MINE: ENTROPY-CODED MEMORY-EFFICIENT INFERENCE FOR LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present MINE (Memory-efficient INference Engine), a lossless entropy coding approach that accelerates single-stream inference of Large Language Models (LLMs), where existing lossless compression methods yield speedups only for specific data types such as `BF16` and under additional assumptions. Single-stream decoding is strongly memory-bound, since every weight must be streamed from memory once per generated token and requests cannot be batched in local or low-latency deployments. At the core of MINE is dtANS-LLM, a GPU entropy coding scheme adapted from the prior dtANS to the needs of LLM inference. It exploits the non-uniform distribution of weights, which persists even after quantization, and exposes hyper-parameters that trade compression ratio against decoding cost. We auto-tune these parameters for each tensor and GPU to obtain the Pareto front between model size and decoding latency. MINE integrates into vLLM as a plugin and selects the best kernel configuration for a given memory budget. On Llama 3.1 8B Instruct with a 2.1-bit EntQuant checkpoint, MINE achieves speedups of 2.52× and 1.48× over the fastest baseline on an RTX 4060 Ti and an RTX 5090, at effective bitrates of 2.24 and 3.43 bits per weight. MINE is available as open source.

1 INTRODUCTION

The dramatically increasing use of Large Language Models (LLMs) makes highly efficient inference a crucial requirement. Inference requires streaming and decoding the entire LLM network from RAM while each weight is used only once per request, which is a highly memory-bound setting (Gholami et al., 2024). In data center deployments this is mitigated by collecting many requests into batches, but other settings such as local LLMs or low-latency applications require good single-stream performance. But even in general, the increasing size of LLMs and the current RAM scarcity pose the need for memory-efficient LLM representations.

The most straightforward approach is lower-precision floating-point representations to reduce the memory footprint of the weights while losing accuracy (Micikevicius et al., 2022; Rouhani et al., 2023; NVIDIA et al., 2025). Many post-training quantization methods can be seen as a continuation of this idea to custom grids (Dettmers & Zettlemoyer, 2023) and non-trivial choices of levels (Frantar et al., 2023; Cheng et al., 2024; 2025b; Badri & Shaji, 2023). Others adapt to the importance of the individual weights (Tseng et al., 2024; ggml-org, 2023; Lin et al., 2024). An alternative is Quantization-Aware Training (Jacob et al., 2018; Liu et al., 2025), which can go as far as only supporting three states for each weight (Ma et al., 2024).

An orthogonal approach, which can also be combined with quantization, is lossless compression of LLMs. The weights of LLMs are not distributed uniformly, a property that also holds after applying most quantization methods (Tan et al., 2026). Lossless coding methods exploit these statistical redundancies (Hao et al., 2024; Zhang et al., 2025; Putzky et al., 2026) to compress the weights, where the limit is given by the weights' entropy. They can increase inference throughput since the saved memory allows for larger batch sizes (Tan et al., 2026) or using simplified coding schemes that leverage additional assumptions on the domain (Fan et al., 2026). However, to date, none of these works provides a speedup using generic entropy coding for single-stream decoding on GPU.

