, and everything below 4 bits or with 4-bit activations depends heavily on model size, method and task. Damage shows up first in small models, hard reasoning, long contexts, and models trained on very many tokens.

Headline results

Setting	Finding	Source
W8A8 FP8, Llama-3.1 8B–405B	"Effectively lossless" across scales	Kurtic et al. 2024
W8A8 INT8, same models	1–3% loss when well tuned	Kurtic et al. 2024
W4A16 INT, same models	"Rivals 8-bit": about 96% recovery at 8B, 97–99% at 70B/405B on Open LLM Leaderboard v2	Kurtic et al. 2024
Total-bit budget, 19M–176B models	4-bit is almost universally optimal for accuracy per model bit; only smaller groups and better data types improve on it	Dettmers & Zettlemoyer 2022
Llama-3-8B, WikiText-2 PPL (FP16 6.1)	GPTQ: 6.5 at 4-bit, 8.2 at 3-bit, 210 at 2-bit; plain round-to-nearest: 8.5 at 4-bit	Huang et al. 2024
Reasoning models, 1.5B–70B	W8A8 and W4A16 near-lossless; W4A4KV4 loses about 3 points at 14–32B but 11–12 at 1.5–7B	Liu et al. 2025
Long context (64K+ inputs)	8-bit loses about 0.8% on average; 4-bit up to 59%, worst on long and non-English inputs	Mekala et al. 2025
NVFP4 vs MXFP4, Llama-3.3-70B MMLU	81.9 vs 77.8 (FP8 82.5)	NVIDIA 2025

GGUF on one model

llama.cpp's own scoreboard for Llama-3-8B on WikiText-2, with KL divergence against FP16 logits (source). KLD is the better metric: perplexity can stay flat while the token distribution drifts.

Type	Size (GiB)	PPL	Mean KLD	Same top token
F16	14.97	6.233	0	100%
Q8_0	7.96	6.234	0.0014	97.7%
Q6_K	6.14	6.253	0.0055	96.0%
Q5_K_M	5.33	6.289	0.0108	
Q4_K_M	4.58	6.407	0.031	91.9%

Type	Size (GiB)	PPL	Mean KLD	Same top token
Q3_K_M	3.74	6.888	0.102	
IQ2_M	2.74	8.60	0.326	
Q2_K	2.96	9.75	0.445	71.1%
IQ1_S	1.88	60.7	2.25	

The same table shows Llama-2-7B's Q4_K_M at a KLD of 0.0127 versus 0.031 for Llama-3-8B, and for Llama-2-70B Q4_K_M costs only +1.2% perplexity. The "knee" is between Q4 and Q3 for 7–8B models and moves lower as models get bigger.

Where degradation shows first

- **Small models.** Fewer parameters means less redundancy per weight. A 3-bit 32B reasoning model (about 14.5 GB) beat a BF16 7B model by about 17% on LiveCodeBench, so for a fixed memory budget a bigger model at lower bits usually wins (Liu et al.).
- **Heavily trained models.** Llama-3 (15T+ tokens) is consistently more sensitive than Llama-2. Kumar et al. formalize it: PTQ damage grows with pretraining data, to the point where more tokens can hurt a model destined for quantization.
- **Hard reasoning, math, code.** Errors compound over long chains. For a 32B reasoning model at W4A4KV4, AIME dropped 3.9 points, MATH-500 1.2, GSM8K 0.0. Aggressive settings (3-bit weights, W4A4KV4) also make outputs longer, especially at 1.5B.
- **Long context and multilingual.** Small per-token errors in attention accumulate across many keys; 4-bit results on 64K+ inputs are strongly model-dependent (under NF4, Qwen-2.5-72B held up while Llama-3.1-70B dropped 32% on the same task).
- **Calibration mismatch.** GPTQ-style methods overfit their calibration set. Using reasoning traces as calibration data added 9.8 points at 3-bit on a 1.5B reasoning model.

