

Is 2:4 Sparsity Still Worth It? Semi-Structured Sparsity Versus Quantization in LLM Inference

Xiuyi Qin¹

ABSTRACT

NVIDIA tensor cores have supported 2:4 semi-structured sparsity since Ampere, and the pattern composes with quantization. When the two are combined, the practical question is which step to take next: from a given precision, add 2:4 sparsity or drop to the next lower precision? We compare the two at equal bits, with sparsity metadata counted, and at equal nominal tensor-core tiers, over all six GEMM operands of a transformer layer, with kernels measured on three GPU generations and online pruning and quantization included. Our results show that as long as a lower precision is available, adding 2:4 is the worse step, both less accurate and slower, because the energy that 2:4 discards from each group of four along the hardware-forced axis does not shrink with bits, while quantization error does. Once quantization reaches the lowest precision the hardware supports, 2:4 still speeds up linear layers by 1.3–1.7 $\times$, and on an H100 a retrained 2:4 model served in FP8 by vLLM reaches 27% higher prefill throughput than the dense FP8 model, with GSM8K accuracy on par. Its quality has to be won back by retraining, however, which restores perplexity long before reasoning; on Blackwell, whose sparse FP4 uses a pairwise 4:8 pattern, the loss is larger and our retraining budget did not recover it.

1 INTRODUCTION

Semi-structured 2:4 sparsity keeps at most two non-zeros in every group of four consecutive elements along the reduction axis of a matrix multiplication, and NVIDIA’s sparse tensor cores run such a multiplication at up to twice the dense rate (Mishra et al., 2021). However, hardware and research on one side and inference software on the other have diverged on whether the feature is worth using. NVIDIA has kept sparse tensor cores in every GPU generation since Ampere and added a sparse FP4 mode in Blackwell, and systems research keeps building on them, for activations (Haziza et al., 2025), for keys and values (Wang & Wang, 2026) and for attention scores (Yu et al., 2026). In 2026, by contrast, vLLM and LLM Compressor, among the most widely used open-source stacks for compressed LLM inference, removed their 2:4 support because few users relied on it (vLLM maintainers, 2026b; LLM Compressor maintainers, 2026). A systematic comparison is urgently needed to settle the question.

Because 2:4 sparsity composes with quantization, the choice a practitioner actually faces is about the next step: **from a given precision, add 2:4 sparsity, or drop to the next lower precision?** The answer depends on whether a lower

precision is still available. We call the lowest dense precision that a GPU’s tensor cores offer for LLM inference the *quantization floor*: FP8 on Ada and Hopper, NVFP4 on Blackwell. Above the floor, 2:4 has to beat one more step of quantization. At the floor, no such step remains, and this is where the real value of 2:4 emerges.

Existing comparisons of pruning and quantization cannot settle this choice for 2:4. Kuzmin et al. (2023) compare unstructured pruning with quantization at nominal compression, mostly for weights, and predict the winner from kurtosis, which measures how heavy-tailed a tensor’s values are. Harma et al. (2025) show that pruning should precede quantization when the two are combined, and Frantar et al. (2025) fold both into a scaling law for pre-training. SLiM (Mozaffari et al., 2025) reports that 4-bit values with 2:4 sparsity beat 2-bit quantization, but at nominal bits; with its indices counted, that configuration costs 3.1 bits. SLiM also measures its speedup on GPUs, against dense 16-bit models; none of these studies compares adding 2:4 with dropping to a lower dense precision on the hardware. The 2:4 question differs from theirs in four ways. A 2:4 tensor stores a position index next to every kept value, so it costs more than half of its dense counterpart; among prior studies, only Maximov et al. (2026) count such metadata, for 8:16 patterns. Sparsity and lower precision each promise twice the tensor-core throughput, but the datasheet says nothing about what each delivers. Most operands at inference time, namely activations, queries, keys, values and attention prob-

¹Independent Researcher, San Jose, CA, USA. Correspondence to: Xiuyi Qin <xiuyiq@acm.org>.

abilities, are produced on the fly and must be pruned and compressed before every GEMM, a cost that benchmarks of the sparse GEMM alone leave out. And the hardware fixes which operand of a GEMM can be sparse and along which axis, so statistics of the whole tensor, such as kurtosis, miss what 2:4 actually removes.

We compare the two steps at equal bits, with the 2:4 index counted, and at equal tensor-core tiers (Section 3), for each of the six operand families of a transformer layer, sparsified along the axis the hardware forces. The comparison covers every layer of two 8B models and is timed on Ada, Hopper and Blackwell GPUs, including the online pruning and quantization of dynamic operands.

Above the floor, our results show that dropping one precision step is more accurate than adding 2:4 at every storage budget, and that at every tensor-core tier some dense format is both faster and more accurate (Section 4). Part of the reason is accounting: 2:4 on $2m$ -bit values costs exactly as many bits as dense $(m+1)$ -bit values, so 2:4 with INT8 values competes with INT5, not INT4. The larger part is the error itself (Section 5). The two values that 2:4 drops from each group of four carry a share of the tensor’s energy, the *2:4 energy loss*, that no number of bits for the kept values recovers, whereas quantization error halves with every added bit. Computed from the tensor alone, the 2:4 energy loss tracks the measured gap closely, though it cannot call a close contest, and it explains why Hadamard rotation widens the gap for most operands.

At the floor, 2:4 delivers a real speedup (Section 6). Stacked on FP8, it speeds up linear layers by 1.3–1.7 $\times$, and a retrained 2:4 model served by vLLM is faster than the dense FP8 model in prefill and decode, with GSM8K accuracy on par; serving the same weights with dense kernels removes the gain. The price is retraining. One-shot pruning at the floor is unusable, and along our retraining runs perplexity recovers within a quarter billion tokens while GSM8K accuracy stalls far below that of Sparse-Llama, a model retrained thirteen times longer on different data. Perplexity, the common yardstick for retrained sparse models, therefore overstates how much they have recovered. Blackwell’s sparse FP4 uses a pairwise 4:8 pattern (NVIDIA, 2026): its kernels run 1.4 $\times$ faster than dense NVFP4, but the pattern discards more of every tensor, cannot reuse models retrained for 2:4, and at our budget does not recover usable reasoning accuracy.

Code, kernels and measurements are available at https://github.com/PuppyQ08/24_quant_compare.

2 BACKGROUND AND RELATED WORK

Sparse tensor cores. A sparse MMA instruction takes its first operand A in compressed form, the kept values plus a 2-

bit index for each, and uses the indices to select the matching elements of the dense operand B . The multiplier array thus covers twice the reduction depth per cycle, which bounds the speedup at $2\times$ over the dense instruction of the same precision (Mishra et al., 2021). Microbenchmarks of Hopper measure the instruction itself below that bound (Luo et al., 2024). Three constraints follow, and they shape everything in this paper. Only one operand of a GEMM can be sparse. Its groups of four must lie along the reduction axis. And the speedup applies to the multiplication, not to whatever produces the compressed operand. For weights, the standard recipe prunes a trained model and retrains it with the mask fixed (Mishra et al., 2021; Pool & Yu, 2021; Zhou et al., 2021); for LLMs, SparseGPT (Frantar & Alistarh, 2023) and Wanda (Sun et al., 2024b) prune in one shot, and Sparse-Llama retrains Llama-3.1-8B to 2:4 on 13B tokens (Marques et al., 2024). For FP4 data, Blackwell’s sparse tensor cores use a paired 4:8 pattern instead of 2:4 (NVIDIA, 2026).

Quantization. Integer formats store k -bit values with a shared scale per group of elements, typically 128. FP8 (E4M3) needs only one scale per row or channel. Block-scaled 4-bit floating-point formats store E2M1 values with a shared scale per small block: MXFP4 uses a power-of-two scale per 32 elements (Rouhani et al., 2023), and NVFP4 an E4M3 scale per 16 elements plus a per-tensor scale. The tensor cores of Ada and Hopper go down to 8-bit floating point; Blackwell adds FP4 at twice the FP8 rate. On top of these formats, weights are quantized with GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024), activations together with weights with LLM.int8() (Dettmers et al., 2022) and SmoothQuant (Xiao et al., 2023), and KV caches with KIVI (Liu et al., 2024) and KVQuant (Hooper et al., 2024); attention runs in low precision in FlashAttention-3 (Shah et al., 2024) and SageAttention (Zhang et al., 2025), and rotations such as QuaRot (Ashkboos et al., 2024) and SpinQuant (Liu et al., 2025b) remove outliers before low-bit quantization.

Sparsity beyond weights. Activation sparsity is exploited through ReLU-like activations (Mirzadeh et al., 2024; Song et al., 2025), training-free thresholds (Liu et al., 2025a) and N:M patterns (Haziza et al., 2025; An et al., 2025; Alanova et al., 2026). For attention, HieraSparse (Wang & Wang, 2026) applies 2:4 to keys and values, DynSpAttn (Yu et al., 2026) and DFSS (Chen et al., 2023) to scores, and MUSTAFAR (Joo et al., 2025) prunes the KV cache without structure. Each of these systems picks one operand and one axis; our measurements put all of them on one scale and against the quantization each would otherwise use.

The quantization floor. Besides FP8 and FP4, Ada also has INT4 tensor cores, which LLM inference stacks do not use, and we leave them out; the floor is thus a property of

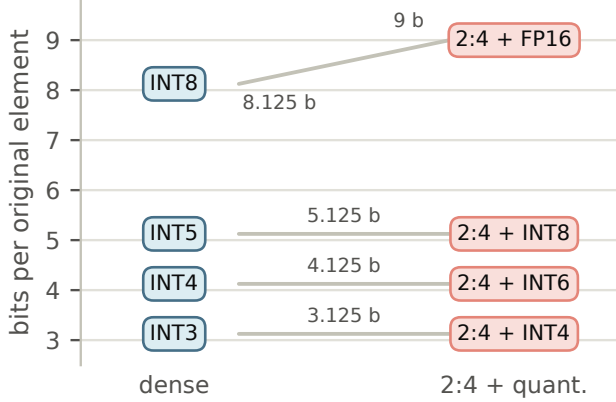


Figure 1. **Equal bits.** Bits per original element, including the 2:4 index (1 bit) and one FP16 scale per 128 elements. Each line joins two formats of identical cost; 2:4 on FP16 values costs more than FP8.

the formats inference stacks run, not of the silicon alone. For storage alone, we take 4-bit weights as the practical floor.

3 METHODOLOGY: COMPARING ON EQUAL TERMS

Adding 2:4 sparsity and dropping to a lower precision both serve one of two purposes: saving memory or saving compute. We compare the two at equal cost for each purpose. At *equal bits* they occupy the same memory, so the only question is which is more accurate. At *equal tensor-core tiers* they promise the same speedup, so we ask both which is more accurate and which is actually faster.

3.1 Equal bits: the same memory

We charge each format for every bit it stores per original element: values, indices, scales and zero points. Dense symmetric INT_k with one FP16 scale per g elements costs $k + 16/g$ bits. A 2:4 tensor keeps half of the elements, so with $\text{INT}_{k'}$ values it stores $k'/2$ value bits and one index bit per original element, plus the same scales, one per g original elements. Writing $k' = 2m$ gives

$$\text{bits}(2:4 + \text{INT}(2m)) = m + 1 + \frac{16}{g} = \text{bits}(\text{INT}(m+1)). \quad (1)$$

Dropping half of the elements thus saves far less than half of the bits. 2:4 with INT8 values costs as much as dense INT5, 2:4 with INT4 values as much as INT3, and 2:4 with FP16 values costs more than FP8 (Figure 1). At equal bits, the real question is whether to drop half of the elements and store the rest with twice the precision, or to keep all of them with one more bit each.

Storage budgets apply to the operands that live in memory:

Table 1. **The six operand families**, the GEMM in which each is the sparse operand, and the axis along which the hardware forces its 2:4 groups. Only weights are static; all other operands are produced at inference time.

