ENTROPACK: FAST AND ACCURATE ENTROPY-CODED WEIGHT COMPRESSION AT ARBITRARY BITRATES

Anonymous authors
Paper under double-blind review

ABSTRACT

Weight compression helps large generative models fit deployment memory budgets, but common fixed-width formats offer only coarse storage choices. Entropy coding supports finer rates, yet the achieved size depends on the quantized weight distribution and coding overhead. Exploiting this flexibility requires accurate rate selection and efficient weight reconstruction for inference. We present EntroPack, an entropy-coded weight compressor that supports arbitrary target bitrates without activation calibration or fine-tuning. It combines row-normalized E_8 lattice quantization with a conditional probability model of lattice coordinates. Sampled storage estimates select the quantization resolution without repeated full-stream encoding. The final coordinates are entropy-coded in independently decodable tiles, enabling fast, fused symbol decoding and numerical weight reconstruction on the GPU. EntroPack supports floating-point and integer weight containers, including BF16, FP16, FP8, and INT8, with storage bitrate controlled independently of numerical precision. We evaluate EntroPack's compression quality and encoding speed on popular diffusion and language models. When compressing the linear-layer weights of the image generator Z-Image-Turbo, EntroPack achieves substantially lower weight and denoiser output errors than fixed-width formats at comparable storage rates, with modest denoising-step overhead. Targeting 4 bits per parameter, it achieves lower weight and denoiser output errors than NF4, including about 24% lower relative L_2 weight error, with less storage.

1 INTRODUCTION

Weight storage constrains the deployment of large generative models. Fixed-width formats, including INT8, FP8, and four-bit formats, provide established compute paths, but a deployment budget may lie between their available storage sizes (Dettmers et al., 2022; 2023; Micikevicius et al., 2022; Darvish Rouhani et al., 2023). Entropy coding allows finer choices by assigning shorter codes to frequent quantized values: adjusting quantization resolution changes both weight error and average storage. This approach is well established in weight compression (Han et al., 2016; Wiedemann et al., 2019), with recent methods supporting fractional rates without activation calibration (Chen et al., 2026; Domb et al., 2026).

Using this flexibility introduces two challenges. First, a requested bitrate must be translated into a quantization resolution (Chen et al., 2026), even though the stored size also depends on the tensor's symbol distribution and coding metadata. Repeatedly encoding full tensors during this search can make compression expensive. Second, inference requires the compressed symbols to be decoded and converted into numerical weights before matrix products. The stream layout and weight representation must support parallel reconstruction to limit added latency (Zhang et al., 2025a). A practical compressor needs inexpensive storage estimation and efficient weight recovery.

In this paper, we present EntroPack, a calibration-free weight compressor that combines fine-grained bitrate control with efficient GPU reconstruction. After row normalization, it quantizes blocks of eight weights onto the E_8 lattice and represents the resulting points as invertible integer fields (Conway & Sloane, 1982). A conditional probability model captures distribution differences between the lattice's integer and half-integer cosets. Using this model and coding metadata, EntroPack estimates storage from sampled rows and searches for a quantization scale matching the requested

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.

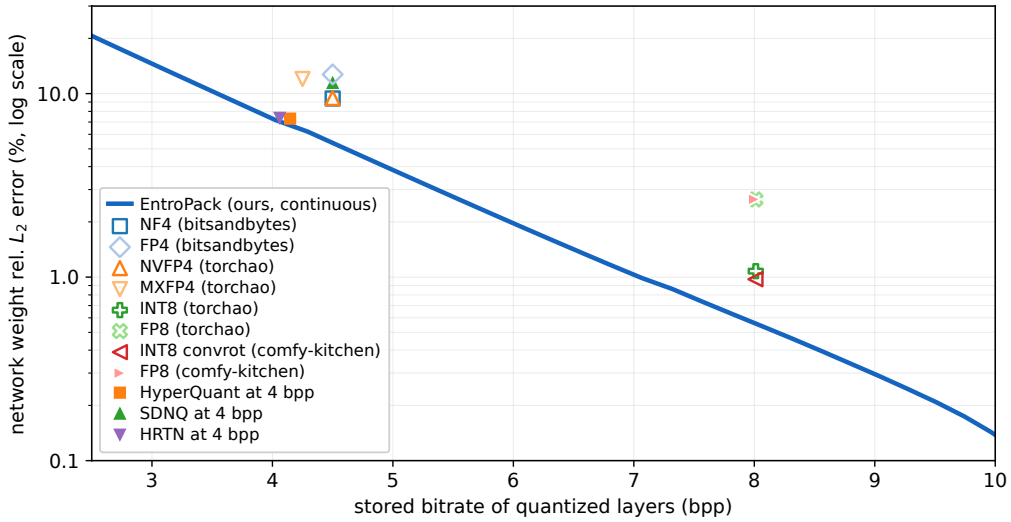

Figure 1: Storage and reconstruction error of compressed linear weights in Z-Image-Turbo.

bitrate, avoiding repeated full-stream encoding. After scale selection, we apply row-scale fitting and optional per-row rate–distortion refinement to further improve weight reconstruction.

EntroPack then encodes the fields with range asymmetric numeral systems (rANS) in independently decodable tiles. This layout permits parallel decoding across tiles while preserving the field order required by the conditional model. The arithmetic inverse of the field representation allows symbol decoding, lattice reconstruction, and row rescaling to be fused on the GPU, without an intermediate symbol tensor or reconstruction codebook. The decoder outputs weights for matrix products in floating-point or integer formats, including FP8 and INT8 for low-precision computation. We evaluate weight reconstruction, inference quality, and runtime after compressing linear weights of the image generator Z-Image-Turbo, the audio-video generator MiniMax-H3, and the language model Qwen3.8-27B. EntroPack supports continuously adjustable target bitrates, and Figure 1 shows that it reconstructs Z-Image-Turbo's linear weights with lower error than the tested fixed-width formats at comparable storage. Our contributions are as follows:

- **Fine-grained rate control.** EntroPack selects lattice quantization resolution from sampled storage estimates specific to each tensor. This provides user-specified weight bitrates without activation calibration or repeated full-stream encoding during rate selection.
- **Compact coding and GPU reconstruction.** Coset-conditioned coordinate coding and independent rANS tiles allow symbol decoding, arithmetic reconstruction, and row rescaling to be fused on the GPU. The representation requires neither an intermediate symbol tensor nor a reconstruction codebook and supports floating-point and integer outputs.
- **Compression quality and runtime.** EntroPack achieves lower weight error than tested fixed-width formats at comparable storage and faster conversion than tested entropy-coded methods. For Z-Image-Turbo's linear weights at a 4 bpp target, conversion takes 3.5 s and reconstruction adds 7.7% to denoising-step time relative to the uncompressed model.

2 Related Work

Entropy-coded quantization. Entropy-constrained quantization jointly considers coding rate and distortion (Chou et al., 1989; Gray & Neuhoff, 1998). In weight compression, Deep Compression applies Huffman coding to quantized weights (Han et al., 2016), while DeepCABAC uses context-adaptive arithmetic coding with rate–distortion optimization (Wiedemann et al., 2019). More recently, EntQuant reduces weight entropy before coding with asymmetric numeral systems (Putzky et al., 2026), and NeuZip combines exponent coding with a precision-reduction option (Hao et al.,

2024). For rate control, HRTN selects scalar quantization scales using Huffman code length (Chen et al., 2026). HyperQuant combines rotated lattices, algebraic constraint removal, and Rice coding with a precomputed relationship between rate and signal-to-noise ratio (Domb et al., 2026).

Structured weight quantization. Weight quantizers differ in both their use of calibration data and their representation of weights. GPTQ and AWQ use activation information for low-bit quantization (Frantar et al., 2022; Lin et al., 2024). In terms of representation, NF4 provides a nonuniform four-bit scalar format (Dettmers et al., 2023). Beyond scalar formats, QuIP and QuIP# combine weight transformations with structured quantization. QuIP# uses a shared E_8-based codebook to quantize blocks of eight weights jointly (Chee et al., 2023; Tseng et al., 2024a). Other structured representations include additive codebooks, grouped and Leech lattices, and trellis quantization (Egiazarian et al., 2024; Zhang et al., 2025b; van der Ouderaa et al., 2026; Tseng et al., 2024b).

Deployment and GPU compression. Deployment-oriented systems such as SDNQ, Quanto, and GGUF provide quantized storage and execution paths (Disty0, 2026; Corvoysier & Hugging Face contributors, 2026; ggml contributors, 2023). At the storage level, lossless weight compressors exploit redundancy in numerical representations to reduce storage without altering weight values (Zhang et al., 2025a; Hershcovitch et al., 2025; Tan et al., 2026). For lossy compression, ZFP and cuSZp address precision- or error-controlled compression of numerical arrays (Lindstrom, 2014; Huang et al., 2023). For parallel entropy coding, DietGPU provides GPU rANS primitives for symbol encoding and decoding (Johnson, 2022), based on asymmetric numeral systems (Duda, 2013).