KV cache quantization

The KV cache is the dominant memory cost at long context and high batch, and it tolerates quantization differently for keys and values.

- **Keys have per-channel outliers, values do not.** KIVI quantizes keys per channel and values per token to 2 bits, keeping a recent window in full precision, for 2.6× lower peak memory and 2.35–3.47× throughput.
- **KVQuant** adds pre-RoPE key quantization and sparse outlier handling: under 0.1 perplexity increase at 3 bits.
- **FP8 KV** is the production default: vLLM supports E4M3 and E5M2 with per-tensor or per-head scales.
- **NVFP4 KV** on Blackwell halves FP8's footprint with under 1% accuracy loss on the reported benchmarks (Qwen3-480B: MMLU-Pro 77.4 vs 78.1 for FP8; RULER-64K 94.6 vs 95.5). Values are dequantized to FP8 before attention (NVIDIA).
- In the reasoning study, KV4 was usually fine on its own; it is the combination with 4-bit activations that hurts.

Low-precision training

- **FP8 at frontier scale.** DeepSeek-V3 (671B total, 14.8T tokens) trained with E4M3 on all GEMM inputs, 1×128 / 128×128 block scales, and FP32 promotion of partial sums every 128 elements, because H800 FP8 accumulation keeps only about 14 bits. Embeddings, output head, gating, norms and attention stayed in BF16/FP32. Relative loss error vs BF16 stayed under 0.25%.
- **4-bit pretraining.** NVIDIA's NVFP4 run: a 12B hybrid Mamba-Transformer on 10T tokens matched FP8 closely (MMLU-Pro 62.58 vs 62.62) using random Hadamard transforms, 2D block scaling, stochastic rounding on gradients, and about 16% of linear layers kept in BF16. Code lagged most (MBPP+ 55.9 vs 59.1). MXFP4 needed about 36% more tokens to match NVFP4's loss.

Rules of thumb

1. FP8 W8A8 (or MXFP8) is the safe default for serving on hardware that has it; treat it as lossless until an eval says otherwise.
2. 4-bit weight-only (INT4 with groups of 128 or less via AWQ/GPTQ, Q4_K_M, NVFP4/MXFP4 weights) costs about 1–4% on 8B models and less on 70B+. Check math, code and long-context evals specifically.
3. Below 4 bits, use importance-aware or vector-quantization methods (imatrix i-quants, QuIP#, QTIP, EXL3) and prefer a larger model at 3 bits over a smaller one at 8.
4. W4A4 needs rotations or NVFP4-class block scaling; MXFP4 activations are noticeably worse than NVFP4.
5. Always measure KL divergence or task evals on your own workload: perplexity on WikiText hides the failures that matter.

7. Hardware

Each accelerator generation has added one narrower native type: FP16 (2017), INT8/INT4 (2018), TF32/BF16 (2020), FP8 (2022), FP6/FP4 with block scaling (2024–25). On NVIDIA and AMD each halving of width roughly doubles peak matrix throughput, but not everywhere, and integer support is being traded away for floats.

Native matrix formats by architecture

Vendor	Architecture (example parts)	Year	Native matrix formats added	Notes
NVIDIA	Volta (V100)	2017	FP16 (FP32 accumulate)	First tensor cores
NVIDIA	Turing (T4, RTX 20)	2018	INT8, INT4	
NVIDIA	Ampere (A100, RTX 30)	2020	TF32, BF16	2:4 structured sparsity doubles the sparse rate
NVIDIA	Ada (L40S, RTX 40)	2022	FP8 E4M3/E5M2	INT4 still present
NVIDIA	Hopper (H100, H200)	2022	FP8 with Transformer Engine	INT4 dropped from tensor cores

