Approaching Shannon Bound with Lossless LLM Weight Compression

Hongshi Tan*, Yao Chen†, Gustavo Alonso§, Weng-Fai Wong*, Bingsheng He*
*School of Computing, National University of Singapore, Singapore
†School of Computer Science and Technology, Huazhong University of Science and Technology, China
§Department of Computer Science, ETH Zurich, Switzerland
hongshi@u.nus.edu, chenyao_cs@hust.edu.cn, alonso@inf.ethz.ch, {dcswwf, dcsheb}@nus.edu.sg

***Abstract*—Large language models (LLMs) now scale to trillions of parameters, driving weight storage into the terabyte regime and creating an acute mismatch with GPU memory capacity. Although lossless compression is widely effective in other domains, it remains underutilized in LLM systems. Through a comprehensive entropy study across models from 1.5B to 405B parameters and numeric formats ranging from bf16 to int4 and AWQ/SQ8, we find that LLM weights contain far less intrinsic randomness than their stored bitwidth implies, their effective entropy is 2–10× lower, indicating that up to a 10× footprint reduction is theoretically achievable without altering any weight values. Leveraging this insight, we introduce a tile-level, on-the-fly lossless decompression framework based on Asymmetric Numeral Systems that aligns decoding with the GEMM tiling pattern of GPU inference. Our design achieves bit-rates within 0.01–0.1 bits of the Shannon limit across a wide range of LLM numerical formats, demonstrating that nearly all statistical redundancy is eliminated. Integrated into the SGLang serving framework with multi-GPU support, our approach increases the maximum batch size of Qwen-14B from 47 to 75, improving throughput by up to 1.2×. On Mixtral-176B, the feasible batch size increases from 20 to 95 (4.8×), yielding up to 1.6× throughput improvement. Compared to state-of-the-art lossless compression approaches NeuZip and DFloat11, our design further improves throughput by up to 11×.**

I. Introduction

Large language models (LLMs) have become the foundational backbone of modern AI systems [27], enabling advanced content generation [22], large-scale data analytics [3], and information-retrieval applications [38]. As model scale continues to grow, so do their capabilities, ranging from stronger reasoning and better generalization to improved adaptability across domains. Yet this rapid expansion places increasing pressure on inference infrastructure and compute hardware.

As shown in Figure 1, LLM parameter counts have grown exponentially from billions to trillions, pushing weight storage into the terabyte range [8], while GPU memory capacity grows much more slowly. Memory, not compute, has become the primary bottleneck: it determines both the maximum deployable model size and the achievable batch size, and reducing the weight footprint therefore directly translates into larger batches and higher inference throughput.

Although lossless compression has proven highly effective in domains such as storage systems and multimedia, its potential for reducing LLM model size and improving inference efficiency remains largely unexplored. Unlike existing model

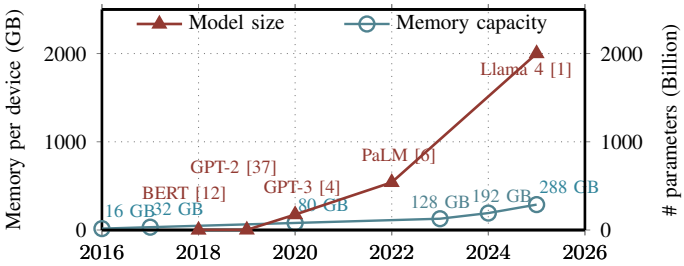


Fig. 1: Model scale increases exponentially while on-device memory growth lags behind.

compression methods such as low rank decomposition [16], [20], [26], [31], [39], [47] and quantization [10], [15], [18], [25], [29], [29], [33], [36], [44], lossless compression preserves the exact numerical representation of model weights and therefore guarantees identical inference results. This property is particularly important for the deployment of reasoning-intensive tasks, where lossy techniques can introduce instability and unpredictable accuracy degradation [24], [28], [30], [42]. Furthermore, lossless compression is orthogonal to existing lossy compression techniques and can be applied on top of well-tuned quantized models to further reduce memory footprint.

State-of-the-art lossless compression methods for LLM weights, such as NeuZip [17] and DFloat11 [49], are largely designed for floating-point representations. However, modern LLM deployments increasingly adopt diverse numerical formats, including FP8, INT8, and low-bit or group-wise quantization, which are not well supported by these approaches. As a result, existing techniques do not generalize well across emerging LLM weight formats and leave substantial redundancy unexploited.

This raises two fundamental questions for lossless LLM compression: (1) How much strictly lossless compression is theoretically achievable across both unquantized and modern quantized formats? (2) If significant redundancy exists, how can we design a lightweight, high-throughput GPU execution model that integrates lossless compression directly into inference while preserving bit-exact outputs?

To answer the first question, we conduct a comprehensive, model-wide analysis grounded in Shannon's information

theory. Shannon limit provides a fundamental lower bound on the bits required to represent weights drawn from any underlying distribution, independent of numeric format or hardware layout, and therefore quantifies the minimum achievable footprint under any lossless encoding. Our analysis across widely deployed LLMs and numerical formats reveals that, on average, up to 27% of the stored bits are redundant, indicating significant headroom for footprint reduction without altering weight values (details in Section II). This entropy gap suggests that substantial memory savings, and therefore potentially higher inference throughput, can be unlocked with the right compression and inference integration strategy.

As the entropy analysis reveals substantial redundancy in contemporary LLM weights, we pursue the goal of developing a strictly lossless compression and high-throughput decompression framework for LLM inference. Achieving compression close to the Shannon limit and translating this redundancy into runtime gains, however, is non-trivial: both the codec and its system integration must satisfy stringent constraints imposed by GPU-accelerated LLM execution.

First, compressed bitstreams produced by conventional entropy codecs are not directly addressable. Most entropy coding schemes generate sequential bitstreams, where decoding must proceed in order and cannot easily jump to arbitrary positions. However, GPU GEMM kernels access weights in a strided, tile-granular pattern rather than sequential order. Transformer projection layers consume weights in small tiles, often non-contiguous and repeatedly reused across the M-dimension. This mismatch makes it difficult to directly retrieve the required tiles from a compressed bitstream. A practical lossless codec must therefore reconstruct tiles on demand, without introducing latency, excessive buffering, or additional global-memory traffic. Only by aligning decompression with the native GEMM tiling structure can the decoding overhead be fully amortized across the compute pipeline.

Second, a key challenge is that directly applying existing lossless decoders to LLM inference pipelines leads to inefficient execution and poor performance. State-of-the-art compression frameworks [17], [49] typically follow a decompress-store-compute workflow, where compressed weights are first fully decoded into global memory before being consumed by GEMM kernels. This layer-wise decompression introduces substantial overhead: it requires additional memory capacity to hold the decompressed weights, generates redundant global-memory traffic, and prevents effective overlap between decompression and computation. As a result, naive decompression can become a critical performance bottleneck in real-world LLM serving systems.

To address these challenges, we begin by analyzing the information content of LLM weights and comparing existing lossless compression families, including dictionary-based methods (LZ77/Zstd), symbol-based entropy codes (Huffman, arithmetic coding), and finite-state entropy coders (rANS/tANS/FSE). *Asymmetric Numeral Systems (ANS)* emerges as the only class that simultaneously (1) approaches Shannon-optimal bitrates, (2) supports random access at tile granularity, and (3) enables parallel decoding suitable for GPUs.

Building on this insight, we propose a *tile-level compression with on-the-fly decompression* framework based on ANS that integrates entropy decoding directly into the GPU inference pipeline. Unlike prior approaches that treat compression as a preprocessing or storage optimization, our design elevates compression to an execution primitive by introducing a compressed-weight execution model in which entropy-coded weight tiles become first-class operands of tensor-core computation. Instead of pre-decompressing weights or introducing layer-level synchronization, compressed tiles are decoded into shared memory precisely when the GEMM microkernel consumes them. This design enables decompression throughput close to the weight input rate of the baseline GEMM kernel while preserving strict losslessness without affecting model accuracy.

In summary, we make three key contributions:

- We perform the first systematic information-theoretic analysis of LLM weights across diverse numerical formats beyond floating-point representations, revealing a substantial entropy gap to the theoretical Shannon bound. Our results show that, in large models, up to a $\mathbf{5}\times$ reduction in memory footprint is achievable without any accuracy loss.
- We propose a compressed-weight execution model for GPU inference that integrates entropy decoding directly into the GEMM pipeline. The design introduces tile-addressable compressed streams, warp-cooperative decoding, and shared-memory reconstruction of weight tiles, enabling decompression and tensor-core computation to proceed concurrently without intermediate global-memory materialization.
- We propose a GPU kernel that integrates seamlessly with standard LLM inference frameworks such as SGLang, enabling larger batch sizes and achieving up to $1.6\times$ higher throughput. Our evaluation further demonstrates that the proposed approach outperforms existing state-of-the-art lossless compression techniques.

II. Information Redundancy in Contemporary LLM Weight

Large language models consist of many transformer layers, each layer performing several projections with high-dimensional matrix multiplications between the weight matrices and the hidden-state activations. These projection weights dominate the static memory footprint of the model and must be frequently accessed from device memory during inference.

A. Heavy-Tailed Distributions in LLM Weights

LLM weights exhibit heavy-tailed statistical distributions [33] and, importantly, the distribution can vary substantially across layers. Most existing studies evaluate compression [16], [20], [26], [31], [39], [47] and quantization [10], [15], [18], [25], [29], [29], [33], [36], [44] primarily from a numerical-format perspective, emphasizing bit width, scaling, and calibration strategies. However, such approaches do not

explicitly capture the underlying data distribution or its cross-layer variability, leaving significant opportunities for further redundancy reduction unaddressed.

To evaluate the potential for further reducing the memory footprint of model weights, we begin with an information-theoretic analysis based on Shannon's source coding theorem. In information theory, the Shannon limit defines the theoretical lower bound on the average number of bits required to represent data without loss. This bound is determined by the Shannon entropy of the source distribution, which quantifies the intrinsic information content of the weight values.