3 METHOD

3.1 OBJECTIVE AND PIPELINE

Let $W \in \mathbb{R}^{R\times C}$ be a weight matrix with $N = RC$ elements stored in numerical data type (dtype) τ, where C is divisible by eight. EntroPack produces a compressed representation $\mathcal{C}$ and reconstructs $\hat{W}$ in the same dtype. We measure relative reconstruction error and stored rate as

$$E_{\mathrm{rel}}(W,\hat{W}) = \sqrt{\frac{\sum_{r=1}^{R}\sum_{j=1}^{C}(\hat{W}_{rj}-W_{rj})^2}{\sum_{r=1}^{R}\sum_{j=1}^{C}W_{rj}^2}}, \qquad R_{\mathrm{stored}}(\mathcal{C}) = \frac{8B_{\mathrm{stored}}(\mathcal{C})}{N}, \tag{1}$$

where B_{stored} is the actual stored size in bytes. For a requested rate R_{tgt} in bpp, the goal is to minimize reconstruction error while keeping the stored rate close to the request. The dtype specifies numerical values independently of compressed bit width.

Figure 2 shows the compression and reconstruction pipeline. Encoding starts by selecting a quantization scale on a sample of rows. For each candidate scale, we map normalized weights to lattice points and represent these points as integer fields, whose statistics determine an estimated storage cost. At the selected scale, we quantize the full tensor and fit one reconstruction scale per row. Optional rate–distortion optimization (RDO) refines the per-row quantization before the final fields are rANS-encoded into independent tiles. The resulting compressed representation contains the tile payloads and the information needed to decode them.

The decoding path reverses these operations: rANS decoding recovers the fields, the algebraic inverse reconstructs lattice points, and row rescaling and dtype conversion produce numerical weights.

3.2 ROW-NORMALIZED LATTICE REPRESENTATION

Normalization and nearest points. We normalize row magnitudes so that a common scale can control quantization resolution across rows. The E_8 lattice combines the geometric efficiency of vector quantization with a structured nearest-point algorithm (Conway & Sloane, 1999). We convert W to FP32, denoted by x, and compute the root-mean-square (RMS) magnitude of each row, $\mu_r = \max(\sqrt{C^{-1}\sum_j x_{rj}^2}, \eta)$, with $\eta = 10^{-12}$. We divide row r by μ_r and partition it into vectors $X_v \in \mathbb{R}^8$. A shared quantization scale $s > 0$ determines their lattice points:

$$p_v(s) = \mathrm{nearest}_{E_8}(X_v/s). \tag{2}$$

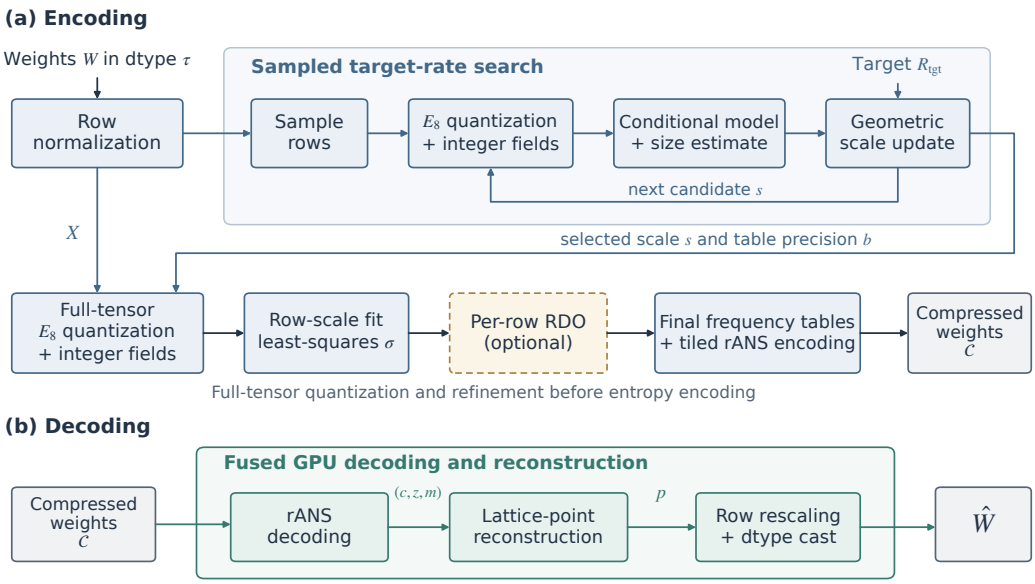

Figure 2: EntroPack encoding and decoding. (a) Sampled scale selection precedes full-tensor quantization, row-scale fitting, optional RDO, and tiled rANS encoding. (b) Symbol decoding, lattice reconstruction, row rescaling, and dtype conversion are fused on the GPU.

For a block in row r, the corresponding weight approximation is $s\mu_r p_v(s)$. Thus s controls quantization in normalized coordinates, while μ_r restores the row's original magnitude.

Write $D_8 = \{z \in \mathbb{Z}^8 : \sum_i z_i \equiv 0 \pmod 2\}$, so that $E_8 = D_8 \cup (D_8 + \frac{1}{2}\mathbf{1})$. The nearest point in each coset is obtained by coordinate rounding followed, when needed, by a parity correction (Conway & Sloane, 1982). For $Y = X_v/s$, let $f = \text{round}(Y)$, $d = Y - f$, $\pi = (\sum_i f_i) \bmod 2$, and $j^\star = \arg\max_j |d_j|$. With e_j denoting the jth coordinate basis vector,

$$\text{nearest}_{D_8}(Y) = f + \pi\,\text{sign}(d_{j^\star})e_{j^\star}. \tag{3}$$

We compare this candidate with $\text{nearest}_{D_8}(Y - \frac{1}{2}\mathbf{1}) + \frac{1}{2}\mathbf{1}$ and select the closer one. Coordinate ties round to the nearest even integer. Parity correction breaks equal-residual ties by the first coordinate and uses $\text{sign}(0) = +1$. These conventions fix the point recovered during decoding.

An invertible field representation. Each p_v lies in one of the two cosets. Let $c \in \{0, 1\}$ identify that coset and write $z = p_v - \frac{c}{2}\mathbf{1} \in D_8$. The eighth integer coordinate satisfies a parity constraint determined by the first seven. Define

$$\pi = (\textstyle\sum_{i=1}^{7} z_i) \bmod 2, \qquad m = (z_8 - \pi)/2. \tag{4}$$

The point is represented by $(c, z_1, \ldots, z_7, m)$ and recovered as

$$p_v = (z_1, \ldots, z_7, 2m + \pi) + \tfrac{c}{2}\mathbf{1}. \tag{5}$$

The fields uniquely identify each point and permit arithmetic recovery. In EntroPack, this representation supplies the symbols for the conditional entropy model described next.

3.3 Coset-conditioned probability model

The coordinate fields can have different distributions in the two cosets. We capture this dependence by conditioning their probabilities on the coset. Let $a = (z_1, \ldots, z_7, m)$ denote the eight coordinate fields, and let θ collect the distribution parameters. We model the coset and coordinates as

$$q_\theta(c, a) = q_\theta(c) \prod_{i=1}^{8} q_\theta(a_i \mid c), \tag{6}$$

using seventeen categorical distributions: one for the coset and two for each coordinate field. For any given set of quantized vectors, we estimate these distributions from the corresponding field counts.

The pooled alternative codes the coset and each coordinate field independently. With exact empirical probabilities, conditioning reduces its ideal coding cost by

$$[H(c) + \textstyle\sum_i H(a_i)] - [H(c) + \sum_i H(a_i \mid c)] = \sum_i I(a_i; c), \tag{7}$$

where H denotes entropy and I mutual information, both in bits. The model captures dependence between each coordinate field and the coset, while treating the coordinates as conditionally independent. The net stored-rate gain also depends on frequency quantization and table storage. We measure it at fixed reconstructions in Appendix A.3.

We represent probabilities by normalizing counts to integer frequencies summing to 2^b, where b is the probability-table precision in bits. Every observed symbol receives a positive frequency. Let $n_{t,z}(s)$ be the count of symbol z in distribution t at quantization scale s, and let $f_{t,z}(s)$ be its normalized frequency. The probability is $f_{t,z}(s)/2^b$, giving the modeled code length in bits

$$C_{\mathrm{CE}}(s, b) = \sum_t \sum_{z: n_{t,z} > 0} n_{t,z}(s)\big(b - \log_2 f_{t,z}(s)\big). \tag{8}$$

This quantity is evaluated from symbol counts and provides the coding-cost term for scale selection.