Vendor	Architecture (example parts)	Year	Native matrix formats added	Notes
NVIDIA	Blackwell (B200, GB200, RTX 50)	2024–25	FP6, FP4, block-scaled MXFP8/6/4 and NVFP4	Scales applied inside the MMA (tcgen05 block_scale kinds)
NVIDIA	Blackwell Ultra (B300, GB300)	2025	Faster NVFP4	INT8 and FP64 cut sharply
NVIDIA	Rubin (VR200)	2026	NVFP4-first; architecture blog also describes a 3-bit lookup-table operand format	Samples Feb 2026, production 2H 2026
AMD	CDNA3 (MI300X, MI325X)	2023	FP8 (FNUZ only), BF16, INT8, TF32	FNUZ encodings differ from NVIDIA FP8
AMD	CDNA4 (MI350X, MI355X)	2025	OCP FP8, MXFP8, MXFP6, MXFP4	FP6 runs at the FP4 rate; TF32 now emulated via BF16
AMD	MI400 (MI455X)	2026	Same MX/OCP set, INT8	HBM4
Intel	Gaudi 3	2024	FP8 E4M3/E5M2, BF16	FP8 matmul at the same rate as BF16
Intel	Xeon AMX: Sapphire Rapids / Granite Rapids / Diamond Rapids	2023 / 2024 / 2027	INT8 + BF16 / FP16 / FP8	Diamond Rapids slipped to 2027, so no shipping Xeon has AMX-FP8
Google	TPU v4 / v5e / v5p / v6e / Ironwood (TPU7x)	2021–2025	BF16 throughout; INT8 on v5e, v6e; FP8 on v5p, TPU7x	FP32 accumulate
Google	TPU 8t / 8i	announced Apr 2026	Native FP4	No GA date yet
Apple	M5 GPU "Neural Accelerators"	2025	Matmul units; Metal tensor ops gained BF16 and INT8/INT4 tensors in OS 26.x	Apple gives no per-format rates
Apple	Neural Engine		FP16, INT8 activations with 8/4-bit weights	INT8×INT8 fast path on A17 Pro/M4
Arm	Armv8.6-A (i8mm, BF16), SME2	2019+	INT8 and BF16 matrix-multiply instructions; SME2 LUT decompress of 2/4-bit	Neoverse, Apple, phones
Qualcomm	Hexagon NPU (Snapdragon 8 Elite Gen 5/6)	2025–26	INT2, INT4, INT8, INT16, FP8, FP16	No FP4 listed

Peak dense throughput by format

Vendor figures, dense (no sparsity), per GPU or chip, in TFLOPS or TOPS. A blank cell means the format is not native or not published. These are marketing peaks: real kernels reach 50–80% on big GEMMs and far less on small ones.

Part	BF16	FP8	FP4	INT8	Memory BW (TB/s)
NVIDIA A100	312			624	1.56 (40 GB)
NVIDIA H100 SXM	989	1,979		1,979	3.35
NVIDIA B200 (HGX, per GPU)	2,250	4,500	9,000	4,500	8
NVIDIA B300 / GB300		5,000	15,000	small fraction of B200	8
NVIDIA Rubin	4,000	17,500	35,000 (training); 50,000 sparse inference	not listed	19.2 (blog: up to 22)
NVIDIA RTX 4090		660.6 (FP16 accum) / 330.3 (FP32 accum)			
NVIDIA RTX 5090	209.5 (FP16, FP32 accum)	838 / 419	1,676		
AMD MI300X	1,307	2,615		2,615	5.3
AMD MI355X	2,500	5,000	10,100 (MXFP6 same)	5,000	8
AMD MI455X	5,000	20,100 (unlabeled)	40,300 MXFP4 (unlabeled)		23.3
Intel Gaudi 3	1,678	1,678			3.7
Google TPU v6e	918			1,836	1.64
Google Ironwood (TPU7x)	2,307	4,614			7.37

Sources: A100, H100, HGX B200/B300/Rubin, Blackwell Ultra, Vera Rubin NVL72, Rubin architecture, RTX Blackwell whitepaper, MI300X, MI355X, MI455X, Gaudi 3 whitepaper, TPU v6e, TPU7x. B200 per-GPU figures are HGX 8-GPU totals divided by 8; NVIDIA's Blackwell Ultra blog quotes slightly higher figures (10 PF NVFP4) for what appears to be the GB200 configuration. HGX B300 lists 3 POPS INT8 for eight GPUs against 72 for HGX B200.