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

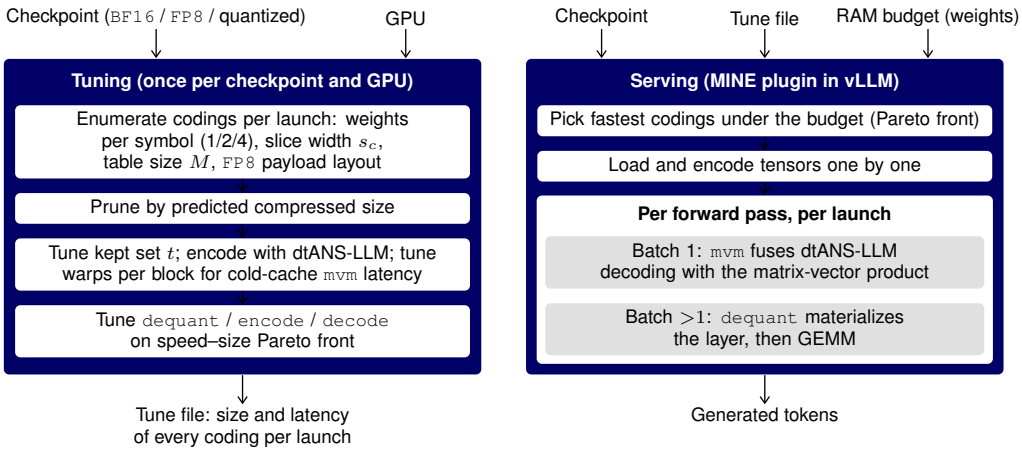


Figure 1: Overview of MINE. Left: tuning, done once for a given checkpoint and GPU; right: inference with the MINE plugin in vLLM.

Contributions. In this paper, we present MINE (Memory-efficient INference Engine), which uses fast lossless entropy coding to compress (quantized) LLMs and achieve inference speedup in the single-stream setting. We adapt an existing method (Schätzle et al., 2026) of entropy coding for GPUs to the particular needs of inference and introduce hyper-parameters to adapt to different settings. Using auto-tuning we then compute the Pareto front between size and single-stream decoding latency to obtain the best possible latency for a given memory budget. We provide highly optimized GPU kernels: `mvm` to perform the needed matrix-vector multiplication and `dequant` to materialize tensors to global memory for prefill. Our specific contributions are as follows:

- We present the entropy coding variant dtANS-LLM, adapted from the existing dtANS (Schätzle et al., 2026) for the specific needs of single-stream inference in LLMs.
- We auto-tune the hyper-parameters of dtANS-LLM for best performance given the characteristics of the tensors and the compute-to-memory-bandwidth ratios of the GPU.
- We integrate dtANS-LLM into the vLLM inference engine (Kwon et al., 2023) as a plugin which we name MINE. Given a memory budget, MINE selects the most promising combination of kernels for performance.
- We evaluate MINE on Llama 3.1 8B Instruct (Grattafiori et al., 2024). On a low-entropy checkpoint (EntQuant (Putzky et al., 2026) at 2.1 bits) it achieves speedups of 2.52× and 1.48× over the fastest baseline on an RTX 4060 Ti and RTX 5090, respectively. The effective bitrates are 2.24 and 3.43 bits per weight, showing that the optimal parameter choices indeed depend on the card.
- We make MINE available as an open-source repository.

2 Overview

We give an overview of MINE and how it is integrated into the existing vLLM engine (Kwon et al., 2023) to efficiently serve entropy-coded checkpoints (model in a specific format/quantization) under a memory budget. MINE consists of two parts, as shown in Figure 1: the tuning (left), which is done once for a given checkpoint and GPU, and the actual inference (right).

Tuning. For the tuning, we first enumerate all (feasible) codings over some design choices: combining 1/2/4 weights into a symbol to code, the number of columns s_c to process per thread, the size of our decoding tables M, and how to represent `FP8` values in the decoding tables. Codings that do not yield a compression in expectation are pruned. Then, for some data types, we search the number of values t to entropy-code (the others are escaped and stored in full precision). We encode using dtANS-LLM and tune how many warps to put in a block to minimize cold-cache `mvm` latency. We compute the runtime vs. model size Pareto front and, for each point on it, also tune warps per block

for the other kernels `dequant` / `encode` / `decode`. Since the described search space is still vast, we will show later how we further prune it to keep tune times manageable and runtimes competitive. The final output is collected in a tune file.

