Q3.6: METADATA-CONDITIONED LOSSLESS COMPRESSION FOR QUANTIZED LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-bit quantization has become a popular way to deploy large language models (LLMs) on resource-constrained devices. However, after a model has already been quantized, residual redundancy in the quantized weights remains largely unexploited, while further quantizing to even lower bit widths can cause unacceptable generation-quality loss. This motivates dedicated lossless compressors for quantized LLMs: they reduce storage after quantization while preserving the exact quantized weights. Existing lossless LLM compressors target floating-point formats such as bfloat16 (BF16), re-quantize before coding, or treat weights as generic byte streams; none conditions on the native metadata of a fixed low-bit object. We propose **`Q3.6`**, a lossless compressor for already-quantized LLMs with bit-exact decoding. **`Q3.6`** builds on the observation that stored metadata, such as scales, predicts the symbol distribution within the corresponding weight blocks. It derives deterministic entropy buckets from the native metadata and encodes each bucket with a matched table-based entropy model, so blocks with similar metadata share a bucket and a bitstream. **`Q3.6`** stores less than a strong Zstandard (zstd) baseline in every evaluated setting, covering three models, five quantized formats, and three bit widths, and saves about 10% on average relative to the uncompressed target tensors. In storage-offloaded serving at 1.5 GB/s, **`Q3.6`** gives the smallest stream and achieves the fastest decoding among the evaluated GPU codecs. A fused decode-plus-GEMM kernel consumes this stream, lowering time per output token by 7 to 10%. The freed GPU memory provides 16% to 90% more KV cache and up to 2.1× throughput in vLLM at 8 to 24 GB GPU budgets, with tokens identical to the stock engines. Code is available at `https://anonymous.4open.science/r/iclr2027-code-4B00`.

1 INTRODUCTION

Large language model (LLM) deployment increasingly relies on low-bit quantization to reduce storage, memory footprint, and inference cost (Gong et al., 2025; Frantar et al., 2023; Lin et al., 2024), with serving formats such as W4A16/INT4 layouts and MXFP4 (NVIDIA Corporation, 2025; Rouhani et al., 2023). Quantization, however, is lossy. Moving an already-quantized model to an even lower bit width can change generation behavior and requires task-level evaluation, especially in extreme low-bit regimes (Liu et al., 2025). Once a model has been quantized to an acceptable operating point, a safer question is whether its serialized representation can be made smaller without changing the quantized weights at all.

Lossless post-quantization compression addresses this question: it reduces storage after quantization while preserving the exact quantized object and therefore the model's behavior. Two obstacles stand in the way. **Metadata conditioning is unexplored:** prior lossless LLM compressors target floating-point formats, re-quantize the model before coding, or treat weights as generic byte streams (Section 2). Recent concurrent work also studies fixed low-bit objects and GPU ANS decoding, but does not use the native per-block quantization metadata as an entropy-routing context (Tan et al., 2026). We study that missing conditioning signal while keeping the serialized low-bit object fixed. **Generic compressors ignore quantization structure:** DEFLATE/ZIP and Zstandard (Deutsch, 1996; Collet & Kucherawy, 2021) are strong baselines but treat a quantized model as an unconditional byte stream. This ignores a key property of quantized weights: each tensor or block stores

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

its low-bit symbols together with native quantization metadata, the decoder-available fields that interpret the payload. This metadata is already stored for dequantization and bit-exact recovery, yet byte-stream compression never uses it as statistical context.

Q3.6 is a lossless compressor built on the structure shared by low-bit quantized objects: discrete payload symbols plus native quantization metadata. The key idea is to use the native metadata, which the decoder already has, as entropy context. We show that metadata is predictive of payload entropy: groups with similar scales or scale codes often have similar symbol distributions, and their conditional entropy can be lower than that of the global payload stream. **Q3.6** therefore derives deterministic entropy buckets from native metadata and encodes each bucket with a matched asymmetric numeral systems (ANS) model (Duda, 2013). The decoder recomputes the same bucket assignments from stored metadata and route metadata, decodes each bucket stream, and scatters the recovered symbols back to the original tensor. Figure 1 shows the effect on payload rates. With native metadata, route metadata, and coding tables charged, **Q3.6** still stores less than the strengthened zstd baseline in every evaluated setting (Table 1).

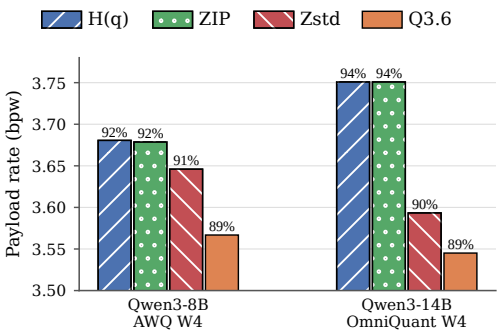

Figure 1: Representative 4-bit-weight (W4) payload-only rates. Percentages are relative to the raw 4 bits-per-weight (bpw) payload size; $H(q)$ is the empirical entropy of the unconditioned payload.

Our contributions are:

- We analyze how native quantization metadata relates to payload entropy. Measurements and the finite-sample analysis in Appendix A show why scales and scale codes can lower the coding rate across models, quantization methods, and bit widths.
- We propose **Q3.6**, a metadata-conditioned lossless compressor for quantized LLM weights. It routes the payload of each tensor or block into entropy buckets derived from the native metadata and encodes each bucket as an independent table-based entropy stream, with bit-exact recovery.
- Across three models, five quantized formats, and three bit widths, **Q3.6** uses less total storage than a strengthened zstd baseline in every evaluated setting, and both its CPU and GPU decoders recover the quantized payload bit for bit.
- For storage-bound serving, our GPU decoder is 2.1 to 2.3 times as fast as the public dietGPU decoder. **Q3.6** produces smaller streams and decodes faster than the evaluated nvCOMP codecs. Streamed through a fused decode-and-multiply kernel that avoids full-model weight materialization, the entropy-coded weights cut the time per output token by 7 to 10% in vLLM, SGLang, and FlexGen, and in vLLM the freed weight buffers yield 16 to 90% more KV cache and up to 2.1× the throughput.

2 Related Work

Quantized LLMs and serving formats. Post-training quantization reduces LLM storage and inference cost through methods such as LLM.int8(), SmoothQuant, GPTQ, AWQ, OmniQuant, and QLoRA/NF4 (Dettmers et al., 2022; Xiao et al., 2023; Frantar et al., 2023; Lin et al., 2024; Shao et al., 2024; Dettmers et al., 2023). 4-bit weight / 16-bit activation (W4A16) formats are widely supported by vLLM, SGLang, TensorRT-LLM, and llama.cpp (Kwon et al., 2023; Zheng et al., 2024; NVIDIA Corporation, 2025; Gerganov & contributors, 2023), while MXFP4 and any4 explore microscaled or learned 4-bit representations (Rouhani et al., 2023; Elhoushi & Johnson, 2025). These methods optimize the lossy mapping; **Q3.6** operates afterward and compresses the exact quantized object losslessly.

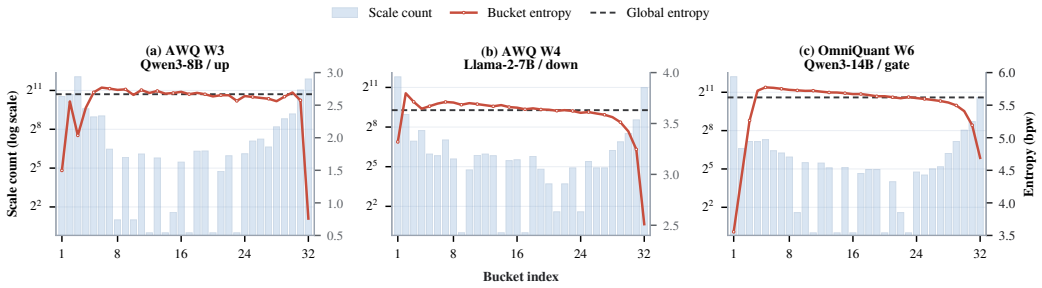

Figure 2: Conditioning on native quantization metadata codes reveals structured entropy variation in quantized payloads (three cases spanning models, bit widths, tensor families, and quantization methods). Buckets are sorted by metadata code; blue bars give the number of scale codes per bucket, red curves the bucket-level payload entropy, and dashed lines the unconditioned global entropy.

General-purpose compression and entropy coding. Ziv–Lempel compression, Huffman coding, DEFLATE, Zstandard, and asymmetric numeral systems (ANS) provide strong bit-exact backends (Ziv & Lempel, 1977; Huffman, 1952; Deutsch, 1996; Collet & Kucherawy, 2021; Duda, 2013), and nvCOMP (NVIDIA Corporation, 2026) and dietGPU (Johnson, 2023) bring such codecs to GPUs. Byte-stream compressors ignore tensor families, groups, and native metadata. Entropy coding conditioned on decoder-side information is established in learned image compression (Ballé et al., 2018) and in DeepCABAC (Wiedemann et al., 2020); **`Q3.6`** uses the native quantization metadata of low-bit LLM weights as that context.

Lossless compression for LLM weights. ZipNN, Huff-LLM, DFloat11, ZipServ, and Unweight compress floating-point weights or exploit runtime structure (Hershcovitch et al., 2024; Yubeaton et al., 2025; Zhang et al., 2025; Fan et al., 2026; Nikulin, 2026). EntroLLM (Sanyal et al., 2025) combines mixed quantization with Huffman coding, and Heilper & Singer (2025) extend ZipNN-style exponent/mantissa coding to FP8/FP4. **`Q3.6`** instead leaves a fixed already-quantized object unchanged, derives deterministic buckets from the decoder-visible scale codes, and charges native metadata, route descriptors, and coding tables. FlexGen (Sheng et al., 2023) streams weights from host memory or disk; **`Q3.6`** reduces these transfers.

Concurrently, Tan et al. (2026) study lossless compression of int4, AWQ, and SQ8 LLM weights and an ANS-based tile-level GPU decoder. Their emphasis is model-wide entropy analysis and inference-aligned tile decompression, whereas **`Q3.6`** conditions bucket codebooks on native quantization metadata for an already-serialized low-bit object and targets format-preserving storage. The two works use different checkpoints, quantization conventions, and deployment pipelines, so their absolute entropy and throughput numbers are not directly comparable; we therefore treat this as concurrent work rather than a numerical baseline.

3 **`Q3.6`**: Metadata-Conditioned Exact Lossless Coding

3.1 Preliminary: Quantized Objects and Metadata

Low-bit LLM serving formats represent each quantized tensor as a structured object rather than as a single array of floating-point values. We use *native quantization metadata* to refer to the quantization parameters and related decoder-available fields required to interpret the payload, such as scales, zero-points, mins, scale codes, or codebooks, and write a quantized object as $Q = (\mathbf{q}, \mathbf{s})$, where the payload $\mathbf{q}$ holds the low-bit symbols or packed qweight stream and $\mathbf{s}$ is the metadata.

In a typical group-wise affine quantizer, a weight in group g is interpreted as $\widehat{w}_{g,j} = \alpha_g(q_{g,j} - z_g)$, where $q_{g,j}$ is the low-bit code, α_g is a scale, and z_g is a zero point or offset. Thus the same code can represent different values under different metadata. In AWQ, $\mathbf{s}$ contains scales and qzeros; in OmniQuant, scales and zeros; and in MXFP4-like formats, block-level scale codes or parameters.

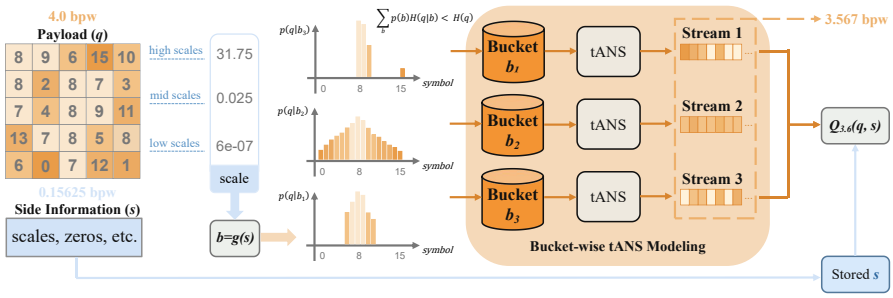

Figure 3: Overview of **Q3.6**. The encoder separates a quantized object into payload and native metadata, derives deterministic route codes from the metadata, partitions the payload into bucket streams, and encodes each with a bucket-specific ANS model. The stored object holds the payload frames, the native metadata, the coding tables, and the route metadata needed for bit-exact recovery.