How throughput scales with width

- **The usual 2× per halving.** B200 goes 2.25 → 4.5 → 9 PF for BF16 → FP8 → FP4; MI355X and H100 follow the same pattern. Multiplier area scales roughly with mantissa width squared, so narrower operands pack more MACs per mm² and per watt.
- **Exceptions.** Gaudi 3 and TPU v5p run FP8 at the BF16 rate, so FP8 there buys memory and bandwidth, not FLOPs. MI355X runs FP6 at the FP4 rate. B300 and Rubin push NVFP4 to 3–4× their FP8 rate by spending area on FP4 at the expense of INT8 and FP64.
- **Accumulator width matters on consumer parts.** GeForce cards run FP8 and FP16 with FP32 accumulation at half the rate of FP16 accumulation; datacenter parts do not have this penalty.

- **Integers are fading on GPUs.** Hopper dropped INT4, Blackwell Ultra gutted INT8, Rubin does not list it. CPUs, NPUs and TPUs v5e/v6e remain the INT8 strongholds.

Compute-bound vs bandwidth-bound

From section 1, a decode step at batch B with b-bit weights has intensity of about 16B/b FLOP/byte, while the ridge point of these chips is in the hundreds (H100: about 295 for BF16, about 590 for FP8).

- **Single-stream decode** is bandwidth-bound. An upper bound is tokens/s ≈ memory bandwidth ÷ bytes of active weights. A 70B dense model at about 4.5 bpw (about 39 GB) on an H100 caps near 3,350 / 39 ≈ 85 tokens/s, against about 24 tokens/s for the same model in BF16 if it fit. Software 4-bit formats capture nearly all of this without native FP4.
- **Prefill, large-batch serving and training** are compute-bound. Only native formats help, which is why FP8 serving and FP8/FP4 training need tensor-core support while llama.cpp-style quants run well on any GPU or CPU.
- **MoE models** shift the balance: only active experts are read per token (gpt-oss-120b activates about 5.1B of 117B parameters), so decode is cheaper, but the full weights must still fit, which makes capacity the binding constraint and 4-bit weights especially attractive.
- **KV cache reads** grow with context and batch and are bandwidth-bound too, which is why FP8 and NVFP4 KV caches pay off directly in long-context throughput.

8. What to pick

Match the format to the bottleneck: software 4-bit for bandwidth-bound local decode, native FP8 or FP4 for compute-bound serving and training.

Situation	Weights	Activations / KV	Why
Consumer GPU, 24–32 GB, llama.cpp or ExLlama	Q4_K_M–Q6_K GGUF, or EXL2/EXL3 at 4–5 bpw; imatrix i-quants (IQ3/IQ2) only to squeeze a bigger model in	FP16 activations; Q8_0 KV cache for long context	Decode is bandwidth-bound; a larger model at about 4 bits beats a smaller one at 8
RTX 40 (Ada) with vLLM or TensorRT-LLM	FP8 or AWQ/GPTQ INT4 (Marlin kernels)	FP8 KV	FP8 is native on Ada
RTX 50 (Blackwell)	NVFP4 checkpoints where published, otherwise as above	FP8 or NVFP4	FP4 speeds prefill; decode still tracks GDDR bandwidth
Apple silicon	MLX 4-bit affine or MXFP4 (gpt-oss)	BF16/FP16	Unified memory makes capacity cheap; generation is bandwidth-bound, M5 mostly speeds prompt processing
CPU only	Q4_K / Q8_0 GGUF	INT8 dot products via VNNI, AMX or i8mm	Integer SIMD is what CPUs accelerate

