

IQR-V3: A Computed Trellis Code under One Scale a Row

Technical report on IQR-V3 for Qwen3.8-27B

Deadcode
radiance project

October 2026

Abstract

IQR-V3 is the third format of the IQR series for Qwen3.8-27B on two AMD R9700 cards. It keeps V2's tail-biting trellis inside GPTQ, but a state's value is no longer read from a trained table: it is computed from the state by three ALU instructions (a 24-bit multiply-add, a mask and a byte-sum dot product), and the trellis grows from 2^{12} to 2^{16} states. The scale is one static f32 a row instead of an E4M3 scale for every 128 weights, applied after the K loop. The GEMMs change most. A one-token GEMV no longer waits on bank-conflicted table gathers: at four bits it takes 0.58 of FP8's time on the largest weight against V2's 0.80, 1.38 $\times$ faster; and 1.67 $\times$ at two bits. At 8 and 16 tokens V3's narrow tile stays at or below FP8's time where V2 needs 1.17–2.11 $\times$ it, 1.7 $\times$ faster than V2 at 16 tokens. At prefill size V3's int8 tile folds like the FP8 tile: a decoder layer's trunk GEMMs take 1.00–1.06 $\times$ FP8's time against V2's 1.19–1.21 $\times$, a 1.15–1.20 $\times$ faster prefill GEMM. Quality improves as well. On the board's private split of 1.05M model-written positions every V3 entry with a trellis, from 2.5 to 7.4 bits a weight, is better than its V2 twin of the same size, by 6–11% in KL up to 6.4 bits and 2% at 6.9 and 7.4, and 23–44% better than V1; IQ4R-L-V3 is 12% below the official FP8 checkpoint at 55% of its size.

1 Background

IQR [1, 2] quantises each linear layer of Qwen3.8-27B on one grid. The weight is rotated by a 128-point Walsh–Hadamard transform, every value is an int8 code times a scale, and the activations are rotated and quantised to int8 per (token, 128 inputs), so one int8 $\times$ int8 matrix core serves every size. Codes are fitted by GPTQ [3] against calibration Hessians. Every format stays at its codes' width in VRAM: the GEMM decodes each weight where it reads it, and nothing is widened at load.

IQR-V2 made the codes a tail-biting bitshift trellis [4, 5] of 2^{12} states run down each column inside GPTQ. Every row of a column shares the column's Hessian diagonal, so the Viterbi search minimises GPTQ's own objective for that column and GPTQ's feedback stays exact. A row's value is a table entry: $\hat{w}_{r,j} = \sigma_{r,g} T[s_r]$, with T a trained table of 2^{12} int8 values and $\sigma_{r,g}$ an E4M3 scale per (row, 128 inputs). V2 cut KL by 21–40% against V1 at equal size, but left two costs in the GEMM:

- **The table.** Every weight's decode is one byte gathered from the 2^{12} -entry table in local data share (LDS). Thirty-two lanes gathering from random addresses collide on LDS banks, so the decode GEMV runs at the LDS's rate, not the memory's (Section 2).
- **The scale a (row, 128).** The prefill tile accumulates int32 over each 128-input block and folds it into f32 with the block's scales. With a scale per row and block, every accumulator slot pays a conversion and a multiply per block, issued between the matrix instructions: V2's prefill GEMM takes about 1.2 $\times$ the FP8 tile's time.

IQR-V3 removes both. Its trellis values are computed from the state by three ALU instructions, and its scale is one f32 a row over the whole input, applied after the K loop. Quality is kept by a larger

trellis (2^{16} states, which costs nothing once there is no table), a scale gain tuned for each width, and a GPTQ damping that no longer over-damps the layer inputs.

2 Where V2's GEMM time goes

At batch 1 a GEMM is a matrix–vector product, and a narrow format should be bound by its bytes: a 4-bit weight read at the card's 600 GB/s takes half the time of an 8-bit one. V2's GEMV reaches 373 GB/s at four bits and 192 GB/s at two: the 4- and 2-bit forms take the same 116–120 μ s on `ffn_gate_up`, against 149 μ s for FP8 (Table 7). The time is not spent on bytes.