Q3.6 is lossless with respect to the quantized object Q: it does not re-quantize the model, change the bit width, or approximate recovered weights; it changes only how the payload $\mathbf{q}$ is represented on disk and keeps the metadata needed to recover the exact quantized tensor.

3.2 Metadata as a Rate-Useful Coding Context

Native metadata provides coding context for the payload. A generic byte compressor models the payload as one unstructured stream, but groups with different scales or scale codes often have different symbol frequencies: the scale is a coarse summary of the local weight regime, and both small-scale (clipping-dominated) and outlier-dominated groups concentrate probability on a few code values. Figure 2 shows this for three representative cases: the per-bucket entropy varies non-monotonically around the global payload entropy. Appendix A gives the finite-sample analysis.

Let $c = g(\mathbf{s})$ be a route code derived deterministically from the metadata and $B = \mathrm{Bucketize}(c)$ the entropy bucket; routing induces the empirical conditional entropy $H(X \mid B) = \sum_b \frac{n_b}{N} H(X \mid B = b)$ over the non-empty buckets (n_b of the N symbols fall in bucket b; X is a payload symbol), which never exceeds $H(X)$ and is strictly below it whenever symbol and bucket are dependent (Appendix A.2). Using stored metadata adds no separately trained predictor or side channel; only the route rule and the coding tables add storage.

3.3 **Q3.6**: Compression Framework

Q3.6 uses the native metadata to route payload symbols into deterministic buckets, then encodes each bucket with a table-based codec. The decoder must reproduce the same routing, and the route metadata is explicitly charged.

ANS coding. An asymmetric numeral systems (ANS) coder (Duda, 2013) represents a discrete distribution by a table of quantized symbol frequencies: the encoder emits a bitstream, and the decoder recovers symbols through table lookups and state updates. **Q3.6** has two ANS backends, the tANS coder of FSE (Collet, 2013) on the CPU and the ANS coder of dietGPU (Johnson, 2023) on the GPU. Low-bit quantized payloads have small alphabets, so **Q3.6** builds one compact ANS model per entropy bucket rather than forcing the whole payload to share one global model.

Metadata-conditioned routing. **Q3.6** stores four components: entropy-coded payload frames, serialized coding tables, native quantization metadata, and compact route metadata (Figure 3). For each group or block, the route code $c_i = g(\mathbf{s}_i)$ is the stored scale code (the 16-bit scale pattern in AWQ and OmniQuant, the 8-bit scale code in MXFP4). Within each tensor family, route codes are partitioned into contiguous buckets using stored split points; the decoder recomputes the same route codes from the native metadata and applies the stored split points, so bucket assignment is deterministic and requires no separately trained predictor; the split points are fitted at encoding time, stored, and charged (Appendix A.1).

Rate condition. For lossless storage reduction, the relevant quantity is conditional rate rather than correlation alone. Let X be a payload symbol drawn from $\mathbf{q}$ and $B = b(S)$ a finite context derived from the metadata S. A bucket-conditioned entropy coder has ideal payload rate $H(X \mid B)$ and is useful only when this rate plus the incremental route and model overheads is below the baseline storage rate:

$$H(X \mid B) + R_{\text{meta}}(B) + R_{\text{model}} < R_{\text{base}}. \tag{1}$$

Here $R_{\text{meta}}(B)$ is the extra rate needed to represent and recover the bucket routing used by B, beyond the native metadata that both the baseline and `Q3.6` store; R_{model} is the rate of the serialized coding tables; and R_{base} is the baseline storage rate. If the baseline is an ideal unconditional code, $R_{\text{base}} = H(X)$ and the condition becomes $I(X;B) = H(X) - H(X \mid B) > R_{\text{meta}}(B) + R_{\text{model}}$: improvements must cover these overheads, not just lower payload entropy (Appendix A.2).

Encoding and decoding. Algorithm 1 gives the storage encoder and decoder. The encoder separates the quantized object into payload symbols and native metadata but does not modify the metadata; it computes route codes from the metadata and fits M_{route}, the state the decoder needs to reproduce the same bucket assignment (family layout, bucket counts, and split points). After routing, each non-empty bucket is encoded with its own table; the tables are stored and charged in the reported rate. The decoder parses M_{route}, recomputes the route codes from the stored metadata, decodes each bucket stream F_b with its own table T_b, and scatters the symbols back to the original layout. Bit-exact recovery follows from two conditions: the codec is lossless within each bucket stream, and routing is a deterministic function of stored metadata plus route metadata, so everything needed to invert the transform is either present in $\mathbf{s}$ or serialized in M_{route} and $\{T_b\}$.

Algorithm 1: `Q3.6` encoding and decoding

Require: Encoder: quantized object $Q = (\mathbf{q}, \mathbf{s})$ with payload groups $\mathbf{q}_i$, native metadata $\mathbf{s}_i$, and tensor family f_i; K_f buckets per family f, the bucket set $\mathcal{B}_f$.
Return: Compressed frames $\{F_b\}$, coding tables $\{T_b\}$, route metadata M_{route}, and the unchanged metadata $\mathbf{s}$.

1: **for** each payload group i **do** derive the route code $c_i \leftarrow g(\mathbf{s}_i)$
2: initialize split points from the route-code histogram, refine them on the payload code length (Appendix A.1) $\rightarrow M_{\text{route}}$
3: **for** each payload group i **do** assign the bucket $b_i \leftarrow \text{Bucketize}(c_i, f_i; M_{\text{route}})$
4: partition the payload into bucket streams $\{\mathbf{q}_b\}_{b \in \mathcal{B}_f}$
5: **for** each non-empty bucket b **do** build the table T_b from $\mathbf{q}_b$ and encode $F_b \leftarrow \text{ANS.Encode}(\mathbf{q}_b, T_b)$
6: store $\{F_b\}$, $\{T_b\}$, M_{route}, and $\mathbf{s}$

Require: Decoder: $\{F_b\}$, $\{T_b\}$, M_{route}, and $\mathbf{s}$.
Return: Recovered quantized object $\widehat{Q}$.

7: parse M_{route} and recover the bucketization rule
8: **for** each payload group i **do** recompute $c_i \leftarrow g(\mathbf{s}_i)$ from the stored metadata and assign $b_i \leftarrow \text{Bucketize}(c_i, f_i; M_{\text{route}})$
9: **for** each non-empty bucket b **do** decode $\widehat{\mathbf{q}}_b \leftarrow \text{ANS.Decode}(F_b, T_b)$
10: scatter the decoded bucket streams back to the original payload layout
11: restore $\widehat{Q} \leftarrow (\widehat{\mathbf{q}}, \mathbf{s})$

3.4 Consuming the Stream on the GPU

Two kernels serve the stored representation in the storage-bound setting, where the weights stream from an SSD or a network tier on every forward pass (Sheng et al., 2023). The decoder preserves the public dietGPU archive format and replaces only the decode kernel, serving each coder-state renormalization from a register window instead of a dependent global load. It produces bit-identical output and runs 2.1–2.3 times as fast as the public decoder (Appendix C.2). During decode, the fused decode-and-multiply kernel keeps decoded weights only in shared memory. A warp decodes each 32 KB block directly into a shared-memory tile serving as the tensor-core operand. Stored scales and split points determine the placement maps; persistent weight state for streamed modules comprises only scales and zeros. For prefill batches above 256 tokens, weights are decoded one

Table 1: Full component breakdown under the reported serialized-storage accounting for MLP gate/up/down payloads across the evaluated models, quantization methods, and bit widths. Under this reported accounting, the zstd column is the baseline total storage rate, and "others" is raw metadata plus route metadata; `Q3.6` payload includes bucket streams and serialized entropy models. The final row is the arithmetic mean over the evaluated target-tensor rows of the size reduction relative to each row's uncompressed payload-plus-native-metadata rate (not a whole-checkpoint rate).

Model	Quant. Method	Bit Width	zstd	Q3.6-FSE			Q3.6-dietGPU		
				payload	others	total	payload	others	total
Qwen3-8B	AWQ	W3	2.880	2.606	0.149	2.755	2.623	0.149	2.772
		W4	3.802	3.567	0.157	3.723	3.580	0.156	3.737
		W6	5.793	5.559	0.173	5.733	5.576	0.172	5.748
	OmniQuant	W3	2.801	2.529	0.149	2.678	2.545	0.149	2.694
		W4	3.744	3.555	0.157	3.712	3.571	0.157	3.728
		W6	5.810	5.602	0.173	5.775	5.621	0.172	5.793
	MXFP4	W4	3.922	3.625	0.250	3.875	3.632	0.250	3.882
Qwen3-14B	AWQ	W3	2.888	2.610	0.149	2.759	2.622	0.148	2.771
		W4	3.805	3.565	0.156	3.722	3.578	0.156	3.735
		W6	5.797	5.560	0.173	5.733	5.575	0.172	5.747
	OmniQuant	W3	2.801	2.515	0.149	2.664	2.530	0.149	2.679
		W4	3.750	3.545	0.157	3.702	3.560	0.156	3.717
		W6	5.818	5.592	0.173	5.765	5.610	0.172	5.782
	MXFP4	W4	3.947	3.644	0.250	3.894	3.650	0.250	3.900
Llama-2-7B	AWQ	W3	2.910	2.640	0.149	2.789	2.657	0.149	2.806
		W4	3.852	3.648	0.157	3.805	3.664	0.156	3.821
		W6	5.850	5.616	0.173	5.789	5.635	0.172	5.807
	OmniQuant	W3	2.816	2.543	0.149	2.691	2.556	0.149	2.705
		W4	3.827	3.613	0.157	3.770	3.628	0.156	3.784
		W6	5.884	5.676	0.173	5.849	5.697	0.172	5.869
	MXFP4	W4	4.083	3.750	0.250	4.000	3.755	0.250	4.005
Average Size Reduction			↓ 7.68%	↓ **9.67%**			↓ **9.34%**		

module at a time into a reusable buffer in the engine's native weight layout (Marlin (Frantar et al., 2024) in vLLM and SGLang), then multiplied by its native kernel. Neither path keeps all model weights resident (Appendix C.3).

4 Experiments

We evaluate `Q3.6` first as a post-quantization storage layer and then as a serving path. Section 4.1 describes the evaluated objects and the accounting, and Table 1 gives the storage results. For the primary codecs, every reported rate charges the native metadata, route metadata, and coding tables needed for recovery and is compared against byte-stream compressors on the same payloads. The negative controls of Section 4.3 are coding-length diagnostics (Appendix B.8).

4.1 Experimental Setup

Models and quantized formats. We evaluate five quantized formats: AWQ, OmniQuant, and MXFP4 in the main tables, and GGUF Q4_K and Any4 as appendix controls. The primary evaluations use Qwen3-8B, Qwen3-14B (Yang et al., 2025), and Llama-2-7B (Touvron et al., 2023) under AWQ and OmniQuant at W3/W4/W6 and under MXFP4 at W4, over the MLP gate/up/down and attention q/k/v/o payloads. GGUF Q4_K (Qwen3-8B) and Any4 (OPT-125M) controls, and a cold-storage control with the MXFP4 checkpoint of gpt-oss-20b, appear in Appendix B.

Bucket configuration. Storage tables pick the bucket count per family by a sweep (Appendix B.3); the decompression benchmarks use 16 buckets per MLP family.

Table 2: Negative-control ablation in total bpw. Both controls retain the coded-stream budget; positive deltas indicate worse compression than **Q3.6**-FSE.