	Tensor	GEMM	Forced axis	Static
W	linear weights	$Y = XW^\top$	input features	yes
X	linear inputs	$Y = XW^\top$	hidden	no
Q	queries	$S = QK^\top$	head dim	no
K	keys	$S^\top = KQ^\top$	head dim	no
V	values	$O^\top = V^\top P^\top$	tokens	no
P	attention probs.	$O = PV$	keys	no

weights, keys and values. Storage is not tied to tensor-core precisions, since weights and caches can be dequantized before the GEMM, and integer formats with group scales are the standard choice for both (Frantar et al., 2023; Lin et al., 2024; Liu et al., 2024; Hooper et al., 2024). We therefore state the identity for integers and compare at four budgets from 9 down to 3.1 bits per element; floating-point formats join wherever their cost fits and are the strongest rivals at the higher budgets. Each 2:4 configuration meets only dense formats that cost the same or less, except MXFP4, an eighth of a bit more expensive than its 2:4 rival (Appendix A).

3.2 Equal tiers: the same promised speed

For compute, we pair formats that promise the same *tensor-core tier*, the nominal throughput relative to dense 16-bit GEMMs given on the datasheet (NVIDIA, 2023b;a). Halving precision doubles the nominal throughput, and so does 2:4 sparsity. In the $2\times$ tier, a GEMM with one 2:4 operand in FP16 (BF16 runs at the same rate) meets a dense GEMM with both operands in FP8 or INT8. In the $4\times$ tier, 2:4 with FP8 operands meets dense 4-bit floating point, which only Blackwell runs; on Hopper, 2:4 is the only way into this tier. The $8\times$ tier, sparse NVFP4 against dense NVFP4, lies at the floor and belongs to Section 6.

In the $2\times$ tier the dense side quantizes both operands of the GEMM, while 2:4 alters only one. A tier states only the nominal throughput, so we also measure what each format actually delivers (Section 4.2).

3.3 Operands and forced axes

A transformer layer has six families of GEMM operands that could be sparse (Table 1). Only one operand per GEMM can be sparse, and that GEMM’s reduction axis fixes the axis of its 2:4 groups. Weights are grouped along input features, queries and keys along the head dimension, values along tokens and attention probabilities along keys. Only weights can be pruned offline and retrained. For attention probabilities we follow DynSpAttn (Yu et al., 2026) and keep the two largest of every four scores before the softmax,

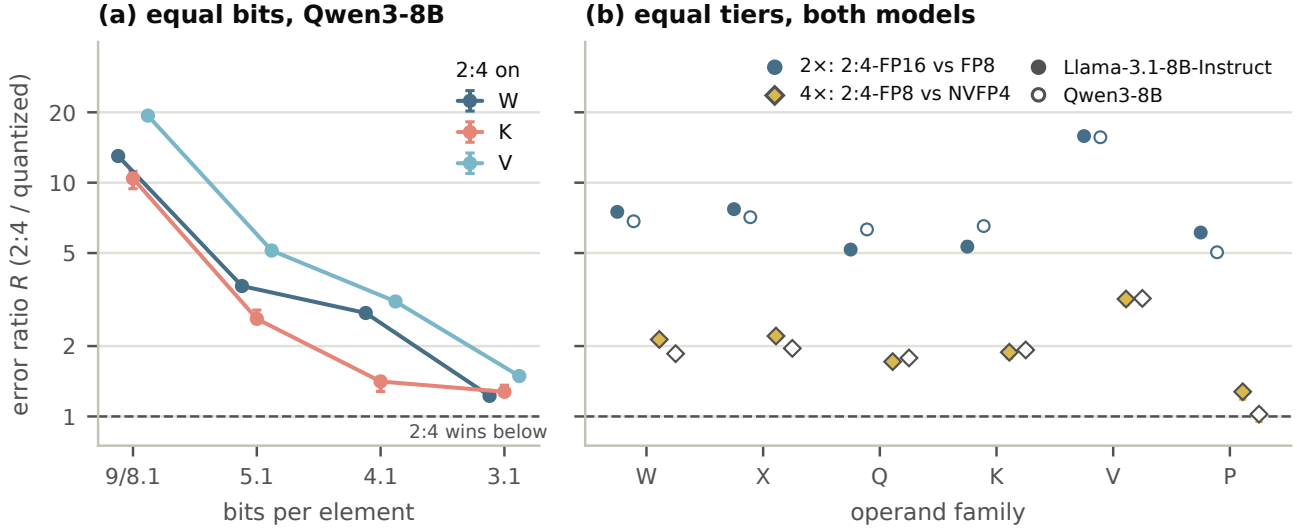


Figure 2. **Accuracy above the floor.** Layer-median error ratio R (2:4 over quantized; below 1, 2:4 would win), 95% bootstrap intervals. (a) Equal bits on Qwen3-8B against the strongest quantizer of each budget (Llama: Figure 8). (b) Equal tiers on both models. All values: Tables 7 and 8.

which then normalizes over the kept entries.

3.4 Measuring accuracy

The primary metric is operator-level relative error against an FP32 reference: the output of a linear layer with real calibration activations, or the output of an attention head. We measure it for every *cell*, a combination of model, operand family, budget or tier, and rival format, and define

$$R = \frac{\text{error of the best 2:4 configuration}}{\text{error of the best quantized configuration}}, \quad (2)$$

so that $R < 1$ means 2:4 wins. Each side picks from an equal-effort menu. The 2:4 side tries magnitude pruning, Wanda and SparseGPT for weights, and per-token shift and variance correction (Alanova et al., 2026) for dynamic operands. The quantized side tries round-to-nearest along its best grouping axis, GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024). Combined formats prune first and then quantize the kept values with fresh scales (Harma et al., 2025). Both sides calibrate on the same C4 sequences (Rafael et al., 2020).

We aggregate in the log domain, so that “twice as bad” and “twice as good” weigh the same: the layer-median R of a cell is the median over layers of the per-layer median over evaluation samples, and its confidence interval comes from a paired bootstrap over samples (Appendix C). Operator error ignores how errors compound across layers, so we also check selected cells at model level, compressing the family in every layer and measuring WikiText-2 perplexity (Merity et al., 2017), GSM8K accuracy (Cobbe et al., 2021) and, for keys, values and probabilities, RULER at 16K tokens (Hsieh

et al., 2024).

3.5 Measuring speed

We time on AWS cloud instances with an L40S (Ada), an H100 (Hopper) and an RTX PRO 6000 (Blackwell, sm_120), using device kernel time from CUPTI on the linear-layer shapes of both models at prefill lengths. Sparse GEMMs use cuSPARSELt with an exhaustive search over algorithm and split-K; dense GEMMs use the library’s default heuristics.

Each side pays for what it needs at run time. Weights are static: pruning, compression and quantization happen offline, and only the GEMM is timed. Every other operand is dynamic, and each side pays for producing its own input. The 2:4 side must select two of every four elements and write the kept values and their indices, optionally quantized; the dense side must quantize. For the 2:4 side we take the faster of cuSPARSELt’s own prune-and-compress and a fused kernel that reads the operand once and writes the compressed form once. The fused kernel reimplements the HieraSparse kernel (Wang & Wang, 2026) in Triton, bit for bit, and optionally fuses FP8 or INT8 quantization. The dense side uses fused per-token FP8 and INT8 quantization and an NVFP4 block quantizer. Every kernel matches a PyTorch reference bit for bit. The per-token shift and variance correction, which win on accuracy for dynamic operands, are not included in the 2:4 timing.

Operands are resident in L2, as in a pipeline where the previous kernel has just written them; preprocessing is also timed with L2 flushed, with the same conclusions (Appendix H). Whole models are served in vLLM for an end-to-end check

operand	comparison	operator R < 1: 2:4 better	Δ perplexity < 0: 2:4 better	Δ GSM8K (pt) > 0: 2:4 better	Δ RULER (pt) > 0: 2:4 better
Llama-3.1-8B-Instruct					
attention probs.	2:4-FP16 vs INT8, 2 $\times$ tier	0.78	-0.02	-3.6	+6.6
attention probs.	2:4-FP8 vs MXFP4, 4 $\times$ tier	0.78	-0.23	+1.2	+2.6
queries	2:4-FP8 vs MXFP4, 4 $\times$ tier	0.85	+1.47	-15.2	-21.4
keys	2:4-FP8 vs MXFP4, 4 $\times$ tier	0.91	+2.26	-24.8	-45.4
keys	2:4+INT4 vs INT3, 3.1 bits	1.05	-0.17	-5.6	-30.0
weights	2:4+INT4 vs INT3, 3.1 bits	1.27	+9.94	-9.6	+1.4
attention probs.	2:4-FP8 vs NVFP4, 4 $\times$ tier	1.28	+0.02	-2.8	+4.0
Qwen3-8B					
queries	2:4-FP8 vs MXFP4, 4 $\times$ tier	0.90	+0.04	-8.4	-19.1
keys	2:4-FP8 vs MXFP4, 4 $\times$ tier	1.00	-0.21	-2.8	-7.9
attention probs.	2:4-FP8 vs NVFP4, 4 $\times$ tier	1.02	-0.15	-0.8	+0.6
attention probs.	2:4-FP16 vs INT8, 2 $\times$ tier	1.13	+0.03	-2.4	+0.8
weights	2:4+INT4 vs INT3, 3.1 bits	1.22	+3.75	-18.4	+10.5
linear inputs	2:4-FP8 vs MXFP4, 4 $\times$ tier	1.25	+1.49	-26.0	-0.7
keys	2:4+INT4 vs INT3, 3.1 bits	1.27	+0.83	+0.0	-11.0
weights	2:4-FP8 vs MXFP4, 4 $\times$ tier	1.33	+4.12	-46.0	+3.9

■ 2:4 better ■ quantized better ■ pale: difference not significant (95%)

Figure 3. **Model quality on the cells closest to parity** ($0.7 < R < 1.4$), sorted by operator-level R ; each cell compresses one operand family in every layer. Changes are 2:4 minus quantized. Coral: 2:4 better; blue: quantized better; pale: not significant at 95%. 2 $\times$, 4 $\times$: tensor-core tiers; 3.1 bits: storage budget.

Table 2. **Speed at equal tier and at the floor.** $t(\text{rival})/t(2:4 \text{ side})$, above 1 means 2:4 is faster; median over the Llama-3.1-8B projections at 2K–16K tokens (ranges: Table 11). W: sparse weights, kernel time; X: sparse activations, preprocessing on both sides. Floor: 2:4 over the dense floor format.

GPU	2 $\times$: vs FP8		4 $\times$: vs NVFP4		floor: 2:4 added	
	W	X	W	X	FP8	NVFP4
L40S	0.89	0.89	–	–	1.65	–
H100	0.74	0.80	–	–	1.42	–
RTX PRO 6000	0.84	0.85	0.66	0.70	1.33	1.40

(Section 6.1).

4 ABOVE THE FLOOR, ONE MORE STEP OF QUANTIZATION WINS

We now run the comparison over all of our measurements: Llama-3.1-8B-Instruct (Llama Team, AI @ Meta, 2024) and Qwen3-8B (Qwen Team, 2025), every layer, all six operand families, four storage budgets and two compute tiers, over 220,000 measured configurations in total.

4.1 Accuracy

Equal bits. 2:4 loses at every storage budget on both models (Figure 2a for Qwen3, Figure 8 for Llama). The gap is more than an order of magnitude at 8 bits and shrinks

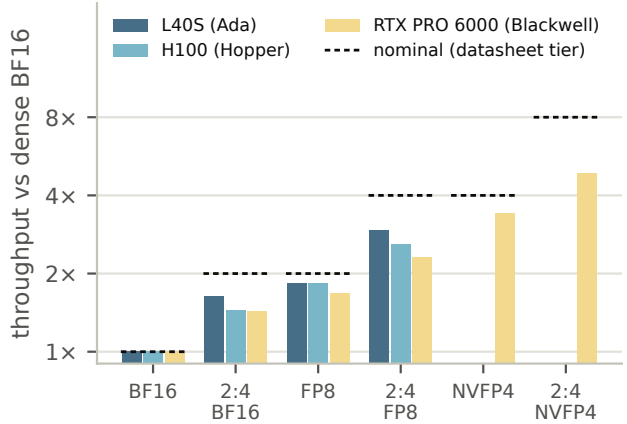


Figure 4. **Nominal against measured throughput** of linear-layer GEMMs relative to dense BF16 (sparse weights, kernel time, median over prefill shapes). Dashed: the nominal tier. NVFP4 and its sparse variant exist only on Blackwell.

steadily as bits fall, but it does not close: at 3.1 bits the closest cell, Llama’s keys against INT3, is still just above parity.