Inference. For the inference, MINE gets as input a checkpoint together with the computed tune file and the RAM budget. The RAM budget applies to the weights only, so the user has to leave some headroom for KV cache and materialized tensors/activations for prefill. MINE then searches the runtime vs. model size Pareto front for the fastest combinations of codings for all kernel launches under the budget. This is a multiple-choice knapsack problem, which we solve using dynamic programming (Kellerer et al., 2004). Then, it loads the uncompressed tensors to the GPU one by one and encodes them. Single-stream decoding (batch size 1) is then served by `mvm`, and for all other batch sizes, `dequant` materializes the weights to global memory followed by a GEMM which might benefit from L2 cache.[1]

3 Background: Fast Fused Entropy Decoding

We start this section by briefly explaining the main aspects for efficient GPU inference. Then, we give a high-level explanation of the entropy coding methods dtANS-LLM builds on. Entropy coding describes the idea of compressing a stream of symbols from an alphabet Ω using close to $H(P)$ bits per symbol where $H(P)$ is the Shannon entropy of the distribution P of the symbols in the stream. $H(P)$ is a lower bound for the size of the compressed stream (Shannon, 1948).

Efficient GPU Code. On NVIDIA GPUs, instructions are issued to *warps* of 32 *lanes*. For memory accesses to global memory to be efficient, neighboring threads have to access consecutive addresses (coalescing). CUDA also exposes fast on-chip *shared memory* which is organized in 32 banks. In each cycle, each bank can provide one 32-bit *word*, and requests causing a c*-way bank conflict* are serialized into c transactions (with 2-way conflicts being free on consumer GPUs for 32-bit accesses (Greg, 2023)). Lastly, warps are grouped into *blocks* which are then executed on *streaming multiprocessors* (SMs). Each SM can execute as many blocks (and thus threads) as the tightest limit over several limited resources allows: threads, registers, and shared memory. The *occupancy* describes the resulting number of threads/warps resident on any SM in parallel (NVIDIA Corporation, 2025).

tANS. *Tabled Asymmetric Numeral Systems* (tANS) (Duda, 2013) is an entropy coding method that achieves rates close to $H(P)$ with reasonable computational cost. The name comes from the existence of a *decoding table* with $M = 2^m$ entries, each containing one of the occurring symbols, a new state x', and a number k of bits to load from a compressed bit stream v. The decoder maintains a state x. It uses the last m bits of x to perform a look-up into the decoding table which yields the decoded symbol and the information needed to construct the next state $x = x' \ll k \mid v_k$ where v_k are the next k bits of v. Encoding works by simulating the actions of the decoder in reverse. The encoded representation consists of a bit stream and a decoding table. The two main issues toward a fast GPU implementation are the dependencies between decoding subsequent symbols (limited instruction-level parallelism (ILP)) and the need to load a variable number of bits from each thread after each symbol (uncoalesced accesses) (Schätzle et al., 2026).

dtANS. Schätzle et al. (2026) alleviated these issues by splitting the decode and state reconstruction. They use tables of size $M = 2^{12}$ and obtain the table indices by rearranging a state of three 32-bit words into an *octet* of 12-bit indices. Then, they can decode the symbols in parallel and obtain 8 *digit/base pairs*, which can be appended to some state x using Mixed Radix Numeral Systems. This increases ILP. Then, as many full 32-bit words as possible are extracted from x. As dtANS limits the base in the table to 2^8, x can grow by at most 2^{64} and at most 2 words can be extracted. Words that cannot be extracted are *conditionally* read from a stream in global memory instead. For these reads to be coalesced, intra-warp communication is required. The third word needed for the next round is always loaded *unconditionally*. Limiting the base to 2^8 causes each symbol to compress to at least $12 - 8 = 4$ bits, which limits the achievable rate for very skewed distributions (i.e., when some symbol has probability higher than $2^8/2^{12} = 6.25\%$).

[1] For `E4M3` we `decode` instead and call the corresponding vLLM kernel.