Case	**Q3.6**-FSE	Shuffled routes	Random buckets	Δ shuffled	Δ random
Qwen3-8B / AWQ / W3	2.7553	2.7899	2.7899	+0.0346	+0.0346
Qwen3-8B / OmniQuant / W6	5.7749	5.8117	5.8117	+0.0367	+0.0368
Llama-2-7B / AWQ / W4	3.8052	3.8130	3.8130	+0.0078	+0.0078

Storage accounting. Storage is reported in bits per weight (bpw). The zstd baseline is $\mathrm{zstd}(\mathrm{payload}) + \mathrm{raw}(\mathrm{metadata})$ and **Q3.6** is $\mathrm{codec}(\mathrm{conditioned\ payload}) + \mathrm{raw}(\mathrm{metadata} + \mathrm{route\ metadata})$, so "others" is raw metadata plus route metadata and native metadata is never omitted. Appendix B.4 gives a cold-storage control in which **Q3.6** also compresses the metadata.

Decompression backends. We report a CPU backend based on FSE (Collet, 2013) and a GPU backend based on dietGPU (Johnson, 2023). Throughput is normalized to one byte per recovered weight (GB/s equals billions of weights per second; Appendix B).

4.2 Main Compression Results

Both target-tensor groups improve over zstd in every evaluated setting. On the two public AWQ W4 checkpoints, rerunning the released storage scripts reproduces the corresponding rows within 0.001 bpw, with bit-exact round trips. On the MLP gate/up/down payloads (Table 1) the mean total-rate reduction is 2.17% for **Q3.6**-FSE and 1.81% for **Q3.6**-dietGPU, and on the attention q/k/v/o payloads (Appendix B.2) it is 3.10% and 2.79%, so the effect is not specific to MLP projections. Relative to the uncompressed target tensors, the average reduction grows from 7.68% (zstd) to 9.67% and 9.34% on MLP payloads, and from 7.21% to 10.07% and 9.78% on attention payloads. The setting is demanding: quantization has already removed most numerical redundancy, and zstd receives byte-aligned qvalue streams for W3 and W6 and beats ZIP/DEFLATE (Appendix B). W3 gains most; route metadata stays below 0.001 bpw. ZipNN, DeepCABAC, and PNG-4 store more on Qwen3-8B/AWQ/W4 (4.124, 3.913, and 4.020 bpw against 3.723; Appendix B.7).

4.3 Attribution Controls

Two negative controls on three representative MLP cases isolate the routing mechanism (Table 2). Both keep the number of streams and the accounting of the coded streams but break the metadata-to-payload correspondence: shuffled routes keep the native route-code distribution but permute the codes within each tensor family, and random buckets ignore the metadata at equal stream budget.

Compression worsens under both controls, so extra streams alone do not explain the gain. The benefit comes from the matched pairing of conditioning information and payload symbols. The two columns agree to four decimals because in both each bucket receives a random subset of groups. Across all 21 MLP configurations, **Q3.6**-FSE stores less than zstd. Metadata-conditioned routing contributes 0.034 bpw on average, about 45% of the gain over zstd, and reverses all three cases in which a single unconditioned tANS table is worse than zstd (Appendix B.8). The bucket-count sweep in Figure 4 improves until metadata growth dominates, as Eq. (1) predicts, and bucketized conditioning nearly matches exact-scale conditioning (Appendix B.3).

4.4 Decompression Throughput

Exact decompression is fast on the GPU: across the 21 MLP cases of Appendix Table 15 the public dietGPU decoder averages 388 GB/s, and the Qwen3-8B/AWQ/W4 payload takes 14 ms.

Table 3 compares the two **Q3.6** decoders of Section 3.4 with the strongest configuration of every nvCOMP GPU codec (NVIDIA Corporation, 2026) on the same payloads under one normalization. **Q3.6** is the smallest object at every bit width, and with our decoder it is also the fastest, 1.9 to 2.5 times as fast as the fastest GDeflate configuration and 2.1 to 4.2 times as fast as its smallest-output configuration; nvCOMP-LZ4 comes within 0.4% at W3 with an object 12% larger and does not

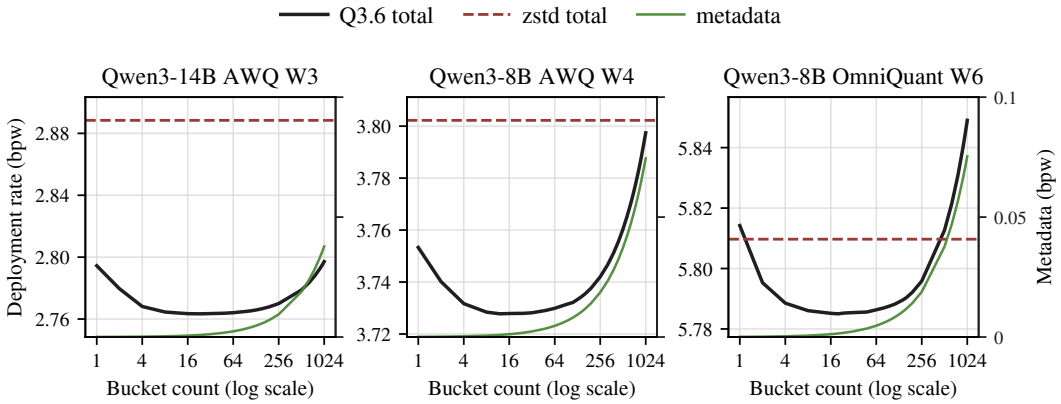

Figure 4: Bucket-count sensitivity. Metadata includes routing/count metadata and prebuilt FSE decoding tables under deployment-overhead ablation accounting.

Table 3: GPU decoding of Qwen3-8B MLP payloads at W3, W4, and W6 on one RTX PRO 6000 Blackwell. The payloads come from a fresh AutoAWQ (Hansen, 2024) run, so their rates differ slightly from Table 1; total bpw is the coded bytes, including the coding tables, plus native and route metadata. Each nvCOMP codec is at its smallest-output configuration over input representation and compression level (GDeflate also at its fastest); throughput is device to device (setup in Appendix C).

	W3		W4		W6	
Codec	total bpw	GB/s	total bpw	GB/s	total bpw	GB/s
nvCOMP-LZ4	3.127	766.5	4.168	563.3	6.177	382.8
nvCOMP-GDeflate, smallest	2.974	182.4	3.834	308.3	5.874	139.8
nvCOMP-GDeflate, fastest	3.060	402.7	3.834	308.3	6.110	209.3
nvCOMP-Deflate	2.951	22.5	3.822	39.3	5.851	19.3
nvCOMP-Zstd	3.026	116.2	3.821	86.4	5.849	46.7
`Q3.6`, public dietGPU decoder	**2.793**	357.9	**3.744**	308.1	**5.786**	223.4
`Q3.6`, our decoder	**2.793**	**769.5**	**3.744**	**650.6**	**5.786**	**522.9**

compress at W4 (4.168 bpw against a raw total of 4.156). Our decoder runs 2.1–2.3 times as fast as the public one on the same streams.

4.5 Serving from Storage

Setup. We serve Llama-2-7B-AWQ with vLLM 0.19 (Kwon et al., 2023), SGLang 0.5.8 (Zheng et al., 2024), and FlexGen (Sheng et al., 2023), and Qwen3-8B-AWQ with vLLM. The engines are unmodified (`Q3.6` is an added quantization backend); raw uses the AWQ-Marlin kernels of vLLM and SGLang, and FlexGen's Llama executor computes in fp16. In every system the quantized weights of each decoder layer are read from storage on every forward pass, so both configurations use storage offloading, including FlexGen's SSD-offloading setting. Embeddings, the output projection, and the norms stay on the GPU. Under the same transport and request settings, *raw* reads the engine's own quantized weights in the layout its kernels expect (the fewest bytes an uncompressed configuration can move), whereas `Q3.6` reads the compressed files and decodes them inside the matrix multiply (Section 3.4). Requests follow FlexGen's main setting, 16 requests per batch with 512 input and 32 output tokens each; TTFT is the time to the first output token of a batch and TPOT the average time per later output token (Appendix C).

Bandwidth condition. On a host with one RTX PRO 6000 Blackwell (96 GB), storage reads are limited to 1.5 GB/s, the rate of commodity SSDs and cloud block storage, by a kernel-level bandwidth cap (Linux cgroup v2 `io.max`) that applies to both configurations (Appendix C). At this rate reading one layer (97 MB) takes 65 ms, whereas decoding and multiplying it takes 5 to 7 ms for a 16-token batch and about 20 ms for a 2048-token prefill (Appendix C.3).

Table 4: Storage-offloaded serving at 1.5 GB/s, **Q3.6** versus raw, AWQ W4 models (batch 16, three interleaved repeats, medians). GPU memory reduction is the raw peak minus the **Q3.6** peak in the same run. Tokens equal the stock engine's in every repeat.

	TTFT (s)			TPOT (s)			GPU memory
System, model	raw	**Q3.6**	change	raw	**Q3.6**	change	reduction (GB)
vLLM, Llama-2-7B	2.250	2.107	−6.4%	2.242	2.058	−8.2%	3.19
SGLang, Llama-2-7B	6.702	6.201	−7.5%	2.242	2.065	−7.9%	3.19
FlexGen, Llama-2-7B	2.308	2.118	−8.3%	2.229	2.078	−6.8%	0.68
vLLM, Qwen3-8B	2.431	2.132	−12.3%	2.407	2.173	−9.7%	3.36

Table 5: Equal-budget vLLM/Llama-2-7B-AWQ capacity and throughput at 1.5 GB/s. KV cache is in tokens and 544-token requests; throughput uses each backend's largest single-wave batch.

	KV cache at start-up					Largest single-wave batch				
	raw		**Q3.6**			raw		**Q3.6**		
GPU budget	tokens	requests	tokens	requests	change	requests	tokens/s	requests	tokens/s	change
8 GB	6,992	12	13,264	24	+90%	12	5.35	23	11.16	2.09×
12 GB	15,152	27	21,440	39	+41%	26	11.60	35	16.99	1.46×
16 GB	23,136	42	29,408	54	+27%	38	16.96	47	22.82	1.35×
24 GB	39,472	72	45,744	84	+16%	63	28.14	73	35.45	1.26×

Results. Across vLLM, SGLang, and FlexGen, **Q3.6** reduces TTFT by 6.4–12.3% and TPOT by 6.8–9.7%, while freeing 0.68–3.36 GB of GPU memory (Table 4). In vLLM and SGLang, the per-token time ratios match the streamed-byte ratios (1.088 for Llama-2-7B, 1.107 for Qwen3-8B) within 0.3%: the smaller transfers reduce latency while decoding is hidden in the fused matrix multiply. Both paths stream the same layers under the same bandwidth cap, isolating the gain from entropy-coded transfers. SGLang's larger absolute TTFT reflects multiple prefill batches for the 16 requests. For Llama-2-7B, the 1.088× per-token speedup ratio also holds at 3 and 6 GB/s, corresponding to PCIe 3 and PCIe 4 NVMe drives (Appendix C).

4.6 Memory Savings for Greater Serving Capacity

vLLM allocates KV cache from the memory left after loading weights. For Llama-2-7B-AWQ, raw retains 3.24 GB of packed qweight; **Q3.6** removes that buffer while retaining scales and zeros and using a 97 MB working set (Appendix Table 18). Table 5 compares KV capacity, batch size, and throughput at equal 8–24 GB GPU budgets (Appendix C).

Without full-model weight buffers, **Q3.6** provides 16–90% more KV cache and 1.26–2.09× vLLM throughput across the four budgets (Table 5). Freed weight buffers admit larger batches while the codec shortens transfers; measured throughput follows the batch ratio times the 1.088 streamed-byte ratio. The effect is greatest at 8 GB, where nearly twice as many requests fit in one scheduler wave. At this budget, **Q3.6** serves 14 Llama-2-13B-AWQ requests where the native backend does not start (Appendix C). FlexGen's raw path already streams layers through a two-layer fp16 buffer, yet **Q3.6** serves 1–2 more requests and delivers 1.12–1.23× throughput (Appendix C.7).