B. Empirical Observations on Model Weights and Numerical Representations

To compute this bound in practice, we start by computing entropy at the granularity of each individual weight tensor and then aggregating the results across the entire model. For each layer l, we treat its weight matrix $W^{(l)}$ as a collection of discrete symbols determined by the chosen numeric format (e.g., FP8 codes, INT4 values, per-channel quantized indices). We build an empirical histogram over these symbols and estimate the Shannon entropy

$$\mathcal{H}^{(l)} = -\sum_i p_i^{(l)} \log_2 p_i^{(l)},$$

where $p_i^{(l)}$ is the empirical probability of symbol i within that tensor.

Because different layers contain different numbers of parameters, we compute the entropy of the full model as a symbol-weighted average of the per-layer entropies:

$$\mathcal{H}_{\text{model}} = \frac{\sum_l \mathcal{H}^{(l)} |W^{(l)}|}{\sum_l |W^{(l)}|},$$

where $|W^{(l)}|$ is the number of elements in layer l. This yields the effective number of bits per weight required under any lossless encoding scheme.

We evaluate widely used open-source LLMs ranging from 1.5B to 405B parameters, including Qwen-1.5B [46], Mistral-7B [21], Qwen-14B [46], DeepSeek-67B [9], Mixtral-176B [40], and Llama-405B [41]. Moreover, we examine the full spectrum of standard numeric formats such as `bfloat16` (bf16), `FP8-E5M2` (fp8), `INT8`, `FP4-E2M1` (fp4), and `INT4`, as well as group-quantized formats from `SmoothQuant` [44] (sq8) and `AWQ` [29] (awq4), allowing us to quantify how much redundancy remains in current weight representations and assess how far existing numeric formats are from the theoretical limit.

Figure 2 shows that the effective bits of LLM weights are dramatically lower than their stored bitwidth, often by a factor of two to six. Across all models, `bfloat16` exhibits about 4–5 bits of redundancy, corresponding to a potential $1.5\times$ reduction in size. Here we refer to *redundancy* as the difference between the stored bitwidth and the estimated entropy of the weight distribution (in bits per weight). For example, if `bfloat16` stores each weight using 16 bits but the measured entropy of the distribution is approximately

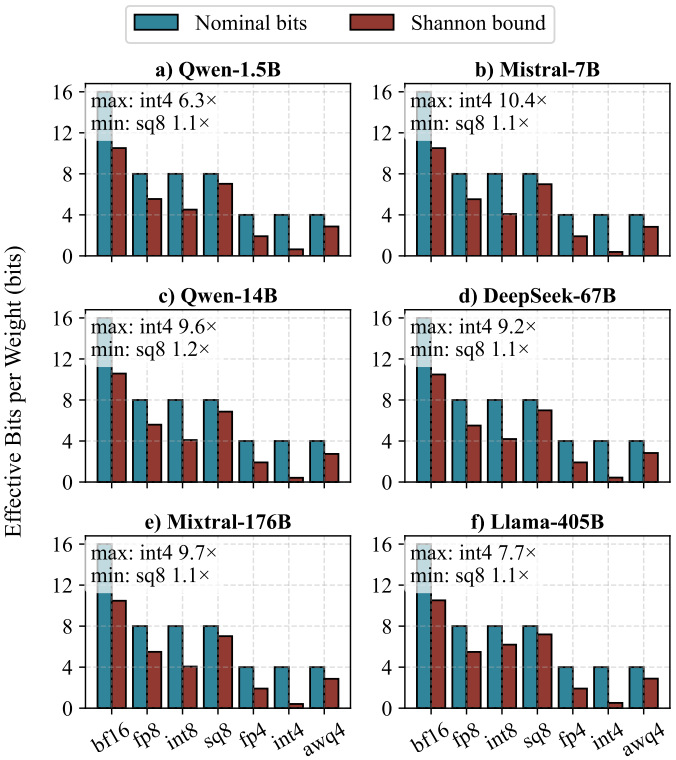


Fig. 2: Remaining entropy gap across models with different data types.

11–12 bits, then about 4–5 bits per weight represent statistical redundancy that can be removed by an optimal lossless coding scheme.

Even in extremely low-bit representations such as `INT4` and `FP4`, substantial redundancy remains because the quantized weight distributions are highly skewed, with only a small subset of symbols appearing frequently due to the heavy-tailed distribution of LLM weights. As shown in Figure 2, these formats exhibit the largest entropy gaps, with entropy ratios reaching 6–10×, indicating significant unused capacity even at very low bitwidths.

In addition, widely deployed group-quantized formats such as `SmoothQuant` and `AWQ` adopt block-based scaling factors to reduce quantization error. While these schemes improve numerical accuracy, they still retain measurable redundancy, typically around 1.1–1.3× above their entropy bound, due to skewed symbol frequencies and the structured metadata introduced by per-group scaling.

These observations indicate that existing models store far more bits than their intrinsic information content requires, suggesting that substantial memory savings of up to ten times remain achievable without sacrificing accuracy.

III. Towards High-Performance LLM Inference with Near-Shannon Limit Compressed Weights

Incorporating compressed weights into LLM inference naturally consists of two stages: **(a) Offline compression**, where weights are compressed during preprocessing using algorithms

TABLE I: Comparison of widely used compression algorithms. Entropy efficiency measures proximity to the Shannon limit, while streaming capability indicates the smallest granularity at which data can be decoded without global synchronization.

Algorithm / Family	Core Principle	Entropy Efficiency	Access Granularity
gzip (DEFLATE) [11]	LZ77 + Huffman	80–90%	Sequential (per block, ∼64 KB)
LZ4 [48]	LZ77 (no entropy stage)	≈80%	Byte-level (continuous)
Zstandard (Zstd) [7]	LZ77 + FSE (rANS)	90–95%	Chunk-level (64 KB–4 MB configurable)
Brotli [2]	Context + Huffman	90–95%	Block-level (windowed)
Huffman Coding [32]	Static symbol code	90–95%	Symbol-level (per byte or token)
Arithmetic Coding [43]	Range interval	98–99%	Bit-level (serial)
BWT / PPM [14]	Transform + context	95–98%	File-level (global transform)
rANS / tANS / FSE [13]	Finite-state entropy	**>99% (near Shannon Limit)**	**Byte-level (fully streaming)**

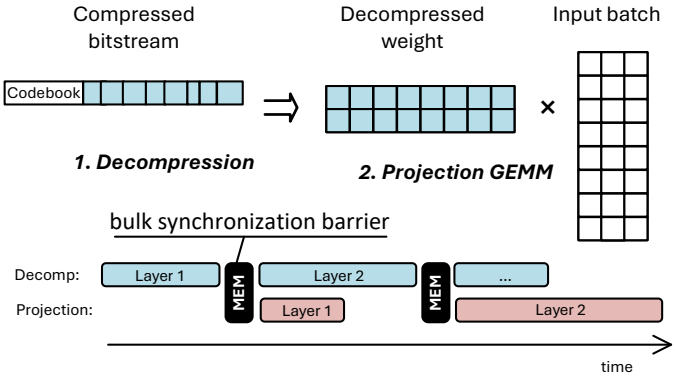


Fig. 3: Existing coarse-grain pipelining between decompression and transformer layer computation.

aggressive enough to approach the Shannon entropy limit; and **(b) On-demand decompression at inference**, where compressed weights must be decoded just before the transformer kernels consume them. As shown in Figure 3, existing systems such as ZipNN [19] decompress an entire layer before launching its GEMM. This places decoding on the critical path, introduces a layer-level synchronization barrier, and produces idle compute, redundant global-memory traffic, and memory stalls.

A. Principles for Selecting Lossless Compression and Decompression Algorithms

Effective lossless compression for LLM inference requires more than a high compression ratio: because GEMM kernels partition projection weights into fine-grained tiles that are repeatedly loaded and reused by the tensor cores, any practical codec must respect this tiled access pattern and integrate seamlessly into the GEMM dataflow rather than treat weights as monolithic tensors.

Principle 1: Near-Shannon bound compression with hardware-realistic throughput. LLM inference reads weights at an extremely high bandwidth. A useful lossless codec must therefore compress close to the entropy limit while also decoding fast enough to keep up with GPU memory throughput. Otherwise, decompression becomes the bottleneck, no matter how good the compression ratio.

Principle 2: Tile-granularity access for decompression. GEMM kernels consume weights tile by tile, not layer by layer. Most compression algorithms cannot jump to an arbitrary tile without decoding everything before it. This forces full-layer decompression and blocks pipelining. A practical codec must allow each tile to be decoded independently, exactly when the GEMM kernel needs it.

Principle 3: Tight integration of decompression with matrix-multiplication tiling. Even with fast, tile-level decoding, decompression must fit directly into the GEMM dataflow. Writing decoded tiles back to global memory wastes bandwidth and breaks overlap. Decompression should instead write directly into shared memory in the same layout used by tensor cores, enabling seamless overlap with computation and avoiding extra memory traffic.

B. Review of Existing Lossless Compression Methods

Given these three principles, we review classical compression algorithms that achieve excellent ratios in general-purpose data. Table I compares widely used compression schemes, highlighting entropy efficiency and access granularity.

1) Dictionary-based compressors (gzip, LZ4, Zstd).: LZ77-style schemes rely on sequential pointer chasing through a sliding dictionary, which prevents random-access decoding of a single weight tile without rebuilding all prior state. Their compression efficiency also degrades at tile granularity, since effectiveness relies on large dictionary contexts. Both properties violate the tile-granularity constraint for pipelined GEMM execution.

2) Symbol-based codecs (Huffman, arithmetic coding).: Huffman coding is fast but limited by integer-length codes, leaving nontrivial gaps to the Shannon limit. Arithmetic coding achieves near-optimal entropy efficiency, but its bit-serial state machine prevents parallel decoding and restricts throughput to only a few GB/s. Both methods fail to meet the two requirements: they neither support tile-level random access nor scale to HBM-level bandwidth during inference.