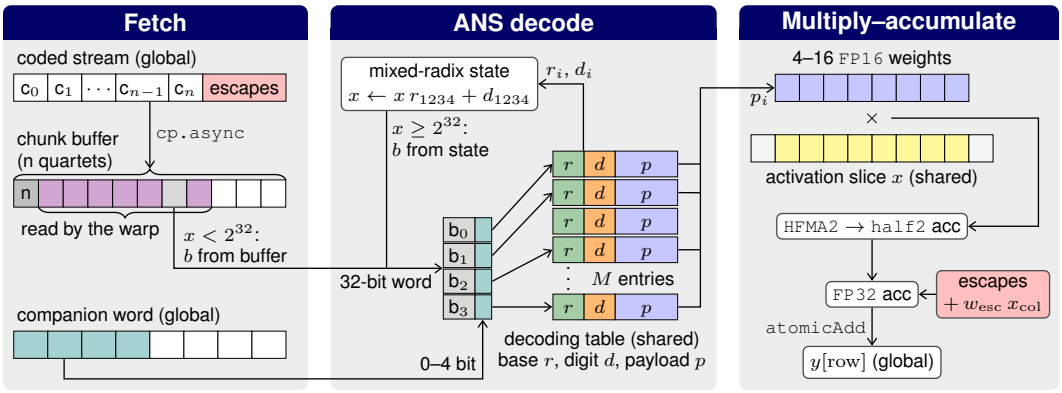


Figure 2: Overview of the performance-critical `mvm` kernel. Compressed data is fetched and decoded into weights which are used in place for fused matrix-vector multiplication with activations.

4 MINE FOR FAST SINGLE-STREAM INFERENCE

In this section, we present MINE for fast single-stream inference. We explain our optimized entropy coding method dtANS-LLM and how we obtain the parameters for optimal single-stream decoding performance under memory constraints. Figure 2 visualizes the performance-critical `mvm` kernel.

Grid Layout. We split each tensor into $s_r \times s_c$ tiles. Each tile is mapped to one warp where each lane processes one row. Tiles are grouped into blocks vertically, such that blocks share input dimensions and can cache their activations. The data type used for activations follows the vLLM baseline for `FP8` and `BF16` but is `FP16` for the quantized checkpoints (vLLM uses `BF16`).

4.1 DTANS-LLM

We describe how we fuse an adapted dtANS (Schätzle et al., 2026) into the kernels efficiently.

Fetching Data. dtANS uses intra-warp communication to load the stream from global memory in coalesced form. As this approach suffers from high latency, we introduce a shared memory buffer which we explain following the left block of Figure 2. The buffer comprises a *chunk* of 1 KiB per warp and is copied using `cp.async`. The first word of every chunk counts after how many iterations n the next chunk has to be loaded (takes the loop iteration off the critical path), upon consumption the remaining data is partially flushed into the lanes (remaining lanes read from the next chunk), and the metadata ensures that only the required part of the last chunk is copied.

Decoding Tables. dtANS-LLM uses the *merged* 32-bit table design from dtANS which reserves 8 bits for a digit and base each such that 16 bits remain for the *payload*. In contrast to dtANS, it stores the base directly instead of its decremented counterpart which limits symbols to 255 slots and thus loses up to $\log_2(256/255) \approx 0.006$ bits for each symbol that would have used 256 in an optimal approximation. While dtANS demonstrated how to use the decremented value without additional instructions, it did not consider the critical path which is shorter for trivial accumulation.

Data Types. dtANS-LLM is designed to support a variety of payloads. For now, these are `BF16`, `FP8`, `FP4`, and `int2` / `int3` / `int4`. Out of these, only `BF16` requires specialized handling because a decoding table for all 2^{16} values would be too large for shared memory. Given that the 1-bit sign and 7-bit mantissa bits are distributed almost uniformly and independently of the 8-bit exponent, we decided to follow the established idea of only encoding the exponent and storing sign and mantissa as a plain byte (Hershcovitch et al., 2025). In our own experiments, we also noticed that the distributions of `FP4` and `int2` / `int3` / `int4` are quite uniform, while those of `FP8` and the exponent of `BF16` are very skewed (many values appear very infrequently if at all).

Given that our table entries support 16-bit payloads, we could combine up to 8 `int2`, 5 `int3`, 4 `int4` / `FP4`, and 2 `BF16` / `FP8` values in a single symbol. Observe how constructing k-tuples over

t values calls for $M \geq t^k$ to make every combination encodable. In practice, to also get a good rate, we need a multiple of that. As tables are stored in shared memory, they should not be too large, and we limit `int2` to Quad mode ($k = 4$) and `int3` / `int4` / `FP4` to Pair mode ($k = 2$). These limits also enable payload designs that allow for efficient unpacking.