The GEMV gives each wave a 16-row tile and each lane four inputs of every 128-input block, so a lane decodes 64 weights a block and its four values of a row become one `v_dot4` against four activation bytes. In V2's compiled loop those 64 weights cost 207 vector-ALU instructions (state extraction, packing, the dot products and the f32 fold) and 64 LDS byte gathers, one per weight, from the 2^{12} -entry table. Thirty-two lanes gathering from random addresses of a 4 KB table collide on the LDS's 32 banks; a gather takes about four LDS cycles, which bounds the loop at about four weights per CU and clock. That is the measured rate: $89 \cdot 10^6$ weights in 116 μ s on 64 CUs at 2.9 GHz. The vector ALUs, meanwhile, are about a fifth busy. A larger trellis would make it worse (a 2^{14} -entry table is 16 KB), and a faster decode needs fewer table reads, not fewer bytes.

At prefill size the decode runs once per weight and is cheap; the K loop is the matrix units', and what costs is the fold. The `int8` tile accumulates each 128-input block in `int32` and adds it into `f32` times the block's scales. With one scale per 128×128 block (FP8's), that is one fused multiply-add per accumulator slot and block; with a scale per row and block (V1 and V2's) it is a conversion, a scale load and a multiply more, and on RDNA4 vector and matrix instructions issue from the same slots. V2's prefill GEMMs take $1.19\text{--}1.21\times$ the FP8 tile's time at 2048 tokens (Table 9).

3 The design

3.1 A computed trellis code

The trellis is V2's: a column of the rotated weight $\tilde{W} \in \mathbb{R}^{N \times K}$ is coded as k -bit symbols $c_1, \dots, c_N$, one per row, and row r 's state is the last L bits, $s_r = (s_{r-1} 2^k + c_r) \bmod 2^L$, in tail-biting runs of 256 rows. What changes is the value of a state. Instead of a table entry it is

$$q(s) = \sum_{b=0}^3 \left\lfloor \frac{h(s)}{2^{8b}} \right\rfloor \bmod 64 - 126, \quad h(s) = (s \cdot A + B) \bmod 2^{32}, \quad (1)$$

the sum of the low six bits of the four bytes of a multiplicative hash, less 126, with $A = 0 \times B5297B$ and $B = 0 \times 2545F491$. On the GPU this is three instructions: a 24-bit multiply-add (`v_mad_u32_u24`: $s < 2^{16}$ and $A < 2^{24}$, so the product is exact modulo 2^{32}), an and with `0x3F3F3F3F`, and a byte-sum dot product (`v_dot4` against `0x01010101`) whose accumulator input carries the -126 . Each value is an integer in $[-126, 126]$: the code lives on IQR's `int8` grid exactly, so the prefill tile reads it with the same `int8` matrix instructions as before.

The sum of four independent uniforms on $\{0, \dots, 63\}$ is near-Gaussian (Irwin–Hall), with standard deviation $\sqrt{4(64^2 - 1)/12} = 36.9$ and support $\pm 3.4\sigma$; over the 2^{16} states its histogram is the binomial-like bell of Figure 1. A hashed code is to a trellis what a random table is: QTIP showed that pseudo-random Gaussian codes perform within a few percent of trained tables [5]. Here the value of the computation is that a table is no longer read at all, so L is no longer limited by the 64 KB of LDS a workgroup may hold: V3 uses $L = 16$, four times V2's states, at no decode cost. (At two bits the quantiser's search on the card holds 2^{15} states, Section 3.4; the 2-bit base uses $L = 15$.)

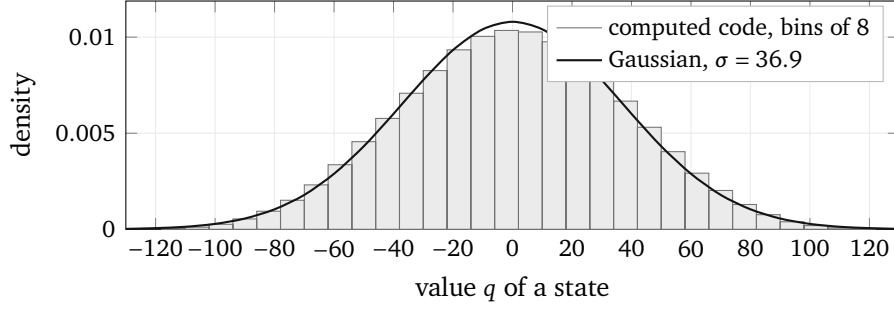


Figure 1. The values of the computed code over all 2^{16} states: a near-Gaussian bell on the int8 grid, bounded at ± 126 ($\pm 3.4\sigma$).

The container still carries the 2^L values as a table, beside a two-word `trellis.mad` plane holding A and B ; the loader checks that the table is the code’s and refuses a container whose table is anything else, so the kernels never decode a weight to values it does not hold.