3) Finite-state entropy coders (rANS, tANS, FSE).: Finite-state entropy coding retains the near-Shannon efficiency of arithmetic coding but replaces serial interval updates with table-driven state transitions, enabling byte-level streaming and parallel decoding at tens to hundreds of GB/s while keeping each tile independently decodable. It is the only codec class that meets both constraints required for high-throughput LLM inference. Prior work, such as DietGPU [23], provides competitive warp-cooperative rANS decoders, offering

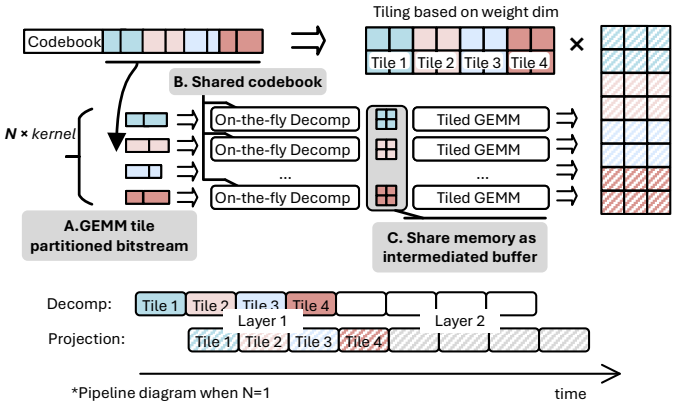


Fig. 4: Tile-aligned on-the-fly decompression, partitions weights into fine-grained tiles, decodes them on demand, and overlaps decompression with GEMM execution using shared memory buffers.

a strong foundation with the potential for further performance optimization and tight integration with the GEMM execution pipeline.

IV. On-the-fly Decompression

Figure 4 illustrates our on-the-fly decompression execution model, which replaces coarse, layer-level decoding with a continuous, tile-aligned dataflow.

The compressed weight bitstream is partitioned into tile-sized ANS substreams, with each tile's starting offset recorded in a lightweight index table. All tiles within the same layer share a compact ANS codebook, which is constructed from the layer's aggregated weight distribution. Sharing a layer-wide codebook maximizes statistical coverage of the underlying distribution while keeping metadata overhead minimal. Each tile can then be decoded independently, exactly at the moment the GEMM kernel needs it, without scanning preceding tiles. The runtime workflow consists of three tightly coupled stages:

A. Tile-partitioned bitstreams. During offline preprocessing, each weight matrix is partitioned into tile-aligned substreams matching the GEMM tile size executed on the SMs. Each tile is entropy-encoded as an independent ANS bitstream using a shared per-layer codebook, and a compact offset entry is stored in the tile index table to enable direct tile access.

B. On-the-fly ANS decompression. At runtime, multiple ANS decoder kernels execute on the SMs, reconstructing weight tiles directly into their on-chip shared memory. This avoids writing decompressed weights back to global memory, substantially reducing global memory bandwidth consumption.

C. Coupled GEMM execution. As soon as a tile is decoded in shared memory, it is immediately consumed by the GEMM kernel in its required swizzled layout. A double-buffered shared-memory workspace ensures that, while one tile is being used for computation, the next tile is being decoded in Tile-aligned on-the-fly .

By aligning decompression with GEMM tile consumption, the proposed design removes layer-level synchronization barriers and eliminates intermediate global-memory traffic. As shown in the timeline comparison, computation for layer i can begin as soon as the first tiles are ready, while later tiles are decoded concurrently, achieving sustained line-rate throughput with negligible overhead.

V. GPU Implementation and Optimizations

In this section, we describe how to integrate the on-the-fly decompression mechanism into GPU execution pipelines.

A. Design Overview

Figure 5 illustrates the complete workflow of the proposed GPU runtime kernel. Building on state-of-the-art GEMM libraries such as CUTLASS, we develop a lightweight plugin-style kernel that can override existing projection operators with minimal changes to library-level GEMM implementations and provides a wrapper for direct invocation in PyTorch.

The design consists of two complementary stages: the offline compression stage, minimizing the static device-memory footprint, and the on-the-fly decompression and swizzled GEMM stage, maximizing runtime efficiency during inference. For the decompression backend, we build upon the ANS kernel implementation from the open-source DietGPU library [23], adapting and extending it to support tile-granular decoding and integration with modern tensor-core GEMM pipelines.

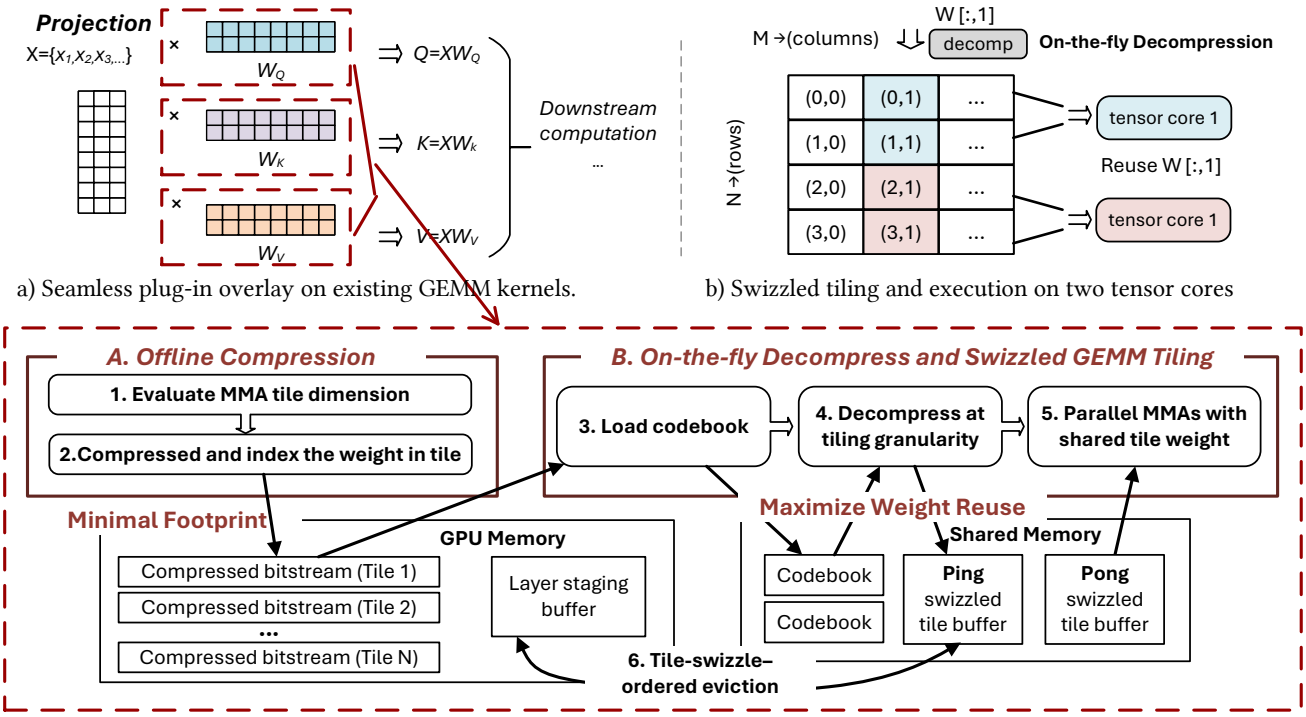
B. Tile Addressable ANS Offline Compression

In the preprocessing phase, each projection matrix (W_Q, W_K, W_V) is first profiled to determine the optimal tensor-core tiling geometry (e.g., 128×32, 256×64, 128×128 and etc.) for downstream GEMM execution. We then aggregate the weight statistics across the entire layer to construct a shared, compact ANS codebook that captures the dominant distributional structure while amortizing codebook overhead of independent compression on chunks. After the codebook is established, the weight tensor is partitioned into tiles aligned with the GEMM tiling geometry. Each tile is then entropy-encoded using ANS with an independently initialized state while sharing the same per-layer codebook. Because ANS supports arbitrary initial states without losing compression efficiency, each tile becomes a fully self-contained substream that can be decoded independently at inference time.

The resulting compressed bitstreams and their tile-offset metadata are stored in GPU memory, producing a compact representation while preserving the exact tile boundaries required by GEMM kernels. Importantly, this stage is *entirely offline* and independent of framework execution. The compressed model can be loaded as a direct drop-in replacement for standard weights, enabling our system to act as a lightweight plugin atop existing LLM inference frameworks such as PyTorch or custom CUDA runtimes.

C. On-the-Fly Decompression Pipelined with GEMM

After the offline compression stage, the compressed weight tensors and their metadata are loaded into GPU memory and used directly during inference. During execution, decompression is performed at the same tile granularity as the GEMM



a) Seamless plug-in overlay on existing GEMM kernels.

b) Swizzled tiling and execution on two tensor cores

Fig. 5: GPU kernel for on-the-fly decompression and swizzled GEMM execution. (a) Projection layers seamlessly integrate the decompression kernel. (b) Offline compression minimizes footprint, while on-the-fly decompression and shared-memory swizzling maximize weight reuse and pipeline overlap.

computation and executed in a fused manner for efficient overlap.

1) Fused Tile Aligned Kernel Design: Algorithm V.1 details the fused execution model that underlies our high-performance ANS decompression path. The kernel consists of two cooperating components: a warp-cooperative rANS decoding kernel and a tile-level GEMM microkernel.

At the beginning of each layer, the rANS decode table is loaded into shared memory by the decoding warp on each SM, enabling low-latency, high-bandwidth table lookups during decompression.

During execution, warp 0 fetches compressed weight tiles from global memory and decodes them in a streaming manner. The decoded symbols are written *directly* into a shared-memory tile $A^{(b)}$, with indices generated in lockstep with the interleaved decode order. This avoids materializing decompressed weights in global memory and significantly reduces global-memory traffic. The rANS decoder therefore acts as a producer that reconstructs weight tiles directly into shared memory, while the remaining warps in the thread block immediately consume the tile for tensor-core GEMM computation.

A double-buffered pipeline overlaps decompression of tile $k{+}1$ with GEMM computation on tile k. Producer and consumer warps synchronize through shared-memory atomic state flags implemented with `cuda::atomic_ref<int, cuda::thread_scope_block>`, ensuring correct ordering with minimal performance overhead. The detailed scheduling of producer and consumer warps on disjoint hardware pipes, and the regime in which decode is fully hidden, are analyzed in Section V-D.