Escapes. By the same logic, `BF16` and `FP8` are effectively limited to Single mode ($k = 1$). However, as many symbols are rare, we restrict the decoding table to the t most frequent symbols. This enables encoding 2 values per symbol if $t^2 \leq M$ but requires a mechanism to handle values that are not in the table. dtANS resolves this issue with a special escape symbol that causes the decoder to read a full-precision value, which in turn requires communication and memory transactions. Differently, dtANS-LLM encodes a 0 instead and adds the actual value to a sparse correction stream. This reconciles fast processing of the dense matrix and the rare values. The correction stream is appended directly to the ANS stream and contains one 32-bit *record* for each escape: a 16-bit weight, an 11-bit column index (relative to the start of the warp's slice), a 1-bit flag (whether the current escape is the last for the row), and a 4-bit counter (for how many rounds to pause before processing the record). Each lane processes the escapes of its assigned row, and memory accesses are coordinated as the conditional loads were before. Thus, the procedure takes as many steps as there are escapes in the row with the most records. In practice, escapes are sufficiently random and thus spread. The 4-bit counter is straightforwardly used to space out reads equally, which gives good performance in practice. We tried more advanced scheduling but it yielded no measurable returns.

Efficient Innermost Loop. In dtANS, storing packed indices and words for the next iteration requires 3 registers each. For different M, this is aggravated: $M = 2^9$ requires 9 each (lowest common multiple of 9 and 32 is 9×32) and $M = 2^{10}$ requires 5. As dtANS-LLM should support multiple M to cater to the different data types, we simplify the innermost loop as shown in the middle block of Figure 2. Our design only takes 8 bits from the ANS state and the remainder from an uncompressed companion stream. The ANS state now only covers a single word and each iteration only decodes a *quartet*. If the number of bits read from the companion stream is a power of two, unrolling the innermost loop such that an entire word is consumed reduces overhead drastically. Hence, we support $M \in \{2^8, 2^9, 2^{10}, 2^{12}\}$ with 0, 1, 2, and 4 companion bits, requiring unrolling of 0, 8, 4, and 2 quartets, respectively. Our scheme has the additional advantage of having less than one word of information in the internal state at the end of the loop whereas dtANS had at most two (this information cannot be used anymore and is thus a constant overhead).

Fused `mvm`. The `mvm` kernel required for fast single-stream inference fuses a dense matrix-vector product of a layer with an activation vector into the decoding. A visualization can be found in the right block of Figure 2. It is highly optimized, accumulates 64 weight-activation products (but at most all of a group) in `FP16` before flushing to `FP32`, unrolls the innermost loop such that the flush and handling of the companion stream do not have to track state, and for quantized checkpoints with row-wise scales/zeros dequantizes efficiently per group/slice in an *epilogue* rather than per weight.

4.2 Auto-Tuning

In this section, we explain how MINE chooses the hyper-parameters for dtANS-LLM that obtain optimal performance for single-stream decoding, and thus the `mvm` kernel is the only runtime tuning objective. We do not only store the fastest coding but compute the speed–size Pareto front to later find the optimal coding under memory constraints (smaller codings might be more computationally expensive to decode). We first explain tuning for the generic case and then illustrate the special handling for the sparse alphabets of `FP8` and `BF16`. MINE also contains a `dequant` kernel that materializes the layer to global memory, and `encode` / `decode` kernels which are not on the critical single-stream decode path. These additional kernels are only tuned on the points that made it to the Pareto front (with the details being omitted for brevity).

Compressed Size Prediction. In order to prune the search space and tune efficiently, we will require accurate estimates of the compressed size. To this end, we use the fact that the compressed size can be closely predicted by the cross-entropy between the distribution and its approximation by the tables. It can be computed efficiently from a histogram (which we keep in a cache) and is a

lower bound on the actual compressed size because dtANS-LLM adds an almost constant overhead. Thus, its argmin is close to the argmin of the compressed size.