3.2 One scale a row

A V3 weight’s value is

$$\hat{w}_{r,j} = \sigma_r q(s_{r,j}), \quad (2)$$

with one f32 scale σ_r for the whole row. There is no scale per 128 inputs: the trellis spreads each column’s choices over the states, and on the rotated, near-Gaussian weights a per-block scale adds little that a long trellis does not already capture. The scale is static, chosen before GPTQ from the row’s rms and a gain g_k for the code width,

$$\sigma_r = g_k \frac{\text{rms}(\tilde{w}_r)}{\text{rms}(q)}, \quad (3)$$

so the code’s spread matches the row’s, widened by g_k . The gain matters because the code is bounded: at $\pm 3.4\sigma$ it clips the rotated weights’ tails, and the more bits a row spends, the more those tails cost. The best gain grows with the width (Table 1): 1.04 at two bits, 1.24 at six and seven, where it lowers the layer error by 37% against the plain rms match.

Two projections want more. The attention output (`attn_output`) and the delta-net output (`ssm_out`) take the heads’ outputs as their input, and at five bits and up their error keeps falling well past the FFN weights’ best gain: on layer 3’s `attn_output` at seven bits it is $4.43 \cdot 10^{-6}$ at 1.24 and $2.22 \cdot 10^{-6}$ at 1.7, and `ssm_out` follows the same curve. These two projections, 8% of the trunk’s weights, take their own gain from five bits on: 1.36, 1.5 and 1.6 at five, six and seven bits. Below five bits both kinds of weight share one gain.

code bits k	2	3	4	5	6	7
states 2^L	2^{15}	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}
gain g_k	1.04	1.08	1.12	1.16	1.24	1.24
gain, head-fed	1.04	1.08	1.12	1.36	1.50	1.60
error at $g = 1$	$3.473 \cdot 10^{-2}$	$8.685 \cdot 10^{-3}$	$2.330 \cdot 10^{-3}$	$6.676 \cdot 10^{-4}$	$2.403 \cdot 10^{-4}$	–
error at g_k	$3.459 \cdot 10^{-2}$	$8.555 \cdot 10^{-3}$	$2.224 \cdot 10^{-3}$	$5.722 \cdot 10^{-4}$	$1.521 \cdot 10^{-4}$	$5.079 \cdot 10^{-5}$
change	–0.4%	–1.5%	–4.5%	–14%	–37%	

Table 1. The scale gain, chosen per width on layer 3’s `ffn_gate_up` from a sweep in steps of 0.04, and for the head-fed projections (`attn_output`, `ssm_out`) on layer 3’s and layer 2’s from a coarser one. The errors are `ffn_gate_up`’s: relative output error $\text{tr}(EHE^T)/\text{tr}(\tilde{W}H\tilde{W}^T)$.

The container also holds one f32 factor per 128-input block shared by every row (`scale.col`), all ones in the shipped recipes; the kernels fold it with the activation's per-128 scale, which they multiply anyway, so it costs nothing and keeps the format open to a column-block scale. Set from each block's share of the weight's energy it changes the FFN weights' layer error by under 1% and raises `attn_output`'s by up to 12%, so the shipped recipes leave it at one.

The scale form saves V2's E4M3 scales: 0.0625 bpw, which at 2–4 bits is 1.5–3% of the model. The row's f32 scale and the block factors cost under 0.01 bpw.

3.3 GPTQ

The search is V2's: columns in order, each trellis-coded over its rows by Viterbi in tail-biting runs of 256, its error fed to the later columns through the upper Cholesky factor of H^{-1} . Two things change.

Damping. GPTQ adds $\lambda = d \cdot \overline{H_{ii}}$ to the Hessian's diagonal. The inputs of the attention and delta-net projections carry massive-activation channels whose H_{ii} is up to $3.7 \cdot 10^4$ times the median: the mean diagonal is nine times the median, and $d = 0.05$ (V1 and V2's damping) adds 45% of the median channel's own curvature. Those inputs were over-damped. V3 uses $d = 0.01$, which lowers the layer error of layer 2's delta-net input projection by 4.6% and of the other projections by about 0.1%; on eight layers quantised at three bits it lowers the KL they add by 3.5%. A further step to 0.002 changes nothing.

The rounding of the code. The Viterbi searches against the values of Equation (1) directly, the same integers the kernels compute.