2) Tile Swizzle and Deterministic Eviction: For large-dimension GEMM, the kernel follows a fixed tile-swizzle traversal order to maximize data reuse and balance shared-memory pressure across thread blocks. Because this access sequence is deterministic and determined by the GEMM tiling schedule, the working set of active tiles may temporarily exceed the available shared-memory capacity. In such cases, decoded tiles are evicted following the deterministic order induced by the tile swizzle traversal. The notion of "recency" is therefore defined by the swizzle access order rather than runtime reuse tracking, allowing the eviction sequence to be determined statically without additional bookkeeping. When a tile is evicted from shared memory, its decompressed form is temporarily stored in a small decompression buffer to avoid redundant decompression if the tile is accessed again shortly thereafter, while the compressed representation remains in global memory.

3) Parallel Warp-Cooperative rANS Decoding: Algorithm V.2 presents the warp-cooperative rANS decoding algorithm used to reconstruct compressed weight tiles on the GPU. Because the rANS state machine is inherently sequential, we expose parallelism by partitioning each tile's compressed bitstream into R independent substreams. Each warp lane maintains its own rANS state and processes one substream, allowing the serial decoding process to be distributed across the warp.

For each decoding step, the lane extracts the low bits of the current rANS state to determine the next symbol, performs

Algorithm V.1: Fused rANS Decompression and GEMM Tile Computation

Input: Compressed bitstreams $\mathcal{B}$, global rANS decode table T, matrices B and C
Output: Updated output tile C

Shared memory:
  Decode table $\tilde{T}$ (copied from T)
  Double-buffered decompressed tiles $A^{(0)}, A^{(1)} \in \mathbb{R}^{M \times K}$
  Tile $B^{(k)} \in \mathbb{R}^{K \times N}$
  Atomic Flags ready[2] indicating buffer readiness
Copy global decode table T into shared memory $\tilde{T}$
Initialize ready[0] $\leftarrow 0$, ready[1] $\leftarrow 0$
for $k = 0$ **to** $K_{\text{tiles}} - 1$ **do**
  $b \leftarrow k \bmod 2$ // current buffer index
  $p \leftarrow 1 - b$ // previous buffer
  Warp 0: rANS decompression into $A^{(b)}$
    RansDecodeTile($A^{(b)}$, $\mathcal{B}[k]$, $\tilde{T}$)
    ready[b] $\leftarrow 1$
  Warps 1..W: GEMM tile compute
    Load $B^{(k)}$ from global memory
    if $k = 0$ **then**
      Wait until ready[b] $= 1$ // first tile
      GemmTile($A^{(b)}, B^{(k)}, C$)
      ready[b] $\leftarrow 0$
    else
      Wait until ready[p] $= 1$ // consume previous buffer
      GemmTile($A^{(p)}, B^{(k)}, C$)
      ready[p] $\leftarrow 0$
  Block-wide synchronize
return C

a shared-memory lookup to retrieve the corresponding table entry, and writes the decoded value directly into the shared-memory.

To maintain correctness of the rANS automaton, each lane independently updates and renormalizes its state by reading additional bits from the compressed stream when the state falls below the renormalization threshold. Because the compressed streams are interleaved across lanes, these renormalization loads are naturally coalesced in global memory, preserving high memory throughput. This warp-level interleaving enables a single SM to decode multiple rANS streams concurrently while maintaining the correctness of the rANS state transitions. As a result, the decoder achieves high parallel efficiency while reconstructing tiles directly into shared memory for immediate consumption by the GEMM kernel.

4) Tile-Level GEMM Microkernel: Algorithm V.3 then illustrates the complementary consumer stage. Here, the GEMM

Algorithm V.2: Warp-cooperative rANS tile decompression

Function RansDecodeTile(A, *stream*, $\tilde{T}$)
  Initialize rANS state s for each lane
  for $i = 0$ **to** $S_{\text{lane}} - 1$ **do**
    $x \leftarrow s.\text{value} \bmod R$ // low bits
    $\sigma,\ f,\ c \leftarrow \tilde{T}[x]$ // shared-memory lookup
    $w \leftarrow \text{DecodeSymbol}(\sigma)$
    Compute (r, c) for symbol index and write $A[r, c] \leftarrow w$ // A is a shared-memory tile
    $s.\text{value} \leftarrow f \cdot \lfloor s.\text{value}/R \rfloor + (x - c)$
    while $s.\text{value} < \text{renorm\_thresh}$ **do**
      $u \leftarrow$ Load 32-bit chunk (coalesced)
      $s.\text{value} \leftarrow (s.\text{value} \ll 32) \mid u$

microkernel reads the decompressed weight tile A directly from shared memory and multiplies it with a $K \times N$ activation tile B. Because the weight tile is already resident in shared memory accessible to the warp, all accesses to $A[m, k]$ are single-cycle, eliminating the bandwidth demands and cache thrashing associated with repeatedly loading large weight matrices from global memory.

We build upon CUTLASS to obtain the flexibility required to support the diverse numeric formats and widely used quantization schemes. Moreover, CUTLASS exposes programmable tensor-core tiling, warp scheduling, and memory layouts, allowing us to integrate tile-level ANS decompression directly into the GEMM pipeline, but our proposed design is not tied to CUTLASS itself.

A key consequence of this organization is that decode and compute share the same on-chip shared-memory footprint, so each weight is decoded exactly once and never reloaded. Combined with the producer-consumer scheduling analyzed in Section V-D, the kernel sustains GEMM at full tensor-core speed while keeping the decoder on the critical path only when batch size is small.

D. Pipeline Overlap and the Batch-Size Regime

The producer (rANS decode) and consumer (tensor-core matrix multiply) execute on physically disjoint pipelines within each streaming multiprocessor: decode warps exercise only the integer and load/store units (probability-table loads, rANS state updates, shared-memory stores into the operand slab), while matrix-multiply warps issue only shared-memory matrix loads and tensor-core multiply–accumulate instructions on the separate tensor pipeline. The warp scheduler co-issues them in the same cycle, and a four-stage shared-memory ring buffer lets decode run several sub-tiles ahead, so each consumer step finds its operands already resident.

The effectiveness of this overlap scales with batch size. Decode cost per sub-tile is approximately constant, dominated by probability-table accesses and renormalization reads; matrix-multiply cost per sub-tile grows with the M-rows processed

Algorithm V.3: Shared-memory GEMM microkernel

Function `GemmTile` (A, B, C)
  Each warp reads its fragment of the shared-memory weight tile A and accumulates into C
  for m, n, k *in tile loops* **do**
    $C[m, n] \mathrel{+}= A[m, k] \cdot B[k, n]$ `// weights`
    $A[m, k]$ `come from shared memory`

by each thread block, since one decoded B fragment is reused across all M-blocks. At small batch sizes the tensor pipeline drains faster than the decoder can produce the next sub-tile and the kernel is decoder-bound; at large batch sizes matrix-multiply work dominates, the consumer never stalls, and tensor utilization recovers toward the uncompressed baseline. The fused kernel therefore achieves near-complete overlap in the high-batch regime and degrades gracefully to an unfused decode-then-GEMM schedule at low batch.

E. Global Memory Access Reduction

Consider a tiled GEMM $C[M \times N] = A[M \times K] \cdot W[K \times N]$ with blocking (M_t, N_t, K_t) and a compute kernel that stages one W–tile of size $K_t \times N_t$ into shared memory while iterating over the M dimension.

Without loss of generality, we assume the projection matrix $W \in \mathbb{R}^{K \times N}$ does not fit in cache at all, so every time the kernel advances along the M-dimension it has to reload the relevant tiles of W from HBM. Since there are M/M_t tiles along the M-dimension, each element of W is therefore loaded once per M_t-tile, so the global-memory traffic for W becomes

$$V_B^{\text{uncompressed}} = \frac{M}{M_t} \cdot KN. \tag{1}$$

In contrast, with our on-the-fly ANS kernel, each element of W is fetched once in compressed form and then reused from shared memory.

$$V_B^{\text{on-the-fly decomp}} = (\frac{M}{M_t} + \alpha - 1) \cdot KN. \tag{2}$$

Despite the fact that decompression throughput may not fully match the raw memory bandwidth of GPUs, our design can still sustain high GEMM throughput. By decoding tiles directly into shared memory and overlapping decompression with computation through carefully orchestrated pipeline scheduling, the system eliminates the repeated global-memory loads that dominate baseline execution. Moreover, the on-the-fly decompression stage introduces no additional global-memory pressure, ensuring that reductions in memory traffic translate directly into end-to-end performance gains.

VI. Evaluation

We evaluate the proposed *on-the-fly decompression* design along three objectives:

1. **Entropy characterization and compression efficiency.** Quantify the entropy of weights across representative LLMs, evaluate achievable compression ratios, and measure the gap between practical rANS-based coding and the Shannon bound.
2. **GPU system-level performance.** Benchmark end-to-end inference performance with on-the-fly decompression under realistic GPU memory budgets, batch sizes, and sequence lengths.
3. **Microbenchmark of the proposed technique.** Evaluate the efficiency and sensitivity of our design across different GEMM dimensions and under both prefill and decode settings.

A. Experimental Setup

Evaluated LLM models. To evaluate the effectiveness and generality of our approach, we conduct experiments across a diverse set of widely used open-source large language models spanning different scales and architectures. The evaluated models include `Qwen-1.5B` (Qwen2-1.5B), `Mistral-7B` (Mistral-7B-v0.3), and `Qwen-14B` (Qwen-14B), which represent commonly deployed dense transformer configurations. To further examine scalability to larger models, we include `DeepSeek-67B` (DeepSeek-LLM-67B) and `Llama-3.1-405B`. In addition, we evaluate mixture-of-experts architectures using `Mixtral-8x22B` (Mixtral-176B total parameters). These models collectively span parameter scales from 1.5B to 405B and cover both dense transformer architectures and modern MoE designs.

