TO COMPRESS OR NOT? PUSHING THE FRONTIER OF LOSSLESS GENAI MODEL WEIGHTS COMPRESSION WITH EXPONENT CONCENTRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The scaling of Generative AI (GenAI) models into the hundreds of billions of parameters makes low-precision computation indispensable for efficient deployment. We argue that the fundamental solution lies in developing low-precision *floating-point* formats, which inherently provide numerical stability, memory savings, and hardware efficiency without dequantization overhead. In this paper, we present a theoretical and empirical study of an *exponent concentration* phenomenon in GenAI weights: exponents consistently exhibit low entropy across architectures and modalities. We show that this arises naturally from α-stable distributions induced by stochastic gradient descent, and we prove tight bounds on the entropy of exponents. Our analysis establishes a theoretical compression limit near FP4.67, which motivates the design of a practical FP8 format. Building on these insights, we propose **Exponent-Concentrated FP8 (ECF8)**, a lossless compression framework with entropy-aware encoding and GPU-optimized decoding. Experiments on LLMs and DiTs up to 671B parameters demonstrate up to 26.9% memory savings and 177.1% throughput acceleration, with perfectly lossless computations, i.e., no deviation in model outputs. Our results establish exponent concentration as a statistical law of trained models and open a principled path for lossless low-precision floating-point design in the FP8 era.

1 INTRODUCTION

The rapid growth of generative AI (GenAI) models, from large language models (LLMs) (Dubey et al., 2024; Team et al., 2023; Hurst et al., 2024; Liu et al., 2024a; Yang et al., 2025) to diffusion transformers (DiTs) (Wan et al., 2025; Batifol et al., 2025; Wu et al., 2025), has led to parameter counts scaling into the hundreds of billions. Such parameter-intensive GenAI models make low-precision computation indispensable. General matrix multiplication (GeMM) (Dongarra et al., 1990; Goto & Geijn, 2008; Springer & Bientinesi, 2018) with reduced precision is the most straightforward way to achieve memory savings and possible acceleration, as it eliminates redundancy in weight representation while directly reducing hardware workloads.

Existing work has primarily focused on integer-based quantization (Zhang et al., 2024; Zhang & Shrivastava, 2024; Yao et al., 2022; Dettmers et al., 2022; Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2024a; Liu et al., 2024b). While effective in reducing model size, these approaches suffer from two fundamental drawbacks: (a) they are *lossy*, often introducing accuracy or generative quality degradation (Zhang et al., 2025); and (b) they incur efficiency penalties in large-batch inference due to the required dequantization procedure and mixed-precision execution (Lin et al., 2024b; Jin et al., 2024). Since integer tensors must be converted back into floating-point values before computing, throughput suffers, limiting their utility in high-performance deployment.

Motivated by the limitations of lossy quantization, DFloat11 (Zhang et al., 2025) revealed that the exponents of BF16 weights in LLMs have far lower entropy than the bitwidth allocated to store them, creating headroom for lossless compression via entropy coding. Yet this finding puzzled the community: Is there a fundamental principle that explains its success? Can the idea extend beyond BF16 to other formats? And most critically, can memory compression be transformed into actual

inference acceleration by getting fast decoding kernels? All of these unanswered questions can be answered positively by our paper.

In this paper, we present a theoretical and empirical analysis revealing an *exponent concentration* phenomenon in GenAI weights: exponents exhibit low entropy (Shannon, 1948) and cluster within narrow ranges across architectures and modalities. We trace this to the heavy-tailed dynamics of stochastic gradient descent (Amari, 1993), which lead neural network weights to follow α-stable distributions (Nikias & Shao, 1995). This provides a rigorous explanation of why exponents concentrate and allows us to bound their entropy. Our analysis proves that the ultimate compression limit corresponds to a floating-point format of roughly FP4.67. While such a fractional format is impractical in modern GPU hardware, it motivates our design of a powerful, lossless FP8 variant.

Building on these insights, we introduce **Exponent-Concentrated FP8 (ECF8)**, a novel lossless compression framework that encodes exponents with entropy-aware coding and efficient GPU decoding. We show that ***ECF8 can transform memory compression into inference acceleration***, delivering up to 26.9% memory savings and up to 177.1% throughput acceleration across state-of-the-art GenAI models, without any observed deviation in generation.

In summary, our contributions are:

- We provide a theoretical analysis of exponent concentration in GenAI weights, proving that α-stable distributions lead to bounded low-entropy exponents with a compression limit near FP4.67.
- We empirically validate exponent concentration across large LLMs and DiTs, showing that layer-wise entropy consistently lies around 2–3 bits.
- We design and implement **ECF8**, a lossless FP8 compression framework with encoding and GPU-optimized decoding, leading to inference acceleration.
- We demonstrate practical benefits on models up to 671B parameters, achieving significant memory reduction and throughput gains while preserving bit-exact fidelity.

2 Exponent Concentration Leads to Lossless Weight Compression In Generative AI

Preliminary. Floating-point numbers in IEEE-754 format consist of three components: a sign bit s, an exponent E, and a mantissa M. In the low-precision computing paradigm of deep learning, the exponents determine the dynamic range of representable values. If the exponent values are highly concentrated rather than spread uniformly, their entropy is low, which directly implies that fewer bits are needed for lossless representation. This motivates analyzing the statistical laws that govern exponent distributions in trained model weights.

2.1 Observation: Low-Entropy Exponents in Generative AI Model Weights

We empirically examine weight matrices from parameter-intensive GenAI models, including transformers used for language and vision tasks. Across diverse architectures (LLMs, diffusion models, and multimodal transformers), we consistently observe that the exponents of weight values occupy a very narrow range and have much lower entropy than expected under an alpha-stable assumption. As shown in Figure 1, histograms of exponent values cluster tightly around a few modes, and the measured Shannon entropy is typically close to 2-3 bits, in contrast to the 8 or more bits allocated in standard floating-point formats. This persistent low-entropy phenomenon strongly suggests an underlying distributional principle.