3.4 The quantiser on the card

The trellis search runs on the GPU. One workgroup holds one 256-row run; its threads hold the states, so that each thread's states include all predecessors of their groups: a step is each group's cheapest predecessor (in registers, or across a few lanes for wide codes), one LDS barrier, and each state's new cost from its group's best. The code's values are computed once per state into registers. At $L = 16$ the two cost buffers of a run fit the 64 KB of LDS from four-bit codes up; at three bits one buffer is used, at the price of a second barrier per step, and at two bits the 2^{16} states' single buffer would not fit, so the 2-bit base uses 2^{15} states. Weights split by row runs across the two cards, and the card's codes are byte-identical to the host's AVX-512 search, which the test suite checks for every form. A 178M-weight `ffn_gate_up` is searched in 12 s at $L = 16$ against 300 s on the host's 16 cores; a whole base converts in under an hour, the time going mostly to the host's double-precision Hessian factorisations.

4 The kernels

All three V3 kernels decode a unit (16 rows $\times$ 128 inputs) of codes the same way: a lane extracts each row's 16-bit state from its column's bit stream (a shift and a mask, or a funnel shift across a word boundary), then evaluates Equation (1). Nothing is staged in LDS and nothing is read but the codes.

GEMV, 2–4 tokens. A lane's 64 decoded values of a unit are packed four to a register and dotted with the activation. The values are kept as their unsigned byte sums, $q + 126$; the 126 comes off once per block and token as 126 times the lane's activation sum, one more `v_dot4` for four values. The block's partial sums are folded with the activation's per-128 scale times the block factor — one f32 for all

rows, so a scalar — and each row’s scale multiplies its sum once, after the K loop. Up to four tokens share a decode; from five, the narrow tile is faster.

GEMV, one token. With a single token there is nothing to share a packed register with, and the packing can go. A value’s masked hash $h \wedge 0x3F3F3F3F$ holds four bytes that sum to $q + 126$; a v_dot4 of it against the token’s activation byte repeated in all four bytes ($x \cdot 0x01010101$) is $(q + 126)x$, the byte sum and the product in one instruction. A weight is then one $v_mad_u32_u24$, one v_and and one v_dot4 , plus its share of the state extraction; the 126 comes off once per block as before. The compiled loop is 336 vector-ALU instructions for 64 weights and no LDS access (Table 2).

GEMV loop, one lane, 64 weights	VALU	LDS	dominant instructions
IQR-V2 (2^{12} -entry table)	207	64	64 <code>ds_load_i8</code> , 48 <code>v_perm</code> , 48 field ops
IQR-V3 (computed code)	336	0	64 <code>v_mad_u32_u24</code> , 65 <code>v_dot4</code> , 89 <code>v_and</code>

Table 2. The 4-bit one-token GEMV’s inner loop as compiled for gfx1201. V3 moves the decode from the LDS, which bounds V2, to the vector ALUs, which V2 left mostly idle.

Narrow tile, 5–64 tokens. The int8 narrow tile (one 16-row weight tile a wave against up to four 16-token tiles) takes each unit decoded and transposed into the matrix units’ fragment order through a small per-wave LDS tile; the fold is again the activation scale times the block factor, and the row scales are applied at the store.

Prefill tile. The prefill tile is the int8 tile V1 and V2 use, with the fold of a weight that has one scale per 32 rows: the int32 accumulator starts each block at $1.5 \cdot 2^{23}$, so the slot read as an f32 is the exact integer sum plus that bias, and the block is folded by one fused multiply-add with a wave-uniform scale, the FP8 tile’s arithmetic. For V3 that scale is the block factor, the same for all rows, and each output is multiplied by its row’s scale when it is stored. The K loop reads no per-row weight scale at all. The weight is decoded once into the tile’s fragment layout by the blocks of the GEMM itself, each value once, overlapped with the other blocks’ K loops.

Exactness. Every kernel computes on the values of Equation (1) times the scales; the selftest checks the GEMV, the narrow tile and the prefill tile against the int8 GEMMs over the host’s decode of the same weight, for every width.

5 Results

5.1 Per weight

Table 3 quantises layer 3’s two FFN weights at every width with both formats and reports the relative error in the metric of the layer’s own input Hessian, $\text{tr}(\Delta H \Delta^\top) / \text{tr}(WHW^\top)$, the quantity GPTQ minimises. V2 is its shipped grid: the trained 2^{12} -state table, an E4M3 scale for every (row, 128 columns) under one f32, damping 0.05. V3 is the computed 2^{16} -state code (2^{15} at two bits) under one f32 scale a row at the gains of Table 1, damping 0.01. Both run the same GPTQ on the same Hessian and the same rotation.