Numeric formats. Across these models, we evaluate a broad spectrum of widely adopted numeric formats, including `bfloat16` (bf16), `FP8-E5M2` (fp8), `INT8`, `FP4-E2M1` (fp4), and `INT4`, as well as state-of-the-art group-quantized formats used in `SmoothQuant` [44] (sq8) and `AWQ` [29] (awq4). This range of representations allows us to evaluate the compressibility and runtime performance of our method across both standard floating-point formats and modern quantized weight representations used in LLM inference.

Hardware platforms. All compression-size and runtime performance experiments for our GPU kernels are conducted on two GPU servers to demonstrate the portability of our design across different GPU generations. The first server is equipped with eight NVIDIA A100 GPUs (80 GB HBM2e, 2 TB/s peak bandwidth). We further evaluate the performance on NVIDIA Hopper H200 GPUs. Experiments are implemented using PyTorch 2.5.1 and CUDA 12.1, with the CUTLASS-based GEMM baselines across platforms to ensure fair comparison.

B. Entropy Characterization and Compression Bound

Figure 6 provides an overview of the effective bits-per-weight achieved by our tile-level ANS compression compared against both the nominal storage formats and the Shannon entropy bound across the evaluated models. We have the following observations.

First, across all models and datatypes, including bf16, fp8, int8, group-quantized formats (sq8, awq4), and low-bit representations (fp4, int4), our ANS bitrates closely track the Shannon bound, typically within 0.01–0.05 bits. This

Model	Sequence Length	Variant	Weight Mem (GB)	KV Mem (GB)	Total Mem (GB)	Max Batch	Throughput (Token/s)	Median TPOT (ms)
Qwen-14B Budget: 80 GB (Single NVIDIA A100 GPU)	1024	Uncompressed	27.5	44.1	75	47	1131	71
		Ours	18.1	56.3	75	**60 (1.3×)**	1217	81
	2048	Uncompressed	27.5	43.1	74	23	548	112
		Ours	18.1	56.3	75	**30 (1.3×)**	651	125
Mixtral-176B Budget: 320 GB (Four NVIDIA A100 GPU)	1024	Uncompressed	261.9	26.3	304	20	241	110
		Ours	163.7	124.6	304	**95 (4.8×)**	391	170
	2048	Uncompressed	261.9	26.3	304	10	190	213
		Ours	163.7	123.4	304	**47 (4.7×)**	257	318

TABLE II: Illustration of memory footprint before and after lossless compression. Compression reduces the weight of memory, freeing capacity for larger KV-cache and enabling larger effective batch sizes and throughput at the same sequence length.

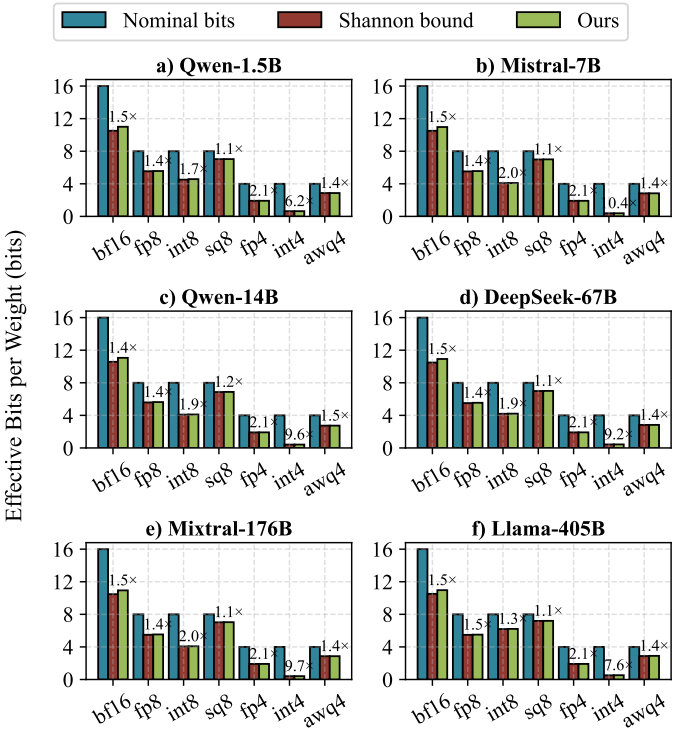


Fig. 6: Effective bit rates of tile-level ANS compression relative to Shannon entropy bounds.

near-perfect overlap indicates that the ANS encoder captures essentially all statistical redundancy in the weight distribution. The agreement is particularly tight for lower-bit formats such as int8, sq8, fp4, and int4, where the symbol alphabet is small and the empirical distribution is sharply peaked, allowing ANS to encode nearly at the theoretical optimum.

Second, the only consistent deviation appears for bf16, where the 16-bit symbol alphabet leads to a broader histogram and a larger frequency table. The finite-precision normalization required for 212-entry frequency tables introduces marginal overhead (roughly 0.1–0.2 bits), which is expected for high-cardinality distributions and well within the limits predicted by finite-precision ANS theory. Importantly, even in this worst case, our ANS bitrate remains within 1.1–1.5× of nominal precision, significantly closer to the entropy limit than any existing lossless scheme.

Third, the figure highlights that nominal storage formats substantially over-allocate bits relative to the intrinsic information content of the weights. For example, bf16 weights often exhibit an effective entropy of only 10–12 bits, int8 typically compresses to 4–5 bits, and even aggressively quantized formats such as int4 retain only 0.6–1.0 bits of true entropy. These gaps are consistent across all model scales, from Qwen-1.5B to Llama-405B, demonstrating that the heavy-tailed, highly structured nature of transformer weight distributions persists uniformly across architectures and sizes.

C. End-to-End Performance Gains from Reduced Weight Memory Footprint

We evaluate the end-to-end inference impact of our reduced memory footprint under realistic device-memory budget constraints using the SGLang serving framework. We modify SGLang [50] to use our ANS-enabled GEMM backend for dense matrix multiplications, comparing it to the default CUTLASS-based kernels in the SGLang runtime.

Because our approach significantly reduces the memory footprint of model weights, it allows substantially larger query batches to fit within a fixed GPU memory budget. We therefore evaluate the resulting end-to-end throughput improvements under realistic inference workloads. Table II reports the memory breakdown, maximum feasible batch size, achieved throughput, and median time-per-output-token (TPOT) for Qwen-14B and Mixtral-176B across two representative sequence lengths (1024 and 2048). For Mixtral-176B, multi-GPU inference is implemented using expert parallelism (EP) across four GPUs. The reported throughput corresponds to measured execution time under SGLang's batching scheduler running on GPUs.

Although our on-the-fly decompression kernel introduces additional computation compared to a pure CUTLASS GEMM kernel used in existing LLM serving systems, reducing the weight footprint directly increases the effective batch size and enables higher serving throughput. For example, on a single A100 with sequence length 1024, Qwen-14B increases the maximum batch size from 47 to 75, improving throughput from 1131 to 1217 tokens/s (1.1×). For longer sequences (2048), throughput increases from 548 to 651 tokens/s (1.2×).

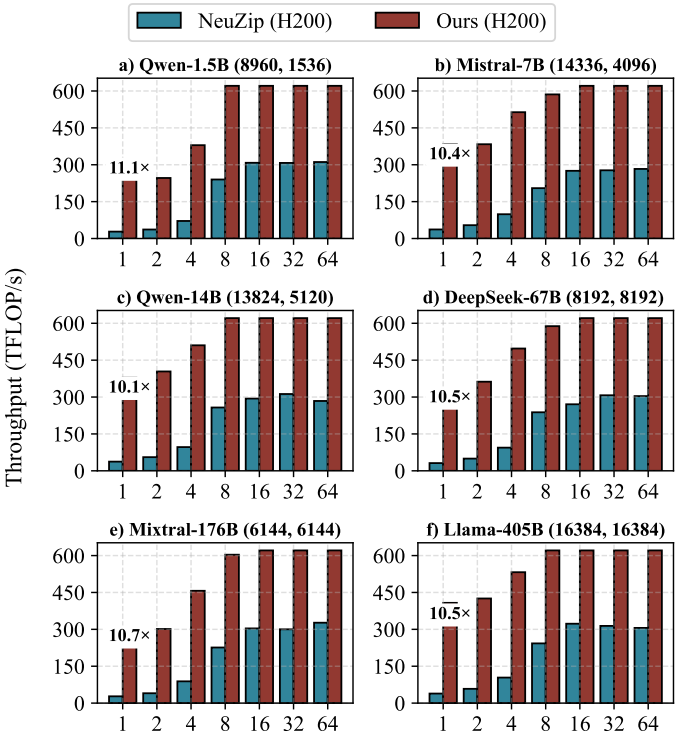


Fig. 7: Comparison between our on-the-fly decompression path and the NeuZip [17] baseline across varying batch sizes on NVIDIA H200. Weight matrix dimensions are given as (a, b)

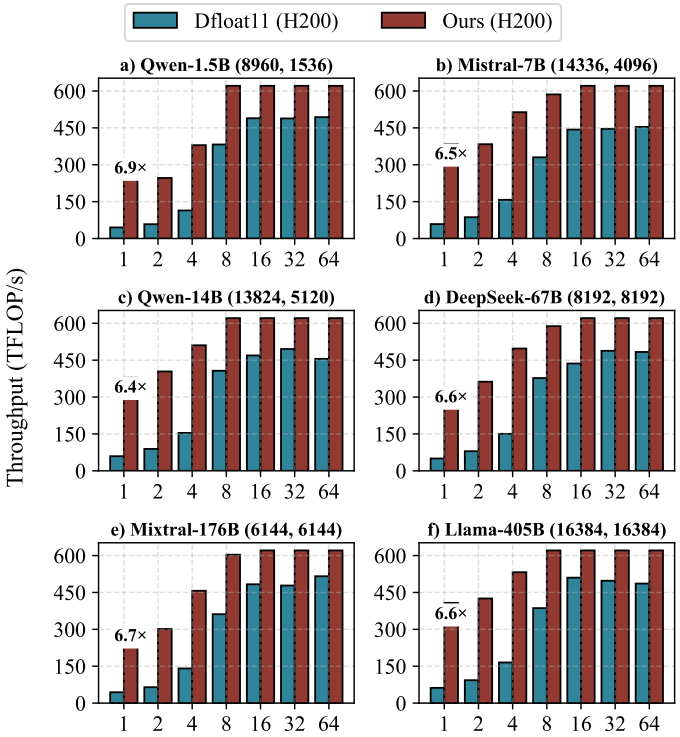


Fig. 8: Comparison between our on-the-fly decompression path and the DFloat11 [49] baseline across varying batch sizes on NVIDIA H200. Weight matrix dimensions are given as (a, b)

The effect is more pronounced for larger models. On four A100 GPUs, Mixtral-176B increases the feasible batch size from 20 to 95 at sequence length 1024, resulting in a throughput improvement from 241 to 391 tokens/s (1.6×). At length 2048, throughput improves from 190 to 257 tokens/s (1.4×). These results highlight an important system-level effect: compression fundamentally shifts the bottleneck of LLM inference from device memory capacity to compute throughput. The gains come from the reduced weight memory, which allows more requests to share the KV-cache capacity, improving batching efficiency under realistic workloads.