3.4 Target-rate scale selection

Scale–rate relationship. The scale s changes both the lattice resolution and the distribution of coded fields. Under a smooth continuous-source approximation and sufficiently fine quantization, the ideal rate of an eight-dimensional scaled lattice is

$$R_{\mathrm{ideal}}(s) \approx h(X)/8 - \log_2 s - (\log_2 v_\Lambda)/8. \tag{9}$$

Here $h(X)$ is the source differential entropy in bits and v_Λ the lattice cell volume, with $v_{E_8} = 1$ (Gray & Neuhoff, 1998). This motivates geometric scale search.

Stored-size estimate. Stored size includes coded symbols and the metadata needed for reconstruction. We estimate it from candidate field counts and the planned tile layout, without producing a codestream. The model cost C_{CE} estimates the symbol cost, but some of this information remains in the coder's final states rather than in emitted payload words. We estimate the payload after this correction, then add the stored states and other metadata. In the tiled rANS format (§3.6), each independently decodable tile uses J coder states and 16-bit payload words. For n_{tile} tiles,

$$\widehat{L}_{\mathrm{payload}} = 16\lceil \max(0, C_{\mathrm{CE}} - 16 J n_{\mathrm{tile}})/16 \rceil. \tag{10}$$

The subtraction approximates information retained in those states, and the remainder is rounded to whole payload words. Adding auxiliary storage gives

$$\widehat{L} = \widehat{L}_{\mathrm{payload}} + L_{\mathrm{restart}} + L_{\mathrm{tables}} + L_{\mathrm{scales}} + \widehat{L}_{\mathrm{other}}, \qquad \widehat{B} = \widehat{L}/8. \tag{11}$$

Here L_{restart} covers the final states and payload offsets needed to start decoding each tile. The remaining terms account for probability tables, row scales, and other metadata.

Sampled rate evaluation. We reuse a fixed sample of complete rows across candidate scales. At each (s, b), we quantize the sample, obtain $C_{\mathrm{CE}}(s, b)$ from field counts, and apply the stored-size estimate above. For N_{sub} sampled weights and estimated size $\widehat{B}_{\mathrm{sub}}(s, b)$, scale selection seeks

$$\widehat{R}_{\mathrm{sub}}(s, b) = 8\widehat{B}_{\mathrm{sub}}(s, b)/N_{\mathrm{sub}} \approx R_{\mathrm{tgt}}. \tag{12}$$

This avoids encoding full streams for candidate scales. Appendix C.1 specifies the sampling settings.

Geometric search. At a fixed probability precision b, we evaluate the geometric midpoint $s = \sqrt{lo\, hi}$ of a scale bracket. When the estimated rate exceeds the target, we replace lo by s. Otherwise, we replace hi by s. After a fixed number of iterations, we use the geometric midpoint of the remaining bracket to quantize the full tensor. If its field alphabets exceed the capacity of the probability representation, we increase b and repeat the search, coarsening s if the maximum supported precision is reached. Algorithm 1 summarizes scale selection and full-tensor quantization. $\mathcal{B}$ lists probability precisions in ascending order and I_s is the number of scale-search steps.

Algorithm 1 Target-rate scale selection and full-tensor quantization.

Require: Tensor W, target R_{tgt}, precisions $\mathcal{B}$, search steps I_s
Normalize rows to obtain X and fix a row sample $\mathcal{S}$ of N_{sub} weights
for $b \in \mathcal{B}$ in ascending order **do**
  Initialize the scale bracket (lo, hi)
  for $i = 1, \ldots, I_s$ **do**
    $s \leftarrow \sqrt{lo\, hi}$
    Quantize $X_{\mathcal{S}}/s$ and obtain field counts $n_{t,z}$ and b-bit frequencies $f_{t,z}$
    $C_{\mathrm{CE}} \leftarrow \sum_{t,z:n_{t,z}>0} n_{t,z}\big(b - \log_2 f_{t,z}\big)$
    Estimate stored bytes $\widehat{B}_{\mathrm{sub}}(s, b)$ from C_{CE} and format overheads
    $\widehat{R}_{\mathrm{sub}} \leftarrow 8\widehat{B}_{\mathrm{sub}}(s, b)/N_{\mathrm{sub}}$
    Set $lo \leftarrow s$ if $\widehat{R}_{\mathrm{sub}} > R_{\mathrm{tgt}}$ and $hi \leftarrow s$ otherwise
  end for
  Set $s \leftarrow \sqrt{lo\, hi}$ and quantize all blocks to form their fields
  Stop the precision search when the full-tensor alphabets fit 2^b
end for
return s, b, lattice points $\{p_v\}$, and fields $\{(c_v, a_v)\}$

3.5 Row-scale refit and rate–distortion optimization

Row-scale refit. The search scale s determines which lattice points are selected. Once those points are fixed, a separate reconstruction scale can reduce row error without changing their coded fields. Writing p_{rj} for the lattice coordinates of row r, we fit this scale by least squares:

$$\sigma_r = \begin{cases} \max\left(\eta, \dfrac{\sum_j x_{rj} p_{rj}}{\sum_j p_{rj}^2}\right), & \sum_j p_{rj}^2 > 0, \\ s\mu_r, & \text{otherwise.} \end{cases} \tag{13}$$

For nonzero lattice rows, this minimizes $\sum_j (x_{rj} - \sigma_r p_{rj})^2$ over scales at least η. The fitted σ_r replaces $s\mu_r$, restoring the row's numerical scale without changing its coded symbols or scale storage.

Per-row rate–distortion refinement. Rows can have different rate–distortion trade-offs even after normalization. RDO compares several quantization resolutions per row and allocates them under a common storage budget. It alternates between selecting row candidates under a fixed probability model and updating that model from the selected fields. Each row's K candidates use scales $\rho_k s$, with ratios in the empirically chosen range $[0.70, 1.45]$. Starting from $\rho = 1$, we add candidates by bisecting the widest remaining interval in log scale, so candidate sets are nested as K increases.

Each candidate is quantized at $\rho_k s$ and refit using equation 13. Let $D_{r,k}$ be its sum of squared reconstruction errors on row r. All rows initially use the baseline at scale s. To compare candidates under a shared coding model, we estimate conditional probabilities from the current row selections, adding $\frac{1}{2}$ to each symbol count before normalization over the coordinate ranges spanned by all candidates. Writing $(c_v^{(k)}, a_v^{(k)})$ for the fields of candidate k, its estimated code length on row r is

$$R_{r,k} = \sum_{v \in r} \Big[-\log_2 q_\theta(c_v^{(k)}) - \sum_{j=1}^{8} \log_2 q_\theta(a_{v,j}^{(k)} \mid c_v^{(k)}) \Big]. \tag{14}$$

We use the baseline's estimated stored size, B bytes, as the allocation target. Let $\widehat{B}_{\mathrm{mix}}$ be the estimated stored bytes for current row selections $k(r)$. The next code-length budget is

$$R_{\mathrm{budget}} = \sum_r R_{r,k(r)} + 8\big(B - \widehat{B}_{\mathrm{mix}}\big). \tag{15}$$

The correction converts the current storage surplus or deficit into bits: an oversized mixture reduces the next code-length budget, while spare storage increases it.

For a rate penalty $\lambda \geq 0$, each row selects

$$\kappa_\lambda(r) = \arg\min_k \big[D_{r,k} + \lambda R_{r,k}\big]. \tag{16}$$

We search for λ to keep the modeled code length within R_{budget}, then update probabilities from the selected fields. This allocation–update procedure runs for T sweeps. Further details are given in Appendix C.2.

3.6 RANS ENCODING AND THE STORED REPRESENTATION

After scale selection, refitting, and any RDO refinement, the quantized fields and reconstruction scales are fixed. We encode these fields with rANS to form the codestream in Figure 2.

Frequency tables and tiles. We recount the final full-tensor fields and construct seventeen shared integer frequency tables using the model in §3.3. Each coordinate table stores a minimum-value offset for zero-based symbol indices. For parallel decoding, we partition the fields into independent tiles of V lattice vectors, each containing $9V$ symbols, and use $J = 32$ interleaved rANS states per tile. This layout assigns one tile to a GPU warp and one state to each lane.

Encoding the fields. For a symbol with integer frequency f and cumulative frequency F in a table totaling $M = 2^b$, rANS updates the coder state ξ as follows (Duda, 2013)

$$\xi' = M\,\lfloor \xi / f \rfloor + (\xi \bmod f) + F. \tag{17}$$

Before this update, renormalization emits payload words as needed to bound the state. Since decoding reverses the updates, we encode each vector's coordinate fields in reverse order, then its coset. The decoder recovers the coset first to select the conditional coordinate tables.

Stored representation. The emitted words form each tile's payload. We concatenate these payloads and store their offsets and terminal rANS states for independent decoding. The compressed representation also contains the frequency tables, alphabet offsets, row reconstruction scales, and tensor layout. Their combined byte count gives the actual stored rate in equation 1. Larger tiles amortize the states and offsets over more weights but expose less decoding parallelism.