Equal tiers. In the 2 $\times$ tier, 2:4 with FP16 values has five to sixteen times the error of dense FP8, for every operand family on both models (Figure 2b). In the 4 $\times$ tier, 2:4 with FP8 values has at least the error of dense NVFP4 for

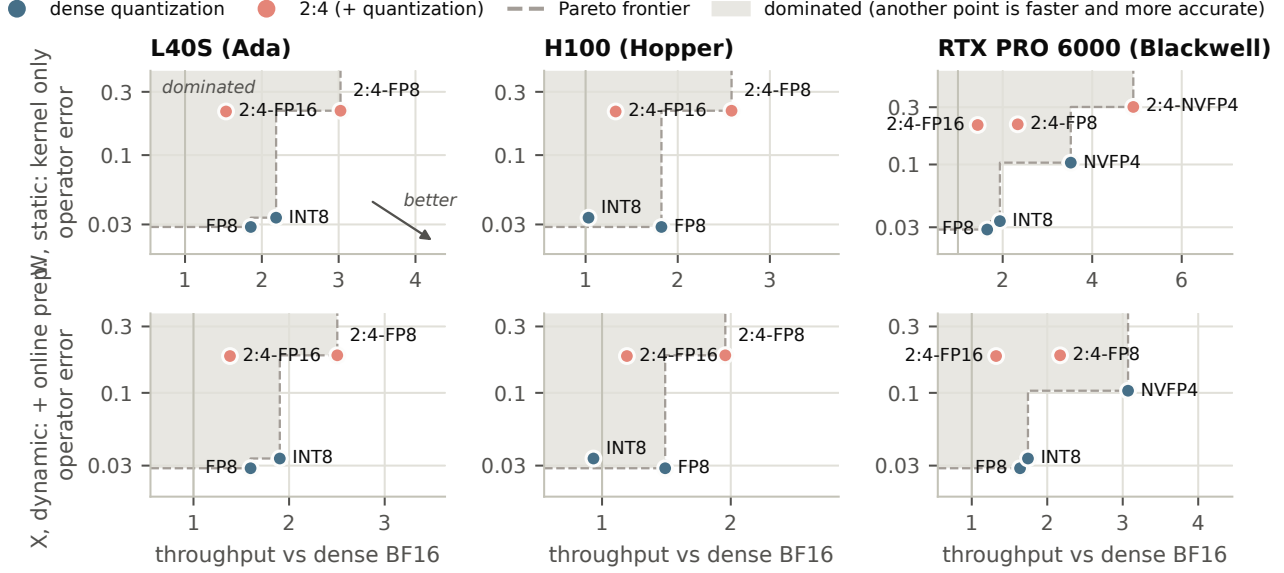


Figure 5. Error against throughput for linear layers (Qwen3-8B; Llama: Figure 10). Each point is one format: throughput relative to dense BF16 (prefill median) against layer-median operator error; lower right is better. Gray: dominated (another format is faster and more accurate); dashed: Pareto frontier. Top: sparse weights; bottom: sparse activations, online preprocessing on both sides.

every family, and attention probabilities come closest. Of the 108 compute cells, 2:4 wins six, all against weak rivals: MXFP4, whose power-of-two block scales waste precision, and row-wise INT8 for attention probabilities. Against FP8 and NVFP4 it wins none.

Model quality. The hardest test for our conclusion at model level uses the fifteen cells whose operator-level ratio is closest to parity, the cells most favourable to 2:4 (Figure 3). On GSM8K, 2:4 is never significantly better than quantization, and it is significantly worse in seven of these cells, including three of the five in which it wins at operator level. 2:4 does significantly better on perplexity or RULER in nine cases. Six are attention probabilities, where keeping the top two of every four scores sharpens attention: it helps long-context retrieval and perplexity, but never reasoning. The other three trade one metric for another: small perplexity gains on keys that RULER contradicts, and better retrieval for weights against INT3 on Qwen3 at the cost of eighteen GSM8K points. Near parity, the operator-level winner is a weak guide, matched by the three metrics in just over half of the cases; far from parity, it matches the perplexity outcome on eleven of twelve representative cells (Table 10).

Why operator error and model quality can disagree. Three differences between a per-layer ratio and a whole model explain the disagreements. First, R is a median over layers, but a few layers can decide model quality. In Llama’s queries and keys, 2:4 beats MXFP4 in about three quarters

of the layers, yet has two to three times its error in the first two layers (Figure 9), and whatever goes wrong there passes through every later layer. Second, operator error is measured with clean inputs, while the model compresses every layer at once, so errors compound through the residual stream and the softmax; Qwen3 has no such outlier layers, and its query cell still reverses at model level. Third, the form of an error matters, not only its size. 2:4 sets whole components to zero, whereas quantization spreads small rounding errors over all of them, and the tasks react differently: retrieval in RULER depends on sharp attention peaks, which pruning queries and keys disturbs and pruning probabilities sharpens, while GSM8K accumulates every small error over long chains of generated tokens. Near parity, these effects outweigh a small difference in operator error; far from parity, they do not.

4.2 Speed

Figure 4 shows what each tier delivers: on all three GPUs, dense FP8 realizes more of its nominal $2\times$ than 2:4 with 16-bit values does. Table 2 confirms it at equal tier. With sparse weights, 2:4 with FP16 values runs slower than dense FP8 on all three GPUs, and on Blackwell 2:4 with FP8 values runs slower than dense NVFP4. With sparse activations, both sides pay for preprocessing and the ratios barely move; preprocessing matters more against the same precision, where only 2:4 pays it (Tables 11 and 12). The one exception, FP16 against INT8 on H100, reflects a weak INT8 GEMM on that GPU, where INT8’s error is a ninth of 2:4’s. The ordering carries over to a serving engine: in

Table 3. The asymmetry on Gaussian data. Relative error of magnitude 2:4 (kept values quantized with round-to-nearest) and of the dense format of equal bits, for i.i.d. Gaussian data with one scale per 128 elements.

Bits	2:4 side	error	dense rival	error
9 / 8.1	2:4 + FP16	36.4%	INT8	0.6%
5.1	2:4 + INT8	36.4%	INT5	5.5%
4.1	2:4 + INT6	36.4%	INT4	11.7%
3.1	2:4 + INT4	37.3%	INT3	27.4%

vLLM on an H100, dense FP8 serves faster than 2:4 with BF16 values, in prefill and in decode, from a smaller model (Table 5).

How a dynamic operand is pruned matters as much as the sparse GEMM. A fused kernel that reads the operand once and writes the kept values and indices once runs close to memory copy bandwidth, whereas cuSPARSELT’s own prune-and-compress path and PyTorch’s semi-structured conversion are several times slower (Table 13) and give away most of what the sparse GEMM gains. Attention shows the limits of 2:4 on dynamic operands (Table 14): queries and keys, reduced over a 128-channel head, roughly break even at equal precision, and for attention probabilities, pruning and compressing P online leaves 2:4 at about half the dense speed, which rules out P as a standalone sparse operand.

4.3 Accuracy and speed together

Figure 5 puts the two together. Above the floor, every 2:4 configuration falls in the dominated region: some dense format is both faster and several times more accurate. 2:4 reaches the Pareto frontier only at its fast end, after it has been stacked on the floor, which is the subject of Section 6.

5 ANALYSIS: WHY 2:4 LOSES

The results of Section 4 follow from a simple asymmetry. Quantization keeps every value and rounds it, and its error shrinks quickly as bits are added. 2:4 sparsity deletes half of the values; the deleted values are lost entirely, and no number of bits spent on the kept ones brings them back. Table 3 shows the effect on Gaussian data at equal bits: the error of 2:4 barely moves across budgets, while the error of the dense rival doubles with every bit removed, so the two would meet only below 3 bits, a precision no dense tensor core supports. The rest of this section makes the argument precise for any tensor.

The energy that 2:4 removes. Let A be the operand that 2:4 prunes, the sparse operand of its GEMM in Table 1, such as the weight matrix of a linear layer or the key matrix of an attention head. Magnitude 2:4 keeps the two largest

of every four elements of A along the forced axis and drops the other two. Its relative error energy is the share of energy in the dropped elements, which we call the *2:4 energy loss*:

$$e_{2:4}(A) = \frac{\sum_g (a_{g1}^2 + a_{g2}^2)}{\|A\|_F^2}, \quad (3)$$

where g runs over the groups of four along the forced axis, a_{g1} and a_{g2} are the two entries of smallest magnitude in group g , and $\|A\|_F^2$, the sum of all squared entries of A , is its total energy. For i.i.d. Gaussian data $e_{2:4} = 0.132$. It is small when each group is dominated by one or two large entries and large when the four entries are alike. Unlike kurtosis, $e_{2:4}$ depends on the axis: the same tensor can have very different values along its rows and its columns, and only the forced axis matters to the hardware.

Equal bits. Once the mask is fixed, pruning and quantization act on disjoint elements: the dropped elements carry the pruning error, and the kept ones carry the quantization error. Their error energies therefore add exactly. In the same notation, let $e_k(A)$ be the relative error energy of a k -bit quantizer on A , and $e_k^{\text{kept}}(A)$ the error of quantizing only the kept elements, normalized by the energy of all of A . At the equal-bit pairing of Equation (1), the ratio of tensor errors is

$$R_{\text{pred}}^2 = \frac{e_{2:4}(A) + e_{2m}^{\text{kept}}(A)}{e_{m+1}(A)}, \quad (4)$$

and 2:4 wins exactly when

$$e_{2:4}(A) < e_{m+1}(A) - e_{2m}^{\text{kept}}(A). \quad (5)$$

The right-hand side shrinks by roughly a factor of four with every bit added to the dense format, because each bit halves the quantization step. The left-hand side does not depend on bits at all. 2:4 is therefore far behind at 8 bits, closer at 4, and can only win once dense quantization error grows to the size of the pruning loss, which for Gaussian data happens only below 3 bits, beneath anything dense tensor cores offer. The inequality also says where help for 2:4 has to come from: not from a better quantizer for the kept values, but from a tensor whose dropped entries carry little energy, or a model trained to do without them, which is what retraining provides at the floor (Section 6).

The same reasoning gives the predicted ratio at the compute tiers, where both operands of the GEMM contribute error (Appendix B). We call R_{pred} , the predicted error ratio, the criterion. Equation (4) is exact for the tensors and contains no fitted parameters: $e_{2:4}$ and the quantization errors are computed from the operands directly. For the GEMM output it is an approximation, because the output error also depends on the other operand. For $Y = XW^\top$, a weight error E gives the output error energy

$$\mathbb{E} \|XE^\top\|_F^2 = \text{tr}(E \Sigma_X E^\top), \quad (6)$$

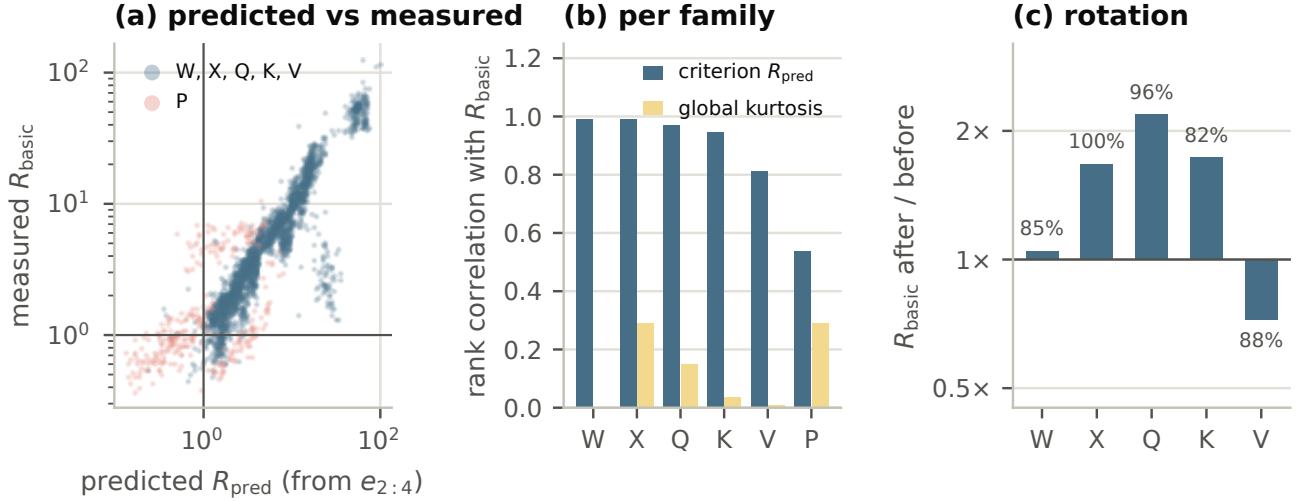


Figure 6. **The criterion.** (a) Predicted R_{pred} against measured R_{basic} for every (model, layer, cell) point. (b) Rank correlation with R_{basic} per operand family, for the criterion and for global kurtosis. (c) Change of R_{basic} under Hadamard rotation (block 128, both operands), median per family; numbers: share of points whose direction the criterion predicts.

where Σ_X is the second-moment matrix of the inputs, which equals the tensor error only for isotropic inputs. The criterion ignores this weighting; Section 5.1 tests how much that matters.