V3 is better at every width on both weights, and the two weights agree to a few tenths of a percent although their shapes and Hessians differ. The lead grows with the width, from 2.4% at two bits to 15% at seven. At low widths the code’s own error dominates and V3 wins by its trellis, sixteen times the states of V2’s table. At high widths the code is fine enough that the scale’s placement shows: V2

k	ffn_gate_up (34816 $\times$ 5120)			ffn_down (5120 $\times$ 17408)		
	V2	V3	Δ	V2	V3	Δ
2	$3.543 \cdot 10^{-2}$	$3.459 \cdot 10^{-2}$	-2.4%	$3.827 \cdot 10^{-2}$	$3.735 \cdot 10^{-2}$	-2.4%
3	$9.077 \cdot 10^{-3}$	$8.544 \cdot 10^{-3}$	-5.9%	$9.845 \cdot 10^{-3}$	$9.263 \cdot 10^{-3}$	-5.9%
4	$2.383 \cdot 10^{-3}$	$2.221 \cdot 10^{-3}$	-6.8%	$2.585 \cdot 10^{-3}$	$2.411 \cdot 10^{-3}$	-6.7%
5	$6.400 \cdot 10^{-4}$	$5.722 \cdot 10^{-4}$	-10.6%	$6.954 \cdot 10^{-4}$	$6.219 \cdot 10^{-4}$	-10.6%
6	$1.784 \cdot 10^{-4}$	$1.521 \cdot 10^{-4}$	-14.7%	$1.947 \cdot 10^{-4}$	$1.655 \cdot 10^{-4}$	-15.0%
7	$5.994 \cdot 10^{-5}$	$5.079 \cdot 10^{-5}$	-15.3%	$6.552 \cdot 10^{-5}$	$5.554 \cdot 10^{-5}$	-15.2%

Table 3. Layer 3, relative error in the input Hessian’s metric, IQR-V2 against IQR-V3 at k bits a weight.

fixes an E4M3 scale per 128 columns before GPTQ runs, so the grid’s step is set by the block’s peak and rounded to three mantissa bits, while V3’s single exact row scale times the tuned gain of Table 1 places the step for the row’s whole spread. After the 128-point Hadamard rotation the blocks of a row have nearly the same spread, so the per-block adaptivity V2 paid for buys little.

The head-fed projections are the exception (Table 4). Here V3 is level with V2 at two and three bits, 5–8% behind at four and five, 16% at six and 56% at seven, with each width at its best gain. Their error is far below the FFN weights’ at every width (at seven bits $2 \cdot 10^{-6}$ against $5 \cdot 10^{-5}$ for attn_output), so V3’s code reaches its resolution there first: the code’s values are integers within ± 126 with a spread of 36.9, and a row scale fitted to the row’s spread leaves the finest steps of a seven-bit code no room. V2’s E4M3 scale for every 128 inputs follows the heads’ outputs and keeps the step small where the block’s values are. The head-fed projections are 8% of the trunk; the FFN weights are 70%, and the ladder shows whether the head-fed rows cost V3 its lead at the top widths (Section 5.2).

k	attn_output, layer 3 (5120 $\times$ 6144)			ssm_out, layer 2 (5120 $\times$ 6144)		
	V2	V3	Δ	V2	V3	Δ
2	$8.010 \cdot 10^{-4}$	$7.933 \cdot 10^{-4}$	-1.0%	$1.467 \cdot 10^{-2}$	$1.484 \cdot 10^{-2}$	+1.2%
3	$2.019 \cdot 10^{-4}$	$1.971 \cdot 10^{-4}$	-2.4%	$3.762 \cdot 10^{-3}$	$3.762 \cdot 10^{-3}$	0.0%
4	$5.282 \cdot 10^{-5}$	$5.562 \cdot 10^{-5}$	+5.3%	$9.873 \cdot 10^{-4}$	$1.050 \cdot 10^{-3}$	+6.3%
5	$1.421 \cdot 10^{-5}$	$1.512 \cdot 10^{-5}$	+6.4%	$2.657 \cdot 10^{-4}$	$2.865 \cdot 10^{-4}$	+7.9%
6	$4.019 \cdot 10^{-6}$	$4.645 \cdot 10^{-6}$	+15.6%	$7.447 \cdot 10^{-5}$	$8.677 \cdot 10^{-5}$	+16.5%
7	$1.392 \cdot 10^{-6}$	$2.169 \cdot 10^{-6}$	+55.9%	$2.547 \cdot 10^{-5}$	$3.965 \cdot 10^{-5}$	+55.7%

Table 4. The head-fed projections, relative error in the input Hessian’s metric, IQR-V2 against IQR-V3 at the gains of Table 1.