3.7 GPU DECODING AND WEIGHT RECONSTRUCTION

Given the stored representation, the decoder follows the lower path of Figure 2. One GPU warp loads a tile's terminal states and locates its payload using the stored offsets, with one state per lane. For a current state ξ', it identifies the symbol whose cumulative-frequency interval contains $u = \xi' \bmod M$ and inverts equation 17:

$$\xi = f\,\lfloor \xi' / M \rfloor + u - F. \tag{18}$$

Renormalization consumes payload words as required to restore the state. Each lane decodes a vector's coset before its coordinate fields, using the coset to choose the conditional tables. Thus entropy decoding recovers the field tuple $(c, z_1, \ldots, z_7, m)$.

The lane then applies the algebraic inverse in equation 5 to recover the lattice point and multiplies it by the stored row scale. Let $\bar{\sigma}_r$ denote that scale read in FP32. Rounding and saturation to the requested dtype τ produce the reconstructed weight:

$$\hat{W}_{rj} = \text{snap}_\tau(\bar{\sigma}_r p_{rj}). \tag{19}$$

These steps fuse symbol recovery, lattice reconstruction, and row rescaling without materializing an intermediate symbol tensor or consulting a reconstruction codebook. The field representation and entropy coder are shared across supported floating-point and integer output dtypes.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We compress linear weights of the image generation model Z-Image-Turbo (Z-Image Team, 2025), the audio-video generation model MiniMax-H3 (MiniMax, 2026), and the language model

Qwen3.8-27B (Qwen Team, 2026). We measure weight storage and reconstruction error, along with model quality and runtime using the compressed weights. All experiments use a single NVIDIA H20 with PyTorch (Paszke et al., 2019). Diffusion models run in DiffSynth-Studio (ModelScope Team, 2024). Additional experiments and ablations are reported in Appendices A and B.

Metrics. We report stored rate, relative L_2 weight error, task fidelity, model conversion time, and inference time. Stored rate in bpp is eight times the total stored bytes, including side information, divided by the total number of weight elements across the evaluated layers. Linear biases are excluded from the denominator. Network weight error follows equation 1, computed jointly over these weights and reported as a percentage. For task fidelity, we compare diffusion outputs with the uncompressed BF16 reference and report perplexity for the language model. Inference timing uses each method's execution path. Evaluation and timing details are provided in Appendix A.1.

Baselines. We compare with NF4 and FP4 from bitsandbytes (Dettmers et al., 2023), INT8, FP8, NVFP4, and MXFP4 from torchao (Or et al., 2025; Micikevicius et al., 2022; Darvish Rouhani et al., 2023; Alvarez et al., 2025), and INT8 with convolutional rotation and FP8 from comfy-kitchen (Comfy Org, 2025). We also evaluate the released HyperQuant, HRTN, and SDNQ implementations (Domb et al., 2026; Chen et al., 2026; Disty0, 2026). Weights that a format cannot represent remain at BF16 and contribute to its reported storage. EntroPack labels give target rates, while columns report stored bpp.

4.2 Rate–distortion and target-rate accuracy

Figure 1 plots the aggregate storage and reconstruction error of Z-Image-Turbo's compressed linear weights. EntroPack consistently achieves lower weight error than the tested fixed-width formats at comparable storage rates. Baseline markers are measured at 4- and 8-bit quantization targets. Across the measured targets, achieved rate increases with the requested rate, while weight error decreases. The absolute relative targeting error, $100|R_{\mathrm{stored}}/R_{\mathrm{tgt}} - 1|$, has a mean of 0.80% and a maximum of 2.13%. At its 4 bpp target, EntroPack stores 4.02 bpp with 7.18% weight error, compared with NF4's 4.50 bpp and 9.41% error. Entropy-coded alternatives are closer: HyperQuant and HRTN achieve 7.31% and 7.38% error at 4.15 and 4.06 bpp, respectively.

4.3 Diffusion model results

We compare Z-Image-Turbo outputs produced with compressed weights against the BF16 reference in Table 1: relative L_2 measures first-step denoising output error, while peak signal-to-noise ratio (PSNR) and structural similarity (SSIM) measure image fidelity. EntroPack compresses weights substantially faster than HRTN and HyperQuant and incurs less inference overhead. Among the evaluated four-bit configurations, it achieves the lowest weight reconstruction error, with a substantial advantage over fixed-width formats. At 4.02 bpp, EntroPack achieves 19.62 dB PSNR and 0.80 SSIM, close to those of HyperQuant and HRTN at their slightly higher stored rates. Increasing EntroPack's storage to 8.06 bpp improves weight reconstruction and image fidelity: weight and output errors fall to 0.54% and 3.92%, and PSNR and SSIM rise to 29.14 dB and 0.95.

At a 4 bpp target, EntroPack compresses the linear weights in 3.5 s, versus 33.2 s for HRTN and 34.2 s for HyperQuant. With on-demand reconstruction, denoising steps take 544.7–555.6 ms at targets of 4, 7, and 8 bpp, only 7.7–9.9% above BF16's 505.7 ms. These times are lower than those of the tested torchao floating-point configurations, HRTN, and HyperQuant.

4.4 Language model results

In Table 2, we compare storage, weight error, WikiText-2 perplexity (PPL) (Merity et al., 2017), and runtime across methods for compressing Qwen3.8-27B's linear weights. At 4.03 bpp, EntroPack's PPL rises from BF16's 5.98 to 6.06, compared with 6.08 for NF4 at 4.50 bpp and 6.04 for HyperQuant at 4.09 bpp. At 7.10 and 8.08 bpp, PPL matches BF16 to the reported precision.

At a 4 bpp target, EntroPack converts Qwen3.8-27B's linear weights in 5.8 s, versus 109.1 s for HRTN and 149.6 s for HyperQuant. Inference overhead depends on the workload. Processing a 32k-token input (prefill) takes 18,709 ms with EntroPack-compressed weights, close to the BF16

method	bpp	weights rel. L_2 (%) ↓	network out rel. L_2 (%) ↓	images (3 seeds, 2 prompts) PSNR ↑	SSIM ↑	inference step (ms) ↓	conversion (s) ↓
BF16 reference	16.00	0.00	0.00	99.00	1.00	505.7	0.0
NF4 (bitsandbytes)	4.50	9.41	25.76	18.97±3.08	0.77±0.09	518.2	0.1
FP4 (bitsandbytes)	4.50	12.73	31.55	17.74±2.08	0.74±0.08	520.0	0.1
NVFP4 (torchao)	4.50	9.47	22.44	19.85±3.45	0.79±0.09	802.6	0.6
MXFP4 (torchao)	4.25	12.02	25.74	18.00±1.86	0.73±0.07	798.8	0.6
INT8 (torchao)	8.01	1.08	4.71	24.90±4.67	0.91±0.06	554.9	0.2
FP8 (torchao)	8.01	2.65	9.47	23.06±3.22	0.87±0.06	577.3	0.2
INT8 convrot (comfy-kitchen)	8.01	0.97	6.35	28.54±5.80	0.93±0.06	464.9	0.1
FP8 (comfy-kitchen)	8.00	2.65	20.86	23.33±4.08	0.88±0.06	348.5	0.0
HyperQuant (4 bpp)	4.15	7.31	23.04	20.28±2.46	0.81±0.07	1041.4	34.2
SDNQ (4 bpp)	4.50	11.43	28.46	17.68±1.97	0.73±0.08	524.1	0.2
HRTN (4 bpp)	4.06	7.38	20.31	20.27±3.37	0.80±0.09	570.3	33.2
EntroPack (4 bpp)	4.02	7.18	20.51	19.62±3.01	0.80±0.09	544.7	3.5
EntroPack (7 bpp)	7.07	0.99	6.72	27.61±4.41	0.93±0.04	551.1	8.0
EntroPack (8 bpp)	8.06	0.54	3.92	29.14±3.35	0.95±0.03	555.6	10.8

Table 1: Weight compression, image fidelity, and runtime for Z-Image-Turbo. Network output error compares first-step denoising outputs with those of the BF16 model.

method	network bpp	rel. L_2 (%) ↓	WikiText-2 PPL ↓	prefill (ms / 32k tokens) ↓	decode (ms / 8 tokens) ↓	conversion (s) ↓
BF16 reference	16.00	0.00	5.98	18639	155.1	0.0
NF4 (bitsandbytes)	4.50	9.28	6.08	18634	210.8	0.3
FP4 (bitsandbytes)	4.50	12.41	6.23	18638	210.7	0.3
NVFP4 (torchao)	4.50	9.49	6.09	19727	1456.7	2.0
MXFP4 (torchao)	4.31	11.80	6.16	19693	1436.4	2.0
INT8 (torchao)	8.01	1.02	5.98	19203	277.2	0.7
FP8 (torchao)	8.01	2.65	5.98	18869	468.4	0.5
INT8 convrot (comfy-kitchen)	8.01	0.99	5.98	16666	161.0	0.2
FP8 (comfy-kitchen)	8.00	2.65	5.98	13210	152.3	0.1
HyperQuant (4 bpp)	4.09	7.31	6.04	27711	2829.4	149.6
SDNQ (4 bpp)	4.50	11.12	6.24	18635	230.8	0.9
HRTN (4 bpp)	4.05	7.44	6.07	18828	454.4	109.1
EntroPack (4 bpp)	4.03	7.19	6.06	18709	272.8	5.8
EntroPack (7 bpp)	7.10	0.98	5.98	18720	321.5	12.6
EntroPack (8 bpp)	8.08	0.54	5.98	18726	341.1	17.3

Table 2: Weight reconstruction, WikiText-2 perplexity, and runtime on Qwen3.8-27B.

reference's 18,639 ms. Autoregressive decoding has less computation over which to amortize weight reconstruction. Our workload uses the checkpoint's Next-N module to draft tokens and the main model to verify eight positions, assuming full draft acceptance at a 32k-token context. The 4 bpp configuration takes 272.8 ms, versus 155.1 ms for BF16.