Consequences. The analysis has three testable consequences. First, the gap narrows as bits fall but does not close at the precisions tensor cores support. Second, which operands come closest depends on their structure along the forced axis, not on their overall distribution, so global kurtosis should predict the outcome poorly. Third, Hadamard rotation (Ashkboos et al., 2024; Liu et al., 2025b) changes both sides: it shrinks the quantizer’s range and makes the four entries of each group more alike, which raises $e_{2:4}$. Attention probabilities lie outside the model, because most of their energy sits in a few large probabilities that 2:4 keeps, while the output loses the many small ones it drops.

5.1 Testing the analysis

Equations (4) and (5) describe the two basic methods, magnitude pruning and round-to-nearest. We therefore test them against R_{basic} , the error ratio measured at the GEMM output when each side uses its basic method; calibrated methods such as SparseGPT and GPTQ compensate errors in ways that statistics of the tensor do not capture.

The first consequence holds: the measured storage ratios of Figure 2 fall with bits and stay above parity, as the Gaussian ones do; the heavier tails of real tensors help 2:4 at high budgets, but nowhere near enough.

Point by point, the criterion tracks the size of the measured gap closely (Figure 6a). For weights, activations, queries,

keys and values, it ranks the points with a correlation of 0.93 and, with no parameters to fit, separates the points where 2:4 wins with an AUC of 0.95. It is not a classifier at the threshold, however (Table 9): 2:4 wins at only 3% of these points, so always predicting quantization is as accurate, and the criterion recognizes only one in eight of 2:4’s wins, because its prediction runs high near parity, most visibly for queries and keys against MXFP4. It explains how large the gap is and why, not who wins a close call, which is settled at model level (Section 4.1). Global kurtosis, the predictor proposed by Kuzmin et al. (2023), barely correlates with the outcome (Figure 6b), and fitted on one model, it separates winners from losers on the other far worse than the parameter-free criterion does. What decides the comparison is the structure of a tensor inside the groups the hardware prunes, not its overall shape.

Rotation tilts the comparison further toward quantization (Figure 6c). Hadamard rotation roughly doubles the error ratio for queries, keys and linear-layer inputs, leaves weights almost unchanged, and favours 2:4 only for values. The criterion predicts the direction of change at 87% of the points, against 68% for always predicting an increase. Since rotation is now a standard step before low-bit quantization, the deployed comparison is less favourable to 2:4 than our unrotated measurements.

The criterion does not hold for attention probabilities, as the analysis anticipates; at model level, 2:4 on probabilities even moves perplexity in opposite directions on the two models. We treat probabilities as a form of attention sparsification rather than compression.

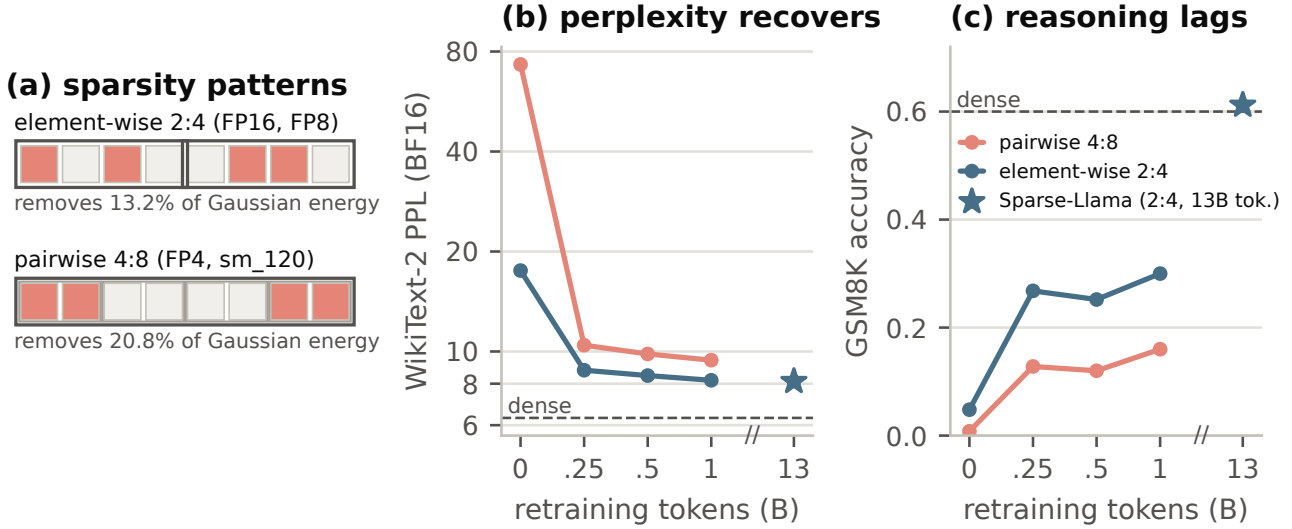


Figure 7. **2:4 at the floor** (Llama-3.1-8B). (a) Kept elements (orange) along eight elements of the reduction axis: element-wise 2:4, and the pairwise 4:8 of Blackwell’s sparse FP4. (b) BF16 WikiText-2 perplexity and (c) GSM8K accuracy along the retraining of our two students, from the one-shot SparseGPT start (0), with Sparse-Llama, retrained for 13B tokens, for reference. Perplexity keeps falling while GSM8K stalls.

Table 4. **The cost of stacking 2:4 on a quantization floor** (Llama-3.1-8B; WikiText-2 perplexity / GSM8K accuracy on 250 problems). Retrained 2:4 (13B tokens) is Sparse-Llama (Marques et al., 2024), quantized with its mask fixed. Ours (1B tokens): element-wise 2:4 except in the hardware NVFP4 row, which uses pairwise 4:8 (4:8p). [†]Element-wise sparse NVFP4 does not exist in hardware and is shown for reference.

Floor	Dense	+ one-shot 2:4	+ retrained 2:4 (13B tok.)	+ retrained, ours (1B tok.)
BF16	6.31 / 0.600	–	8.13 / 0.612	8.19 / 0.300 (2:4), 9.41 / 0.160 (4:8p)
INT4 weights (W4A16)	6.90 / 0.540	18.61 / 0.032	8.45 / 0.580	8.53 / 0.292
FP8 (W8A8)	6.36 / 0.552	17.82 / 0.028	8.19 / 0.632	8.25 / 0.308
NVFP4 (W4A4), pairwise 4:8	6.98 / 0.460	108.83 / 0.020	–	10.22 / 0.132
NVFP4, element-wise 2:4 [†]	6.98 / 0.460	21.74 / 0.020	8.72 / 0.516	8.82 / 0.192

6 2:4 AT THE QUANTIZATION FLOOR

At the floor there is no lower dense format to switch to, and 2:4 can only be added on top. We study three floors for the linear layers, with attention kept in BF16: FP8 weights and activations, the floor of Ada and Hopper; NVFP4 weights and activations, the floor of Blackwell; and 4-bit integer weights, the practical storage floor. 2:4 is applied to the weights, which can be pruned offline and retrained. This section uses the base Llama-3.1-8B, from which both Sparse-Llama and our retrained students are derived.

6.1 How much faster

Stacked on FP8, 2:4 speeds up prefill-shaped linear layers by 1.3 to 1.7 $\times$, depending on the GPU (Table 2), which lifts their throughput to between two and three times dense BF16. On Blackwell, sparse NVFP4 runs 1.4 $\times$ faster than dense NVFP4, for nearly five times dense BF16 (Figure 4). These gains belong to the compute-bound regime. At decode,

Table 5. **End-to-end serving** in vLLM on one H100 (dense: Llama-3.1-8B; 2:4: Sparse-Llama). Prefill: 64 prompts of 2048 tokens; decode: 256 tokens at batch b . No GSM8K difference is significant.

serving	weights (GiB)	prefill (k tok/s)	decode (tok/s)		GSM8K
			$b=1$	$b=64$	
dense BF16	15.0	41.4	136	6,424	0.596
2:4 BF16	9.3	40.7	115	6,477	0.624
dense FP8	8.5	58.9	198	8,556	0.568
2:4 FP8	6.1	74.6	234	9,914	0.612

GEMMs are bound by memory bandwidth and 2:4 saves only bytes, which helps on some GPUs and hurts on others (Table 15 and Figure 11). We did not time the 4-bit weight floor, which serves decode and whose sparse kernels vLLM has since removed (vLLM maintainers, 2026a).

The gain survives in a serving engine (Table 5). Served

with vLLM (Kwon et al., 2023) from LLM Compressor checkpoints (Appendix H), 2:4 on the FP8 floor raises prefill and decode throughput, at batch 1 as at batch 64, and shrinks the weights in memory by more than a quarter. Serving the same weights with dense kernels removes the speedup and leaves GSM8K unchanged, so the gain comes from the sparse kernels alone; it is smaller than at the GEMM level because attention and the other layers do not shrink. With BF16 values, the engine’s 2:4 path is no faster than dense BF16, well short of cuSPARSELt on the same GPU.

6.2 What pairwise 4:8 costs on Blackwell

For FP4 data, Blackwell’s sparse tensor cores do not use element-wise 2:4. cuSPARSELt documents a paired 4:8 pattern (NVIDIA, 2026): in each group of eight elements, two of the four adjacent pairs are kept (Figure 7a), and in the block-scaled GEMM both operands carry one scale per 32 positions of the reduction axis, half the scale resolution of dense NVFP4. We checked the format on an RTX PRO 6000 (sm_120) with cuSPARSELt 0.8.1: element-wise masks give wrong products, while the pairwise format matches our simulation to within rounding; sparse FP8 on the same GPU remains element-wise. What the documentation does not say is what the pattern costs in an LLM.

Pairing costs quality. Keeping pairs rather than elements removes about one and a half times as much energy from Gaussian data, and one-shot pruning to pairs at the NVFP4 floor gives Llama-3.1-8B five times the perplexity that element-wise pruning would (Table 4). Almost all of this comes from the pattern, not from the coarser scales, which barely matter under either mask (Table 16). Nor can a model retrained for element-wise 2:4 be reused: converting Sparse-Llama to pairs discards a third of its trained weights and destroys it.

6.3 What it costs, and what retraining buys

Table 4 gives the cost of stacking 2:4 on each floor. One-shot pruning is unusable on every floor: perplexity more than doubles and GSM8K accuracy collapses to a few percent. Instruction-tuned Llama and Qwen3 behave alike.

Retraining recovers most of the loss. With Sparse-Llama, the general 2:4 model retrained for 13B tokens (Kurtić et al., 2025), the perplexity increase on the INT4 and FP8 floors shrinks to a small fraction of the one-shot increase, and GSM8K accuracy fully returns. No such model exists for the pairwise format, so we retrained two students, one with pairwise 4:8 masks and one with element-wise 2:4 masks as a control, by distillation from the dense model with the one-shot SparseGPT masks fixed (Kurtić et al., 2023), on the same 1B tokens of FineWeb-Edu (Penedo et al., 2024) with one untuned recipe (Table 17).

Two results stand out (Figure 7b,c). First, the pairwise for-

mat costs more than element-wise 2:4 throughout training, and the gap is still clear at 1B tokens. Second, perplexity overstates what retraining has achieved. Along both runs, perplexity keeps falling, while GSM8K accuracy jumps within the first quarter billion tokens and then barely moves. At 1B tokens, the pairwise student has recovered almost all of the perplexity lost to one-shot pruning but only a quarter of the lost GSM8K accuracy, and our element-wise student matches Sparse-Llama’s perplexity while solving half as many GSM8K problems. Sparse-Llama shows that reasoning does return after a much longer run on different data, but perplexity gives no sign of how far a run still has to go. Since retrained sparse models are commonly judged by perplexity, a reasoning task belongs in any judgment of whether retraining is done.