The bits a weight are the same in both columns except for the scales: V2 stores one byte for every 128 weights (0.0625 bits a weight), V3 four bytes a row (under 0.007 bits a weight on these shapes). On the whole model that is about 0.06 bits a weight that V3 hands to the allocator, which spends them on wider codes for the sensitive tensors.

The card quantiser takes 10 to 13 seconds for these weights with V3 against 1 to 3 seconds with V2’s, the cost of sixteen times the trellis states; a full model converts in about half an hour on two cards.

5.2 The ladder

Uniform bases. Before any mix, each width is converted on its own: every trunk projection at k bits, the embedding and head as in V2’s bases. Table 5 scores them on the selection corpus (464K positions, the allocator’s input; not the board’s split). V3 is ahead of V2 at every width from two

to six bits, by 3–7%, and level at seven, each time in 160 MB less: V2’s E4M3 scale for every 128 weights is 0.0625 bpw, V3’s row scale almost nothing.

k	IQR-V2		IQR-V3		ΔKL
	GB	KL	GB	KL	
2	9.730	0.11573	9.562	0.11220	−3.1%
3	12.818	0.02715	12.659	0.02525	−7.0%
4	15.905	0.00733	15.746	0.00710	−3.1%
5	18.993	0.00230	18.834	0.00216	−6.1%
6	22.080	0.00093	21.921	0.00087	−6.5%
7	25.167	0.00052	25.008	0.00052	0%

Table 5. Every trunk projection at k bits, KL against the bf16 model on the selection corpus.

The ladder. Each board entry is a mix of the bases: the allocator of V2’s report splits every band of eight layers into its FFN-up, FFN-down and attention units, puts each unit on any base or on the int8 grid, and takes the mix predicted best at the entry’s size, here at V2’s sizes to the third decimal of a bpw. The entries are scored on the board’s private split, 1.05M positions the model writes itself, against the bf16 model’s top-512 distribution (Table 6). The GEMM column is the trunk’s GEMM time a decoder layer at 2048 tokens for the entry’s own mix, from Table 9’s kernels.

Every V3 entry with a trellis is better than its V2 twin of the same size: by 6–11% in KL from IQ2R to IQ6R (2.5 to 6.4 bits), and by 2% at IQ6R-L and IQ7R. IQ8R is the int8 grid in both formats, the same container, and scores the same; it is in the table as the ladder’s top, not as a comparison. Against V1 the gain is 39–44% up to five bits, 29–30% at six and 23% at IQ6R-L. The margin is widest in the middle of the ladder, where the FFN weights sit at four and five bits, and narrows above six bits for two reasons: more of each entry is on the int8 grid that V2 and V3 share (47% of IQ7R’s trunk), and the head-fed projections give back part of V3’s lead there (Table 4). IQ4R-L-V3 is below the official FP8 checkpoint (KL 0.00389) at 55% of its size, as IQ4R-L-V2 was, now by 12% instead of 3%.

The tail follows the mean. V3’s median KL and its 99th percentile are below V2’s at every entry up to IQ6R-L, and its top-1 agreement is higher up to IQ6R; from IQ6R-L on the two are level (top-1 99.04% against 99.05% at IQ6R-L, p99 0.0055 against 0.0053 at IQ7R). The GEMM column is the trunk’s GEMM time a decoder layer at 2048 tokens for each entry’s own mix: 4935–5236 μs for V3 up to IQ6R-L against V2’s 5850–5921, a 1.13–1.19 $\times$ faster prefill GEMM, within 0.7–6.8% of the FP8 checkpoint’s 4901 μs . IQ7R’s int8 half takes the int8 grid’s time in both formats, and there V3 is 1.07 $\times$ faster; IQ8R, all int8, takes the int8 grid’s 5699 μs in both.

5.3 GEMM speed

All times are one rank of the two-card tensor-parallel split (each card holds half of every weight), measured by the engine’s own kernels on one R9700 at its 210 W cap: the median of seven rounds, with the variants of a shape interleaved within each round so clock drift falls on all of them alike. Ratios are taken within one run; the one-token rows are their own run. FP8 is the engine’s FP8 GEMM with one E4M3 scale per 128×128 block, int8 the IQR8 grid, V1 its 4-bit table form (t4), V2 and V3 their trellis forms at k bits. Every IQR time decodes the stored form in the kernel; no format is widened.