Launch. In most settings, the computational graph of an LLM exposes parallelism to some degree, allowing multiple matrix-vector products to be computed in parallel. Thus, grouping these together to share a single kernel *launch* can significantly reduce overhead, increase device utilization, and improve latency. For simplicity we require these *sections* in a launch to have the same input dimension and quantization method/parameters. For the default transformer architecture, there are 4 launches per layer: q/k/v, gate/up, o_proj, and down_proj. To keep our search space over `mvm` kernels small, guarantee similar work per block, and use resources optimally (mixing pays the maximum of each resource), we choose mode, s_c, and M per launch. Each combination constitutes a *coding*, and which we evaluate depends on the launch. First, modes are chosen depending on the data type: Single and Pair are supported by all and Quad only by `int2`. Second, s_c are chosen from $\{64, 128, 256, 512, 1024, 2048\}$ such that they divide the input dimension, allow sufficient unrolling, and are a multiple of the group size (if applicable). By choice, s_c is also a multiple of the 64 weight-activation products that are accumulated in `FP16` before being flushed to `FP32`. Third, M are taken from $\{256, 512, 1024, 4096\}$ in order until our lower bound increases by more than 1% over the minimum observed until then (larger tables require more shared memory and computation, so they must pay for themselves in smaller size). If vLLM kernels are available, we add their runtime and size to the Pareto front and skip codings whose lower bound exceeds the uncompressed size.

Tuning Grid. For each coding, we want to find the optimal number of blocks per SM and warps per block. We start by computing for each number of blocks per SM how many warps each block can have to still fit in shared memory and the threads per SM limit. Then, we take the maximum number of warps per block that still compiles without register spilling when we provide nvcc with the corresponding launch bounds (we can maintain a cache of compiled binaries). When serving, the cache will be cold and thus we need accurate cold-cache measurements for the time taken by the launches. To this end, we reproduce the kernel as many times as necessary to fill the L2 cache at least twice. Then, we capture the compute graph and take the median runtime of 5 bursts, each of which comprises as many replays as required to spend 400 μs. The data is then added to the front.

Tuning t. For `FP8` and `BF16` there is another axis to tune: the number $t \leq 256$ of weights/exponents to store in the decoding tables. Values that are not stored in the table are escaped and require an additional 4-byte record. By entropy, it is optimal to keep the t most frequent codes (and always 0 as we need it for escapes). We can adapt our lower bound on the compressed size accordingly by adding the size of the records (we already have the necessary histogram). Generally, we must assign at least one table slot for each payload tuple, so $t \leq M$ for Single and $t^2 \leq M$ for Pair mode. As $M \geq 256$, this is trivially fulfilled for Single and for Pair we require $t \leq 16, 22, 32, 64$ for $M = 256, 512, 1024, 4096$, respectively. Since good choices of t depend on the distribution of weights in the tensor, we search it per section but in parallel per launch (to keep the search space linear). So for each coding, we first use the lower bound to find an estimate of the byte-optimal t and tune the warps per block for them. The expected runtime of `mvm` depends on two aspects: the amount of memory to transfer (increases for smaller and larger t) and compute (decreases with t). Thus, reducing t would increase compute without reducing transfer size and consequently is not expected to win. In contrast, increasing t will decrease compute while possibly increasing transfer size, pushing the kernel to the memory-bound regime (left in a roofline plot (Williams et al., 2009)). Thus, for each section, we compute the escapes-saved-per-additional-byte ratio when incrementing t (while t can still be increased). Then, we increment the t with the highest ratio until the memory transfer size is about to increase by more than 1% (or all t are at the cap), tune that selection, and iterate. As before, if the lower bound is above the plain tensor size, we stop considering the coding (which might already happen at minimal t). Lastly, we bisect runtime gaps larger than 1% (on top of measurement noise) to get a smooth front.