5 Conclusion

Q3.6 uses native quantization metadata as entropy context to compress quantized LLM weights losslessly. It stores less than zstd in every evaluated setting after charging native metadata, route descriptors, and the coding tables required for exact recovery. Metadata conditioning contributes about 45% of the average improvement across 21 MLP configurations. In the three cases where a single unconditioned tANS table loses to zstd, metadata routing reverses the result. Its decoder runs 2.1–2.3 times as fast as the public dietGPU decoder, and its fused GPU path decodes weight tiles into shared memory without retaining full-model weights. Smaller streamed weights reduce time per output token by 7–10%, while freeing weight-buffer memory for more KV cache and up to 2.1× vLLM throughput at equal GPU budgets.

AI USE STATEMENT

In this work, generative AI tools were used only to help polish and revise the wording of the manuscript. The authors reviewed the edits and take responsibility for the final text.

ETHICS STATEMENT

Q3.6 changes only the storage and transport of already-quantized weights; it introduces no new model capabilities, training data, or generated content. Smaller and faster-to-load quantized models make existing models easier to copy and redistribute, including models that may be misused; the released artifact therefore redistributes no external model weights and requires users to obtain models and libraries under their original licenses. Appendix E discusses these considerations further.

REPRODUCIBILITY STATEMENT

The storage accounting is defined in Section 4, the route function and bucketization in Appendix A.1, the decompression protocol and hardware in Appendix B, and the serving hardware, the storage-bandwidth condition, the kernel costs, and the memory accounting in Appendix C. The anonymous code repository linked in the abstract contains the codec, the CUDA kernels, the vLLM, SGLang, and FlexGen integrations, and the artifact pipeline. Its experiment drivers and result files cover the public AWQ W4 storage rows, W3/W4/W6 GPU decoding, serving and capacity runs, and the listed operator and round-trip tests. The broader storage tables and appendix controls use the same codec and quantization pipeline, but are not each backed by a dedicated one-command driver in the repository.

REFERENCES

Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. In *International Conference on Learning Representations*, 2018.

Yann Collet. FiniteStateEntropy: New generation entropy coders. https://github.com/Cyan4973/FiniteStateEntropy, 2013. Accessed: 2026-05-04.

Yann Collet and Murray Kucherawy. Zstandard compression and the application/zstd media type. RFC 8878, 2021. URL https://www.rfc-editor.org/rfc/rfc8878.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022. URL https://arxiv.org/abs/2208.07339.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2305.14314.

Peter Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, 1996. URL https://www.rfc-editor.org/rfc/rfc1951.

Jarek Duda. Asymmetric numeral systems: Entropy coding combining speed of huffman coding with compression rate of arithmetic coding, 2013. URL https://arxiv.org/abs/1311.2540.

Mostafa Elhoushi and Jeff Johnson. any4: Learned 4-bit numeric representation for LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 15215–15231. PMLR, 2025. URL https://proceedings.mlr.press/v267/elhoushi25a.html.

Ruibo Fan, Xiangrui Yu, Xinglin Pan, Zeyu Li, Weile Luo, Qiang Wang, Wei Wang, and Xiaowen Chu. ZipServ: Fast and memory-efficient LLM inference with hardware-aware lossless compression. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2026. URL https://arxiv.org/abs/2603.17435.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.17323.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: Mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*, 2024.

Georgi Gerganov and contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. Accessed: 2026-05-04.

Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Yang Yong, Shiqiao Gu, Haotong Qin, Jinyang Guo, Dahua Lin, Michele Magno, and Xianglong Liu. A survey of low-bit large language models: Basics, systems, and algorithms. *Neural Networks*, 192:107856, 2025. doi: 10.1016/j.neunet.2025.107856. URL https://arxiv.org/abs/2409.16694.

Casper Hansen. AutoAWQ: An easy-to-use package for 4-bit quantized models. https://github.com/casper-hansen/AutoAWQ, 2024. Version 0.2.9.

Anat Heilper and Doron Singer. Lossless compression of neural network components: Weights, checkpoints, and K/V caches in low-precision formats. *arXiv preprint arXiv:2508.19263*, 2025.

Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, and Danny Harnik. ZipNN: Lossless compression for AI models, 2024. URL https://arxiv.org/abs/2411.05239.

David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

Jeff Johnson. DietGPU: GPU-based lossless compression for numerical data. Meta Research, https://github.com/facebookresearch/dietgpu, 2023. Accessed: 2026-05-04.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, 2024. URL https://arxiv.org/abs/2306.00978.

Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, Lin Xiao, Yuandong Tian, Bilge Soran, Raghuraman Krishnamoorthi, Tijmen Blankevoort, and Vikas Chandra. ParetoQ: Improving scaling laws in extremely low-bit LLM quantization. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.02631.

Ivan Nikulin. Unweight: Lossless MLP weight compression for LLM inference. Cloudflare Technical Report Cf-TR-2026.04.v1, Cloudflare, 2026. URL https://research.cloudflare.com/nikulin2026/.

NVIDIA Corporation. TensorRT-LLM: Numerical precision. https://nvidia.github.io/TensorRT-LLM/reference/precision.html, 2025. Accessed: 2026-05-01.

NVIDIA Corporation. nvCOMP: GPU data compression. https://developer.nvidia.com/nvcomp, 2026. Version 5.3.0; accessed: 2026-05-06.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. Microscaling data formats for deep learning, 2023. URL https://arxiv.org/abs/2310.10537.

Arnab Sanyal, Gourav Datta, Prithwish Mukherjee, Sandeep P. Chinchali, and Michael Orshansky. EntroLLM: Entropy encoded weight compression for efficient large language model inference on edge devices. *arXiv preprint arXiv:2505.02380*, 2025.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. OmniQuant: Omnidirectionally calibrated quantization for large language models. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2308.13137.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the International Conference on Machine Learning*, 2023.

Hongshi Tan, Yao Chen, Gustavo Alonso, Weng-Fai Wong, and Bingsheng He. Approaching Shannon bound with lossless LLM weight compression. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2026. arXiv:2606.15789.

Hugo Touvron et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

W3C. Portable network graphics (PNG) specification (second edition). W3C Recommendation, 2003.

Simon Wiedemann, Heiner Kirchhoffer, Stefan Matlage, Paul Haase, Arturo Marban, Talmaj Marinč, David Neumann, Tung Nguyen, Heiko Schwarz, Thomas Wiegand, Detlev Marpe, and Wojciech Samek. DeepCABAC: A universal compression algorithm for deep neural networks. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):700–714, 2020.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the International Conference on Machine Learning*, 2023. URL https://arxiv.org/abs/2211.10438.

An Yang et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Patrick Yubeaton, Tareq Mahmoud, Shehab Naga, Pooria Taheri, Tianhua Xia, Arun George, Yasmein Khalil, Sai Qian Zhang, Siddharth Joshi, Chinmay Hegde, and Siddharth Garg. Huff-LLM: End-to-end lossless compression for efficient LLM inference, 2025. URL https://arxiv.org/abs/2502.00922.

Tianyi Zhang, Mohsen Hariri, Shaochen Zhong, Vipin Chaudhary, Yang Sui, Xia Hu, and Anshumali Shrivastava. 70% size, 100% accuracy: Lossless LLM compression for efficient GPU inference via dynamic-length float (DFloat11). In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2504.11651.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

A Route Details and Scale-Conditioned Rate Analysis

A.1 Route Function and Bucketization Details

This subsection gives the implementation details for the deterministic router summarized in Section 3.3. For each payload group or block i, let $f_i \in \mathcal{F}$ denote its tensor family within the evaluated projection set, e.g., gate/up/down for MLP projections or q/k/v/o for attention projections, and let $\mathbf{s}_i$ denote its native quantization metadata. The route function used in our experiments maps this metadata to a discrete scale code

$$c_i = g(\mathbf{s}_i) = \begin{cases} \mathrm{bits}_{16}(\alpha_i), & \text{AWQ/OmniQuant scale } \alpha_i, \\ \mathrm{uint8}(\sigma_i), & \text{MXFP4 scale code } \sigma_i, \end{cases} \tag{2}$$

where bits_{16} denotes the unsigned 16-bit stored code of the native scale, using the bit pattern for 16-bit floating-point scales and the stored integer code for integer scale representations. For low-bit AWQ and OmniQuant payloads, zero points define the decoded-symbol alphabet, e.g., $q - z + \mathrm{offset}$, but are not used by this router.

Bucketization is performed separately for each tensor family. Let $n_{f,c}$ be the number of groups in family f with route code c, let $C_f = \{c : n_{f,c} > 0\}$ be the sorted active code set, and set $K_f = \min(K, |C_f|)$. We partition C_f into K_f contiguous intervals in code order. The initial internal split points are equal-mass quantiles of the route-code histogram:

$$\tau_{f,j} = \min \left\{ c \in C_f : \sum_{u \le c} n_{f,u} \ge \left\lceil \frac{jN_f}{K_f} \right\rceil \right\}, \qquad j = 1, \ldots, K_f - 1, \tag{3}$$

where $N_f = \sum_{c \in C_f} n_{f,c}$. Split points are adjusted only where needed to keep every interval non-empty. Given the stored split points, the bucket id is deterministic:

$$b_i = o_{f_i} + \mathrm{searchsorted}\big((\tau_{f_i,1}, \ldots, \tau_{f_i,K_{f_i}-1}), c_i; \mathrm{right}\big), \tag{4}$$

where o_f is the family-specific route offset. Thus buckets are contiguous scale-code ranges within each tensor family, not learned clusters.

For the storage results, **`Q3.6`** further refines the initialized split points by a bounded local search over adjacent split positions. The objective is the empirical payload code length within the induced buckets,

$$\sum_b \left[n_b \log_2 n_b - \sum_y n_{b,y} \log_2 n_{b,y} \right] = \sum_b n_b H(\widehat{p}_b), \tag{5}$$

where $n_{b,y}$ is the count of payload symbol y in bucket b. The final split points, together with the family ordering and bucket counts, are stored as route metadata. The decoder recomputes $c_i = g(\mathbf{s}_i)$ from the native quantization metadata and applies the stored split points, so correctness requires only that the stored route metadata reproduce the same bucket assignment.

A.2 Finite-Sample Rate Analysis

This subsection states and proves the rate facts used in Section 3. All statements are finite-sample: they hold for one fixed quantized payload and one fixed, deterministic bucket sequence, and they do not assume that the payload symbols are drawn from any probability model.

Setting and notation. Let A be the finite symbol alphabet of the payload and let $x^N = (x_1, \ldots, x_N) \in A^N$ be the payload symbols of one tensor family. Let $s^N = (s_1, \ldots, s_N)$ be the native metadata attached to the symbols (the scale code of the group containing each symbol), and let $b^N = (b_1, \ldots, b_N)$ with $b_i = \phi(s_i) \in \{1, \ldots, K\}$ be the bucket sequence produced by the stored split points through Eq. (4) (ϕ is the map written $B = b(S)$ in Section 3.3). Define the counts

$$n_{k,a} = \big|\{i : b_i = k,\ x_i = a\}\big|, \qquad N_k = \sum_{a \in A} n_{k,a}, \qquad n_a = \sum_k n_{k,a}, \qquad N = \sum_{k,a} n_{k,a}, \tag{6}$$

and the empirical distributions $\hat{p}_a = n_a/N$, $\hat{\pi}_k = N_k/N$, $\hat{p}_{k,a} = n_{k,a}/N$, and, for $N_k > 0$, $\hat{p}_a^{(k)} = n_{k,a}/N_k$. Logarithms are to base 2 and $0 \log 0 = 0$. The empirical entropy, conditional entropy, and mutual information are

$$\hat{H}(X) = -\sum_a \hat{p}_a \log \hat{p}_a, \qquad \hat{H}(X \mid B) = \sum_{k:\, N_k>0} \hat{\pi}_k \Big(-\sum_a \hat{p}_a^{(k)} \log \hat{p}_a^{(k)} \Big), \tag{7}$$

$$\hat{I}(X;B) = \hat{H}(X) - \hat{H}(X \mid B). \tag{8}$$

For a probability vector p on A, the *ideal code* assigns $-\log p_a$ bits to symbol a; an ideal entropy coder attains this length, and Remark A.6 accounts for the redundancy of the ANS backends.

Lemma A.1 (Optimal single-model code length)**.** *For every probability vector p on A,*

$$-\sum_{a \in A} n_a \log p_a \;\geq\; N \hat{H}(X), \tag{9}$$

with equality if and only if $p_a = \hat{p}_a$ for every a with $n_a > 0$.