One token. Table 7 is decode’s case. FP8 runs at the memory’s rate, 600 GB/s, and the int8 grid matches it. V2 does not follow its bytes down: its two- and four-bit forms take the same time,

entry	IQR-V3				vs V2		vs V1		GEMM, μ s	
	GB	bpw	KL	top-1	KL	Δ	KL	Δ	V3	V2
IQ2R	8.29	2.465	0.06807	90.54%	0.07303	−7%	0.12234	−44%	4935	5856
IQ2R-L	9.37	2.786	0.04241	92.62%	0.04698	−10%	0.06937	−39%	4973	5850
IQ3R-S	10.57	3.145	0.02584	94.02%	0.02798	−8%	0.04469	−42%	5010	5850
IQ3R	11.63	3.459	0.01698	95.35%	0.01880	−10%	0.03043	−44%	5022	5850
IQ3R-L	12.77	3.798	0.01129	96.16%	0.01198	−6%	0.01887	−40%	5030	5856
IQ4R-S	13.89	4.131	0.00762	96.90%	0.00832	−8%	0.01327	−43%	5047	5860
IQ4R	14.99	4.458	0.00512	97.45%	0.00562	−9%	0.00904	−43%	5070	5863
IQ4R-L	16.08	4.783	0.00342	97.86%	0.00377	−9%	0.00609	−44%	5100	5868
IQ5R-S	17.02	5.064	0.00249	98.15%	0.00273	−9%	0.00426	−42%	5112	5875
IQ5R	18.23	5.422	0.00182	98.38%	0.00204	−11%	0.00300	−39%	5122	5886
IQ6R-S	20.34	6.049	0.00106	98.83%	0.00115	−8%	0.00151	−30%	5136	5908
IQ6R	21.42	6.372	0.00088	98.93%	0.00094	−6%	0.00123	−29%	5169	5921
IQ6R-L	23.33	6.939	0.00072	99.04%	0.00073	−2%	0.00092	−23%	5236	5915
IQ7R	24.95	7.420	0.00060	99.12%	0.00061	−2%	—	—	5421	5823
IQ8R	27.03	8.040	0.00047	99.18%	0.00047	±0%	—	—	5699	5699

Table 6. The IQR-V3 ladder on the board’s private split (t1 and t2, 1.05M model-written positions): each entry against its V2 and V1 twin of the same name and size.

0.78–0.81 of FP8’s on the FFN weights, bound by the table gathers of Section 2. V3 has no table and follows the bytes: at four bits it takes 0.58 of FP8’s time on `ffn_gate_up` and 0.65 on `ffn_down`, against V2’s 0.80 and 0.81, so a V3 4-bit weight is read 1.38 $\times$ and 1.23 $\times$ as fast as V2’s. At two bits the gap is 1.67 $\times$ and 1.50 $\times$, where the hash-dot GEMV’s three instructions a weight are most of the loop. V3 at four bits is as fast as V1’s 4-bit table form, which spends no ALU on a trellis. V1’s 2-bit vector grid stays the fastest one-token form (0.38 of FP8’s time on `ffn_gate_up`), one byte a four weights indexing a 256-entry codebook, at V1’s quality. On the small `ssm_out` weight, where launch and tail are a larger share, V3 takes 0.79–0.85 of V2’s time.

shape	FP8		V1		V2		V3			
	μ s	int8	k=4		2	4	2	3	4	5
<code>ffn_gate_up</code>	148.8	1.01	0.59	0.78	0.80	0.47	0.54	0.58	0.70	0.81
<code>ffn_down</code>	75.1	1.01	0.68	0.78	0.81	0.52	0.59	0.65	0.77	0.89
<code>ssm_out</code>	29.8	1.01	0.86	0.95	1.00	0.75	0.83	0.85	0.96	1.04

Table 7. One token ($M = 1$): FP8’s time, and every other form’s time relative to it. Lower is faster.

A few tokens. Table 8 follows the 4-bit forms from one token to prefill size. Up to four tokens the GEMV serves every form and the cost of the decode grows with the tokens it is shared by. From five tokens to 64 every form runs the narrow tile, and V2’s per-(row, 128) scales make that tile slower than FP8’s: 1.17–2.11 $\times$ FP8’s time at 8 and 16 tokens, the case of several decode streams or a speculative draft’s verification. V3’s narrow tile folds with one block factor and applies the row scale at the store, and stays at 0.71–1.01 of FP8’s time there: at 16 tokens V3 is 1.69 $\times$ as fast as V2 on `ffn_gate_up`, 1.68 $\times$ on `ffn_down` and 1.70 $\times$ on `ssm_out`. At 64 and 512 tokens V3 takes 0.70–0.85 of V2’s time.