Situation	Weights	Activations / KV	Why
Datacenter, Hopper / MI300X	FP8 W8A8 (per-channel weights, dynamic per-token activations); W4A16 AWQ/GPTQ for latency-bound small batch	FP8 KV cache	FP8 is near-lossless and doubles compute; convert FNUZ vs OCP FP8 when moving checkpoints to or from MI300
Datacenter, Blackwell / MI355X / Rubin	NVFP4 (NVIDIA) or MXFP4/MXFP6 (AMD) for throughput, MXFP8/FP8 as the safe fallback	FP8 or NVFP4 KV	Block-scaled FP4 is the native fast path; validate reasoning, code and long-context evals before switching
Google TPU	BF16, INT8 on v5e/v6e, FP8 on v5p/Ironwood	BF16	Only what the MXU supports gets a speedup
Pretraining	BF16 baseline; FP8 with block scales (DeepSeek-V3 style) or MXFP8 on Blackwell; NVFP4 with a full recipe for the adventurous	Keep embeddings, head, norms, attention softmax, master weights and optimizer state in BF16/FP32	FP8 training is proven at 671B; NVFP4 is proven at 12B/10T tokens
A model you will ship at 4 bits	Fine-tune with QAT, or train natively in NVFP4/MXFP4		QAT recovers most of the PTQ gap below 8 bits

Three checks before trusting a quantized model: KL divergence (or top-token agreement) against the unquantized model on your own prompts, your hardest task evals (math, code, multi-step reasoning), and a long-context test at the lengths you actually use.

Sources

Specs

- OCP 8-bit Floating Point Specification (OFP8), rev 1.1
- FP8 Formats for Deep Learning (NVIDIA, Arm, Intel)
- OCP Microscaling Formats spec v1.1 (Oct 2026)
- Microscaling Data Formats for Deep Learning
- StableHLO FP8 FNUZ RFC
- ROCm FP8 numbers and ROCm precision support
- Introducing NVFP4 (NVIDIA)
- Transformer Engine FP8 primer
- CUTLASS tcgen05 MMA docs

Methods and quality

- LLM.int8() and Dettmers on emergent features
- SmoothQuant and repo
- GPTQ, AWQ, QuaRot, SpinQuant, QuIP#, QTIP, QServe
- QLoRA / NF4, HQQ, ExLlamaV3 / EXL3
- llama.cpp perplexity scoreboard, quantize README, i-quants PR #4773
- Give Me BF16 or Give Me Death?
- The case for 4-bit precision

- Scaling Laws for Precision
- An empirical study of LLaMA3 quantization
- Quantization Hurts Reasoning?
- Does quantization affect long-context tasks?
- KIVI, KVQuant, vLLM quantized KV cache, NVFP4 KV cache (NVIDIA)
- DeepSeek-V3 technical report, Pretraining LLMs with NVFP4
- BitNet b1.58, BitNet b1.58 2B4T, gpt-oss model card

Hardware

- NVIDIA: Volta whitepaper, Turing, Ampere, L40S, Hopper, HGX, Blackwell Ultra, RTX Blackwell whitepaper, Vera Rubin NVL72, Rubin architecture
- AMD: MI300X, MI355X, CDNA 4 whitepaper, MI455X
- Intel: Gaudi 3 whitepaper, ISA extensions reference rev 062, AMX-TMUL guide
- Google: TPU system architecture, v6e, TPU7x, TPU 8t and 8i
- Apple: M5 announcement, Metal tensor tech talk, MLX on M5, Core ML quantization
- Arm and Qualcomm: Armv8.6-A, SME2, Snapdragon 8 Elite Gen 5 brief

Not verified from a vendor page: RTX 50 block-scaled MMA support by name (NVIDIA calls it FP4 with FP8 block scales), whether Apple GPUs ran BF16 natively before M5, and the per-format rates of Apple's Neural Accelerators.