2.2 Analysis: Exponent Concentration of α-Stable Variables

Set-up and notations. Let $X \in \mathbb{R}$ be a continuous random variable representing a neural network weight. In the IEEE-754 floating-point representation, a nonzero real number x is encoded as:

$$x = (-1)^s \cdot 2^E \cdot M, \quad \text{with } E \in \mathbb{Z}, M \in [1, 2),$$

where s is the sign bit, E is the exponent, and M is the mantissa. We define:

$$E = \lfloor \log_2 |X| \rfloor ,$$

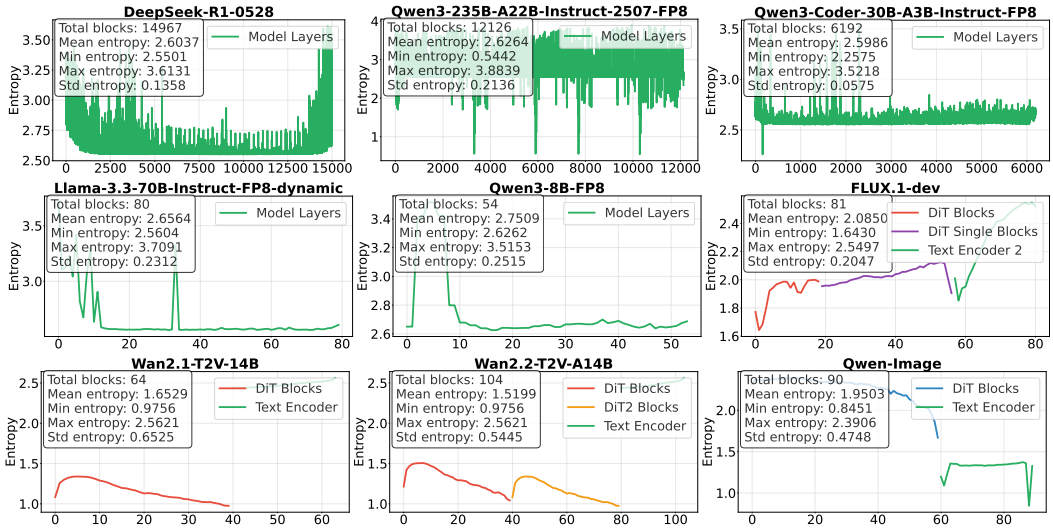

Figure 1: Entropy analysis across transformer blocks for different model architectures. The x-axis represents the block index within each model, and the y-axis shows the entropy values. Different colors indicate different block types within each architecture.

as the *floating-point exponent* of X. Our aim is to analyze the entropy and compression limit of E when X follows a symmetric α-stable distribution:

$$X \sim S_\alpha(\beta = 0, \gamma, \delta), \quad \text{with } \alpha \in (0, 2].$$

2.2.1 Why Neural Network Weights Follow α-Stable Distributions

Trained neural network weights are formed by accumulating stochastic updates over time. Under stochastic gradient descent (SGD), the update rule $\theta_{t+1} = \theta_t - \eta \cdot \nabla \mathcal{L}(\theta_t; \xi_t)$ introduces heavy-tailed noise from random mini-batch sampling. Empirical work shows that the gradient noise often exhibits power-law tails $\mathbb{P}(|\Delta_t| > x) \sim x^{-\alpha}$ with $\alpha < 2$. By the Generalized Central Limit Theorem, sums of such heavy-tailed variables converge to α-stable distributions. Thus, neural network weights after many updates approximately follow symmetric α-stable laws, providing the theoretical foundation for our analysis.

2.2.2 Exponent Entropy Concentration

Theorem 2.1 (Exponent Entropy Concentration). *Let $E = \lfloor \log_2 |X| \rfloor$ where $X \sim S_\alpha(\beta = 0, \gamma, \delta)$. Then E follows a discrete two-sided geometric distribution with parameter $q = 2^{-\alpha}$:*

$$\mathbb{P}(E = k) = \frac{1 - q}{1 + q} \cdot q^{|k|}, \quad k \in \mathbb{Z}.$$

The Shannon entropy of E is bounded by:

$$\frac{\alpha}{1 + 2^{-\alpha}} \leq H(E) \leq \frac{\alpha}{1 - 2^{-\alpha}}.$$

In particular, $H(E)$ is finite for all $\alpha > 0$.

Proof. For large $|x|$, the tails of an α-stable distribution behave like $\mathbb{P}(|X| > x) \sim C_\alpha x^{-\alpha}$. Thus, the probability of exponent $E = k$ is:

$$\mathbb{P}(E = k) \approx \mathbb{P}(2^k \leq |X| < 2^{k+1}) = C_\alpha 2^{-k\alpha}(1 - 2^{-\alpha}),$$

which corresponds to a two-sided geometric distribution. The entropy of this distribution is:

$$H(E) = -\sum_{k \in \mathbb{Z}} \mathbb{P}(E = k) \log_2 \mathbb{P}(E = k) = h_2\left(\frac{1 - q}{1 + q}\right) + \frac{2q}{1 + q} \cdot \frac{|\log_2 q|}{1 - q},$$

where $h_2(p) = -p \log_2 p - (1 - p) \log_2 (1 - p) \leq 1$ is the binary entropy. Plugging $q = 2^{-\alpha}$ and bounding terms gives the inequality. □

Interpretation. This theorem shows that the floating-point exponents of α-stable weights do not spread evenly across the integer line but instead concentrate around zero, decaying geometrically with rate $2^{-\alpha}$. The entropy bound demonstrates that this concentration is strong enough to guarantee finite entropy regardless of α, and tighter concentration (smaller α) leads to smaller entropy. Practically, this means that exponents carry very limited uncertainty, which enables efficient compression.

2.3 Impact: Exponent Concentration Guides Lossless Compression

Corollary 2.2 (Compression Limit)**.** *The minimal expected number of bits required to losslessly encode the exponent E is:*

$$L_{\min} = H(E).$$

Therefore, exponent values of neural network weights drawn from an α-stable distribution can be encoded in:

$$O\left(\frac{\alpha}{1-2^{-\alpha}}\right) \textit{ bits on average}.$$

Numerical instance ($\alpha = 2$). When $\alpha = 2$ (the Gaussian-like case), we have $2^{-\alpha} = 1/4$. The entropy bounds give:

$$1.6 \;\leq\; H(E) \;\leq\; 2.67.$$

Thus, the exponent itself has a compression limit of about 2.67 bits in the extreme case. However, a floating-point representation also requires one sign bit and several bits for the mantissa to preserve numerical precision. Even with a minimal mantissa allocation (e.g., $\sim$ 1 bit) plus the sign, the absolute floor is around:

$$2.67 \;+\; 1 \;+\; 1 \;\approx\; 4.67 \text{ bits}.$$

In practice, it is infeasible to implement a "FP4.67" or even FP5 format efficiently due to alignment and hardware constraints. Therefore, our proposed FP8 format (ECF8) represents a practical engineering choice: it is close to the entropy-driven theoretical limit, while retaining sufficient mantissa precision and hardware compatibility for efficient inference.