We also report the median TPOT to capture the latency impact of the compressed execution path. Our design primarily targets throughput-oriented serving workloads by increasing the effective batch capacity under a fixed memory budget. While TPOT slightly increases due to the additional on-the-fly decompression cost, the increase is modest compared to the substantial throughput gains enabled by larger batching for the throughput-oriented LLM model serving.

D. Comparison to SOTA Lossless Compression LLM

We compare our fused on-the-fly decompression pipeline with two recent lossless LLM compression systems, NeuZip [17] and DFloat11 [49] on NVIDIA H200 hardware. We evaluate six representative LLMs of different scales, and each subplot reports the achieved throughput in TFLOP/s as the batch size increases from 1 to 64 of a single layer of projection with 4096 input tokens. The annotation in each plot indicates the peak relative improvement of our method compared with baseline.

Both prior approaches target floating-point formats (e.g., FP16/BF16) and perform layerwise decompression, where an entire compressed layer must first be reconstructed in global memory before GEMM execution. This introduces additional memory traffic and synchronization overhead. In contrast, our method decodes compressed weights at tile granularity and feeds them directly into the GEMM pipeline. This eliminates global-memory materialization of decompressed layers and overlaps decompression with tensor-core computation.

Figure 7 and Figure 8 show that our approach outperforms both baselines across all evaluated models and batch sizes on NVIDIA H200. Compared with NeuZip, the fused pipeline achieves up to $\sim 10\times$ higher throughput, while outperforming DFloat11 by ~ 6–$7\times$. These gains show that tight integration of entropy decoding with tiled GEMM execution is essential for high-performance compressed LLM inference.

E. Performance Analysis

To evaluate the performance of the on-the-fly decompression GPU kernel, we conducted three complementary experiments that examined the system from different perspectives. First, we present a step-by-step optimization breakdown that quantifies the performance improvement from a naive ANS decoder to our final fused decompression GEMM pipeline. Second, we evaluated kernel-level performance across different GPU architectures (A100 and H200) under a representative inference

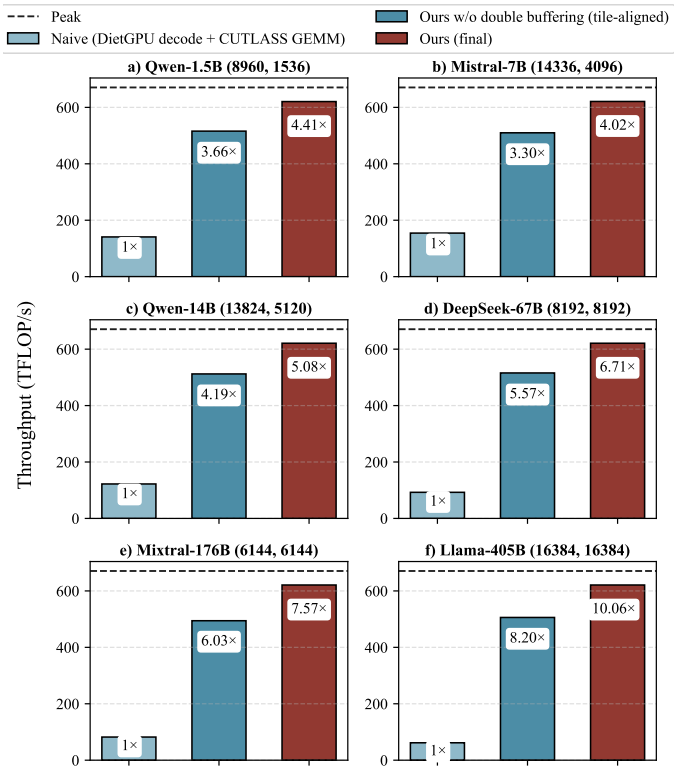


Fig. 9: Breakdown of throughput improvements from a naive ANS decoder to the fused decompression GEMM.

workload with sequence length 4096. Finally, we compare our approach with KTransformer [5], a system designed to handle out-of-memory (OOM) scenarios by offloading model weights, to demonstrate the end-to-end advantages of the compression-based approach for large-model inference.

1) Optimization breakdown of the fused decompression pipeline: Figure 9 shows the step-by-step throughput improvements from a naive GPU implementation (DietGPU ANS decode + CUTLASS GEMM) to our final fused decompression GEMM kernel.

The first improvement comes from aligning decompression with the GEMM tiling schedule. When decoding and GEMM are executed as separate stages, frequent synchronization and global-memory traffic between the two kernels is needed. Our tile-aligned pipeline eliminates these barriers by decoding weight tiles directly into the layout required by the GEMM microkernel, allowing decompression and computation to proceed in a tightly overlapped fashion. This optimization alone improves throughput by $3.3\times$–$8.2\times$ across models (e.g., $3.66\times$ for Qwen-1.5B and $8.20\times$ for Llama-405B).

Despite achieving decompression throughput at the GPU register level in the TB/s range, the decoding stage can still slow down the overall GEMM execution. Therefore, we introduce double buffering in shared memory to prefetch and decode the next tile while the current tile is being consumed by GEMM. This producer–consumer pipeline hides most of the decompression latency and further increases utilization of tensor cores. With this optimization, the final kernel achieves $4.0\times$–$10.1\times$ speedup over the naive baseline (e.g., $4.41\times$ for Qwen-1.5B, $6.71\times$ for DeepSeek-67B, and $10.06\times$ for Llama-405B). The gains become more pronounced for larger matrices, where the decompression overhead can be better amortized and the overlapped pipeline keeps the compute units closer to the peak GEMM throughput.

2) Cross-GPU kernel performance on A100 and H200: Figure 10 and Figure 11 compare our fused on-the-fly decompression kernel with the CUTLASS baseline across varying batch sizes on A100 and H200 GPUs. Across both architectures, throughput increases with batch size as tensor-core utilization improves. As batching grows, our implementation approaches the performance of the native CUTLASS GEMM and in several cases slightly exceeds it.

On A100, our kernel achieves performance close to the baseline across all evaluated models, typically within about $1.0\times$–$1.1\times$ of CUTLASS at larger batch sizes. This shows that integrating tile-level decompression into the GEMM pipeline introduces minimal overhead while maintaining high tensor-core utilization.

The advantage becomes more visible on H200. Due to its larger on-chip memory capacity and improved memory subsystem, the tile-level pipeline can overlap decompression and computation more effectively. As a result, our implementation not only matches but occasionally surpasses the CUTLASS baseline, achieving up to about $1.2\times$ speedup at larger batch sizes. These results demonstrate that the fused decompression GEMM design scales well across GPU architectures and becomes increasingly effective as the batch size grows.

3) End-to-End comparison with KTransformer under memory constraints: Figure 12 reports the prefill-stage throughput of a single representative linear layer drawn from six LLMs, comparing our on-the-fly decompression path against KTransformer across varying batch sizes. In KTransformer, the raw full-precision weight matrix for this layer does not fit into GPU memory, requiring the layer weight to reside in CPU memory and be streamed over PCIe during execution.

With our lossless tile-level compression, the compressed form of the same layer fits fully in GPU memory. As a result, execution remains entirely on-device, eliminating CPU–GPU streaming and yielding consistently higher throughput, up to $7.7\times$ in our layer-level evaluations. This demonstrates that our method provides a practical alternative for improving out-of-memory performance, enabling high throughput without any loss of numerical accuracy.

Figure 13 shows that the decode stage exhibits even larger improvements, where up to $18.1\times$ throughput improvements are achieved, as weight access becomes the dominant bottleneck when the baseline must stream parameters over PCIe. By eliminating these host–device transfers, our on-the-fly decompression path substantially increases throughput.

F. Metadata Overhead Analysis

We explicitly quantify the metadata footprint of our ANS-compressed tiles. Each compressed tile stores a per-tile offset entry into the compressed buffer, while a single ANS codebook

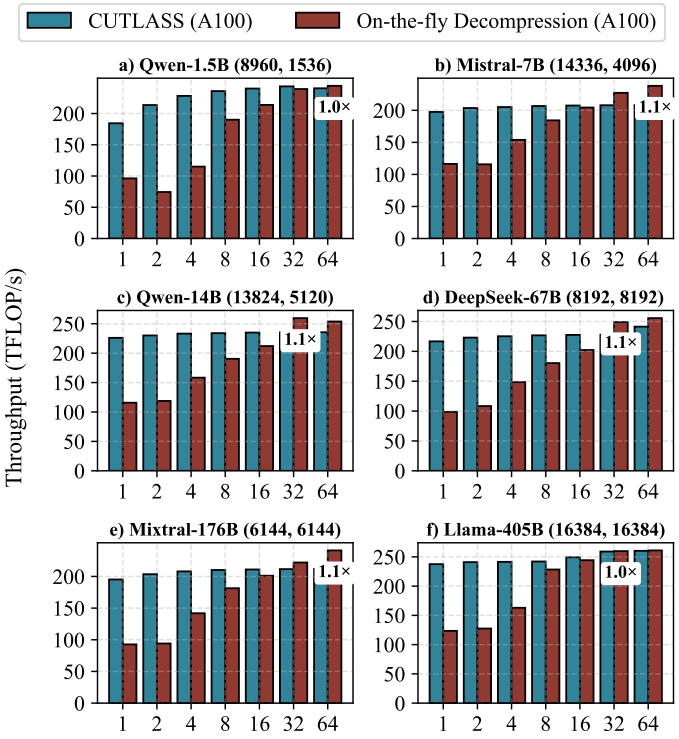


Fig. 10: Comparison between our on-the-fly decompression path and the CUTLASS [34] baseline across varying batch sizes with 4096 input tokens on NVIDIA A100. Weight matrix dimensions are given as (a, b)