5 Conclusion

We presented EntroPack, a calibration-free weight compression method that combines arbitrary target bitrates with efficient GPU reconstruction. Lattice quantization and conditional entropy modeling define the coded representation, and sampled storage estimates guide rate selection without repeated full-stream encoding. The resulting fields are encoded in independent rANS tiles, allowing symbol decoding and weight reconstruction to be fused on the GPU without intermediate symbol tensors or reconstruction codebooks. By separating storage bitrate from numerical precision, this representation supports floating-point and integer compute formats. The same codec can also compress FP8- or INT8-quantized weights and reconstruct them for low-precision inference. Experiments on diffusion and language models show accurate bitrate control and lower weight error than the tested fixed-width formats at comparable storage, with fast encoding and modest inference overhead on compute-intensive workloads such as diffusion denoising and language model prefill.

AI USE STATEMENT

We used generative AI tools to assist with parts of the algorithm implementation and runtime optimization, literature retrieval, review of theoretical claims, and manuscript editing. The authors are responsible for the final implementation, experimental results, and content of the paper.

REFERENCES

Eduardo Alvarez, Omri Almog, Eric Chung, Simon Layton, Dusan Stosic, Ronny Krashinsky, and Kyle Aubrey. Introducing NVFP4 for Efficient and Accurate Low-Precision Inference. NVIDIA Technical Blog, June 2025. URL https://developer.nvidia.com/blog/?p=102000.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees. In *Advances in Neural Information Processing Systems*, volume 36, pp. 4396–4429, 2023. doi: 10.52202/075280-0196.

Jiale Chen, Yalda Shabanzadeh, Elvir Crnčević, Torsten Hoefler, and Dan Alistarh. The Geometry of LLM Quantization: GPTQ as Babai's Nearest Plane Algorithm. In *The Fourteenth International Conference on Learning Representations*, 2026.

Philip A Chou, Tom Lookabaugh, and Robert M Gray. Entropy-constrained vector quantization. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(1):31–42, 1989.

Comfy Org. Comfy Kitchen, 2025. URL https://github.com/Comfy-Org/comfy-kitchen. Software. Accessed 2026-09-22.

J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der mathematischen Wissenschaften*. Springer, New York, 3rd edition, 1999. doi: 10.1007/978-1-4757-6568-7.

John Conway and Neil Sloane. Fast quantizing and decoding algorithms for lattice quantizers and codes. *IEEE Transactions on Information Theory*, 28(2):227–232, 1982.

David Corvoysier and Hugging Face contributors. Optimum Quanto, 2026. URL https://github.com/huggingface/optimum-quanto/tree/2290af22625527d0c91ff4862d750b573d2fb0ac. Software, revision dated 2026-08-19.

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. October 2023.

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*, volume 35, pp. 30318–30332, 2022. doi: 10.52202/068431-2198.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, volume 36, pp. 10088–10115, 2023. doi: 10.52202/075280-0441.

Disty0. SDNQ: SD.Next Quantization Engine, 2026. URL https://github.com/Disty0/sdnq/tree/06c83b3878d7240da5d8b8c94ae798af9a047e6b. Software, version 0.2.7, revision dated 2026-08-30.

Yuval Domb, Hadar Sackstein, and Tomer Solberg. HyperQuant: a rate-distortion-optimal quantization pipeline for large language and diffusion models. *arXiv preprint arXiv:2606.23406*, 2026.

Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 12284–12303. PMLR, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323*, 2022.

ggml contributors. GGUF, 2023. URL `https://github.com/ggml-org/ggml/blob/master/docs/gguf.md`. Format specification. Accessed 2026-09-22.

Robert M. Gray and David L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, 1998. doi: 10.1109/18.720541.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

Yongchang Hao, Yanshuai Cao, and Lili Mou. NeuZip: memory-efficient training and inference with dynamic compression of neural networks. In *NeurIPS 2024 Workshop on Efficient Natural Language and Speech Processing (ENLSP-IV)*, 2024.

Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Or Ozeri, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, et al. ZipNN: lossless compression for AI models. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 186–198. IEEE, 2025.

Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2023.

Jeff Johnson. DietGPU: GPU-based lossless compression for numerical data, 2022. URL `https://github.com/facebookresearch/dietgpu`. Software. Accessed 2026-09-22.

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 on-device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, volume 6, pp. 87–100, 2024.

Peter Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 12 2014. doi: 10.1109/TVCG.2014.2346458.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu, Neil Burgess, Sangwon Ha, Richard Grisenthwaite, Naveen Mellempudi, Marius Cornea, Alexander Heinecke, and Pradeep Dubey. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

MiniMax. MiniMax-H3, 2026. URL `https://www.modelscope.cn/models/MiniMax/MiniMax-H3`. Model card. Accessed 2026-09-22.

ModelScope Team. DiffSynth-Studio, 2024. URL `https://github.com/modelscope/DiffSynth-Studio`. Software. Accessed 2026-09-22.

Andrew Or, Apurva Jain, Daniel Vega-Myhre, Jesse Cai, Charles David Hernandez, Zhenrui Zheng, Driss Guessous, Vasiliy Kuznetsov, Christian Puhrsch, Mark Saroufim, et al. Torchao: Pytorch-native training-to-serving model optimization. *arXiv preprint arXiv:2507.16099*, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8024–8035, 2019.

Patrick Putzky, Martin Genzel, Mattes Mollenhauer, Sebastian Schulze, Thomas Wollmann, and Stefan Dietzel. Float8@2bits: Entropy Coding Enables Data-Free Model Compression. *Preprint arXiv:2601.22787*, 2026.

Qwen Team. Qwen3.8-Max: A New Bar for Coding and Cowork, August 2026. URL `https://qwen.ai/blog?id=qwen3.8`.

Hongshi Tan, Yao Chen, Gustavo Alonso, Weng-Fai Wong, and Bingsheng He. Approaching Shannon bound with lossless LLM weight compression. *arXiv preprint arXiv:2606.15789*, 2026.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: even better LLM quantization with Hadamard incoherence and lattice codebooks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 48630–48656, 2024a.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: quantization with trellises and incoherence processing. In *Advances in Neural Information Processing Systems*, volume 37, pp. 59597–59620, 2024b.

Tycho F. A. van der Ouderaa, Mart van Baalen, Paul Whatmough, and Markus Nagel. Leech lattice vector quantization for efficient LLM compression. In *ICML 2026 Workshop on Resource-Adaptive Foundation Model Inference (AdaptFM)*, 2026.

Simon Wiedemann, Heiner Kirchhoffer, Stefan Matlage, Paul Haase, Arturo Marban, Talmaj Marinc, David Neumann, Ahmed Osman, Detlev Marpe, Heiko Schwarz, et al. DeepCABAC: Context-adaptive binary arithmetic coding for deep neural network compression. In *Joint ICML Workshop on On-Device Machine Learning and Compact Deep Neural Network Representations (ODML-CDNNR)*, pp. 1–4, 2019.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clément Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: state-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.

Z-Image Team. Z-Image: An Efficient Image Generation Foundation Model with Single-Stream Diffusion Transformer. *arXiv preprint arXiv:2511.22699*, 2025.

Tianyi Zhang, Mohsen Hariri, Shaochen Henry 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*, volume 38, pp. 98966–98994, 2025a.

Xi Zhang, Xiaolin Wu, Jiamang Wang, and Weisi Lin. Learning grouped lattice vector quantizers for low-bit LLM compression. In *Advances in Neural Information Processing Systems*, volume 38, pp. 110672–110698, 2025b.