Prefill. At 2048 tokens the decode is once per weight and the time is the K loop’s. Table 9 sums the GEMMs of a decoder layer, every trunk projection of the model at one width, averaged over the 64 layers (48 linear-attention and 16 full-attention). FP8’s block-scaled tile takes 4901 μ s a layer. The

M	ffn_gate_up			ffn_down			ssm_out		
	V1	V2	V3	V1	V2	V3	V1	V2	V3
1	0.59	0.80	0.58	0.68	0.81	0.65	0.86	1.00	0.85
2	0.61	0.80	0.64	0.68	0.80	0.69	0.91	1.06	0.89
4	0.70	0.88	0.81	0.82	0.97	0.90	1.14	1.32	1.13
8	1.02	1.17	0.75	1.46	1.52	1.01	1.97	2.11	0.89
16	0.79	1.20	0.71	1.06	1.56	0.93	0.97	1.50	0.88
64	1.21	1.52	1.09	1.60	2.11	1.50	1.28	1.86	1.31
512	1.33	1.37	1.13	1.38	1.49	1.27	1.40	1.46	1.23
2048	1.17	1.19	1.04	1.18	1.18	1.02	1.26	1.26	1.07

Table 8. The 4-bit forms’ time relative to FP8’s at M tokens.

int8 grid, V1 and V2 fold a scale per (row, 128) and take $1.15\text{--}1.21\times$ that, V2 the slowest. V3 folds like FP8 and is within 0–6% of it: 0.998 at two bits, 1.028 at four, 1.058 at seven. Against V2 that is a $1.20\times$ faster prefill GEMM at two bits, $1.16\times$ at four and $1.15\times$ at six, at every width faster than the int8 grid itself. The residue against FP8 grows with the width, with the bits the tile decodes, and it is the one place V3 is still behind FP8.

k	V1		V2		V3	
	μs	/FP8	μs	/FP8	μs	/FP8
2	5642	1.151	5862	1.196	4893	0.998
3	5728	1.169	5845	1.193	5006	1.021
4	5666	1.156	5860	1.196	5037	1.028
5	5709	1.165	5872	1.198	5117	1.044
6	5764	1.176	5910	1.206	5133	1.047
7	–	–	5948	1.214	5184	1.058
FP8 4901 μs (1.000), int8 grid 5699 μs (1.163)						

Table 9. Trunk GEMM time of one decoder layer at 2048 tokens, every projection at k bits, one rank. V1’s 6-bit row is its integer form with a zero point.

6 Summary

IQR-V3 replaces V2’s table with a computed code and V2’s block scales with one scale a row, and both changes pay in the GEMM. With no table the decode is ALU work the GEMV had idle, so a narrow weight is finally read about as fast as its bytes allow; with one scale a row the prefill tile folds like FP8’s, and the int8 path that V1 and V2 paid 15–21% for at prefill size now costs 0–6%. The larger trellis that the computed code makes free, a gain for each width (and a larger one for the projections fed by the attention heads), and a lighter GPTQ damping make the format better per bit than V2 at the same time: every trellis entry of the ladder is ahead of its V2 twin, by 6–11% in KL up to 6.4 bits and 2% above, and 23–44% ahead of V1.

What is left is at the top widths. On the projections fed by the attention heads, 8% of the trunk, V3’s integer code reaches its resolution before V2’s per-block scales do, and from six bits on those weights are behind V2’s (Table 4). The prefill GEMM is still up to 6% behind FP8’s at seven bits, the decode of the wider codes. Both are the code’s: a code with a wider spread for the wide widths, still computed and still within int8, is the next step.

References

- [1] Deadcode. IQR: GPTQ integer quantisation on a rotated int8 grid. Technical report on IQR-V1, radiance project, 2026.
- [2] Deadcode. IQR-V2: trellis-coded GPTQ columns on a rotated int8 grid. Technical report on IQR-V2, radiance project, 2026.
- [3] E. Frantar, S. Ashkboos, T. Hoefler, D. Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *ICLR*, 2023.
- [4] M. W. Marcellin, T. R. Fischer. Trellis coded quantization of memoryless and Gauss–Markov sources. *IEEE Trans. Commun.* 38(1):82–93, 1990.
- [5] A. Tseng, Q. Sun, D. Hou, C. De Sa. QTIP: Quantization with trellises and incoherence processing. *NeurIPS*, 2024.