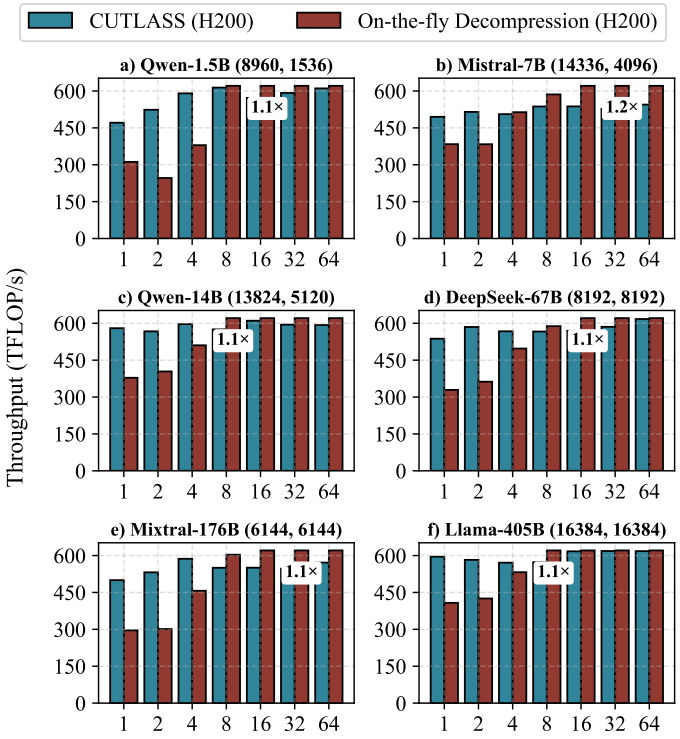


Fig. 11: Comparison between our on-the-fly decompression path and the CUTLASS [34] baseline across varying batch sizes with 4096 input tokens on NVIDIA H200. Weight matrix dimensions are given as (a, b)

is shared across all tiles in the layer. The compact representation is therefore

$$\text{Metadata} = \underbrace{4\,\text{B} \cdot N_{\text{tiles}}}_{\text{offset table}} + \underbrace{(2^b \cdot 4\,\text{B})}_{\text{shared codebook}}, \tag{3}$$

where b is the ANS probability bits (default $b = 12$). For fp16 weights, the tile count is $N_{\text{tiles}} = (K/K_{\text{tile}}) \cdot (N/N_{\text{tile}})$ and we report both percentage of uncompressed size and the effective-bit overhead ($16 \times$ metadata/weight).

a) A100 (32×128): The overhead on global memory is only 0.052%–0.108% of fp16 weights, i.e., an effective-bit overhead of 0.0083–0.0173 bits/weight. This is sufficiently small that the end-to-end compression rate remains close to the Shannon bound for the payload entropy.

b) H200 (64×256): Since H200 has a larger share memory size compared to A100, it allows large tile size up to 64×256, and the metadata overhead becomes only 0.015%–0.072% of uncompressed fp16 weights, corresponding to 0.0024–0.0115 effective bits per weight. In this regime, the overhead is further reduced, so the achieved bitrate is very close to the Shannon limit.

TABLE III: Metadata overhead for `bfloat16` weights (shared codebook + offset table).

Model	A100 (32×128)			H200 (64×256)		
	KB	% of layer	Eff. bits	KB	% of layer	Eff. bits
Qwen-1.5B	29.1	0.108%	0.0173	19.3	0.072%	0.0115
Mistral-7B	72.0	0.063%	0.0100	30.0	0.026%	0.0042
Qwen-14B	83.5	0.060%	0.0097	32.9	0.024%	0.0038
DeepSeek-67B	80.0	0.061%	0.0098	32.0	0.024%	0.0039
Mixtral-176B	52.0	0.071%	0.0113	25.0	0.034%	0.0054
Llama-405B	272.0	0.052%	0.0083	80.0	0.015%	0.0024

VII. Related Work

Data compression on GPU. GPU-based compression has been explored extensively in HPC and ML systems to reduce memory traffic and accelerate data movement. nvCOMP [35] is the most widely deployed GPU compression library, offering optimized CUDA implementations of LZ4, Snappy, GDeflate, and Bitcomp. However, nvCOMP is closed source and exposes only coarse-grained, host-driven APIs, preventing developers from triggering decompression from inside GPU kernels or at higher granularity. Such lack of device-level control makes it impossible to fuse decompression directly into GEMM pipelines or overlap decoding with computation. Although more entropy-efficient codecs exist, integrating them into LLM inference requires tight alignment with tile-level weight access. Our work addresses this gap via tile-granular ANS decompression tightly coupled with GEMM execution, enabling optimizations that nvCOMP's opaque interface cannot support.

NVIDIA inline compression is a hardware mechanism that compresses cache lines in the GPU memory hierarchy to reduce DRAM traffic; it is designed for general workloads

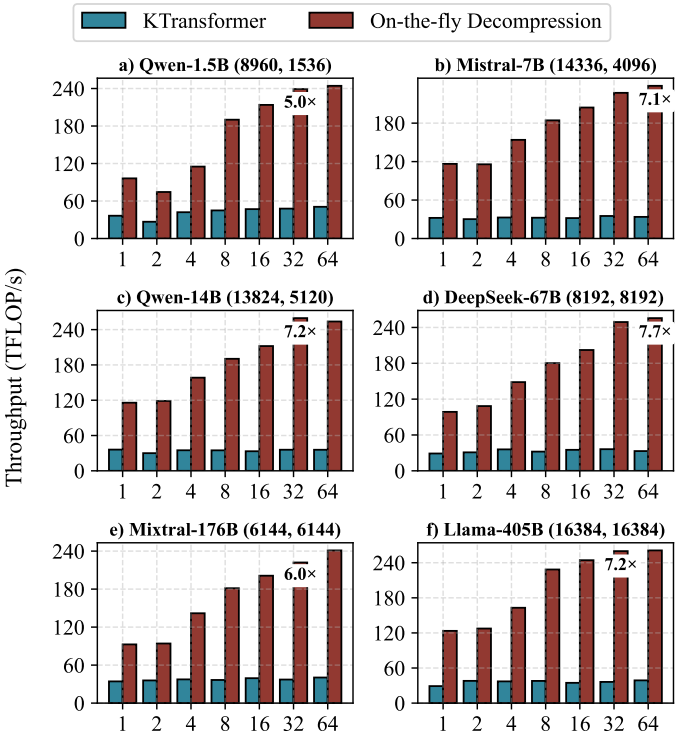


Fig. 12: Comparison between our on-the-fly decompression path and the KTransformer [5] baseline across varying batch sizes in the prefill stage with a 4096-token sequence length. Weight matrix dimensions are given as (a, b)

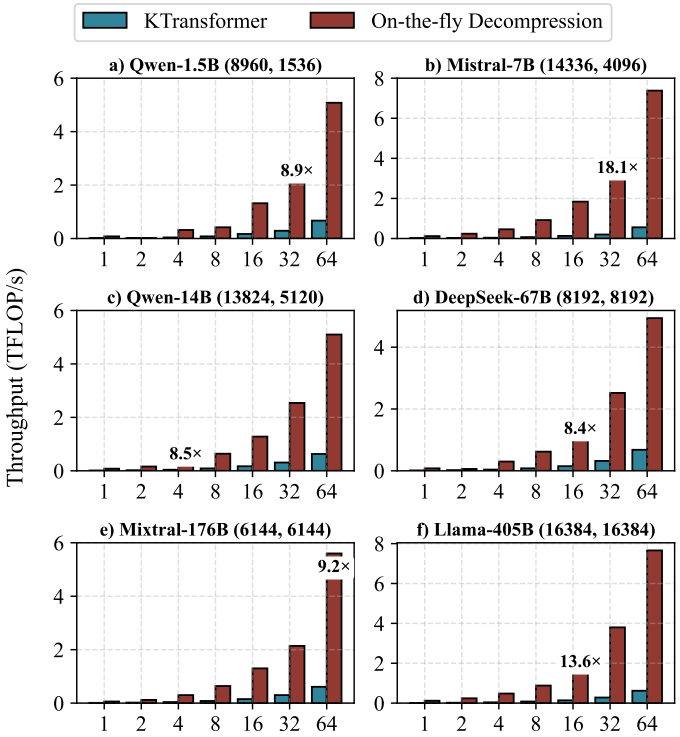


Fig. 13: Comparison between our on-the-fly decompression path and the KTransformer [5] baseline across varying batch sizes in the decode stage. Weight matrix dimensions are given as (a, b)

without application-level knowledge. In contrast, our work focuses on model-aware compression of LLM weights and targets the inherent redundancy in the weight distribution itself. Rather than relying on opportunistic hardware compression, our approach enables near Shannon's limit compression through distribution-aware encoding integrated with the inference pipeline. As a result, the two techniques operate at different layers of the system stack and are complementary.

Model compression with GPU codecs. LLM.265 [45] repurposes video codecs (H.264/H.265) as tensor compressors for LLMs. While leveraging NVENC/NVDEC hardware is attractive, video engines sustain only $\sim$1.1–1.3 GB/s, two to three orders of magnitude below HBM bandwidth, making them the bottleneck. Furthermore, optimizing for perceptual quality rather than numerical fidelity yields accuracy degradation exceeding 5% at sub-3-bit. Our method is strictly lossless and designed for GPU-resident LLM inference.