3 ECF8: Lossless LLM Weight Compression with Exponent Concentration

We present ECF8, a lossless compression algorithm that exploits the statistical properties of FP8 weights in pre-trained generative AI models. Our method consists of three core components: an encoding scheme based on Huffman coding, a parallel GPU decoding kernel for variable-length decoding, and a dynamic tensor management system that reduces memory footprint during inference.

3.1 Encoding

Our CUDA-based Huffman encoding pipeline transforms neural network weights into a compressed format optimized for parallel GPU decoding through three sequential stages. First, we generate optimal Huffman codes by analyzing weight exponent frequencies and constructing the corresponding binary tree. Second, we build hierarchical lookup tables that enable efficient variable-length code decoding using 8-bit subtables aligned with GPU memory architecture. Third, we encode weight exponents into a compressed bitstream while computing synchronization metadata that enables autonomous parallel decoding across GPU thread blocks.

Huffman code generation. Neural network weights in FP8 format allocate 4 bits for exponents, yet empirical analysis reveals that exponent entropy is substantially lower due to non-uniform frequency distributions. We exploit this entropy gap through Huffman coding, which assigns variable-length prefix-free codes with shorter sequences for frequent exponents. Our encoding procedure extracts exponents from model weights, computes their empirical frequency distribution $p(x)$ for exponent values $x \in \{0, 1, \ldots, 15\}$, and constructs the optimal Huffman tree minimizing expected code length $\mathbb{E}[\ell] = \sum_x p(x)\ell(x)$. To ensure GPU compatibility, we constrain maximum code length to 16 bits, requiring frequency adjustment for rare symbols while preserving near-optimality. This constraint is rarely violated in transformer layers empirically.

Hierarchical lookup table construction. We construct a multi-level lookup system that processes variable-length Huffman codes through sequential 8-bit operations. The system comprises two components: a *cascaded decode table* mapping codes to symbols, and a *length table* storing code lengths indexed by symbol value.

The cascaded structure organizes codes by their byte-aligned prefixes. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_k\}$ denote the set of all byte-aligned prefixes extracted from Huffman codes, ordered by length. For each prefix p_i of length $8j$ bits, we construct a lookup subtable LUT_i with 256 entries. Each entry $LUT_i[b]$ for byte value $b \in \{0, \ldots, 255\}$ contains either

- a decoded exponent $x \in \{0, \ldots, 15\}$ if the code $p_i \| b$, where $\|$ denotes concatenation, corresponds to a complete Huffman code,
- or a pointer value $256 - \text{index}(p_{i'})$ directing lookup to subtable $LUT_{i'}$ for longer prefix $p_{i'} = p_i \| b$.

This design bounds memory usage at $O(|\mathcal{P}| \cdot 256)$ entries while maintaining $O(\lceil \ell_{\max}/8 \rceil)$ lookup time for codes of length $\ell_{\max}$. The length table $L[x]$ stores the bit length of each symbol x, enabling proper bitstream advancement during decoding. Figure 2 illustrates the construction process of the *cascaded decode table* using a simplified example using the characters "a", "b", "c", "d", and "e" as symbols rather than exponents $x \in \{0, \ldots, 15\}$.

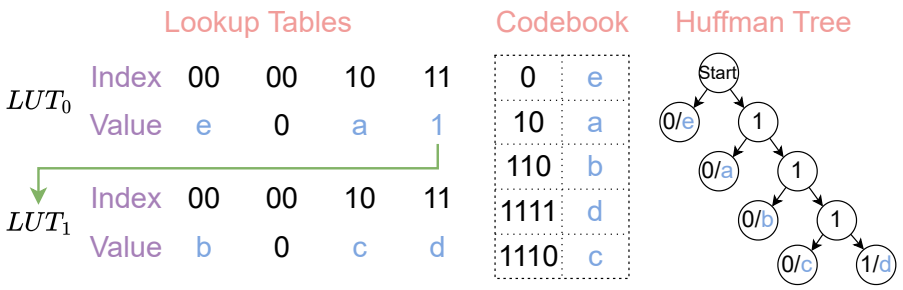

Figure 2: A simplified illustration of lookup table construction. A Huffman tree is built from the string "aaabbcddeeeee". Lookup tables are configured to be 2-bit, so each subtable has 4 entries. Codes for symbols "e" and "a" are at most 2 bits long and appear directly in the first table. In contrast, codes for "b", "c", and "d" exceed 2 bits and begin with "11", so entry "11" in the first table points to a secondary table. The pointer value is 1, indicating subtable 1. A second lookup in this subtable resolves the final symbol.

Encoding and synchronization metadata generation. We encode exponents into a compressed bitstream while generating coordination metadata for parallel GPU decoding. Given fixed parameters B (bytes per thread) and T (threads per block), we sequentially process symbols $x_1, x_2, \ldots, x_n$, concatenating their Huffman codes into a continuous bitstream and extracting complete bytes for storage. Since variable-length codes may span thread boundaries, we compute *gap values* to ensure proper synchronization. For thread t processing bytes $[tB, tB + 2)$, the gap g_t represents the bit offset from the previous thread:

$$g_t = \left(\sum_{i=0}^{t-1} \sum_{x_j \in \text{symbols}(i)} \ell(x_j) \right) \bmod 8B$$

where $\text{symbols}(i)$ denotes the set of symbols processed by thread i. Additionally, we compute *output positions* o_b for each thread block b, tracking the cumulative count of symbols processed by preceding blocks. This metadata enables autonomous thread block operation during parallel decompression without requiring inter-block synchronization.