APPENDIX

Appendix A presents the evaluation protocol, video and audio results, design ablations, and codec measurements. Appendix B studies low-precision inference and per-layer storage allocation. Appendix C gives search settings, the RDO algorithm, and GPU implementation details. Appendix D discusses limitations and future research directions.

A SUPPLEMENTARY EXPERIMENTS

This section provides the evaluation protocol and supplementary results on video and audio generation. The ablations measure the storage savings from the entropy model and the error–time trade-off of per-row refinement. We also characterize GPU reconstruction under different tile sizes and workloads, and evaluate compression across floating-point and integer dtypes.

A.1 EVALUATION DETAILS

For each model, methods share a weight population and quality evaluator. Runtime uses each method's execution path, including weight reconstruction where required. These settings apply to all experiments in the main paper and appendix.

Quantized populations. Z-Image-Turbo evaluations cover 276 linear-weight tensors in the diffusion transformer, excluding the text encoder and other modules. MiniMax-H3 evaluations cover the 259 linear-weight tensors in its quantization configuration. For Qwen3.8-27B, we compress 518 quantizable two-dimensional weight tensors from the trunk, visual tower, and Next-N block, excluding the token embedding and output projection. Unless noted, layer-level experiments use the 10240×3840 Z-Image-Turbo weight `noise_refiner.0.feed_forward.w1`, called the reference layer. RDO is disabled except in its dedicated ablations.

Baseline handling. To keep the evaluated population fixed, weights unsupported by a format remain at BF16 and count toward its storage and element totals. This affects 27 Qwen visual-tower layers with widths incompatible with MXFP4, and one MiniMax layer for HRTN. Comfy-kitchen INT8 and FP8 quantize both weights and activations. The other methods quantize weights only.

Image fidelity. Z-Image-Turbo generations use eight denoising steps, two prompts, and seeds 42, 7, and 123. Each compressed-weight output is compared with the BF16 output for the same prompt and seed. PSNR and SSIM summarize image fidelity as mean $\pm$ standard deviation over these six pairs, while network output error compares the first-step denoising outputs. The BF16 reference's 99 dB PSNR is the metric's zero-error cap. Artifacts contain prompts and per-seed measurements.

Video and audio fidelity. For MiniMax-H3, we generate 120 frames at 768×1344 using 20 denoising steps and seeds 0, 1, and 2. Video fidelity is measured by PSNR and SSIM against matched BF16 outputs. Audio is extracted from the generated files and compared through log-mel spectrogram relative L_2 error. We report mean and standard deviation over seeds, with the same 99 dB PSNR cap for the reference. Prompts and per-seed results accompany the artifacts.

Language model quality. We evaluate Qwen3.8-27B perplexity on WikiText-2 raw test text with the checkpoint's tokenizer and the Transformers library (Wolf et al., 2020). For this quality comparison, each method's reconstructed weights are inserted into the BF16 model. The shared evaluator uses 1,024-token target windows, a stride of 512 tokens, and up to 512 preceding tokens of context. Runtime uses each method's execution representation for the workloads below.

Diffusion runtime. Diffusion inference columns report GPU device time per denoising step. Each method uses its native quantized matrix product when available for the workload and otherwise reconstructs weights on each call. Step time includes this reconstruction and averages steps 2 and 3 of a three-step timing run. It excludes prompt encoding, the first step, final image, video, or audio decoding, and MiniMax transfers between pipeline stages.

Language model runtime. We time prefill and autoregressive decoding with a 32k-token context. Prefill measures one pass over the context. Autoregressive timing combines seven Next-N draft steps with a trunk pass verifying eight positions, assuming full draft acceptance. Both include weight reconstruction required by the method's execution path.

Conversion and codec time. Conversion time sums synchronized wall-clock measurements of per-layer weight quantization and compression after warm-up. For the dtype and tile-size experiments, codec timings use CUDA events around complete compression or decompression calls. We report the minimum of fifteen measurements after three warm-up runs.

A.2 Additional diffusion results: MiniMax-H3

MiniMax-H3 extends the diffusion evaluation to joint video and audio generation. Table 3 compares weight compression, output fidelity, and runtime under the evaluation protocol above. Video and audio outputs are compared with BF16 generations using matched prompts and seeds.

EntroPack achieves weight reconstruction close to the tested entropy-coded baselines while requiring less conversion time. At the 4 bpp target, it stores 4.03 bpp with 7.11% weight error, compared with HRTN and HyperQuant at 4.17 and 4.21 bpp. These baselines provide higher video and audio fidelity at substantially higher conversion costs. At targets of 7 and 8 bpp, EntroPack's weight error falls to 0.97% and 0.53%, with corresponding audio log-mel errors of 14.48% and 11.39%.

The longer denoising steps also make reconstruction a smaller fraction of runtime than on Z-Image-Turbo. EntroPack adds 0.5–0.8% to BF16 step time across these targets. Model conversion takes 4.9–14.2 s, between the tested fixed-width methods and the HRTN and HyperQuant configurations.

method	bpp	weights	video		audio	inference	conversion
		rel. L_2 (%) ↓	PSNR ↑	SSIM ↑	log-mel rel. L_2 (%) ↓	step (ms) ↓	(s) ↓
BF16 reference	16.00	0.00	99.00	1.00	0.00	26209.4	0.0
NF4 (bitsandbytes)	4.50	9.14	15.86±2.34	0.60±0.10	34.14±3.26	26269.1	0.4
FP4 (bitsandbytes)	4.50	12.03	14.05±2.45	0.52±0.10	29.94±5.10	26274.0	0.5
NVFP4 (torchao)	4.50	9.77	16.79±2.04	0.67±0.07	28.66±2.24	27785.9	2.7
MXFP4 (torchao)	4.25	11.51	15.19±2.27	0.59±0.08	27.42±4.40	27739.9	2.7
INT8 (torchao)	8.01	0.98	24.98±1.84	0.89±0.05	13.98±6.49	26864.5	0.9
FP8 (torchao)	8.07	2.60	20.27±1.48	0.80±0.06	21.79±2.56	26599.5	0.7
INT8 convrot (comfy-kitchen)	8.01	0.89	25.25±0.28	0.91±0.02	15.67±7.83	24285.8	0.3
FP8 (comfy-kitchen)	8.00	2.65	18.95±2.48	0.76±0.08	20.43±5.17	21232.1	0.2
HyperQuant (4 bpp)	4.21	7.31	17.28±2.08	0.70±0.10	24.91±1.85	32312.4	201.1
SDNQ (4 bpp)	4.50	10.24	14.00±0.33	0.52±0.03	25.03±1.49	26350.6	1.3
HRTN (4 bpp)	4.17	6.84	17.51±1.94	0.72±0.03	25.92±2.73	26581.3	139.6
EntroPack (4 bpp)	4.03	7.11	15.94±3.35	0.63±0.13	26.66±2.40	26338.5	4.9
EntroPack (7 bpp)	7.10	0.97	25.10±1.89	0.91±0.03	14.48±7.85	26421.9	10.8
EntroPack (8 bpp)	8.08	0.53	24.68±4.29	0.89±0.06	11.39±4.40	26431.4	14.2

Table 3: Weight compression, video and audio fidelity, and runtime on MiniMax-H3. Fidelity metrics are reported as mean ± standard deviation over three seeds.

A.3 Entropy-model ablation

Coset conditioning captures differences between the two cosets' coordinate distributions, while parity reduction exploits the constraint on the eighth coordinate. We isolate their storage contributions by comparing coding configurations with identical reconstructed weights.

For each target of 2, 4, and 8 bpp, we compress Z-Image-Turbo's reference linear layer with EntroPack. We decode its integer fields and re-encode the lattice points under three configurations, retaining the row scales and output dtype. Tile capacity is 16,384 symbols, and probability-table precision is 11, 11, and 12 bits at the three targets, respectively. The variants reuse the baseline quantization scale s without a new search, so reconstruction is identical within each column. Column labels give the baseline's target rate, while actual stored rates may differ.

EntroPack uses the parity-reduced fields $(c, z_1, \ldots, z_7, m)$ and seventeen probability tables: one for c and two for each coordinate field. The *9 pooled tables* variant keeps these fields but merges the two coset-specific tables for each coordinate, removing coset conditioning. The *direct* z_8 variant

keeps coset conditioning but replaces m with z_8, removing the parity reduction in equation 4. Each variant preserves the lattice points and builds probability tables from its own fields. We verify exact recovery of the input fields from all nine encoded rANS streams.