Proof. Let $A^+ = \{a : n_a > 0\}$. If $p_a = 0$ for some $a \in A^+$ the left-hand side is infinite and there is nothing to prove, so assume $p_a > 0$ on A^+. Then

$$-\sum_{a \in A^+} n_a \log p_a = N\Big(-\sum_{a \in A^+} \hat{p}_a \log \hat{p}_a \Big) + N \sum_{a \in A^+} \hat{p}_a \log \frac{\hat{p}_a}{p_a} = N\hat{H}(X) + N \sum_{a \in A^+} \hat{p}_a \log \frac{\hat{p}_a}{p_a}. \tag{10}$$

By the log-sum inequality,

$$\sum_{a \in A^+} \hat{p}_a \log \frac{\hat{p}_a}{p_a} \;\geq\; \Big(\sum_{a \in A^+} \hat{p}_a \Big) \log \frac{\sum_{a \in A^+} \hat{p}_a}{\sum_{a \in A^+} p_a} \;=\; \log \frac{1}{\sum_{a \in A^+} p_a} \;\geq\; 0, \tag{11}$$

since $\sum_{a \in A^+} \hat{p}_a = 1$ and $\sum_{a \in A^+} p_a \leq 1$. Equality requires equality in both steps, i.e., $\hat{p}_a/p_a$ constant on A^+ and $\sum_{a \in A^+} p_a = 1$, which together give $p_a = \hat{p}_a$ on A^+. □

Proposition A.2 (Conditioning never lengthens the ideal payload)**.** *Let*

$$L^\star_{\mathrm{global}} = \min_p \Big(-\sum_a n_a \log p_a \Big), \qquad L^\star_{\mathrm{cond}} = \min_{p^{(1)},\dots,p^{(K)}} \Big(-\sum_k \sum_a n_{k,a} \log p_a^{(k)} \Big) \tag{12}$$

be the shortest ideal code lengths with one model for the whole family and with one model per bucket, respectively, the minima being taken over probability vectors on A. Then

$$L^\star_{\mathrm{global}} = N\hat{H}(X), \qquad L^\star_{\mathrm{cond}} = N\hat{H}(X \mid B), \tag{13}$$

$$L^\star_{\mathrm{global}} - L^\star_{\mathrm{cond}} = N\hat{I}(X;B) = N\, D_{\mathrm{KL}}\big(\hat{p}_{X,B} \,\big\|\, \hat{p}_B \otimes \hat{p}_X\big) \;\geq\; 0, \tag{14}$$

where $\hat{p}_{X,B} = (\hat{p}_{k,a})_{k,a}$ and $\hat{p}_B \otimes \hat{p}_X = (\hat{\pi}_k \hat{p}_a)_{k,a}$. Equality holds if and only if $n_{k,a}/N = (N_k/N)(n_a/N)$ for all k and a, that is, if and only if symbol and bucket are empirically independent.

Proof. The first identity is Lemma A.1. For the second, the objective separates into one term per bucket; for a bucket with $N_k > 0$, Lemma A.1 applied to the counts $(n_{k,a})_{a \in A}$ gives the minimum $-\sum_a n_{k,a} \log(n_{k,a}/N_k) = N_k \hat{H}(X \mid B = k)$, attained at $p^{(k)} = \hat{p}^{(k)}$, while an empty bucket contributes zero. Summing over k gives $N\hat{H}(X \mid B)$. For the third, use $n_a = \sum_k n_{k,a}$ to write

$$\begin{aligned} N\hat{H}(X) - N\hat{H}(X \mid B) &= \sum_{k,a} n_{k,a} \Big(\log \frac{n_{k,a}}{N_k} - \log \frac{n_a}{N} \Big) \\ &= N \sum_{k,a} \hat{p}_{k,a} \log \frac{\hat{p}_{k,a}}{\hat{\pi}_k \hat{p}_a} = N\, D_{\mathrm{KL}}\big(\hat{p}_{X,B} \,\big\|\, \hat{p}_B \otimes \hat{p}_X\big), \end{aligned} \tag{15}$$

where the sums run over the pairs with $n_{k,a} > 0$. Both arguments of the divergence are probability vectors on $\{1,\dots,K\} \times A$, so the divergence is non-negative by the log-sum inequality, with equality if and only if $\hat{p}_{k,a} = \hat{\pi}_k \hat{p}_a$ for all (k,a). □

Proposition A.3 (Bucketizing exact metadata)**.** *Because every bucket is a function of the metadata, $b_i = \phi(s_i)$, the bucketized conditional entropy is never below the exact-metadata conditional entropy:*

$$\hat{H}(X \mid B) \;\geq\; \hat{H}(X \mid S), \qquad \hat{H}(X \mid B) - \hat{H}(X \mid S) = \hat{I}(X; S \mid B) \;\geq\; 0. \tag{16}$$

Proof. All quantities are entropies of the joint empirical distribution of the triples (x_i, s_i, b_i), so the chain rule and the non-negativity of conditional mutual information apply to them as to any distribution. Because b_i is a function of s_i, conditioning on (B, S) is the same as conditioning on S, hence $\hat{H}(X \mid S) = \hat{H}(X \mid B, S)$. The chain rule gives $\hat{H}(X \mid B) - \hat{H}(X \mid B, S) = \hat{I}(X; S \mid B)$, which is a $\hat{\pi}$-weighted average of divergences of the form in Proposition A.2 and is therefore non-negative. □

Proposition A.3 is the reason the bucketized payload rate of Appendix B.3 lies above the exact-scale rate $\hat{H}(X \mid S)$. The observed payload gap is 0.00013 bpw, and 0.00255 bpw once route and model metadata are charged, while the bucketized code uses 2112 streams instead of one stream per active scale value (5968), which keeps the number of coding tables that a decoder must hold small.

Corollary A.4 (Rate condition with overhead)**.** *Let $M_{\mathrm{meta}}(B)$ be the number of bits of route metadata needed to recover b^N from the stored native metadata, let M_{model} be the number of bits of the serialized coding tables, and write $R_{\mathrm{meta}}(B) = M_{\mathrm{meta}}(B)/N$ and $R_{\mathrm{model}} = M_{\mathrm{model}}/N$. In the reported serialized-storage accounting the native quantization metadata is charged to both the baseline and* **Q3.6***, so $M_{\mathrm{meta}}(B)$ counts only the incremental route metadata. The bucket-conditioned length*

$$L_{\texttt{Q3.6}} = N\hat{H}(X \mid B) + M_{\mathrm{meta}}(B) + M_{\mathrm{model}} \tag{17}$$

is smaller than a baseline of rate R_{base} (length NR_{base}) if and only if

$$\hat{H}(X \mid B) + R_{\mathrm{meta}}(B) + R_{\mathrm{model}} < R_{\mathrm{base}}, \tag{18}$$

which is Eq. (1)*. When the baseline is the best single-model ideal code, $R_{\mathrm{base}} = \hat{H}(X)$, the condition is equivalent to*

$$\hat{I}(X; B) > R_{\mathrm{meta}}(B) + R_{\mathrm{model}}. \tag{19}$$

Proof. Divide both lengths by N; the equivalent form follows from $\hat{H}(X) - \hat{H}(X \mid B) = \hat{I}(X; B)$ (Proposition A.2). □

Proposition A.5 (Split refinement targets the conditioned length)**.** *For a fixed number of buckets, the objective of Eq.* (5) *equals $L^{\star}_{\mathrm{cond}}$. Hence the local search over split points acts directly on the ideal conditioned payload length (it returns the best split configuration it evaluates), and it leaves the number of coding tables, and with it the bucket count that enters M_{model}, unchanged.*

Proof. For each bucket b, with $n_b = \sum_y n_{b,y}$,

$$\begin{aligned} n_b \log n_b - \sum_y n_{b,y} \log n_{b,y} &= -\sum_y n_{b,y}\big(\log n_{b,y} - \log n_b\big) \\ &= -\sum_y n_{b,y} \log \frac{n_{b,y}}{n_b} = n_b \hat{H}(X \mid B = b). \end{aligned} \tag{20}$$

Summing over buckets gives $N\hat{H}(X \mid B) = L^{\star}_{\mathrm{cond}}$ by Proposition A.2. The search moves split points between adjacent non-empty intervals and never merges or creates intervals, so the number of buckets is fixed. □

Remark A.6 (Redundancy of the ANS backends). An ANS coder with t-bit probability precision (2^t states in tANS) encodes bucket k with a quantized probability vector $\tilde{p}^{(k)}$ whose entries are multiples of 2^{-t} rather than with $\hat{p}^{(k)}$. Up to the small state-dependent deviation of ANS from an ideal code, its length is $N_k \hat{H}(X \mid B = k) + N_k\, D_{\mathrm{KL}}\big(\hat{p}^{(k)} \,\|\, \tilde{p}^{(k)}\big)$ plus a bounded per-stream cost for state initialization and flushing. These terms are part of the measured payload column of every table, which counts the coded bytes actually produced, so the reported rates are realized lengths, not the ideal lengths of this subsection.

A.3 SCALE-CONDITIONED CODE CONCENTRATION

This subsection formalizes the two low-entropy mechanisms named in Section 3.2. Let Z be a zero-point-centered local weight and $S > 0$ the group scale stored as native quantization metadata. The quantizer forms the normalized ratio $U = Z/S$ and the code

$$D = \mathrm{clip}_{[-Q,Q]}\big(\mathrm{round}(U)\big) \in \{-Q, \dots, Q\}, \tag{21}$$

an alphabet $\mathcal{A}$ of $2Q+1$ values. The relevant object is the conditional distribution of U within a scale band S_b, i.e., over the groups whose scale satisfies $S \in S_b$. Writing $p_a(S_b) = P(D = a \mid S \in S_b)$, round-to-nearest followed by clipping gives

$$p_a(S_b) = \begin{cases} P\big(U < -Q + \frac{1}{2} \;\big|\; S \in S_b\big), & a = -Q, \\ P\big(a - \frac{1}{2} \le U < a + \frac{1}{2} \;\big|\; S \in S_b\big), & -Q < a < Q, \\ P\big(U \ge Q - \frac{1}{2} \;\big|\; S \in S_b\big), & a = Q, \end{cases} \tag{22}$$

and we write $H_b := H(D \mid S \in S_b) = -\sum_{a=-Q}^{Q} p_a(S_b) \log p_a(S_b)$ for the code entropy within the band.

Lemma A.7 (Grouping bound)**.** *Let D take values in a finite set $\mathcal{A}$, let $E \subseteq \mathcal{A}$ be non-empty with non-empty complement, and let $\delta = P(D \in E)$. Then*

$$H(D) \;\le\; H_2(\delta) + \delta \log |E| + (1-\delta) \log |\mathcal{A} \setminus E|, \qquad H_2(\delta) = -\delta \log \delta - (1-\delta) \log(1-\delta). \tag{23}$$

Proof. Let $J = \mathbf{1}\{D \in E\}$. Since J is a function of D, $H(D) = H(D, J) = H(J) + H(D \mid J)$. Here $H(J) = H_2(\delta)$, and

$$H(D \mid J) = \delta\, H(D \mid J = 1) + (1 - \delta)\, H(D \mid J = 0) \;\le\; \delta \log |E| + (1 - \delta) \log |\mathcal{A} \setminus E|, \tag{24}$$

because given $J = 1$ the code lies in E, given $J = 0$ it lies in $\mathcal{A} \setminus E$, and the entropy of a distribution on a finite set is at most the logarithm of the size of the set. □

Proposition A.8 (Two low-entropy regimes)**.** *Fix a scale band S_b.*

- *(i)* Central concentration. *Let $\epsilon = P(D \neq 0 \mid S \in S_b) = P\big(U < -\frac{1}{2}$ or $U \ge \frac{1}{2} \;\big|\; S \in S_b\big)$. Then $H_b \le H_2(\epsilon) + \epsilon \log(2Q)$.*
- *(ii)* Boundary concentration. *Let $\eta = P(|D| < Q \mid S \in S_b) = P\big(-Q + \frac{1}{2} \le U < Q - \frac{1}{2} \;\big|\; S \in S_b\big)$. Then $H_b \le H_2(\eta) + (1 - \eta) + \eta \log(2Q - 1)$.*

In particular $H_b \to 0$ as $\epsilon \to 0$ in case (i), and $\limsup H_b \le 1$ bit as $\eta \to 0$ in case (ii), the remaining bit being the sign of the clipped code.