3.2 Decoding

The decoding process reconstructs original exponent values from the Huffman-encoded bitstream using a highly optimized CUDA kernel that exploits GPU parallelism through hierarchical memory management and coordinated thread execution. The decoding algorithm operates in five distinct phases: **i. memory initialization**, where each thread allocates a register buffer of fixed size

(bytes per thread plus additional lookahead bytes) and the thread block establishes shared memory for coordination and output staging; **ii. data loading**, where threads load their assigned segments of the encoded bitstream from global memory into thread-local registers; **iii. parallel counting**, where each thread performs initial decoding based on gap values to determine the number of decodable symbols, followed by a parallel reduction algorithm across the thread block to compute cumulative symbol counts and establish output positions for each thread (ensuring threads write to non-overlapping memory regions); **iv. coordinated decoding**, where threads decode their full symbol sequences and write results to shared memory using the previously computed output positions; and **v. global memory write-back**, where decoded symbols are transferred from shared memory to global memory via coalesced writes. This approach minimizes global memory access overhead while maximizing parallelism through careful coordination of thread-local computation and block-level synchronization primitives. A detailed description of the decoding algorithm is provided in Algorithm 1.

3.3 Tensor Management

Our tensor management system enables on-demand weight decompression during model inference through strategic memory allocation and layer-wise processing. We implement a just-in-time decompression mechanism using PyTorch forward hooks that intercept layer execution and perform weight reconstruction immediately before computation. The system maintains a single pre-allocated GPU memory buffer of size equal to the largest layer's weight tensor, eliminating dynamic memory allocation overhead during inference. For each layer ℓ_i in the sequential model architecture, the forward hook invokes our CUDA decompression kernel to reconstruct weights W_i from their compressed representation and writes the result to the shared memory buffer. Upon completion of layer ℓ_i's forward pass, the buffer becomes available for layer ℓ_{i+1}, enabling memory-efficient inference with constant GPU memory overhead regardless of model depth.

4 Experiments

In this section, we test nine models spanning autoregressive language models, diffusion transformers, and mixture-of-experts variants from 8B to 671B parameters. We evaluate ECF8 across two critical dimensions: compression effectiveness and end-to-end inference performance. Specifically, we aim to address the following two fundamental research questions related to the deployment of production-scale GenAI models:

- **RQ1:** What memory reduction can production-scale transformers achieve while maintaining bit-exact weight reconstruction?
- **RQ2:** How does weight compression affect memory footprint and inference latency, can we achieve faster inference under the same memory budget?

4.1 Lossless Memory Saving for FP8 Weights

This section addresses **RQ1** regarding the rate of memory reduction for production-scale transformers. ECF8 achieves memory reductions from 9.8% to 26.9% across all evaluated models. LLMs demonstrate consistent reductions between 9.8% and 14.8%, while DiTs achieve higher compression ratios, with the Wan2.2-T2V-A14B model reaching 26.9% reduction. Figure 3 shows that the compression is lossless.

ECF8 delivers memory reductions from 9.8% to 26.9% across all evaluated models, with effectiveness correlating strongly with architecture type. Language models achieve moderate but consistent reductions between 9.8% and 14.8%, while diffusion transformers show substantially higher compression potential, with the Wan2.2-T2V-A14B model reaching 26.9% reduction.

This pattern validates our theoretical foundation: FP8 exponent distributions in trained networks contain exploitable redundancy through lossless compression. Compression effectiveness remains remarkably stable across model scales, from 8B parameter Qwen3-8B-FP8 to 671B parameter DeepSeek-R1-0528, indicating that ECF8's performance depends on fundamental weight distribution properties rather than model size.

Table 1: Memory savings and throughput improvements under fixed memory constraints. For throughput evaluation, DeepSeek-R1-0528 is tested on 8×H200 systems, Qwen3-235B-A22B-Instruct-2507-FP8 on 4×H200 systems, and the remaining models on single GH200 (96GB) systems. Memory savings and throughput improvements under fixed memory constraints demonstrate ECF8's practical deployment benefits.

Model	Memory Change (GB)	Memory ↓ (%)	Supported Machine	Throughput ↑ (%)
DeepSeek-R1-0528	623.19 → 530.26	14.8	8×H100 (80 GB)	150.3
Qwen3-235B-A22B-Instruct-2507-FP8	217.77 → 185.98	14.4	4×H100 (80 GB)	35.9
Llama-3.3-70B-Instruct-FP8-dynamic	63.76 → 54.69	13.4	1×H100 (80 GB)	11.3
Qwen3-Coder-30B-A3B-Instruct-FP8	27.85 → 23.69	14.3	1×RTX5090 (32 GB)	23.7
Qwen3-8B-FP8	6.47 → 5.61	9.8	1×RTX4070 (12 GB)	12.6
FLUX.1-dev	10.52 → 8.29	14.1	1×RTX4070 (12 GB)	177.1
Wan2.1-T2V-14B	17.40 → 12.65	25.4	1×RTX4080 (16 GB)	55.1
Wan2.2-T2V-A14B	30.49 → 21.85	26.9	1×RTX4090 (24 GB)	108.3
Qwen-Image	26.20 → 20.56	21.0	1×RTX4090 (24 GB)	126.6

The practical impact extends far beyond storage efficiency. Memory reductions enable deployment on lower-capacity hardware: the 14.8% reduction for DeepSeek-R1-0528 allows 8×H100 deployment instead of requiring 8×H200 or 2-node 8×H100 systems, while the 25.4% reduction for Wan2.1-T2V-14B fits within single RTX4080 constraints where uncompressed models exceed memory limits.

Under fixed memory constraints, ECF8 enables throughput improvements ranging from 11.3% to 177.1% by supporting larger batch sizes within the same memory budget. These gains compound ECF8's benefits beyond simple storage efficiency to deliver measurable inference performance improvements. Details of inference speed evaluation are presented in Section 4.2.

Figure 3: Images generated by ECF8-compressed Qwen-Image model, demonstrating pixel-perfect reconstruction quality compared to the original FP8 model. The images generated by the original FP8 model are shown in Figure 4.

Figure 3 provides direct evidence of ECF8's lossless compression property through visual quality assessment. The ECF8-compressed Qwen-Image model generated these images using identical random seeds and inference parameters as the uncompressed FP8 baseline. Each output is pixel-by-pixel identical to the original model's results, confirming zero quality degradation.