The floors therefore differ: on FP8, retraining at the scale of Sparse-Llama makes the speedup usable end to end; on Blackwell’s sparse FP4, our 1B-token retraining leaves GSM8K far below the dense floor. Partial stacking does not avoid retraining either: sparsifying the units with the lowest predicted cost first beats random and kurtosis-based orders, but at half of the linear-layer FLOPs buys only 1.14–1.25× (Appendix G).

7 CONCLUSION

At equal bits and equal tensor-core tiers, dropping one precision step was more accurate than adding 2:4 at every storage budget, and at every tier some dense format was both faster and more accurate, because the energy 2:4 discards along the forced axis does not shrink with bits. At the FP8 floor, a retrained 2:4 model served by vLLM on an H100 gained 27% in prefill throughput over the dense FP8 model, with GSM8K accuracy on par. Whether 2:4 pays off at the floor is decided by retraining more than by kernels: Blackwell’s pairwise sparse FP4 is fast but not yet usable at our budget, and perplexity, which recovered within a quarter billion tokens, is a poor guide to when retraining is done.

REFERENCES

- Alanova, S., Kazistova, K., Galaeva, E., Kostromina, A., Smirnov, V., Redko, D., Dontsov, A., Zhelnin, M., Bur-naev, E., and Shvetsov, E. Motivating next-gen accelerators with flexible N:M activation sparsity via benchmarking lightweight post-training sparsification approaches. In *Proceedings of ACL 2026: Industry Track*, 2026. arXiv:2509.22166.
- An, T., Cai, R., Zhang, Y., Liu, Y., Chen, H., Xie, P., Chang, S., Yao, Y., and Wang, G. Amber pruner: Leveraging N:M activation sparsity for efficient prefill in large language models. *arXiv preprint arXiv:2508.02128*, 2025.

- Ashkboos, S., Mohtashami, A., Croci, M. L., Li, B., Cameron, P., Jaggi, M., Alistarh, D., Hoefler, T., and Hensman, J. QuaRot: Outlier-free 4-bit inference in rotated LLMs. In *Advances in Neural Information Processing Systems*, 2024.
- Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., and Li, J. LongBench: A bilingual, multitask benchmark for long context understanding. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.
- Chen, Z., Qu, Z., Quan, Y., Liu, L., Ding, Y., and Xie, Y. Dynamic N:M fine-grained structured sparse attention mechanism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. GPT3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022.
- Frantar, E. and Alistarh, D. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, 2023.
- Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.
- Frantar, E., Evci, U., Park, W., Houlsby, N., and Alistarh, D. Compression scaling laws: Unifying sparsity and quantization. *arXiv preprint arXiv:2502.16440*, 2025.
- Harma, S. B., Chakraborty, A., Kostenok, E., Mishin, D., Ha, D., Falsafi, B., Jaggi, M., Liu, M., Oh, Y., Subramanian, S., and Yazdanbakhsh, A. Effective interplay between sparsity and quantization: From theory to practice. In *International Conference on Learning Representations*, 2025.
- Haziza, D., Chou, T., Choudhary, D., Wehrstedt, L., Massa, F., Yu, J., Jeong, G., Rao, S., Labatut, P., and Cai, J. Accelerating transformer inference and training with 2:4 activation sparsity. In *ICLR 2025 Workshop on Sparsity in LLMs*, 2025. *arXiv:2503.16672*.
- Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M. W., Shao, Y. S., Keutzer, K., and Gholami, A. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems*, 2024.
- Hsieh, C.-P., Sun, S., Krizan, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., and Ginsburg, B. RULER: What’s the real context size of your long-context language models? In *Conference on Language Modeling*, 2024.
- Joo, D., Hosseini, H., Hadidi, R., and Asgari, B. MUSTA-FAR: Promoting unstructured sparsity for KV cache pruning in LLM inference. In *Advances in Neural Information Processing Systems*, 2025.
- Kurtic, E., Kuznedelev, D., Frantar, E., Goin, M., and Alistarh, D. Sparse fine-tuning for inference acceleration of large language models. *arXiv preprint arXiv:2310.06927*, 2023.
- Kurtić, E., Marques, A., Kurtz, M., Alistarh, D., and Pandit, S. 2:4 sparse Llama: Smaller models for efficient GPU inference. Red Hat Developer blog, <https://developers.redhat.com/articles/2025/02/28/24-sparse-llama-smaller-models-efficient-gpu-inference>, 2025.
- Kuzmin, A., Nagel, M., van Baalen, M., Behboodi, A., and Blankevoort, T. Pruning vs quantization: Which is better? In *Advances in Neural Information Processing Systems*, 2023.
- Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with PagedAttention. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2023.
- Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. AWQ: Activation-aware weight quantization for On-Device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, 2024.
- Liu, J., Ponnusamy, P., Cai, T., Guo, H., Kim, Y., and Athiwaratkun, B. Training-free activation sparsity in large language models. In *International Conference on Learning Representations*, 2025a.
- Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *International Conference on Machine Learning*, 2024.
- Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., and Blankevoort, T. SpinQuant: LLM quantization with learned rotations. In *International Conference on Learning Representations*, 2025b.
- Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

- LLM Compressor maintainers. LLM Compressor v0.11.0 release notes: removal of Sparse24 quantization formats and sparse model compression. <https://github.com/vllm-project/llm-compressor/releases/tag/0.11.0>, 2026.
- Luo, W., Fan, R., Li, Z., Du, D., Wang, Q., and Chu, X. Benchmarking and dissecting the Nvidia Hopper GPU architecture. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.
- Marques, A., Kurtić, E., Kurtz, M., Alistarh, D., Pandit, S., and Shahsavan, F. 2:4 sparse Llama FP8: SOTA performance for NVIDIA Hopper GPUs. Red Hat Developer blog, <https://developers.redhat.com/articles/2024/12/18/24-sparse-llama-fp8-sota-performance-nvidia-hopper-gpus>, 2024.
- Maximov, E., Kuzkina, Y., Kanametov, A., Prutko, A., Zhelنين, M., Shvetsov, E., and Goncharov, A. From 2:4 to 8:16 sparsity patterns in LLMs for outliers and weights with variance correction. In *Proceedings of ACL 2026: Industry Track*, pp. 957–965, 2026.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.
- Mirzadeh, I., Alizadeh, K., Mehta, S., del Mundo, C. C., Tuzel, O., Samei, G., Rastegari, M., and Farajtabar, M. ReLU strikes back: Exploiting activation sparsity in large language models. In *International Conference on Learning Representations*, 2024.
- Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.
- Mozaffari, M., Yazdanbakhsh, A., and Mehri Dehnavi, M. SLiM: One-shot quantization and sparsity with low-rank approximation for LLM weight compression. In *International Conference on Machine Learning*, 2025.
- NVIDIA. NVIDIA H100 tensor core GPU datasheet. <https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet>, accessed 2026-10-20, 2023a.
- NVIDIA. NVIDIA L40S GPU datasheet. <https://resources.nvidia.com/en-us-l40s/l40s-datasheet-28413>, accessed 2026-10-20, 2023b.
- NVIDIA. cuSPARSELt documentation: Data types (cusparseLtSparsity_t, cusparseLtPruneAlg_t). <https://docs.nvidia.com/cuda/cusparseLt/types.html>, accessed 2026-10-20, 2026.
- Penedo, G., Kydlíček, H., Ben Allal, L., Lozhkov, A., Mitchell, M., Raffel, C., von Werra, L., and Wolf, T. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.
- Pool, J. and Yu, C. Channel permutations for N:M sparsity. In *Advances in Neural Information Processing Systems*, 2021.
- Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21, 2020.
- Rouhani, B. D., Zhao, R., More, A., Hall, M., Khodamoradi, A., Deng, S., Choudhary, D., Cornea, M., Dellinger, E., Denolf, K., Stosic, D., Elango, V., Golub, M., Heinecke, A., James-Roxby, P., Jani, D., Kolhe, G., Langhammer, M., Li, A., Melnick, L., Mesmakhosroshahi, M., Rodriguez, A., Schulte, M., Shafipour, R., Shao, L., Siu, M., Dubey, P., Micikevicius, P., Naumov, M., Verrilli, C., Wittig, R., Burger, D., and Chung, E. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.
- Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems*, 2024.
- Song, C., Han, X., Zhang, Z., Hu, S., Shi, X., Li, K., Chen, C., Liu, Z., Li, G., Yang, T., and Sun, M. ProSparse: Introducing and enhancing intrinsic activation sparsity within large language models. In *International Conference on Computational Linguistics (COLING)*, 2025.
- Sun, M., Chen, X., Kolter, J. Z., and Liu, Z. Massive activations in large language models. In *Conference on Language Modeling*, 2024a.
- Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models. In *International Conference on Learning Representations*, 2024b.
- vLLM maintainers. [Quantization][Deprecation] Remove Marlin 24 (vLLM pull request #32688, merged 2026-01-28). <https://github.com/vllm-project/vllm/pull/32688>, 2026a.

- vLLM maintainers. [Sparse24] [Deprecation] Remove Sparse24 CT integration and kernels (vLLM pull request #36799, merged 2026-03-23, released in v0.19.0). <https://github.com/vllm-project/vllm/pull/36799>, 2026b.
- Wang, H. and Wang, C. HieraSparse: Hierarchical semi-structured sparse KV attention. *arXiv preprint arXiv:2604.16864*, 2026.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.
- Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, 2023.
- Yu, X., Fan, R., Li, Z., Luo, W., Gong, G., and Chu, X. DynSpAttn: Efficient attention via dual-side dynamic sparsity on sparse tensor cores. In *International Conference on Supercomputing (ICS)*, 2026.
- Zhang, J., Wei, J., Zhang, P., Zhu, J., and Chen, J. SageAttention: Accurate 8-bit attention for plug-and-play inference acceleration. In *International Conference on Learning Representations*, 2025.
- Zhou, A., Ma, Y., Zhu, J., Liu, J., Zhang, Z., Yuan, K., Sun, W., and Li, H. Learning N:M fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021.

AI USAGE DISCLOSURE

We used Claude (Anthropic), through the Claude Code agent, as a research assistant. It wrote and debugged most of the measurement, kernel and analysis code, ran and monitored the experiments, produced the figures and tables from the result files, and helped condense and revise the text. The author posed the research question and designed the comparison framework, the choice of the next compression step judged at equal bits and at equal tensor-core tiers; decided the measurement protocol, the models, tasks and hardware, and every later change to them; chose which follow-up experiments to run; judged which results support which claims; and wrote the first draft of the paper and determined its structure and argument. The author checked the results and takes full responsibility for the content.

A BIT ACCOUNTING

Table 6. Bits per original element ($g = 128$ original elements per FP16 scale unless stated). Element-wise 2:4 stores a 2-bit index per kept value (1 bit per element); pairwise 4:8 stores two 2-bit pair indices per eight elements (0.5 bit per element) and one E4M3 scale per 32 positions, i.e. per 16 kept values.

Configuration	Formula	Bits
Dense INT k , symmetric	$k + 16/g$	8.125 / 5.125 / 4.125 / 3.125
Dense INT k , asymmetric	$k + 32/g$	INT4: 4.25
FP8, per-row or per-channel scale	$8 + \approx 0$	8.0
MXFP4 (E8M0 scale per 32)	$4 + 8/32$	4.25
NVFP4 (E4M3 per 16, FP32 per tensor)	$4 + 8/16$	4.5
2:4 + FP16	$8 + 1$	9.0
2:4 + INT k	$k/2 + 1 + 16/g$	5.125 / 4.125 / 3.125
Sparse NVFP4, pairwise 4:8 (sm_120)	$4/2 + 4/8 + 8/32$	2.75

Budgets. At 9/8.1 bits, 2:4 + FP16 meets INT8 and FP8; at 5.1 bits, 2:4 + INT8 meets INT5 (symmetric and asymmetric) and NVFP4; at 4.1 bits, 2:4 + INT6 meets INT4 (symmetric and asymmetric) and MXFP4; at 3.1 bits, 2:4 + INT4 meets INT3 (symmetric and asymmetric). Where costs cannot match exactly, the dense side gets fewer bits (FP8 and INT8 at 8.0 and 8.125 against 9, NVFP4 at 4.5 against 5.125), except MXFP4, which costs an eighth of a bit more than 2:4 + INT6. An ideal encoding of the six possible 2:4 patterns would need $\log_2 6/4 \approx 0.65$ bits per element instead of the hardware’s 1 bit. Without scales, adding 2:4 to k -bit data compresses it by only $2k/(k + 2)$.