Proof. Both statements are Lemma A.7 applied to the conditional distribution of D given $S \in S_b$. For (i) take $E = \{a \in \mathcal{A} : a \neq 0\}$, so $|E| = 2Q$ and $|\mathcal{A} \setminus E| = 1$; the term $(1 - \epsilon) \log 1$ vanishes. For (ii) take $E = \{a \in \mathcal{A} : |a| < Q\}$, so $|E| = 2Q - 1$ and $\mathcal{A} \setminus E = \{-Q, Q\}$, whose size has logarithm 1. The characterizations of ϵ and η in terms of U follow from the case formula for $p_a(S_b)$. The limits follow from $H_2(\delta) \to 0$ as $\delta \to 0$. □

Proposition A.8 explains why the dependence of bucket entropy on the scale need not be monotone. A band in which most normalized ratios satisfy $|U| < \frac{1}{2}$, i.e., scales that are large relative to the typical local weight as in outlier-dominated groups, has a code concentrated at 0; a band in which most ratios fall outside the unclipped interior, i.e., scales that are small relative to the local weights so that clipping dominates, has a code concentrated on $\{-Q, Q\}$. Both are low-entropy regimes, and which one appears depends on the conditional distribution of Z/S within the band. In Figure 2 the buckets at the two ends of the scale-code range have the lowest entropies, consistent with these two mechanisms. Whether a band is rate-useful is then decided by Corollary A.4: scale-derived contexts pay only when the empirical mutual information they induce exceeds the route and model overhead.

Table 6: Additional quantized formats under the reported serialized-storage accounting used in Section 4.

Format	Model	Backend	zstd total	**Q3.6** total	Delta
GGUF Q4_K	Qwen3-8B-Q4_K_M	FSE	4.342190	4.325744	-0.016446
Any4	OPT-125M	dietGPU	4.338551	4.315155	-0.023396

B Additional Experimental Results

This appendix collects the auxiliary storage controls and the full decompression benchmark behind Section 4. Storage is reported in bits per weight (bpw). In the decompression benchmark, p95 is the empirical 95th-percentile latency; GB/s denotes mean throughput normalized to one byte per recovered weight (billions of weights per second) across all backends and formats.

Hardware and software. The decompression benchmark ran on a dual-socket Intel Xeon Platinum 8470Q host with Python 3.12.3 on Linux 5.15; GPU measurements use one NVIDIA RTX PRO 6000 Blackwell. The zstd baseline uses level 19. The GPU decompression benchmark uses a fixed 16x3 bucket configuration (16 buckets for each of the three MLP tensor families; configurations are written as buckets per family times number of families), table log 12, three warm-up runs, and 30 timed runs. All reported rows pass exact payload roundtrip checks for the corresponding backend. Storage measurements are deterministic for fixed inputs and accounting rules.

Quantized objects. Public quantized checkpoints do not exist for most of the evaluated model, format, and bit-width combinations, so the quantized objects of Tables 1 and 7 are produced by us from the public bf16 checkpoints of Qwen3-8B, Qwen3-14B, and Llama-2-7B with the released pipeline. AWQ objects use AutoAWQ (Hansen, 2024) (activation-aware scale search with clipping, 128 calibration sequences of 512 tokens, group size 128, asymmetric round-to-nearest with integer zero points and fp16 scales) at W3, W4, and W6. OmniQuant objects use the OmniQuant reference implementation (learnable weight clipping and equivalent transformation) at group size 128 with fp16 scales and b-bit zero points, so their native metadata is $(16 + b)/128$ bpw. MXFP4 objects convert the bf16 weights to E2M1 elements with one 8-bit E8M0 scale per block of 32, that is, 0.25 bpw of native metadata. The route code is the stored 16-bit scale pattern for AWQ and OmniQuant and the 8-bit scale code for MXFP4 (Appendix A.1); the Qwen3-8B-AWQ checkpoint used for serving is the public one, whose scales are stored in bfloat16. For the public Qwen3-8B-AWQ and TheBloke/Llama-2-7B-AWQ W4 checkpoints, rerunning the released storage scripts reproduces the corresponding Table 1 rows within 0.001 bpw, and all qweight, zero-point, and scale bits pass exact round-trip checks.

B.1 Additional Quantized Formats

Beyond the MLP and attention evaluations, Table 6 applies **Q3.6** to GGUF, the llama.cpp file format, with the Q4_K quantization of Qwen3-8B, and to Any4-style objects of OPT-125M, under the accounting of the main experiments: the zstd baseline compresses only the payload and keeps the native metadata raw, while **Q3.6** compresses the conditioned payload and charges raw metadata plus route metadata. The paper's aggregate storage claims rest on the AWQ, OmniQuant, and MXFP4 results of Tables 1 and 7.

B.2 Attention Target-Tensor Results

Table 7 gives the full attention q/k/v/o breakdown summarized in Section 4.2, under the accounting of Table 1: the zstd column is the baseline total rate, "others" is raw metadata plus route metadata, and the **Q3.6** payload includes the bucket streams and the serialized entropy models. As in Table 1, the final row is the arithmetic mean over the rows of the size reduction relative to each row's uncompressed payload-plus-native-metadata rate.

Table 7: Full component breakdown under the reported serialized-storage accounting for attention q/k/v/o target payloads.

Model	Quant. Method	Bit Width	zstd	Q3.6-FSE payload	Q3.6-FSE others	Q3.6-FSE total	Q3.6-dietGPU payload	Q3.6-dietGPU others	Q3.6-dietGPU total
Qwen3-8B	AWQ	W3	2.896	2.505	0.149	2.654	2.517	0.148	2.665
		W4	3.821	3.562	0.156	3.719	3.576	0.156	3.732
		W6	5.877	5.556	0.173	5.729	5.571	0.172	5.743
	OmniQuant	W3	2.861	2.552	0.149	2.701	2.565	0.148	2.713
		W4	3.820	3.601	0.156	3.757	3.614	0.156	3.771
		W6	5.870	5.657	0.173	5.830	5.676	0.172	5.848
	MXFP4	W4	3.955	3.639	0.250	3.889	3.648	0.250	3.898
Qwen3-14B	AWQ	W3	2.891	2.508	0.149	2.657	2.518	0.148	2.667
		W4	3.814	3.555	0.157	3.712	3.569	0.156	3.725
		W6	5.876	5.556	0.174	5.730	5.572	0.172	5.744
	OmniQuant	W3	2.859	2.550	0.149	2.698	2.562	0.148	2.710
		W4	3.818	3.601	0.157	3.757	3.614	0.156	3.771
		W6	5.870	5.658	0.174	5.832	5.677	0.172	5.849
	MXFP4	W4	3.959	3.650	0.250	3.900	3.659	0.250	3.909
Llama-2-7B	AWQ	W3	2.859	2.512	0.149	2.661	2.525	0.148	2.674
		W4	3.810	3.565	0.157	3.722	3.578	0.156	3.735
		W6	5.836	5.535	0.173	5.708	5.551	0.172	5.723
	OmniQuant	W3	2.814	2.530	0.149	2.678	2.541	0.148	2.690
		W4	3.819	3.602	0.157	3.759	3.615	0.156	3.771
		W6	5.882	5.668	0.174	5.842	5.685	0.172	5.857
	MXFP4	W4	4.047	3.757	0.250	4.007	3.756	0.250	4.006
Average Size Reduction			↓ 7.21%		↓ **10.07%**			↓ **9.78%**	

B.3 Scale-Conditioning Oracle Analysis

For Qwen3-8B/AWQ/W4 we compare the bucketized model with an oracle that conditions on the exact scale value of every group, under the reported serialized-storage accounting (route descriptors and symbol-probability tables charged, prebuilt FSE decoding tables not). The bucketized model nearly matches exact-scale conditioning: the payload-entropy gap is 0.00013 bpw, and the deployment-accounted gap is 0.00255 bpw, while the bucketized code uses 2112 bucket routes instead of 5968 exact scale codes, which keeps the number of coding tables a decoder must hold small. The bucket-count tradeoff in Figure 4 uses deployment-overhead ablation accounting, where metadata includes routing/count metadata and prebuilt FSE decoding tables; this accounting differs from the headline serialized-storage accounting of Table 1.

B.4 Cold-Storage Full-Object Control

The main paper keeps the native metadata raw because serving runtimes may need it directly addressable. Table 8 drops this requirement: the baseline compresses the whole quantized object as one byte stream with zstd, and **Q3.6** keeps its conditioned payload and compresses the native and deployment metadata with zstd as well. The rows cover an OmniQuant W4 checkpoint of Llama-2-7B, the released MXFP4 checkpoint of gpt-oss-20b (a 20B-parameter mixture-of-experts model), and the GGUF Q4_K checkpoint of Qwen3-8B; in all three, the cold-storage variant of **Q3.6** stays smaller than zstd of the original object.

Table 10: Auxiliary ZIP/DEFLATE control over the primary MLP gate/up/down payloads under the reported serialized-storage accounting. Delta is ZIP/DEFLATE total bpw minus zstd total bpw; positive values mean zstd is the stronger baseline. Reductions are relative to ZIP/DEFLATE.

Group	Rows	ZIP-zstd	FSE red.	dietGPU red.
All	21	0.171	6.63%	6.28%
W3	6	0.409	16.43%	15.98%
W4	9	0.034	2.37%	2.07%
W6	6	0.139	3.20%	2.91%

Table 8: Cold-storage full-object control under the reported cold-storage full-object accounting. The baseline is zstd(original quantized object); **`Q3.6`** uses dietGPU(payload) + zstd(native quantization metadata) + zstd(deployment metadata). All entries are bpw and pass exact roundtrip checks.

Case	zstd(original object)	**`Q3.6`**+zstd(metadata)	Delta
Llama-2-7B / OmniQuant / W4	3.7764	3.7278	-0.0487
gpt-oss-20b / MXFP4 / W4	3.9107	3.8748	-0.0359
Qwen3-8B / GGUF Q4_K	4.3544	4.2601	-0.0943

B.5 Baseline Representation Control

For W3 and W6, the main paper uses zstd on byte-aligned logical qvalue streams instead of zstd on the original bit-packed payload. This control checks that the choice does not weaken the baseline. Table 9 reports the difference

$$\mathrm{zstd}(\text{unpacked uint8 qvalues}) - \mathrm{zstd}(\text{bit-packed payload})$$

on payload bpw. Negative values mean the byte-aligned logical-qvalue baseline is stronger.

Table 9: W3/W6 zstd representation control under the reported payload-rate control. Delta is payload bpw for byte-aligned logical-qvalue zstd minus bit-packed-payload zstd.

Group	Rows	Mean delta	Min delta	Max delta
All W3/W6	12	-0.185	-0.242	-0.104
W3	6	-0.145	-0.207	-0.104
W6	6	-0.226	-0.242	-0.218

B.6 Auxiliary ZIP/DEFLATE Control

The legacy ZIP/DEFLATE control in Table 10 uses the same payload representation as the MLP gate/up/down evaluation: W3 and W6 use the byte-aligned logical qvalue stream, while W4 and MXFP4 use the native payload stream. Native metadata remains raw. zstd is stronger than ZIP/DEFLATE in every evaluated setting, supporting its use as the primary byte-stream baseline. The larger reductions against ZIP/DEFLATE show how much a byte-stream compressor leaves unused when it ignores the quantization metadata.

B.7 Lossless Coders for Neural Networks

Table 11 compares **`Q3.6`** with lossless coders for neural-network weights and with an image codec on the Qwen3-8B/AWQ/W4 MLP payloads, under the accounting of Table 1 (native metadata charged raw). ZipNN (Hershcovitch et al., 2024) separates and entropy-codes the bytes of floating-point weights; on the packed 4-bit payload most chunks take its pass-through path, and the payload shrinks by 0.8% (4.000 to 3.968 bpw). DeepCABAC (Wiedemann et al., 2020) codes quantized network weights with context-adaptive binary arithmetic coding, and PNG-4 stores the payload with the PNG image codec (W3C, 2003). **`Q3.6`** is the smallest, by 0.19 to 0.40 bpw. DFloat11, Huff-LLM, ZipServ, and Unweight target floating-point weights (Section 2).