This lossless property proves critical for production deployment, where model behavior must remain completely unchanged while achieving significant memory savings. The diverse generated content demonstrates that ECF8 preserves full generative capabilities across different image domains and styles without introducing any artifacts or quality loss.

4.2 Improvement on Inference Speed under Fixed Memory Footprint

This section addresses **RQ2** regarding the inference acceleration undera fixed memory footprint. We evaluate inference speed improvements by measuring maximum achievable throughput within fixed memory budgets, simulating real-world scenarios where hardware capacity limits batch sizes.

In conclusion, the memory savings achieved by ECF8 translate directly into substantial inference performance improvements under realistic deployment constraints. The rationale behind this observation is that smaller memory footprints allow for larger batch sizes, which in turn enable higher throughput, as well as shorter per-request latency. For instance, under a 640 GB memory limit, ECF8 DeepSeek-R1-0528 sustains a batch size of 16, delivering 150.3% higher throughput and 60.1% lower per-request latency than FP8, which can only accommodate a batch size of 2.

Table 2: Comparison of FP8 and ECF8 across LLMs of varying scales. Under fixed memory constraints, we report per-request latency (s), throughput (tokens/s), and maximum batch size (each batch generates 1024 tokens). DeepSeek-R1-0528 and Qwen3-235B-A22B-Instruct-2507 are evaluated on H100 GPUs (141 GB), while other models are evaluated on a single GH200 GPU (96 GB).

Model / Constraint		Max Batch Size		Per Request Latency (s)			Throughput (tokens/s)		
Model	**Constraint**	**FP8**	**ECF8**	**FP8**	**ECF8**	**↓ (%)**	**FP8**	**ECF8**	**↑ (%)**
DeepSeek-R1-0528	640 GB	2	16	660.65	263.95	60.1	1.55	3.88	150.3
Qwen3-235B-A22B-Instruct-2507-FP8	240 GB	32	64	107.56	79.14	26.4	9.52	12.94	35.9
Llama-3.3-70B-Instruct-FP8-dynamic	80 GB	32	48	24.80	22.28	10.2	41.28	45.96	11.3
Qwen3-Coder-30B-A3B-Instruct-FP8	32 GB	16	32	107.33	86.70	19.2	9.54	11.80	23.7
Qwen3-8B-FP8	12 GB	16	24	4.90	4.35	11.2	208.80	235.22	12.6

Language model inference acceleration. Table 2 results demonstrate consistent throughput improvements across language models ranging from 11.3% to 150.3%. The DeepSeek-R1-0528 model shows the most dramatic improvement, achieving 150.3% higher throughput (3.88 vs 1.55 tokens/s) by enabling 8× larger batch sizes (16 vs 2) within the 640 GB memory constraint. This improvement stems from ECF8's ability to reduce the model's memory footprint from 623GB to 530GB, creating sufficient headroom for larger batches.

Smaller models exhibit more modest but still significant gains. The Qwen3-8B-FP8 model achieves 12.6% throughput improvement by increasing batch size from 16 to 24 within a 12GB constraint. Even with constrained memory budgets typical of consumer hardware, ECF8 consistently enables meaningful performance improvements through more efficient memory utilization.

Table 3: Comparison of FP8 and ECF8 across popular DiTs. Reported metrics include end-to-end (E2E) latency (s), step latency (ms), and GPU peak memory (MB). All experiments are conducted with the DiffSynth library on a single GH200 GPU (96 GB) using an identical batch size, random seed, and prompt. DiffSynth enables VRAM management by default, dynamically offloading and reloading model components between GPU and CPU to reduce peak memory usage.

Model	**DType**	**E2E Latency (s)**	**Step Latency (ms)**	**Memory (MB)**	**Memory ↓ (%)**	**Latency ↓ (%)**
FLUX.1-dev	ECF8	13.15 ± 0.08	438.4 ± 2.8	14274	12.1	45.9
	FP8	24.29 ± 0.10	809.5 ± 3.4	16243	0.0	0.0
Wan2.1-T2V-14B	ECF8	460.67 ± 0.92	9213.4 ± 18.5	18036	7.6	3.3
	FP8	476.21 ± 3.34	9524.3 ± 66.8	19529	0.0	0.0
Wan2.2-T2V-A14B	ECF8	461.41 ± 1.06	9228.2 ± 21.3	27560	17.8	4.0
	FP8	480.45 ± 0.68	9608.9 ± 13.5	33517	0.0	0.0
Qwen-Image	ECF8	49.05 ± 0.07	1226.3 ± 1.7	25766	7.9	55.9
	FP8	111.14 ± 1.39	2778.4 ± 34.8	27963	0.0	0.0

Diffusion model inference acceleration. Diffusion models are primarily compute-bound rather than memory-bound, with latency dominated by extensive denoising computations across multiple timesteps. VRAM management, which dynamically offloads and reloads model components between GPU and CPU memory, represents a common deployment strategy for diffusion models to handle memory constraints during inference. As shown in Table 3, ECF8 consistently outperforms FP8 baselines across four representative diffusion transformer architectures, achieving memory reductions from 7.9% to 17.8% and latency improvements from 3.3% to 55.9% under controlled experimental conditions using the DiffSynth library. FLUX.1-dev exhibits the most substantial gains with 45.9% end-to-end latency reduction (13.15s vs 24.29s) and 12.1% memory savings, while Qwen-Image demonstrates exceptional step-level efficiency with 55.9% per-step latency improvement (1226.3ms vs 2778.4ms). The video generation models Wan2.1-T2V-14B and Wan2.2-T2V-A14B show more modest latency improvements of 3.3-4.0%, though Wan2.2-T2V-A14B achieves 17.8%

memory savings. These results validate that ECF8's compact weight representation reduces communication overhead during the frequent weight loading operations characteristic of VRAM-managed diffusion inference, translating storage efficiency into measurable performance gains across diverse model architectures and scales. Beyond single-batch performance gains, the reduced peak memory consumption enables larger batch sizes within fixed memory constraints, translating to remarkable throughput improvements of 55.1% to 177.1% as demonstrated in Table 1.