B PREDICTED RATIO AT COMPUTE TIERS

At a compute tier, both operands of the GEMM contribute error. Treating their relative errors as additive in the output, the $2\times$ and $4\times$ tiers give

$$R_{\text{pred}}^2 = \frac{e_{2:4}(A)}{e_8(A) + e_8(B)} \quad (2\times), \quad (7)$$

$$R_{\text{pred}}^2 = \frac{e_{2:4}(A) + e_8^{\text{kept}}(A) + e_8(B)}{e_4(A) + e_4(B)} \quad (4\times), \quad (8)$$

where B is the dense operand and e_8 and e_4 are the errors of the 8-bit and 4-bit formats. In the $2\times$ tier, the pruning loss of one operand must beat the 8-bit rounding error of two, and dropping half of the elements costs far more than rounding them to 8 bits.

C MEASUREMENT DETAILS AND TABLES

Models. Llama-3.1-8B-Instruct (revision 0e9e39f) and Qwen3-8B (revision b968826) for the comparison above the floor; Llama-3.1-8B base (revision d04e592) and Sparse-Llama, RedHatAI/Sparse-Llama-3.1-8B-2of4 (revision c28c229, the general model, not a task-tuned variant), at the floor.

Samples and positions. Each layer is evaluated on 16 samples: 8 LongBench documents (Bai et al., 2024) (two each from 2WikiMQA, HotpotQA, Qasper and GovReport, truncated to 16,384 tokens) and 8 C4 validation documents. Errors are measured at causal-safe query positions, the last position of each 128-token block. Both sides calibrate on 128 C4 sequences of 2048 tokens, with statistics from the uncompressed model. Linear-layer weights and inputs pool the four projection sites (fused QKV, output, fused gate/up, down). With grouped-query attention, keys and values are measured per KV head, queries and probabilities per query head. The measurements comprise 221,408 rows, half of them with Hadamard rotation.

Aggregation. The layer-median R of a cell is the median over layers of the per-layer median of $\log R$ over samples and projection sites; its 95% interval comes from a paired bootstrap over samples with 10,000 resamples. R_{basic} is the same ratio with the basic method on each side (magnitude pruning against round-to-nearest). The criterion baseline is a logistic regression on $\log(\text{excess kurtosis} + 3)$, fitted on one model and tested on the other.

Is 2:4 Sparsity Still Worth It?

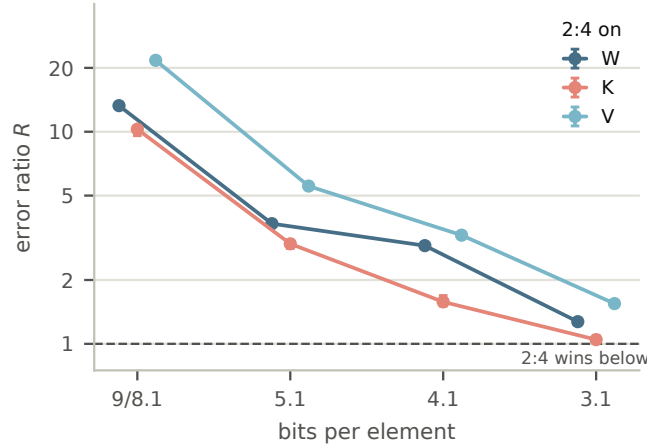


Figure 8. Figure 2a (equal bits) for Llama-3.1-8B-Instruct.

Table 7. Storage budgets: layer-median R against the strongest quantizer (in parentheses).

Model	Family	9 / 8.1 b	5.1 b	4.1 b	3.1 b
Llama	W	13.27 (FP8-row)	3.69 (NVFP4-g16)	2.90 (INT4-g128)	1.27 (INT3-g128)
Llama	K	10.28 (FP8-row)	2.96 (NVFP4-g16)	1.58 (MXFP4-g32)	1.05 (INT3-g128)
Llama	V	21.71 (FP8-row)	5.54 (NVFP4-g16)	3.25 (INT4-g128)	1.55 (INT3-g128-asym)
Qwen3	W	13.00 (FP8-row)	3.61 (NVFP4-g16)	2.77 (INT4-g128)	1.22 (INT3-g128)
Qwen3	K	10.43 (FP8-row)	2.62 (NVFP4-g16)	1.41 (MXFP4-g32)	1.27 (INT3-g128)
Qwen3	V	19.36 (FP8-row)	5.13 (NVFP4-g16)	3.10 (INT4-g128)	1.49 (INT3-g128)

Table 8. Compute tiers: layer-median R per family and rival. Bold: $R < 1$; all six such cells face MXFP4 or row-wise INT8 for P. MXFP4 is simulated.

Model	Family	2× vs FP8	2× vs INT8	4× vs NVFP4	4× vs MXFP4
Llama	W	7.50	7.70	2.13	1.48
Llama	X	7.70	9.25	2.21	1.48
Llama	Q	5.17	19.05	1.72	0.85
Llama	K	5.33	20.33	1.88	0.91
Llama	V	15.83	2.31	3.18	2.07
Llama	P	6.12	0.78	1.28	0.78
Qwen3	W	6.84	5.00	1.86	1.33
Qwen3	X	7.12	4.96	1.96	1.25
Qwen3	Q	6.31	14.91	1.78	0.90
Qwen3	K	6.52	16.57	1.92	1.00
Qwen3	V	15.64	3.52	3.20	2.00
Qwen3	P	5.04	1.13	1.02	0.64

Table 9. How well the criterion separates the points where 2:4 wins at operator level ($R_{\text{basic}} < 1$). Always predicting quantization is as accurate as the criterion for W-V; the criterion’s value lies in ranking (AUC, rank correlation), not in classification at the threshold. AUC and Spearman as in Figure 6; kurtosis fitted on the other model.

Points	n	2:4 wins	accuracy		balanced accuracy	recall of 2:4 wins	AUC		Spearman	
			always quant.	criterion			criterion	kurtosis	criterion	kurtosis
W, X, Q, K, V	3400	3.2%	96.8%	96.8%	0.56	12%	0.945	0.776	0.93	0.10
P	272	47.4%	52.6%	66.2%	0.66	71%	0.773	0.627	0.54	0.29
W-V, within 25% of parity	152	41.4%	58.6%	56.6%	0.49	5%	—	—	—	—

D MODEL-LEVEL CHECKS

A model-level cell compresses its family in every layer, each side using its most frequent winning method in the operator-level measurements; keys and values are compressed block-causally, with the current 128-token block kept dense. WikiText-2 perplexity uses 141 windows of 2048 tokens, each beginning with the BOS token, as every calibration sequence does (GPTQ calibrated on such sequences fits Llama’s massive BOS activations (Sun et al., 2024a) and fails on windows without it); GSM8K uses the first 250 test problems with 8-shot chain of thought (Wei et al., 2022), greedy decoding and strict matching (95% interval about ± 6 points); RULER uses five tasks at 16K tokens with 25 items each. Prompts are plain text without a chat template, so Qwen3’s thinking mode is not engaged; GSM8K generates at most 256 tokens and stops at the next question, and RULER uses each task’s own limit. Differences between the two sides are tested with a paired bootstrap over items (GSM8K, RULER) or windows (perplexity), 10,000 resamples; a difference that is not significant is not evidence of equivalence. On the fifteen boundary cells ($0.7 < R < 1.4$), operator-level R agrees with perplexity on 4 of 7 Llama and 5 of 8 Qwen3 cells, with GSM8K on 4 of 7 and 6 of 7, and with RULER on 3 of 7 and 3 of 8. RULER was planned for keys, values and probabilities; the six weight, input and query cells were added later and run on L40S GPUs, both sides on the same machine. A configuration shared with an H100 run reproduces its RULER score within 0.6 points.

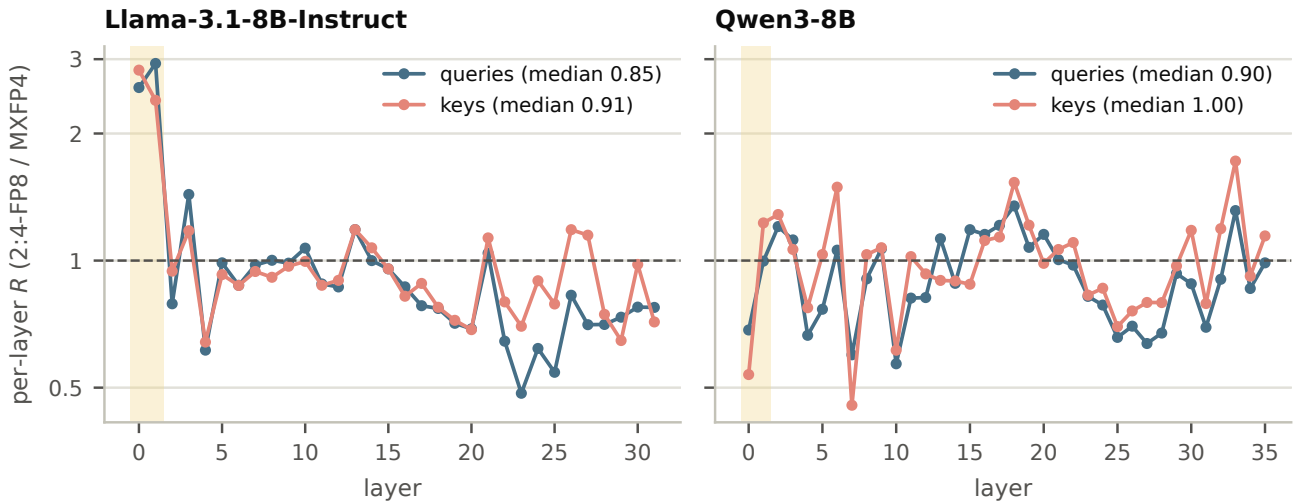


Figure 9. Per-layer operator-level R of queries and keys against MXFP4 ($4\times$ tier), the cells whose median favours 2:4 while model quality does not. Shaded: the first two layers.

Table 10. Operator-level R against model-level perplexity (Llama-3.1-8B-Instruct, WikiText-2; uncompressed 7.23). Two cells per family, one among the most and one among the least favourable to 2:4. These cells were chosen and evaluated in the pilot (eight layers, L40S GPUs) before the full measurements, so their R and perplexity differ slightly from Figure 3; for example, keys at 3.1 bits have $R = 0.97$ over the pilot layers and 1.05 over all 32.

Family	Cell	R (operator)	PPL, 2:4 side	PPL, quantized side	Agree
K	storage 3.1 b, vs INT3-g128	0.969	7.86	8.03	yes
K	storage 9/8.1 b, vs INT8-g128	40.6	7.82	7.23	yes
P	compute $4\times$, 2:4-FP8 vs MXFP4	0.848	7.29	7.52	yes
P	compute $2\times$, 2:4-FP16 vs FP8	6.18	7.29	7.23	yes
Q	compute $4\times$, 2:4-FP8 vs MXFP4	0.777	10.80	9.34	no
Q	compute $2\times$, 2:4-FP16 vs INT8	17.4	10.65	7.23	yes
V	storage 3.1 b, vs INT3-g128-async	1.58	9.17	7.30	yes
V	storage 9/8.1 b, vs INT8-g128	40.8	9.16	7.23	yes
W	storage 3.1 b, vs INT3-g128	1.27	22.98	12.37	yes
W	storage 9/8.1 b, vs INT8-g128	52.6	21.36	7.23	yes
X	compute $4\times$, 2:4-FP8 vs MXFP4	1.50	14.44	9.69	yes
X	compute $2\times$, 2:4-FP16 vs INT8	9.07	14.14	7.34	yes

E HARDWARE MEASUREMENTS

We use PyTorch 2.10, CUDA 12.8 and cuSPARSELt 0.8.1. Times are means of 100 calls after 20 warm-ups. Linear shapes are the four Llama-3.1-8B projections at $M \in \{2048, 4096, 16384\}$ (12 shapes); Qwen3-8B shapes give ratios within 0.04. Floors: FP8 uses per-channel weight scales and dynamic per-token activation scales; NVFP4 uses E2M1 values, an E4M3 scale per 16 elements and an FP32 tensor scale; INT4 weights use GPTQ with groups of 128.