Table 11: Lossless coders on the Qwen3-8B/AWQ/W4 MLP payloads, total bpw under the accounting of Table 1.

Method	total bpw
Uncompressed (payload and native metadata)	4.156
ZipNN	4.124
PNG-4	4.020
DeepCABAC	3.913
zstd	3.802
`Q3.6`-dietGPU	3.737
`Q3.6`-FSE	**3.723**

Table 12: Decomposition of the gain over zstd on the MLP gate/up/down payloads (total bpw). FSE 1x1 is a single unconditioned tANS table; backend gain is zstd minus FSE 1x1, and conditioning gain is FSE 1x1 minus `Q3.6`-FSE.

Model	Method	Bit	zstd	FSE 1x1	`Q3.6`-FSE	backend gain	conditioning gain
Qwen3-8B	AWQ	W3	2.880	2.790	2.755	0.090	0.035
	AWQ	W4	3.802	3.754	3.723	0.048	0.031
	AWQ	W6	5.793	5.775	5.733	0.018	0.042
	OmniQuant	W3	2.801	2.705	2.678	0.096	0.027
	OmniQuant	W4	3.744	3.742	3.712	0.002	0.030
	OmniQuant	W6	5.810	5.815	5.775	−0.005	0.040
	MXFP4	W4	3.922	3.942	3.875	−0.020	0.067
Qwen3-14B	AWQ	W3	2.888	2.795	2.759	0.093	0.036
	AWQ	W4	3.805	3.755	3.722	0.050	0.033
	AWQ	W6	5.797	5.773	5.733	0.024	0.040
	OmniQuant	W3	2.801	2.695	2.664	0.106	0.031
	OmniQuant	W4	3.750	3.739	3.702	0.011	0.037
	OmniQuant	W6	5.818	5.815	5.765	0.003	0.050
	MXFP4	W4	3.947	3.954	3.894	−0.007	0.060
Llama-2-7B	AWQ	W3	2.910	2.810	2.789	0.100	0.021
	AWQ	W4	3.852	3.814	3.805	0.038	0.009
	AWQ	W6	5.850	5.808	5.789	0.042	0.019
	OmniQuant	W3	2.816	2.701	2.691	0.115	0.010
	OmniQuant	W4	3.827	3.781	3.770	0.046	0.011
	OmniQuant	W6	5.884	5.871	5.849	0.013	0.022
	MXFP4	W4	4.083	4.064	4.000	0.019	0.064

B.8 Routing and Backend Controls

Two controls separate the routing from the entropy backend. Table 13 changes the backend at fixed routing: replacing zstd by a single unconditioned tANS table (1x1) already moves the total rate, in either direction, helping on Qwen3-14B/AWQ/W3 (2.888 to 2.795 bpw) and on Llama-2-7B/MXFP4/W4 (4.083 to 4.064 bpw) but slightly hurting on Qwen3-8B/OmniQuant/W6 (5.810 to 5.815 bpw); metadata-conditioned routing then lowers every row further, to 2.759, 4.000, and 5.775 bpw, and is the only component that improves all three cases. Table 12 extends the 1x1 control to all 21 MLP rows of Table 1: the backend gain is zstd minus the single unconditioned tANS table, and the conditioning gain is that table minus `Q3.6`-FSE. Conditioning gives 0.009 to 0.067 bpw, the larger share in 10 of the 21 rows, and the whole margin over zstd in the three rows where the unconditioned table is worse than zstd.

Table 14 keeps the routing and replaces the backend, compressing the same routed streams with zstd. The shuffled-route and random-bucket negative controls of the main text keep the stream and metadata budget while breaking the metadata-to-payload correspondence. They are coding-length diagnostics rather than decodable codecs: the permutation or assignment is not stored or charged, which only favors the controls.

Table 13: Ablation on metadata-conditioned bucketization. All entries are total bpw under the same reported serialized-storage accounting as the primary gate/up/down evaluation. Configurations are written as buckets per family times number of families: 1x1 is a single unconditioned tANS table shared by all groups, 1x3 is one table per MLP family without scale routing, and "best" uses the bucket count selected by the storage sweep with refined split points, as in the main tables.

Case	zstd	FSE 1x1	FSE 1x3	FSE best	dietGPU 1x1	dietGPU 1x3	dietGPU best
Qwen3-14B / AWQ / W3	2.888	2.795	2.794	2.759	2.780	2.780	2.771
Llama-2-7B / MXFP4 / W4	4.083	4.064	4.064	4.000	4.070	4.070	4.005
Qwen3-8B / OmniQuant / W6	5.810	5.815	5.814	5.775	5.797	5.797	5.793

Table 14: Backend control under the same metadata-conditioned routing. All entries are total bpw under the reported serialized-storage accounting. The bucketed-zstd control is charged raw native quantization metadata and route metadata; unlike `Q3.6`-FSE, it has no serialized entropy-model table.

Case	zstd	Bucketed zstd	`Q3.6`-FSE	Bucketed zstd − `Q3.6`
Qwen3-8B / AWQ / W3	2.8795	2.8541	2.7553	+0.0987
Llama-2-7B / AWQ / W4	3.8517	3.8355	3.8052	+0.0303

Table 15: GPU payload decompression benchmark over the MLP gate/up/down payloads with the public dietGPU decoder on one GPU. Times are milliseconds; GB/s is per recovered weight at one byte per weight.

Model	Method	Bit	mean ms	p95 ms	GB/s
Qwen3-8B	AWQ	W3	11.95	11.96	454.8
Qwen3-8B	AWQ	W4	14.15	14.16	384.2
Qwen3-8B	AWQ	W6	19.49	19.50	278.9
Qwen3-8B	OmniQuant	W3	11.74	11.75	463.1
Qwen3-8B	OmniQuant	W4	14.07	14.07	386.5
Qwen3-8B	OmniQuant	W6	19.57	19.58	277.7
Qwen3-8B	MXFP4	W4	11.91	11.93	456.4
Qwen3-14B	AWQ	W3	23.30	23.42	459.1
Qwen3-14B	AWQ	W4	27.65	27.67	386.8
Qwen3-14B	AWQ	W6	38.17	38.18	280.2
Qwen3-14B	OmniQuant	W3	22.43	22.45	476.8
Qwen3-14B	OmniQuant	W4	26.88	26.92	397.9
Qwen3-14B	OmniQuant	W6	37.91	38.32	282.1
Qwen3-14B	MXFP4	W4	22.73	22.74	470.5
Llama-2-7B	AWQ	W3	9.53	9.54	454.0
Llama-2-7B	AWQ	W4	11.38	11.39	380.4
Llama-2-7B	AWQ	W6	15.56	15.58	278.2
Llama-2-7B	OmniQuant	W3	9.34	9.35	463.3
Llama-2-7B	OmniQuant	W4	11.35	11.31	381.7
Llama-2-7B	OmniQuant	W6	15.67	15.68	276.2
Llama-2-7B	MXFP4	W4	9.42	9.44	459.5

B.9 Full Decompression Benchmark

Table 15 lists every case of the GPU decompression benchmark summarized in Section 4.4; it uses the public dietGPU decoder, and Table 3 reports the faster decoder.

C Serving Details

C.1 Hardware, Software, and the Bandwidth Condition

The serving experiments of Section 4.5 run on a cloud host with one NVIDIA RTX PRO 6000 Blackwell (96 GB) and network-attached block storage; vLLM 0.19.0 (torch 2.10, CUDA 12.8),

SGLang 0.5.8 (torch 2.9.1), FlexGen (the FlexLLMGen repository at upstream `main`) with the authors' Llama executor, and public dietGPU with 32 KB blocks. The read bandwidth of the serving process tree is fixed at 1.5 GB/s with the cgroup v2 I/O controller (`io.max`, `rbps`); every repeat records its read rate over its wall time, and it is 1.50 GB/s in every reported run. Transport is identical for both configurations: `O_DIRECT` reads (8 threads, 16 MB requests) into ten pinned slots with model-wide read-ahead, host-to-device or direct-from-pinned consumption up to three layers ahead on a separate CUDA stream, and CUDA events guarding slot reuse. The raw configuration streams the engine's post-load quantized weights from a layer-major file (Marlin `qweight`, zeros, and scales; in FlexGen the AWQ tensors, dequantized into the executor's fp16 pool), and the **`Q3.6`** configuration streams the **`Q3.6`** files (in FlexGen through a tensor wrapper, so the fp16 pool is never allocated). The first forward pass of the raw configuration compares every streamed tensor with the stock engine's; the first forward pass of the **`Q3.6`** configuration compares every module's output with Marlin's.

Request shape and GPU budgets. Every serving run uses 512 input and 32 output tokens, the main setting of FlexGen (Sheng et al., 2023), with identical prompts, greedy decoding, and a forced generation length; the engines admit the whole batch and run the prefill of a batch as one forward pass (SGLang splits it over several). TTFT is the wall time of a batch that generates one token (prefill plus the first token), and TPOT is the wall time of the 32-token batch minus TTFT, divided by 31. In Table 4 the SGLang TTFT is larger in absolute terms because SGLang admits the 16 requests over several prefill batches, and the FlexGen TPOT gain is smaller than vLLM's because FlexGen's dense fp16 compute is a larger share of its per-token time.

GPU budgets and the largest single-wave batch. The GPU budgets of Table 5 hand the engine the fractions 0.084, 0.126, 0.167, and 0.251 of the 96 GB card, the memory of 8, 12, 16, and 24 GB cards; the CUDA context, the weights, and the engine's profiled activation peak are charged inside the budget as they would be on such a card, and the remainder becomes the KV cache. "Requests" is the KV cache in tokens divided by the 544 tokens of one request. The largest single-wave batch of each backend is found by a fixed-point search: a two-token probe at batch b must finish in one scheduler wave (one prefill plus two decode passes) and $544b$ tokens must fit the KV cache that the engine's profiling leaves at that batch, since the prefill activation peak grows with b (which is why the largest single-wave batch is below the start-up request count); otherwise b is set to that KV cache divided by 544 and probed again. The throughput run at the accepted batch then reports output tokens per second over the wall time of the 32-token batch, three alternated repeats, medians; a run with more than 33 forward passes would indicate a second wave and is backed off by 5%.

Second host and other bandwidths. Table 5 was measured on a second host with the same GPU model (RTX PRO 6000 Blackwell Server Edition, Intel Xeon Platinum 8470Q, CUDA 12.8, vLLM 0.19). On this host the artifact is served from host memory through the same reader, with the aggregate read rate paced at 1.5 GB/s in the reader. Under this cap the batch-16 runs match Table 4 within 0.1% (2.241 s per token for the raw configuration and 2.060 s for **`Q3.6`**, at 1.50 GB/s), and the KV capacities agree with the first host's within one 16-token block. The same host served the artifact at 3 and 6 GB/s, the rates of PCIe 3 and PCIe 4 NVMe drives: the per-token times fall to 1.120 and 1.030 s and to 0.559 and 0.513 s (raw and **`Q3.6`**, batch 16), the ratio stays 1.088, and tokens remain identical; the byte ratio carries over unchanged to any storage tier on which the transfer dominates: the kernel consumes the stream from host memory at up to 11 GB/s, so any tier up to that rate stays transfer-bound.

The decoder comparison of Table 3 runs on a second machine of the same GPU model (RTX PRO 6000 Blackwell Server Edition, Intel Xeon Platinum 8470Q, CUDA 13.0, nvCOMP 5.3.0) with the artifact pipeline of the released code; its W3, W4, and W6 objects are produced with AutoAWQ 0.2.9 from the bf16 Qwen3-8B checkpoint (128 calibration sequences of 512 tokens, group size 128, asymmetric rounding with integer zero points and fp16 scales) and routed with 16 buckets per family. For Qwen3-8B-AWQ, whose checkpoint stores scales in bfloat16, the route codes are the bfloat16 bit patterns; the runtime converts the engine's fp16 scales back to those patterns when it generates the forward maps.