5 Related Work

Weight compression of GenAI models. Quantization (Hubara et al., 2018) has become the dominant compression paradigm, which reduces memory by conversion of 16-bit weights to lower precision formats. LLM.int8() (Dettmers et al., 2022) achieved practical 8-bit inference through mixed-precision matrix multiplication, and delivered 2× memory reduction by processing most operations in INT8 while it handled outliers in FP16. SmoothQuant (Xiao et al., 2023) enabled W8A8 quantization by migration of activation outliers into weights, while GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2024a), and SpinQuant (Liu et al., 2024b) pushed boundaries to 4-bit precision through second-order optimization, activation-aware selection, and learned rotations. However, quantization is inherently lossy and can cause unpredictable performance degradation, which translates to significant revenue losses at scale. DFloat11 (Zhang et al., 2025) addressed this by exploitation of the low entropy of BF16 weights, and proposed lossless compression via entropy-coded dynamic-length floats with efficient GPU decompression, which achieved 30% memory reduction with bit-exact reconstruction. Nevertheless, these methods are specifically designed for BF16 weights. With the emergence of native FP8 models (Micikevicius et al., 2022) as the future trend in GenAI, there remains an unaddressed need for efficient lossless compression techniques tailored to FP8 weights, which motivates our work.

Weight distribution analysis of neural networks. A growing body of work indicates that the weight matrices of deep neural networks are not well captured by classical Gaussian assumptions but instead exhibit heavy-tailed behavior that can often be modeled by α-stable distributions. On the optimization side, Gurbuzbalaban et al. (2021) showed that stochastic gradient descent gives rise to heavy-tailed dynamics, where the properly rescaled iterates converge in distribution to α-stable random variables, with heavier tails (smaller α) linked to improved generalization. From the perspective of infinite-width limits, Jung et al. (2021) proved that networks with heavy-tailed initializations converge to α-stable processes, while Lee et al. (2023) extended these results to architectures with dependent weights, showing that heavy tails also induce sparsity and compressibility. On the spectral side, Mahoney & Martin (2019) and Martin & Mahoney (2021) analyzed trained weight matrices and observed power-law spectral densities, $\rho(\lambda) \propto \lambda^{-\alpha}$, with most exponents in the range $\alpha \in (2, 4)$ across diverse architectures. Taken together, these findings suggest that heavy-tailed statistics are a robust and recurring feature of well-trained networks. As we show in Section 2, α-stable distributions lead to exponent concentration and, consequently, low entropy, offering opportunities for lossless compression via entropy coding.

6 Conclusion

We revisited the problem of efficient model deployment for generative AI systems through the lens of low-precision computation. While integer quantization has been the prevailing approach, its lossy nature and reliance on dequantization limit its scalability in high-throughput environments. We demonstrated that neural network weights obey an *exponent concentration* principle, rooted in α-stable dynamics of stochastic gradient descent, which ensures that exponents carry bounded low entropy. This theoretical insight yields a fundamental compression limit around FP4.67. Motivated by this, we developed **ECF8**, a practical, lossless FP8 format with entropy-aware encoding and GPU-efficient decoding. Our experiments across LLMs and diffusion transformers show that ECF8 reduces memory consumption by up to 26.9% and improves throughput by as much as 177.1%, all while preserving bit-exact fidelity. Beyond immediate systems benefits, this work highlights exponent concentration as a universal property of trained models and lays the foundation for principled design of next-generation low-precision floating-point formats for GenAI.

Ethics Statement

This work focuses on improving the efficiency of generative AI systems. We do not anticipate any direct negative ethical implications. On the contrary, our approach has the potential to reduce the energy consumption and carbon footprint of AI deployment by lowering GPU requirements. Furthermore, by improving accessibility and reducing computational costs, our work can help democratize the use of generative AI technologies for a broader range of users.

Reproducibility Statement

We have taken several steps to facilitate reproducibility. An anonymous GitHub repository containing the full source code and experiment scripts is available at `https://anonymous.4open.science/r/ecf8-anonymous-7649`. All theoretical results are accompanied by detailed proofs in Section 2. Together, these resources ensure that our results can be independently verified and extended.

References

Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5 (4-5):185–196, 1993.

Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, et al. FLUX.1 Kontext: Flow matching for in-context image generation and editing in latent space. *arXiv e-prints*, pp. arXiv–2506, 2025.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:30318–30332, 2022.

Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

Mert Gurbuzbalaban, Umut Simsekli, and Lingjiong Zhu. The heavy-tail phenomenon in SGD. In *International Conference on Machine Learning (ICML)*, pp. 3964–3975. PMLR, 2021.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research (JMLR)*, 18(187):1–30, 2018.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the Association for Computational Linguistics (ACL)*, pp. 12186–12215, 2024.

Paul Jung, Hoil Lee, Jiho Lee, and Hongseok Yang. α-stable convergence of heavy-tailed infinitely-wide neural networks. *arXiv preprint arXiv:2106.11064*, 2021.

Hoil Lee, Fadhel Ayed, Paul Jung, Juho Lee, Hongseok Yang, and Francois Caron. Deep neural networks with dependent weights: Gaussian process mixture limit, heavy tails, sparsity and compressibility. *Journal of Machine Learning Research (JMLR)*, 24(289):1–78, 2023.

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. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024a.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. QServe: W4A8KV4 quantization and system co-design for efficient LLM serving. *arXiv preprint arXiv:2405.04532*, 2024b.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. SpinQuant: LLM quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024b.

Michael Mahoney and Charles Martin. Traditional and heavy tailed self regularization in neural network models. In *International Conference on Machine Learning (ICML)*, pp. 4284–4293. PMLR, 2019.

Charles H Martin and Michael W Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *Journal of Machine Learning Research (JMLR)*, 22(165):1–73, 2021.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

Chrysostomos L Nikias and Min Shao. *Signal processing with alpha-stable distributions and applications*. Wiley-Interscience, 1995.

Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

Paul Springer and Paolo Bientinesi. Design of a high-performance gemm-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 44(3):1–29, 2018.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

Chenfei Wu, Jiahao Li, Jingren Zhou, Junyang Lin, Kaiyuan Gao, Kun Yan, Sheng-ming Yin, Shuai Bai, Xiao Xu, Yilei Chen, et al. Qwen-Image technical report. *arXiv preprint arXiv:2508.02324*, 2025.