Table 4 reports bpp calculated from each representation's actual storage, including coding metadata. At fixed reconstruction error, coset conditioning gives its largest saving at the 2 bpp target, reducing stored rate from 2.07 to 2.02 bpp. Parity reduction saves 0.10–0.13 bpp across the evaluated targets by coding m instead of the eighth integer coordinate z_8.

source model	target 2 bpp	target 4 bpp	target 8 bpp
9 pooled tables (no coset conditioning)	2.07	4.03	8.01
direct z_8 (no parity reduction)	2.12	4.15	8.13
EntroPack (17 tables + parity reduction)	2.02	4.02	8.00

Table 4: Measured stored rates for coding ablations on Z-Image-Turbo's reference linear layer. Each column uses the same quantized points and row reconstruction scales.

A.4 Optional per-row rate–distortion refinement

Per-row refinement seeks lower weight error through additional encoding work. Candidate count controls the available row resolutions, while sweep count controls updates to the row allocation and shared probability model. We vary these settings separately on Z-Image-Turbo's linear weights at a 3 bpp target. Allocations use the single-scale baseline's estimated storage budget, and we compare reconstruction error and encoding time at similar stored rates.

With five candidates per row, Figure 3 varies the number of allocation sweeps from the single-scale baseline at zero sweeps. Most of the observed improvement occurs in the first sweep: weight error falls from 14.35% to about 14.11%, with little change thereafter. Additional sweeps continue to increase encoding time, reaching 26.3 s at eight sweeps.

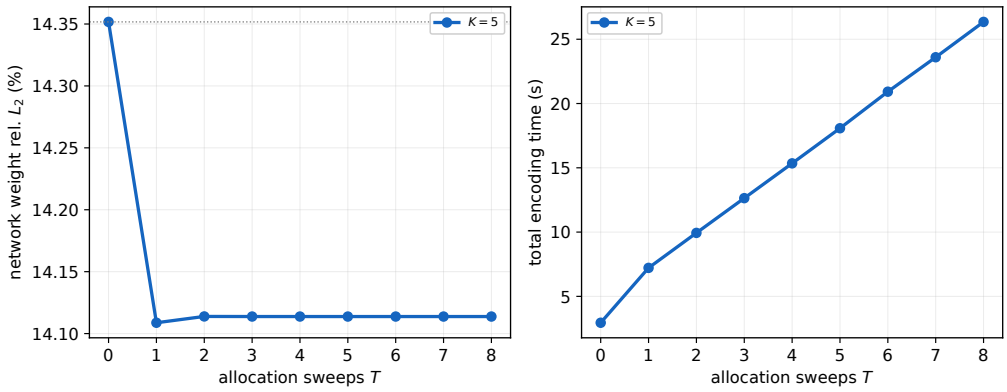

Figure 3: RDO sweep count on Z-Image-Turbo at a target of 3 bpp with five candidate scales. Left: network weight error. Right: total model encoding time.

We then hold the sweep count at one and vary the number of candidates in Figure 4. Candidate scale ratios form nested sets within the fixed empirical range $[0.70, 1.45]$ relative to the baseline scale. Larger sets reduce error in this experiment, with diminishing gains: $K = 5$ gives 14.11% error and $K = 20$ gives 14.01%. Encoding time rises from 4.2 s at $K = 1$ to 17.5 s at $K = 20$.

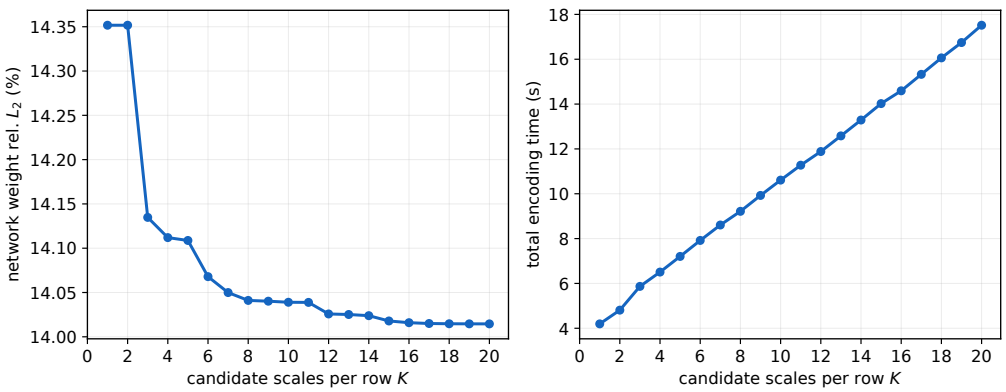

Figure 4: RDO candidate count on Z-Image-Turbo at a target of 3 bpp with one allocation sweep. Left: network weight error. Right: total encoding time.

A.5 Reconstruction cost and tile overhead

Inference overhead depends on decoder speed and the computation that reuses each weight. Figure 5 uses the reference layer compressed at a 4 bpp target.

The left panel of Figure 5 measures how tile size affects decoding throughput and stored rate. Throughput counts reconstructed BF16 bytes per second, and stored rate includes all metadata. Over this sweep, smaller tiles decode faster but use more storage for terminal states and offsets, reflecting the cost of exposing more independent decoding work.

The right panel of Figure 5 holds the compressed weights fixed and varies the number of input tokens processed in one linear-layer call. For each input size, we compare a BF16 matrix product using resident, uncompressed weights with a call that first reconstructs the weights and then performs the same matrix product. The vertical axis reports the percentage increase in total call time relative to the uncompressed case. This overhead falls from 132% at 256 tokens to 9.4% at 4096 tokens and 2.6% at 16,384 tokens. Larger matrix products amortize reconstruction over more computation.

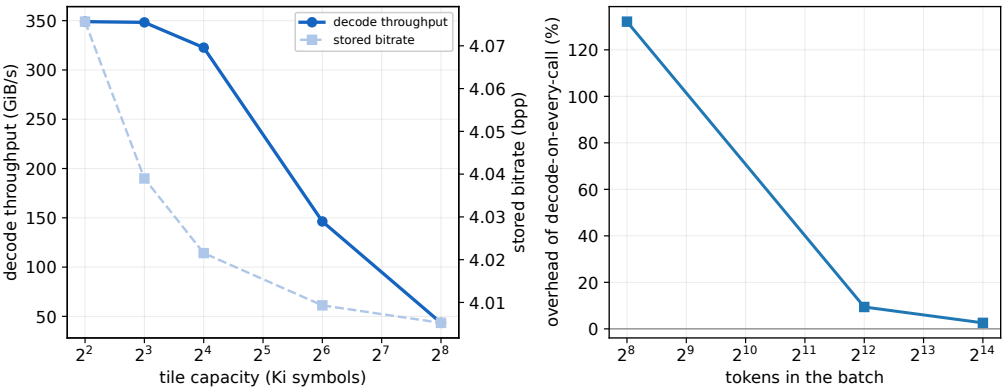

Figure 5: Reconstruction cost on the reference layer at a 4 bpp target. Left: decode throughput and stored rate versus tile size. Right: linear-layer latency overhead relative to resident BF16 weights, versus the number of input tokens processed together.

Table 5 quantifies the tile metadata cost under the default configurations. Each tile stores terminal rANS states and a payload offset so that decoding can start independently. The table reports this tile metadata storage in bytes and as a fraction of total compressed size. It occupies 1.80–3.60% of storage across the displayed targets and is included in every reported bitrate.

target bpp	tile (symbols)	tile (vectors)	tiles	b	tile metadata (bytes)	tile metadata (%)
2	32,768	3,640	1,351	11	178,336	1.80
4	16,384	1,820	2,701	11	356,536	1.80
7	8,192	910	5,402	11	713,068	2.06
8	4,096	455	10,803	12	1,426,000	3.60

Table 5: Tile configurations and metadata storage on the reference layer. Tile metadata comprises rANS states and payload offsets, with percentages relative to total stored bytes.

A.6 Numerical dtype measurements

EntroPack controls storage bitrate separately from the dtype of the reconstructed weights. To evaluate this property, Table 6 reports compression of seven dtype versions of the reference layer. FP8 inputs use per-row scaling, and integer inputs use per-tensor scaling before casting, with an offset for unsigned values. Each tensor is reconstructed in its input dtype and compared with that input, so the reported error measures compression after the initial format conversion.

Stored rates remain close to targets across these containers. Encoding takes 11.0–36.3 ms and decoding 0.23–0.41 ms on this layer.

container	target bpp	stored bpp	rel. L_2 (%) ↓	encode (ms) ↓	decode (ms) ↓
FP32	4.00	4.02	7.19	11.7	0.31
FP32	8.00	8.05	0.51	35.5	0.41
FP16	4.00	4.02	7.19	11.7	0.31
FP16	8.00	8.05	0.51	36.3	0.40
BF16	4.00	4.02	7.19	11.5	0.23
BF16	8.00	8.05	0.53	36.2	0.34
FP8 E4M3FN	4.00	4.02	7.54	11.7	0.32
FP8 E5M2	4.00	4.02	7.15	11.8	0.31
INT8	4.00	4.02	8.17	11.3	0.32
UINT8	4.00	4.02	7.76	11.0	0.32

Table 6: Stored rate, reconstruction error, and codec time for seven dtypes on the reference layer. Errors use each dtype's input as the reference, with the UINT8 offset removed.