Table 11. Speed ratios at the same precision and at equal tier. W: sparse weights, kernel only. X: sparse activations, online preprocessing on both sides. $8\times$: pairwise 4:8 NVFP4 against dense NVFP4.

GPU	operand	same precision			equal tier			
		BF16	FP8	INT8	$2\times/\text{FP8}$	$2\times/\text{INT8}$	$4\times/\text{NVFP4}$	$8\times/\text{NVFP4}$
L40S	W, kernel	1.63 [1.49, 1.98]	1.65 [1.52, 1.90]	1.27 [1.14, 1.41]	0.89 [0.78, 0.92]	0.72 [0.62, 0.77]	–	–
L40S	X, + prep.	1.43 [1.25, 1.95]	1.59 [1.50, 1.75]	1.24 [1.12, 1.35]	0.89 [0.70, 0.92]	0.70 [0.56, 0.78]	–	–
H100	W, kernel	1.44 [1.37, 1.57]	1.42 [1.34, 1.57]	2.39 [2.16, 2.62]	0.74 [0.62, 0.78]	1.29 [1.02, 1.39]	–	–
H100	X, + prep.	1.32 [1.18, 1.44]	1.31 [1.24, 1.47]	2.03 [1.84, 2.30]	0.80 [0.68, 0.89]	1.28 [1.18, 1.37]	–	–
RTX PRO 6000	W, kernel	1.43 [1.32, 1.75]	1.33 [1.23, 1.50]	1.34 [1.17, 1.42]	0.84 [0.73, 0.92]	0.76 [0.70, 0.85]	0.66 [0.61, 0.70]	1.40 [1.36, 1.46]
RTX PRO 6000	X, + prep.	1.32 [1.02, 1.67]	1.29 [1.16, 1.57]	1.30 [1.15, 1.40]	0.85 [0.67, 0.90]	0.77 [0.60, 0.82]	0.70 [0.63, 0.75]	–

Table 12. Share of online preprocessing in each side’s time (sparse activations).

GPU	2:4-FP16 side	2:4-FP8 side	dense FP8 side	dense NVFP4 side
L40S	0.04 [0.01, 0.17]	0.11 [0.03, 0.23]	0.06 [0.02, 0.16]	–
H100	0.10 [0.04, 0.14]	0.24 [0.09, 0.34]	0.16 [0.05, 0.26]	–
RTX PRO 6000	0.04 [0.01, 0.13]	0.09 [0.03, 0.18]	0.06 [0.02, 0.17]	0.15 [0.06, 0.23]

Is 2:4 Sparsity Still Worth It?

Table 13. Preprocessing kernels with L2 flushed: time in μ s and, in parentheses, compulsory bytes over time as a fraction of measured copy bandwidth. A copy reads and writes every byte, so a read-dominated kernel can exceed 1. HieraSparse’s original TileLang kernel ran only on L40S.

GPU	operand	fused 2:4	fused 2:4+FP8	FP8	NVFP4	cuSPARSELt	PyTorch tile	HieraSparse
L40S	P 65536×16384	6753 (0.74)	3806 (1.10)	4714 (1.01)	4714 (0.86)	26948	9777	5189
L40S	V 128×16384	10 (1.00)	12 (0.70)	10 (0.91)	15 (0.55)	35	11	8
L40S	X 4096×4096	48 (1.61)	51 (1.28)	48 (1.56)	49 (1.29)	396	56	54
L40S	X 16384×14336	1269 (0.86)	823 (1.11)	978 (1.07)	907 (0.98)	5950	1961	1100
H100	P 65536×16384	1530 (0.84)	1728 (0.63)	1742 (0.71)	2263 (0.47)	47354	4796	–
H100	V 128×16384	6 (0.45)	10 (0.22)	7 (0.33)	14 (0.15)	275	13	–
H100	X 4096×4096	22 (0.93)	32 (0.52)	28 (0.69)	41 (0.40)	531	81	–
H100	X 16384×14336	280 (1.01)	401 (0.59)	394 (0.69)	500 (0.46)	9369	1052	–
RTX PRO 6000	P 65536×16384	2277 (1.01)	1892 (1.02)	2141 (1.03)	1844 (1.02)	15086	3330	–
RTX PRO 6000	V 128×16384	5 (0.93)	7 (0.54)	6 (0.67)	9 (0.40)	26	9	–
RTX PRO 6000	X 4096×4096	24 (1.47)	29 (1.03)	31 (1.10)	25 (1.19)	216	34	–
RTX PRO 6000	X 16384×14336	461 (1.09)	420 (1.01)	454 (1.06)	393 (1.05)	3312	695	–

Table 14. Attention shapes as standalone GEMMs per KV group (four query heads, head dimension 128), sequence length 4096 / 16384: speed ratio with kernel time only $\rightarrow$ with online preprocessing. Entries above 2 on L40S at length 4096 reflect slow dense kernels chosen by the library heuristic at these narrow shapes.

GPU	family	same prec. FP16	same prec. FP8	2 $\times$ vs FP8	4 $\times$ vs NVFP4	prep. share
L40S	Q	0.83 $\rightarrow$ 0.82 / 0.83 $\rightarrow$ 0.82	0.96 $\rightarrow$ 0.96 / 0.99 $\rightarrow$ 0.99	0.84 $\rightarrow$ 0.84 / 0.92 $\rightarrow$ 0.92	– / –	0.01 / 0.00
L40S	K	0.84 $\rightarrow$ 0.84 / 1.01 $\rightarrow$ 1.01	1.04 $\rightarrow$ 1.04 / 0.95 $\rightarrow$ 0.95	1.03 $\rightarrow$ 1.03 / 0.88 $\rightarrow$ 0.88	– / –	0.01 / 0.00
L40S	P	3.90 $\rightarrow$ 0.51 / 1.77 $\rightarrow$ 0.38	1.55 $\rightarrow$ 1.25 / 1.63 $\rightarrow$ 1.31	0.72 $\rightarrow$ 0.79 / 0.92 $\rightarrow$ 0.74	– / –	0.87 / 0.78
L40S	V	1.02 $\rightarrow$ 1.01 / 1.03 $\rightarrow$ 1.03	2.94 $\rightarrow$ 2.73 / 1.01 $\rightarrow$ 1.01	0.41 $\rightarrow$ 0.42 / 0.58 $\rightarrow$ 0.59	– / –	0.01 / 0.00
H100	Q	0.84 $\rightarrow$ 0.80 / 0.88 $\rightarrow$ 0.87	0.96 $\rightarrow$ 0.95 / 0.98 $\rightarrow$ 0.98	0.84 $\rightarrow$ 0.87 / 0.82 $\rightarrow$ 0.82	– / –	0.05 / 0.01
H100	K	0.84 $\rightarrow$ 0.82 / 0.86 $\rightarrow$ 0.86	1.10 $\rightarrow$ 1.09 / 1.14 $\rightarrow$ 1.14	0.98 $\rightarrow$ 0.98 / 0.95 $\rightarrow$ 0.95	– / –	0.03 / 0.00
H100	P	1.49 $\rightarrow$ 0.44 / 1.73 $\rightarrow$ 0.38	1.28 $\rightarrow$ 0.93 / 1.50 $\rightarrow$ 1.07	0.86 $\rightarrow$ 1.04 / 0.85 $\rightarrow$ 1.08	– / –	0.70 / 0.78
H100	V	0.97 $\rightarrow$ 0.93 / 1.01 $\rightarrow$ 1.00	0.98 $\rightarrow$ 0.96 / 1.02 $\rightarrow$ 1.01	0.58 $\rightarrow$ 0.62 / 0.53 $\rightarrow$ 0.53	– / –	0.05 / 0.01
RTX PRO 6000	Q	0.91 $\rightarrow$ 0.89 / 1.15 $\rightarrow$ 1.15	1.02 $\rightarrow$ 1.01 / 1.06 $\rightarrow$ 1.06	0.91 $\rightarrow$ 0.93 / 1.03 $\rightarrow$ 1.03	1.12 $\rightarrow$ 1.11 / 1.00 $\rightarrow$ 1.00	0.02 / 0.00
RTX PRO 6000	K	0.72 $\rightarrow$ 0.71 / 0.91 $\rightarrow$ 0.91	1.01 $\rightarrow$ 1.01 / 0.94 $\rightarrow$ 0.94	0.75 $\rightarrow$ 0.76 / 0.91 $\rightarrow$ 0.91	1.04 $\rightarrow$ 1.04 / 0.95 $\rightarrow$ 0.95	0.01 / 0.00
RTX PRO 6000	P	1.75 $\rightarrow$ 0.40 / 1.76 $\rightarrow$ 0.46	1.41 $\rightarrow$ 1.18 / 1.60 $\rightarrow$ 1.23	1.02 $\rightarrow$ 1.01 / 0.91 $\rightarrow$ 0.92	0.76 $\rightarrow$ 0.93 / 0.90 $\rightarrow$ 0.96	0.77 / 0.74
RTX PRO 6000	V	1.23 $\rightarrow$ 1.20 / 1.05 $\rightarrow$ 1.05	1.30 $\rightarrow$ 1.27 / 1.01 $\rightarrow$ 1.01	0.72 $\rightarrow$ 0.74 / 0.52 $\rightarrow$ 0.52	0.71 $\rightarrow$ 0.75 / 0.58 $\rightarrow$ 0.59	0.03 / 0.00

Table 15. Decode-shaped GEMMs ($M \in \{16, 128\}$), L2 flushed before every call. cuSPARSELt does not support 8-bit sparse activations with 16 rows.

GPU	operand	BF16	FP8	2 $\times$ /FP8	4 $\times$ /NVFP4	8 $\times$ /NVFP4
L40S	W, kernel	1.72 [1.58, 1.79]	1.46 [1.33, 1.70]	0.90 [0.86, 1.04]	–	–
L40S	X, + prep.	0.98 [0.90, 1.02]	1.03 [1.02, 1.04]	0.62 [0.61, 0.63]	–	–
H100	W, kernel	0.77 [0.63, 1.11]	1.15 [0.95, 1.30]	0.76 [0.67, 0.84]	–	–
H100	X, + prep.	0.72 [0.55, 0.92]	0.93 [0.91, 0.95]	0.80 [0.65, 0.84]	–	–
RTX PRO 6000	W, kernel	1.63 [1.20, 2.61]	1.28 [0.98, 1.55]	0.85 [0.71, 0.98]	1.35 [0.99, 1.59]	1.65 [1.21, 2.65]
RTX PRO 6000	X, + prep.	1.02 [0.97, 1.56]	0.95 [0.83, 1.22]	0.67 [0.58, 0.92]	0.92 [0.71, 1.05]	–

Is 2:4 Sparsity Still Worth It?

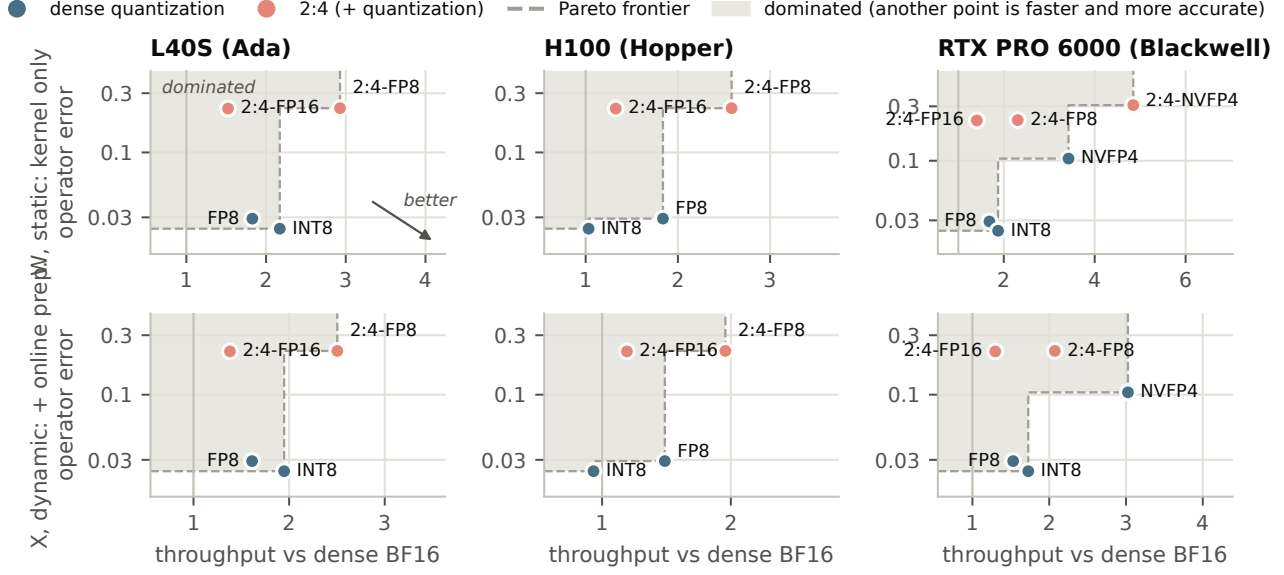


Figure 10. Figure 5 for Llama-3.1-8B-Instruct.