Quantization, pruning, and low-rank compression. Extensive work reduces model size using lossy compression techniques, including low-bit quantization [10], [18], [25], [29], group-wise and adaptive quantization [15], [29], [33], [36], [44], pruning [16], [31], [47], and low-rank decomposition [20], [26], [39]. While effective, these methods inevitably introduce approximation error or accuracy loss. Our approach is orthogonal: it is fully lossless and can further compress quantized or pruned models. As our entropy analysis shows, even low-bit formats such as FP4, INT4, SmoothQuant, and AWQ retain significant statistical redundancy that lossless compression can exploit.

VIII. Conclusion

We identify a significant gap between the storage bitwidth of LLM weights and their information-theoretic entropy, revealing substantial redundancy even in low-bit formats. To exploit it, we propose a tile-level on-the-fly ANS decompression framework aligned with the GEMM execution pipeline. By decoding weight tiles directly in shared memory and overlapping decompression with tensor-core computation, our fused GPU kernel achieves near-optimal memory reduction without performance overhead, outperforms state-of-the-art lossless methods, and enables larger batch sizes within the memory budget, yielding up to 1.6× throughput improvement for LLM serving. *Future work* will explore extending on-the-fly entropy coding to the *KV cache* for efficient long-context decoding.

Acknowledgment

This research/project is supported by the Ministry of Education AcRF Tier 2 grant (No. MOE-T2EP20224-0020) and Tier 1 grant (No. T1 251RES2315) in Singapore, the Google South & Southeast Asia Research Award 2025.

REFERENCES

[1] M. AI, "The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation," Blog post, Apr. 2025. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[2] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A general-purpose data compressor," *ACM Transactions on Information Systems (TOIS)*, vol. 37, no. 1, pp. 1–30, 2018.

[3] A. Alaparthi, P. Loh, and R. Marcus, "Scalellm: A technique for scalable llm-augmented data systems," in *Companion of the 2025 International Conference on Management of Data*, 2025, pp. 11–14.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. Nips '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[5] H. Chen, W. Xie, B. Zhang, J. Tang, J. Wang, J. Dong, S. Chen, Z. Yuan, C. Lin, C. Qiu, Y. Zhu, Q. Ou, J. Liao, X. Chen, Z. Ai, Y. Wu, and M. Zhang, "KTransformers: Unleashing the full potential of CPU/GPU hybrid inference for MoE models," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 2025.

[6] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of machine learning research*, vol. 24, no. 240, pp. 1–113, 2023.

[7] Y. Collet, *RFC 8878: Zstandard Compression and the'application/Zstd'Media Type*. RFC Editor, 2021.

[8] M. Davies, N. Crago, K. Sankaralingam, and C. Kozyrakis, "Efficient llm inference: Bandwidth, compute, synchronization, and capacity are all you need," *arXiv preprint arXiv:2507.14397*, 2025.

[9] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, W. Zeng, W. Zhao, W. An, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, X. Q. Li, X. Jin, X. Wang, X. Bi, X. Liu, X. Wang, X. Shen, X. Chen, X. Zhang, X. Chen, X. Nie, X. Sun, X. Wang, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yu, X. Song, X. Shan, X. Zhou, X. Yang, X. Li, X. Su, X. Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Y. Zhang, Y. Xu, Y. Xu, Y. Huang, Y. Li, Y. Zhao, Y. Sun, Y. Li, Y. Wang, Y. Yu, Y. Zheng, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Tang, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Wu, Y. Ou, Y. Zhu, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Zha, Y. Xiong, Y. Ma, Y. Yan, Y. Luo, Y. You, Y. Liu, Y. Zhou, Z. F. Wu, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Huang, Z. Zhang, Z. Xie, Z. Zhang, Z. Hao, Z. Gou, Z. Ma, Z. Yan, Z. Shao, Z. Xu, Z. Wu, Z. Zhang, Z. Li, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Gao, and Z. Pan, "DeepSeek-V3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2412.19437

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in neural information processing systems*, vol. 36, pp. 10 088–10 115, 2023.

[11] P. Deutsch, *Rfc1951: Deflate Compressed Data Format Specification Version 1.3*. RFC Editor, 1996.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[13] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *2015 Picture Coding Symposium (PCS)*. IEEE, 2015, pp. 65–69.

[14] M. Effros, "PPM performance with BWT complexity: A fast and effective data compression algorithm," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1703–1712, 2002.

[15] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[16] S. Gao, C.-H. Lin, T. Hua, Z. Tang, Y. Shen, H. Jin, and Y.-C. Hsu, "Disp-llm: Dimension-independent structural pruning for large language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 72 219–72 244, 2024.

[17] Y. Hao, Y. Cao, and L. Mou, "Neuzip: Memory-efficient training and inference with dynamic compression of neural networks," *arXiv preprint arXiv:2410.20650*, 2024.

[18] D. He, A. Jaiswal, S. Tu, L. Shen, G. Yuan, S. Liu, and L. Yin, "AlphaDecay: Module-wise Weight Decay for Heavy-Tailed Balancing in LLMs," *arXiv preprint arXiv:2506.14562*, 2025.

[19] M. Hershcovitch, A. Wood, L. Choshen, G. Girmonsky, R. Leibovitz, O. Ozeri, I. Ennmouri, M. Malka, P. Chin, and S. Sundararaman, "Zipnn: Lossless compression for ai models," in *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*. IEEE, 2025, pp. 186–198.

[20] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models." *ICLR*, vol. 1, no. 2, p. 3, 2022.

[21] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7B," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[22] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[23] J. Johnson, "DietGPU: GPU implementation of a fast generalized ANS (asymmetric numeral system) entropy encoder and decoder, with extensions for lossless compression of numerical and other data types in HPC/ML applications," 2025. [Online]. Available: https://github.com/facebookresearch/dietgpu

[24] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, and H. Yuen, "A study of BFLOAT16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[25] T. Kim, S. Kim, S. Han, W. Cho, Y. Choi, Y. Bae, and S. Hong, "Outlier Matters: A Statistical Analysis of LLM Tensor Distributions and Quantization Effects," in *2025 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC)*. IEEE, 2025, pp. 1–6.

[26] S. A. Koohpayegani, K. L. Navaneet, P. Nooralinejad, S. Kolouri, and H. Pirsiavash, "Nola: Compressing lora using linear combination of random basis," *arXiv preprint arXiv:2310.02556*, 2023.

[27] J. Li, Y. Yang, Y. Bai, X. Zhou, Y. Li, H. Sun, Y. Liu, X. Si, Y. Ye, and Y. Wu, "Fundamental capabilities of large language models and their applications in domain scenarios: A survey," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 11 116–11 141.

[28] Z. Li, Y. Su, R. Yang, C. Xie, Z. Wang, Z. Xie, N. Wong, and H. Yang, "Quantization meets reasoning: Exploring llm low-bit quantization degradation for mathematical reasoning," *arXiv preprint arXiv:2501.03035*, 2025.

[29] J. Lin, J. Tang, H. Tang, S. Yang, Wei-Ming Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of machine learning and systems*, vol. 6, pp. 87–100, 2024.

[30] R. Liu, Y. Sun, M. Zhang, H. Bai, X. Yu, T. Yu, C. Yuan, and L. Hou, "Quantization hurts reasoning? an empirical study on quantized reasoning models," *arXiv preprint arXiv:2504.04823*, 2025.

[31] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21 702–21 720, 2023.

[32] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.

[33] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in

International Conference on Machine Learning. PMLR, 2020, pp. 7197–7206.

[34] NVIDIA Corporation, "CUTLASS: CUDA templates and python dsls for high-performance linear algebra," 2025. [Online]. Available: https://github.com/NVIDIA/cutlass

[35] ——, "nvCOMP: GPU library for high-performance lossless compression and decompression," 2025. [Online]. Available: https://docs.nvidia.com/cuda/nvcomp/

[36] J. Pan, C. Wang, K. Zheng, Y. Li, Z. Wang, and B. Feng, "Smoothquant+: Accurate and efficient 4-bit post-training weightquantization for llm," *arXiv preprint arXiv:2312.03788*, 2023.

[37] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[38] M. Raza, Z. Jahangir, M. B. Riaz, M. J. Saeed, and M. A. Sattar, "Industrial applications of large language models," *Scientific Reports*, vol. 15, no. 1, p. 13755, 2025.

[39] S. Ren and K. Zhu, "Low-rank prune-and-factorize for language model compression," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, 2024, pp. 10 822–10 832.

[40] M. A. Team, ""Cheaper, better, faster, stronger — mixtral 8×22B"," Apr. 2024. [Online]. Available: https://mistral.ai/news/mixtral-8x22b

[41] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[42] X. Wang, A. Amayuelas, K. Zhang, L. Pan, W. Chen, and W. Y. Wang, "Understanding reasoning ability of language models from the perspective of reasoning paths aggregation," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 50 026–50 042.

[43] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[44] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 38 087–38 099.

[45] C. Xu, Y. Wu, X. Yang, B. Chen, M. Lentz, D. Zhuo, and L. W. Wills, "LLM. 265: Video Codecs are Secretly Tensor Codecs," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture®*, 2025, pp. 445–460.

[46] A. Yang, B. Yu, C. Li, D. Liu, F. Huang, H. Huang, J. Jiang, J. Tu, J. Zhang, J. Zhou, J. Lin, K. Dang, K. Yang, L. Yu, M. Li, M. Sun, Q. Zhu, R. Men, T. He, W. Xu, W. Yin, W. Yu, X. Qiu, X. Ren, X. Yang, Y. Li, Z. Xu, and Z. Zhang, "Qwen2.5-1M technical report," *arXiv preprint arXiv:2501.15383*, 2025.

[47] Y. Yang, Z. Cao, and H. Zhao, "Laco: Large language model pruning via layer collapse," *arXiv preprint arXiv:2402.11187*, 2024.

[48] J. Zhang, Z. Shen, S. Yang, L. Meng, C. Xiao, W. Jia, Y. Li, Q. Sun, W. Zhang, and X. Lin, "High-Ratio Compression for Machine-Generated Data," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–27, 2023.

[49] T. Zhang, M. Hariri, S. Zhong, V. Chaudhary, Y. Sui, X. Hu, and A. Shrivastava, "70% size, 100% accuracy: Lossless llm compression for efficient gpu inference via dynamic-length float (dfloat11)," in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[50] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62 557–62 583, 2024.