B Additional applications

EntroPack can combine compressed storage with low-precision computation and can assign different rates to individual layers. We evaluate these two uses on Z-Image-Turbo, first retaining FP8 or INT8 matrix products and then reallocating a fixed storage budget among layers.

B.1 FP8 and INT8 computation with compressed storage

The dtype measurements above concern reconstruction of individual tensors. Here we evaluate whether compressed FP8 and INT8 weights retain the runtime benefits of low-precision computation during image generation. We first convert Z-Image-Turbo's linear weights to row-scaled FP8 or INT8, then compress them with EntroPack. During inference, weights are reconstructed in those formats for low-precision matrix products. Table 7 compares these routes with uncoded FP8 and INT8, direct BF16 compression, and the uncompressed BF16 model.

Both compressed low-precision routes reduce weight storage while preserving faster denoising than the BF16 reference. At the 4 bpp target, they store 4.02 bpp, approximately half the 8.01 bpp of their uncoded counterparts. Denoising steps take 392 and 393 ms, compared with 506 ms for BF16 and 343 ms for uncoded FP8. These times include weight reconstruction and matrix multiplication.

Weight error uses the original BF16 weights as its reference. For FP8 and INT8, it includes initial format conversion and compression. At the 6 bpp target, direct BF16 compression has 1.89% weight error, compared with 3.11% for the FP8 route and 2.37% for the INT8 route.

route	stored bpp	rel. L_2 (%) ↓	PSNR ↑	step (ms) ↓
BF16 reference	16.00	0.00	99.00	505.7
FP8 e4m3 container, uncoded	8.01	2.65	22.21	343.1
INT8 container, uncoded	8.01	1.07	20.10	342.5
EntroPack (4 bpp), FP8 container	4.02	7.98	19.52	391.7
EntroPack (4 bpp), INT8 container	4.02	7.34	18.88	392.5
EntroPack (4 bpp), direct BF16	4.02	7.18	19.62	544.7
EntroPack (6 bpp), FP8 container	6.06	3.11	22.31	392.2
EntroPack (6 bpp), INT8 container	6.06	2.37	20.12	392.0
EntroPack (6 bpp), direct BF16	6.06	1.89	26.34	548.0

Table 7: Storage, weight error, image fidelity, and runtime with compressed FP8 and INT8 weights on Z-Image-Turbo. Fidelity uses the uncompressed BF16 model as the reference.

B.2 Aggressive network-wide weight compression

At very low bitrates, a uniform target may spend too little storage on layers that strongly affect image structure. Table 8 compares uniform and mixed allocations of the same budget to Z-Image-Turbo's linear weights. The mixed allocation assigns 3 bpp to modulation, embedding, and output layers and approximately 2.47 bpp to the remaining layers. Both use approximately 2.5 stored bpp overall. At this matched storage, the mixed allocation raises mean SSIM from 0.64 to 0.67.

allocation	network bpp	rel. L_2 (%) ↓	PSNR ↑	SSIM ↑
uniform 2.5 bpp	2.50	20.64	16.82	0.64
sensitive@3 + rest@2.47	2.50	20.76	16.42	0.67

Table 8: Uniform and mixed per-layer rate allocation on Z-Image-Turbo at approximately 2.5 stored bpp. Both allocations are evaluated against the same BF16 reference.

Figure 6 illustrates the effect of the allocation with generations using the same prompt and seed. In this example, the mixed allocation produces more consistent illumination on the face and raised hand, improving the visual coherence of the generated image.



(a) Uniform allocation (b) Mixed allocation

Figure 6: Z-Image-Turbo generations with uniform and mixed rate allocation at 2.50 stored bpp, using the same prompt, random seed, and denoising settings.

C ADDITIONAL METHODOLOGICAL DETAILS

We specify the search settings, per-row allocation algorithm, and GPU coding implementation.

C.1 SCALE-SEARCH SETTINGS

The scale search in Algorithm 1 uses twelve geometric-bisection steps on $[10^{-3}, 8]$ and a nominal sample budget of 2^{18} eight-dimensional vectors. To compare candidate scales on the same data, we sample complete rows with a shape-dependent seed, holding them fixed throughout the search.

Probability-table precision b determines the number of distinct symbols that can receive positive frequencies. We start at 11 bits for targets up to 7 bpp and at 12 bits above that. If a full-tensor field alphabet exceeds 2^b, we increase b and repeat scale selection, up to 15 bits. If the alphabet still exceeds this capacity, we coarsen the scale by setting $s \leftarrow 1.05\, s\, A_{\max}/2^b$ and requantizing until the alphabets fit, where $A_{\max}$ is the largest coordinate alphabet.

C.2 RDO ALLOCATION ALGORITHM

Algorithm 2 details the refinement in §3.5 under the baseline's estimated storage budget. Candidate fields, fitted row scales, and distortions are precomputed. Each sweep updates the shared probability model and selects row candidates through a common rate-penalty search.

Algorithm 2 Per-row refinement using the baseline storage budget.

Require: Weights W, baseline (s, b), ratios $\{\rho_k\}$ including 1, sweeps T
For each ρ_k, quantize at $\rho_k s$ and fit row scales by equation 13
Record candidate fields, scales $\sigma_{r,k}$, and squared errors $D_{r,k}$
Initialize all $k(r)$ to the baseline; let B be its estimated stored bytes
for $t = 1, \ldots, T$ **do**
    Fit the shared model q_θ to the current row selections
    $R_{r,k} \leftarrow -\sum_{v \in r} \log_2 q_\theta(c_v^{(k)}, a_v^{(k)})$ for every (r, k)
    Estimate the current mixture's stored bytes $\widehat{B}_{\mathrm{mix}}$
    $R_{\mathrm{budget}} \leftarrow \sum_r R_{r,k(r)} + 8(B - \widehat{B}_{\mathrm{mix}})$
    Define $\kappa_\lambda(r) = \arg\min_k [D_{r,k} + \lambda R_{r,k}]$, $\quad L(\lambda) = \sum_r R_{r,\kappa_\lambda(r)}$
    $\widehat{k} \leftarrow \kappa_0$
    if $L(0) > R_{\mathrm{budget}}$ **then**
        Set $\lambda_{\mathrm{lo}} = 0$ and bracket λ by doubling a positive upper bound
        for $j = 1, \ldots, 20$ **do**
            $\lambda \leftarrow \lambda_{\mathrm{hi}}/2$ if $\lambda_{\mathrm{lo}} = 0$, otherwise $\sqrt{\lambda_{\mathrm{lo}} \lambda_{\mathrm{hi}}}$
            Raise λ_{lo} to λ if $L(\lambda) > R_{\mathrm{budget}}$; otherwise lower λ_{hi} to λ and retain $\widehat{k} \leftarrow \kappa_\lambda$
        end for
    end if
    $k \leftarrow \widehat{k}$
end for
return the selected fields and row scales $\{\sigma_{r,k(r)}\}$

C.3 TILED RANS IMPLEMENTATION

Tile layout and storage. Independent tile decoding requires a saved set of rANS states and a location in the payload. Each tile has $J = 32$ 32-bit terminal states and a 32-bit payload offset, requiring $4J + 4 = 132$ bytes of tile metadata. For a configured size of S symbols, a tile holds $V = \max(32, \lfloor S/9 \rfloor)$ eight-dimensional vectors. Table 5 lists the defaults, from $S = 32768$ at 2 bpp to $S = 4096$ at 8 bpp. The tensor also stores a final boundary offset and FP32 row scales.

State normalization. The implementation uses 32-bit rANS states maintained in $[2^{15}, 2^{31})$ and 16-bit payload words. Renormalization transfers words between the state and payload to preserve

this range. The supported probability precisions $b \in \{9, \ldots, 15\}$ give table frequencies $M = 2^b \leq 2^{15}$, so at most one payload word is needed per symbol.

Decoder lookup tables. Lookup tables recover a symbol and its frequency interval from the low bits of the current rANS state. They reside in shared memory or packed global memory, depending on alphabet size and available device resources.

D LIMITATIONS AND FUTURE WORK

EntroPack's flexible storage rates require additional work during model conversion and inference. Scale search, probability modeling, and entropy coding increase conversion cost relative to a single fixed-width quantization pass. Reducing repeated data passes could lower this cost. During inference, the decoder materializes a dense weight tensor before each matrix product, making reconstruction harder to amortize in small workloads. Fusing entropy decoding and weight reconstruction with matrix multiplication could avoid this intermediate tensor and reduce memory traffic.

Activations and key–value caches are produced during inference and would require online compression. Extending EntroPack beyond static weights to these tensors would therefore require probability modeling and rate selection to track changing distributions with low latency.