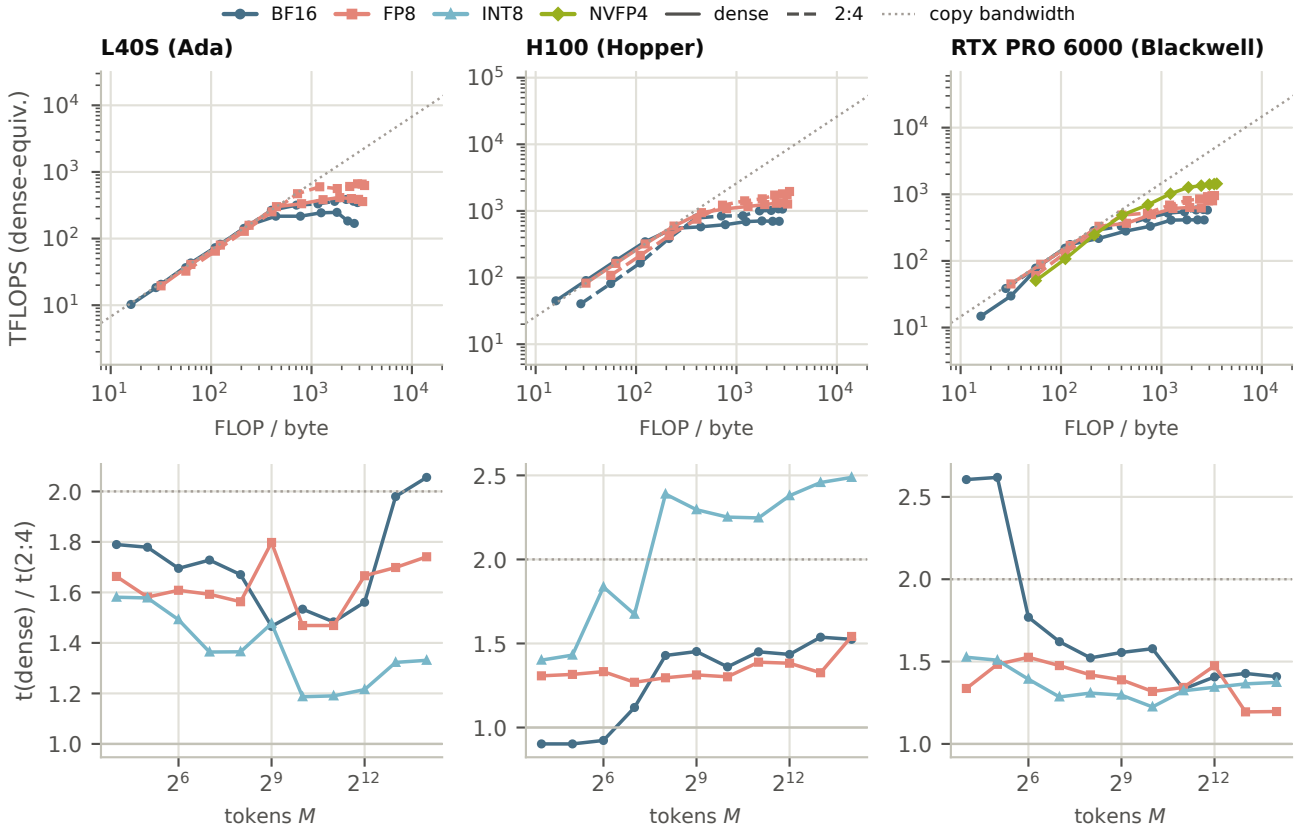


Figure 11. Top: throughput against arithmetic intensity for the gate/up projection (14336×4096 , M from 16 to 16,384), L2 flushed; solid dense, dashed 2:4; dotted, measured copy bandwidth. Bottom: 2:4 speedup over the same dense format against M .

F SPARSE FP4: MASK PATTERN AND SCALE GRANULARITY

Table 16. One-shot SparseGPT 2:4 on the weights at the NVFP4 floor (Llama-3.1-8B base, WikiText-2 perplexity), with the mask pattern and the scale block of the dense operand varied independently. Weights keep one E4M3 scale per 32 positions (16 kept values) throughout. Only the pairwise mask with 32-position scales runs on sm_120; the other corners are simulated.

Mask on the weights	activation scale per 16	per 32
element-wise 2:4	21.74	22.39
pairwise 4:8	105.23	108.83 (hardware)
dense NVFP4 floor	6.98	–

G PARTIAL STACKING

Units are (layer, projection) pairs. Each order makes the first units one-shot 2:4 until a share f of the linear-layer FLOPs is sparse: by the cost the criterion predicts, by global kurtosis, or at random (five seeds).

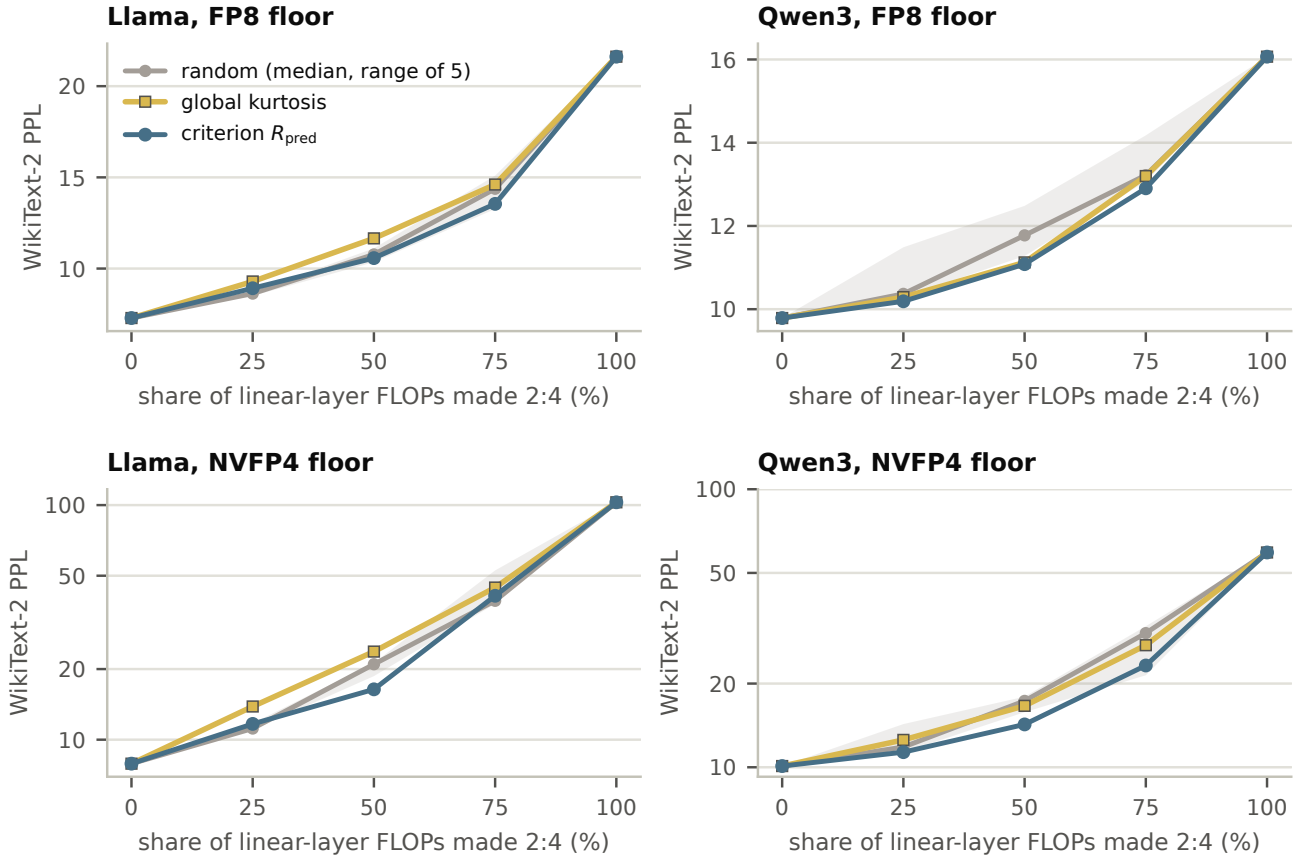


Figure 12. Stacking one-shot 2:4 on the FP8 floor in the order given by the criterion, by global kurtosis, or at random.

H TIMING DETAILS

Guarded profiling. After other kernels have run in a process, a `torch.profiler` session can silently drop its first kernel events; short timings then come out too fast, and an algorithm search keeps configurations that appear to take no time. Every measurement therefore starts with 512 tiny pad kernels that absorb the lost events, and it is accepted only if each timed kernel is captured a whole number of times per call.

Clocks and libraries. We record SM clock, power and throttle reasons before and after every run and use only GPUs that reach their normal clocks under load. With cuSPARSELt 0.8.1 preloaded, PyTorch’s bundled 0.7.1 is also mapped into the process, so our `ctypes` calls bind the preloaded library explicitly and record its version.

Cache state. An L2 cache of 50–128 MB holds whole activations at moderate M and compressed weights at decode, so repeated calls run faster than a pipeline that reads from DRAM. Preprocessing, decode-shaped GEMMs and the roofline sweep are therefore also timed with L2 flushed before every call; the prefill conclusions hold under both.

End-to-end serving. vLLM 0.11.2 is the last release that installs on our H100 host (glibc 2.28) and still serves 2:4 checkpoints, whose support was removed in 0.19 (vLLM maintainers, 2026b). LLM Compressor 0.8.1 writes the checkpoints in the compressed-tensors format: FP8 with per-channel weight scales and dynamic per-token activation scales on every linear layer except the LM head, and, for Sparse-Llama, the 2:4 bitmask format, which vLLM serves with its CUTLASS sparse GEMMs. Each configuration runs alone on one H100 with CUDA graphs, a 4096-token context, 85% memory utilization and prefix caching off. Prefill and decode prompts are random tokens; each timing is the median of three runs after a warm-up, and decode ignores the end-of-sequence token. Weight memory is the model memory vLLM reports after loading and includes the BF16 embeddings and LM head, which 2:4 does not shrink. As a control, Sparse-Llama’s weights were also served from dense-format checkpoints: throughput matches the dense rows of Table 5 within 3%, and GSM8K matches the 2:4 rows.

I RETRAINING DETAILS

Table 17. Retraining recipe.

Start	One-shot SparseGPT on Llama-3.1-8B, all linear weights; pairwise 4:8 and element-wise 2:4
Objective	Fixed-mask SquareHead distillation: KL divergence to the dense teacher on the next token plus the mean over layers of $\ h_s - h_t\ ^2 / \ h_t\ ^2$; pruned positions reset to zero after every step
Data	FineWeb-Edu sample-10BT, packed into 2048-token sequences with BOS per document; both students see the same sequences in the same order
Budget	7,629 steps (999,948,288 tokens); checkpoints at steps 1,908, 3,815 and 7,629
Optimizer	AdamW ($\beta = 0.9, 0.95$; no weight decay), peak learning rate 2×10^{-5} , 100 warm-up steps, cosine decay to 10%, gradient clipping 1.0; batch 64×2048 tokens; FSDP2 with FP32 master weights and BF16 compute
Hardware	4 H100 per student, about 16 hours

No hyperparameter was tuned, and each student was trained once. Format cost at 1B tokens (pairwise minus element-wise, BF16): +1.22 perplexity, 95% interval [1.15, 1.29]; −0.14 GSM8K, interval [−0.20, −0.09].