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 *International Conference on Machine Learning (ICML)*, pp. 38087–38099. PMLR, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:27168–27183, 2022.

Tianyi Zhang and Anshumali Shrivastava. LeanQuant: Accurate and scalable large language model quantization with loss-error-aware grid. *arXiv preprint arXiv:2407.10032*, 2024.

Tianyi Zhang, Jonah Yi, Zhaozhuo Xu, and Anshumali Shrivastava. KV cache is 1 bit per channel: Efficient large language model inference with coupled quantization. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:3304–3331, 2024.

Tianyi Zhang, Yang Sui, Shaochen Zhong, Vipin Chaudhary, Xia Hu, and Anshumali Shrivastava. 70% size, 100% accuracy: Lossless LLM compression for efficient GPU inference via dynamic-length float. *arXiv preprint arXiv:2504.11651*, 2025.

APPENDIX

A THE USE OF LARGE LANGUAGE MODELS (LLMS)

We used LLMs as a general-purpose assistive tool during the preparation of this work. Specifically, LLMs were employed for:

- **Writing assistance:** refining grammar, improving clarity of sentences, and suggesting alternative phrasings without altering the technical content.
- **Formatting:** generating LaTeX code snippets for tables, figures, and algorithm blocks, which were then verified and adapted by the authors.
- **Brainstorming:** offering initial organizational structures for sections and summarizing related work, which the authors subsequently reviewed, fact-checked, and rewrote.

No parts of the research methodology, experiments, data analysis, or conclusions were generated by LLMs. The authors take full responsibility for the final content of the paper. LLMs were not used for creating or fabricating research results. As per ICLR policy, LLMs are not listed as authors.

B GENERATIVE AI MODELS USED FOR EVALUATION

We evaluate a diverse set of state-of-the-art generative AI models spanning language, code, image, and video modalities. All models are released in FP8 or have community-supported FP8 versions.

B.1 LLMS

- DeepSeek-R1-0528[1]: a 671B-parameter reasoning-focused mixture-of-experts (MoE) model, one of the first native FP8 LLMs, achieving strong performance on complex reasoning tasks.
- Qwen3-235B-A22B-Instruct-2507-FP8[2]: a 235B-parameter MoE FP8 model tuned for instruction following, reasoning, mathematics, coding, and tool use.
- Llama-3.3-70B-Instruct-FP8-dynamic[3]: a 70B dense instruction-tuned model using symmetric FP8 quantization for weights and activations, improving throughput and reducing memory cost.
- Qwen3-Coder-30B-A3B-Instruct-FP8[4]: a 30.5B-parameter MoE FP8 model specialized for code generation and agentic tool use, supporting long contexts up to 256K tokens (extendable to 1M with Yarn).
- Qwen3-8B-FP8[5]: an 8B dense FP8 model released by Qwen, serving as a lightweight but competitive baseline.

B.2 DITS

- FLUX.1-dev[6]: a 16B DiT for text-to-image generation, originally in BF16 with community FP8 implementations by community.
- Wan2.1-T2V-14B[7]: a 14B DiT for text-to-video generation, BF16 release with FP8 support by community.
- Wan2.2-T2V-A14B[8]: a 30B-parameter MoE DiT for text-to-video generation, representing a major architectural upgrade. Released in BF16 with FP8 support by community.
- Qwen-Image[9]: a 20B DiT-based model for image generation and editing, notable for high-fidelity text rendering and precise editing. Released in BF16 with FP8 implementations by community.

[1] https://huggingface.co/deepseek-ai/DeepSeek-R1-0528
[2] https://huggingface.co/Qwen/Qwen3-235B-A22B-Instruct-2507-FP8
[3] https://huggingface.co/RedHatAI/Llama-3.3-70B-Instruct-FP8-dynamic
[4] https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct-FP8
[5] https://huggingface.co/Qwen/Qwen3-8B-FP8
[6] https://huggingface.co/black-forest-labs/FLUX.1-dev
[7] https://huggingface.co/Wan-AI/Wan2.1-T2V-14B
[8] https://huggingface.co/Wan-AI/Wan2.2-T2V-A14B
[9] https://huggingface.co/Qwen/Qwen-Image

C Image Generation Settings

We use the open-source DiffSynth implementation[10] for image generation. All DiT models are run with the default settings for image/video resolution, inference steps, and guidance scale. A fixed random seed (2025) is applied to guarantee comparability between the FP8 and ECF8 models. The prompts employed are listed in Table 4. The images generated by the FP8 Qwen-Image model are presented in Figure 4, and those generated by the ECF8 Qwen-Image model are shown in Figure 3.

Table 4: Prompts used for image generation.

Prompt
A futuristic neon-lit cityscape with a cheerful cyberpunk character in a glowing high-tech exosuit, smiling with holographic tattoos and a reflective visor. The atmosphere is bright and vibrant with neon blues and purples, blending anime and sci-fi concept art aesthetics. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A playful surrealist artwork where colorful balloons float through a sunny meadow, and a joyful faceless figure relaxes in midair. The palette is light and cheerful with splashes of gold and pastel tones, evoking a sense of carefree happiness. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
An anime female character, lofi style, soft colors, gentle natural linework, key art, emotion is happy. Hand drawn with an award-winning anime aesthetic and a well-defined nose. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A festive parade scene with a vibrant character at the center of a confetti-filled street, smiling brightly, with balloons and streamers in the background. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A radiant anime-style character standing in a glowing crystal meadow, surrounded by rainbows and magical sparkles, smiling with pure happiness. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A cheerful fantasy scene where a character rides a friendly dragon while flying a kite, filled with vibrant colors and joyful energy. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A pastel illustration of a character sitting by a glowing campfire, with warm lanterns floating above, smiling peacefully. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A joyful carnival sunset with a character in front of a Ferris wheel, holding cotton candy, illuminated by golden evening light. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A futuristic festival scene where a neon-clad character dances joyfully under holographic fireworks. Holding a sign saying ECF8 IS FAST AND LOSSLESS.
A serene beach scene with a character building a sandcastle under a rainbow, while dolphins leap in the distance. Holding a sign saying ECF8 IS FAST AND LOSSLESS.