C.2 Decoder Details

Nsight Compute shows the public decoder to be latency-bound rather than bandwidth-bound: with 32 KB blocks and eight projections per launch it keeps 18.9 of 48 warps per SM active and has no eligible warp in 74% of cycles. Our decoder removes the stalls. Each lane keeps one 16-bit word of the backward-read stream in a three-chunk register window and serves every renormalization word through a single warp shuffle instead of a dependent global load; the table word is carried across iterations so that the next shared-memory lookup issues before the refill test; full blocks run with compile-time trip counts; and thread blocks are assigned to streams in proportion to their number of compressed blocks (an exclusive prefix sum computed on the host), so short streams no longer idle while long ones form a tail. The kernel runs at 30.4 active warps and 62% issue-slot utilization (public decoder: 19%), executes 34 instead of 42 warp instructions per 32 symbols, and its output is compared byte for byte with the public decoder's on every stream before any timing is recorded.

C.3 Fused Kernel Numerics and Cost

Numerics. Against an fp64 reference computed from the decoded symbols, maps, and scales, the maximum absolute error of a layer-0 module is 3×10^{-7} to 5×10^{-5} at output magnitudes of 3 to 200, for 1, 16, and 2048 tokens and all four module types; a dequantize-then-multiply path in fp16 sits at 6×10^{-4} to 6×10^{-2} on the same inputs. Against vLLM's Marlin kernel (Frantar et al., 2024) on the first prefill of all 128 modules of Llama-2-7B, the relative Frobenius error is 2.0×10^{-4} to 3.2×10^{-4}. The greedy tokens equal the stock engine's in every run of Sections 4.5 and 4.6, and in a 512-input, 512-output generation of 8 requests (vLLM, Llama-2-7B-AWQ) every one of the 4,096 generated positions equals the stock engine's.

Cost. One decoder layer of Llama-2-7B costs 0.84 ms at up to 16 tokens and 18.7 ms for a 2048-token prefill with the stream in device memory, and 5.1 to 7.3 ms and 20 to 22 ms with the stream read from pinned host memory, the configuration of Section 4.5 (Table 16). At 1.5 GB/s a layer's 97 MB of compressed bytes (the average over layers; layer 0 in Table 16 is 83 MB) take 65 ms to arrive, so both costs stay hidden behind the transfer; from host memory the kernel consumes compressed bytes at up to 11.3 GB/s, which is the storage bandwidth up to which it keeps pace.

Table 16: Fused-kernel operator cost on layer 0 of Llama-2-7B against Marlin with resident weights: milliseconds per call, median of 10, M tokens; "MB" is the module's compressed size and "pinned" reads the stream from host memory as in the end-to-end runs. Above 256 tokens the end-to-end runs use the prefill branch (decode into a Marlin-layout buffer, then Marlin).

Module	MB	decode only	device, M=16	device, M=2048	pinned, M=16	pinned, M=2048	Marlin, 16 / 2048
qkv_proj	14.7	0.141	0.232	3.59	1.28	4.29	0.041 / 0.75
o_proj	7.1	0.130	0.189	2.28	0.48	2.21	0.040 / 0.26
gate_up_proj	41.4	0.218	0.255	9.53	2.22	9.94	0.045 / 1.35
down_proj	19.9	0.151	0.224	3.45	1.16	5.75	0.044 / 0.65
layer	83.1	0.64	0.84	18.7	5.1 to 7.3	19.7 to 22.2	0.17 / 3.0

Table 16 reports the operator cost of the fused kernel on layer 0 of Llama-2-7B against Marlin with resident weights. With the stream in device memory the kernel is bound by the number of independent decode chains: a module has 512 to 1376 32 KB blocks, each a serial chain of 1024 dependent decode steps, which is 0.1 to 0.5 waves of four-warp thread blocks on 192 SMs. With the stream in pinned host memory the 64-byte window refills are bound by the PCIe transaction rate, which is the 11.3 GB/s quoted above; with resident weights the engines' native kernels remain the right choice. For prefill batches above 256 tokens the fused kernel would be compute-bound, so the module's streams are decoded into one reusable Marlin-layout buffer (43 MB for Llama-2-7B, 48 MB for Qwen3-8B; in FlexGen a 90 MB fp16 buffer) and multiplied by the engine's own kernel: about 20 ms per layer for 8,192 tokens, still below the 65 ms transfer.

C.4 SHORT REQUESTS

Table 17 repeats the main comparison with 128 input and 128 output tokens at the same 1.5 GB/s. The per-token times and their ratios are within 0.2% of Table 4, since every forward pass is bound by the same transfer; the FlexGen memory is lower because the prefill of 16 short requests needs less activation memory, and the differences between configurations are unchanged.

Table 17: Storage-offloaded serving at 1.5 GB/s with short requests, AWQ W4 models (batch 16, 128 input and 128 output tokens, three interleaved repeats, medians). GPU memory reduction is the raw peak minus the **Q3.6** peak in the same run. Tokens equal the stock engine's in every repeat.

System, model	TTFT (s) raw	**Q3.6**	change	TPOT (s) raw	**Q3.6**	change	GPU memory reduction (GB)
vLLM, Llama-2-7B	2.276	2.101	−7.7%	2.242	2.061	−8.1%	3.10
SGLang, Llama-2-7B	6.783	6.183	−8.8%	2.243	2.065	−7.9%	3.13
FlexGen, Llama-2-7B	2.244	2.129	−5.1%	2.229	2.078	−6.8%	0.73
vLLM, Qwen3-8B	2.340	2.148	−8.2%	2.406	2.173	−9.7%	3.29

C.5 MEMORY ACCOUNTING

Table 18 lists the memory of the vLLM configurations for Llama-2-7B in the runs of Table 4. The **Q3.6** configuration frees the Marlin `qweight` of every quantized module (3.24 GB, 4.0 of the 4.156 bits per weight of resident weight state); scales and zeros (0.156 bpw) and a 97 MB working set remain resident: the 43 MB prefill buffer, per-module maps generated just in time, an 8 MB workspace, 1 MB of decoder scratch, and a 512-token accumulator chunk. In FlexGen the configuration avoids the executor's 810 MB fp16 pool and adds 134 MB (the 90 MB fp16 prefill buffer, maps, workspace, and scratch). The driver-level (nvidia-smi) reduction is smaller than the allocated one because PyTorch keeps partially used allocator segments reserved. FlexGen is measured with each configuration in its own process.

Table 18: GPU and host memory of the vLLM configurations, Llama-2-7B-AWQ, batch 16, 512 input and 32 output tokens (GB unless stated).

Configuration	allocated at rest	peak allocated	peak reserved	nvidia-smi peak	host pinned
stock (resident, no streaming)	50.12	50.87	51.00	51.81	0
raw streaming	50.12	50.87	51.00	51.81	1003 MB
Q3.6 streaming	46.93	47.68	48.14	48.93	928 MB

C.6 LLAMA-2-13B AT THE FOUR BUDGETS

Table 19 repeats the budget experiment of Table 5 with Llama-2-13B-AWQ. Its packed 4-bit weights are 6.34 GB (6.59 GB with scales and zeros), the **Q3.6** stream is 6.06 GB (byte ratio 1.089), and the resident weights including the embeddings and the output projection are 7.3 GB. With the native backend the 8 GB budget is too small: after the weights are loaded, 0.11 GB remain for the KV cache, less than the 0.45 GB one request needs, so vLLM does not start. **Q3.6** keeps the weights off the GPU, serves 14 requests at 8 GB, and has 6.40 GB more KV cache at 12 to 24 GB: at 12 GB it serves 2.7 times as many requests as the native backend.

Table 19: Llama-2-13B-AWQ at equal GPU budget (vLLM): KV cache at start-up in GB, in tokens, and in concurrent 544-token requests. Raw is the engine's native `awq_marlin` backend; change is in requests.

	raw			**Q3.6**			
GPU budget	KV cache	tokens	requests	KV cache	tokens	requests	change
8 GB	does not start (0.11 GB left)			6.52	7,952	14	–
12 GB	4.39	5,360	9	10.80	13,168	24	+167%
16 GB	8.58	10,464	19	14.98	18,288	33	+74%
24 GB	17.15	20,928	38	23.55	28,736	52	+37%

C.7 FlexGen at the Four Budgets

Table 20 repeats the budget experiment of Table 5 in FlexGen. FlexGen's raw configuration already streams the weights layer by layer and holds only a two-layer fp16 buffer of dequantized weights, so neither configuration keeps the model's weights on the GPU and the comparison measures what **Q3.6** adds on top of streaming. FlexGen allocates the KV cache for each generation rather than at start-up, so capacity here is the largest batch whose 512-input, 32-output generation completes within the budget. The budget is charged as for vLLM: the memory the process uses outside PyTorch's allocator after a warm-up, 0.82 GB, counts against it, and PyTorch's allocator is capped at the rest. Throughput is measured at that batch and 1.5 GB/s, with three alternated repeats and medians, on the host of Table 5, where a batch-16 run matches the FlexGen row of Table 4 within 1% (2.243 and 2.061 s per token); on this host the ratio of FlexGen's per-token times is 1.088, the byte ratio, as in vLLM and SGLang. **Q3.6** saves the net 0.68 GB of Table 4 and serves 1 to 2 more requests at every budget, and its throughput gain matches the batch ratio times the byte ratio of 1.088 within 0.2%, as the transfer-bound model predicts. Tokens are identical in every run.

Table 20: Equal GPU budget, FlexGen, Llama-2-7B-AWQ, 1.5 GB/s: the largest batch of 544-token requests whose generation completes, and its output tokens per second (three alternated repeats, medians). Raw streams the native AWQ tensors into a two-layer fp16 buffer.

	Largest batch			Output tokens/s		
GPU budget	raw	**Q3.6**	change	raw	**Q3.6**	change
8 GB	15	17	+13%	6.69	8.24	1.23×
12 GB	26	27	+4%	11.60	13.10	1.13×
16 GB	36	38	+6%	16.05	18.43	1.15×
24 GB	57	59	+4%	25.42	28.58	1.12×

D Limitations

Q3.6 is a metadata-conditioned exact lossless coding method for quantized model tensors. Its storage gains require native quantization metadata that the decoder can read and that predicts the payload distribution; they shrink where that dependence is weak or where tensors are too small to amortize the route and table overhead. The evaluation covers broad target-tensor sets and representative format controls.

The serving results of Section 4.5 are for the storage-bound setting, in which the quantized weights stream from an SSD, a cloud block device, or a network tier on every forward pass; their benefit comes from moving fewer bytes and from not holding decoded weights on the GPU, and the fused kernel keeps pace with storage tiers up to about 11 GB/s. When the weights are resident in GPU memory the engines' native kernels remain the right choice, and the storage layer of Section 4 is the part of **Q3.6** that applies. The capacity gains of Section 4.6 come from not materializing the weights. In vLLM and SGLang, whose Marlin kernels take whole resident modules, Section 4.6 compares the engines' native layout with the fused path; FlexGen's raw configuration already streams layer by layer, and against it **Q3.6** serves 1 to 2 more requests and delivers 1.12 to 1.23 times the throughput (Appendix C.7). A streaming path on an nvCOMP codec would move more bytes per token than **Q3.6**, since every nvCOMP codec produces a larger object (Table 3).

The GPU budgets are fractions of one 96 GB card rather than measurements on smaller cards. The end-to-end measurements use one GPU model and three serving systems; other accelerators require the released kernels to be rebuilt.

E BROADER IMPACT

`Q3.6` reduces the storage footprint of already-quantized LLMs while preserving the exact quantized tensors. Smaller compressed models reduce storage, transfer, and restoration costs in research and deployment, especially for teams that manage several quantized variants. The method does not introduce new model capabilities, training data, or generated content; its effect is on the representation and transport of existing quantized weights.

The same storage reduction can also make existing models easier to copy, archive, and redistribute, including models that may be used in harmful applications. To limit release-related risk, the submission artifact does not redistribute external model weights and requires users to obtain models, quantization assets, and third-party libraries under their original licenses and terms. Any deployment of compressed quantized models inherits the safety, privacy, and misuse considerations of the underlying model and serving system.