Figure 4: Images generated by the FP8 Qwen-Image model. These are pixel-wise identical to the images generated by the ECF8-compressed model (see Figure 3).

[10] https://github.com/modelscope/DiffSynth-Studio

D SOFTWARE AND HARDWARE

Our experiments are conducted using PyTorch 2.7.1, CUDA 12.8, Transformers 4.56.0, and Diffusers 0.34.0. The hardware configuration varies based on model size requirements: we employ 8×H200 GPUs (141 GB memory each) for DeepSeek-R1-0528 due to its substantial memory demands, 4×H200 GPUs (141 GB memory each) for Qwen3-235B-A22B-Instruct-2507-FP8, and a single GH200 GPU (96 GB memory) for all remaining models in our evaluation suite.

E NOTATIONS

Table 5: Symbols in Algorithm 1.

Symbol	Description	Type / Shape
LUT	Hierarchical lookup tables	$n_{\text{luts}} \times 256$ integers
$encoded$	Encoded byte stream	n_{bytes} bytes
$packed$	Packed sign/mantissa nibbles	$\lceil n_{\text{elem}}/2 \rceil$ bytes
$outpos$	Block output positions	$(n_{\text{blocks}}+1)$ 64-bit integers
$gaps$	Packed 4-bit gap values (two per byte), over all threads	$\lceil (n_{\text{blocks}} \cdot T)/2 \rceil$ bytes
L	64-bit sliding bit window	64-bit integer
S	16-bit tail buffer	16-bit integer
f	Free bits in headroom	8-bit integer
c	Symbols counted per thread	32-bit integer
o_{start}	Thread output start index	32-bit integer
o_{end}	Thread output end index	32-bit integer
o_{base}	Block output base index	32-bit integer
$localbf$	Thread register buffer	Bytes of length $B + 2$
$writebf$	Shared write buffer	$outpos[b+1] - outpos[b]$ bytes
$accum$	Shared prefix-sum array	$T + 1$ 32-bit integers
n_{luts}	Number of lookup tables	32-bit integer
n_{bytes}	Length of encoded stream	32-bit integer
n_{elem}	Number of output elements	32-bit integer
B	Bytes per thread window	Constant integer
T	Number of threads per block	Constant integer
$outputs$	Decoded FP8 output bytes	n_{elem} bytes
b	Thread-block index	32-bit integer
t	Thread index within block	32-bit integer
t_g	Global thread index ($b \cdot T + t$)	64-bit integer
g	Per-thread gap (extracted from $gaps$)	4-bit integer
x	Decoded symbol (exponent code)	8-bit integer
q	The packed sign/mantissa nibble	4-bit integer
b_ℓ	Bit-length of the codeword for x	8-bit integer

F PSEUDOCODE FOR ECF8 DECOMPRESSION

Table 5 summarizes the notations used in this section.

Algorithm 1 Block-level decompression from ECF8 to FP8

Require: LUT, $encoded$, $packed$, $outpos$, $gaps$, n_{luts}, n_{bytes}, n_{elem}
Ensure: $outputs$

Parallel for each thread t in block b of size T:
Global thread id $t_g \leftarrow b \cdot T + t$.
Load $B + 2 = 10$ bytes from $encoded$ at offset $t_g \cdot B$ into local buffer $localbf$.
Form a 64-bit head L from $localbf[0..7]$ so the oldest byte is the most significant; form a 16-bit tail S from $localbf[8..9]$.
Extract 4-bit gap g from packed $gaps$ using t_g: $g \leftarrow \big(gaps[\lfloor t_g/2 \rfloor] \gg (4 - (t_g \bmod 2) \cdot 4)\big) \,\&\, 0x0f$. Set $L \leftarrow L \ll g$, $f \leftarrow g$, $c \leftarrow 0$.
Phase 1: symbol counting
while $f < 16$ **do**
    $x \leftarrow LUT[L \gg 56]$
    if $x \geq 240$ **then** $x \leftarrow LUT\big[256(256 - x) + ((L \gg 48) \,\&\, 255)\big]$
    end if
    $b_\ell \leftarrow LUT\big[256(n_{\text{luts}} - 1) + x\big]$; $L \leftarrow L \ll b_\ell$; $f \leftarrow f + b_\ell$; $c \leftarrow c + 1$
end while
Stitch tail: $L \leftarrow L \vee (S \ll (f - 16))$; $f \leftarrow f - 16$.
while $2 + \lfloor f/8 \rfloor < B$ **do**
    Decode one symbol as above; update L, f, c.
end while
Block-level prefix sum
Each thread writes its count c to shared array $accum[t]$; thread 0 writes $accum[0] \leftarrow outpos[b] + c$.
Perform in-place up-sweep and down-sweep on $accum[0..T-1]$ to obtain exclusive starts.
Thread 0 sets $accum[0] \leftarrow outpos[b]$ and $accum[T] \leftarrow outpos[b{+}1]$.
Set $o_{\text{base}} \leftarrow accum[0]$, $o_{\text{start}} \leftarrow accum[t]$, $o_{\text{end}} \leftarrow \min(o_{\text{start}} + c,\ n_{\text{elem}})$.
Phase 2: decode and assemble FP8
Reinitialize L, S, f.
while $f < 16$ **and** $o_{\text{start}} < o_{\text{end}}$ **do**
    Decode next symbol x.
    $q \leftarrow packed[\lfloor o_{\text{start}}/2 \rfloor] \ll \big((o_{\text{start}} \bmod 2) \cdot 4\big)$.
    $byte \leftarrow (x \ll 3) \vee (q\&0x80) \vee ((q \gg 4)\&0x07)$.
    $writebf[o_{\text{start}} - o_{\text{base}}] \leftarrow byte$.
    $o_{\text{start}} \leftarrow o_{\text{start}} + 1$; update L, f using $b_\ell = LUT[256(n_{\text{luts}}{-}1) + x]$.
end while
Stitch remaining: $L \leftarrow L \vee (S \ll (f - 16))$; $f \leftarrow f - 16$.
while $o_{\text{start}} < o_{\text{end}}$ **do**
    Decode and pack as above to fill $writebf$.
end while
Output write-back
Synchronize all threads.
Each thread copies its slice of $writebf$ to $outputs$.