Floats. Lastly, for `FP8`, we tune two payload layouts for Pair (Single stores `FP16`): `E4M3` which has to be converted to `FP16` and `E5M2` which can simply be used as the high byte of `FP16` (Micikevicius et al., 2022). The former has a constant overhead while the latter requires escaping values that are not representable. In practice, the latter works well only for EntQuant (Putzky et al., 2026) because we observe that most codes lie at the subnormal floor, which can be represented in `E5M2`.

5 Related Work

Various works have explored schemes to quantize the weights of LLMs as discussed in Section 1. To date, far fewer have considered lossless entropy coding due to inherent overhead. Decompressing into a global buffer (Hao et al., 2024; Zhang et al., 2025; Putzky et al., 2026) yields good compression but negative speedups for single-stream inference due to the additional latency. Some works use the freed RAM for larger batch sizes to increase throughput (Tan et al., 2026; Liberman et al., 2026). Other approaches employ domain-specific techniques to obtain good compression and little speedup (Fan et al., 2026; Liberman et al., 2026). In Section 6, we compare MINE in the challenging single-stream decoding setting to the four of these works with source code available.

There are also works employing entropy coding to (de)compress to/from global memory (Weißenberger & Schmidt, 2018; 2019; Yamamoto et al., 2020; Johnson, 2022; Lin et al., 2023; NVIDIA Corporation, 2020) and shared memory (NVIDIA Corporation, 2026c). Recently, a fast fused entropy decoding method has been proposed for another workload (Schätzle et al., 2026). There have also been considerations to implement entropy coding in hardware (Lal et al., 2017; Yubeaton et al., 2025; Cheng et al., 2025a).

6 Evaluation

We evaluate MINE on Llama 3.1 8B Instruct (Grattafiori et al., 2024) in 19 formats/quantizations and on two GPU cards. The model has a convenient size and fits one card uncompressed but not the other. First, we show an extensive steady-state single-stream decoding throughput comparison between MINE, vLLM (Kwon et al., 2023; Frantar et al., 2025; inclusionAI, 2026), and related works (Zhang et al., 2025; Hao et al., 2024; Putzky et al., 2026; Fan et al., 2026). Then, we contrast these results with the prefill performance (for which MINE is not optimized).

Experimental Setup. We conducted our experiments on two GPUs: an RTX 4060 Ti / RTX 5090 with 16 / 32 GB of VRAM, 32 / 96 MB L2 cache, and 34 / 170 streaming multiprocessors (SMs) with 128 KB L1 cache each (of which up to 100 KB can be used as shared memory). We highlight that the former card has 2.64 ALUops/bandwidth-bit compared to 2.21 (19% more).[2] They were put in the same machine with an Intel Xeon w5-2465X with 33.75 MB L3 cache and 128 GB DDR5 ECC RAM. As an operating system, we installed Ubuntu 24.04.4 LTS with kernel 7.0.0-31-generic and NVIDIA driver 595.84 (open). MINE's kernels were compiled via torch using nvcc 13.1.115, with flags `-O3 --use_fast_math` and pinned architecture `-gencode arch=compute_89,code=sm_89` / `arch=compute_120,code=sm_120`.

Checkpoints. We evaluated MINE on 19 checkpoints of Llama 3.1 8B Instruct: `BF16`; `E4M3` (Micikevicius et al., 2022); `MXFP4` (Open Compute Project, 2023; Rouhani et al., 2023); `NVFP4` (NVIDIA et al., 2025); EntQuant with 2.1 / 3.0 / 3.9 bits of entropy per weight (Putzky et al., 2026); asymmetric RTN with 2 bits (groups of 64) and 3 / 4 bits (groups of 128) (Dettmers & Zettlemoyer, 2023); GPTQ with groups of 128 and 2 (asymmetric), 3 (asymmetric), and 4 (symmetric) bits (Frantar et al., 2023); symmetric AutoRound with 2 bits (groups of 128 / 64 (Cheng et al., 2025b)) and 4 bits (groups of 128) (Cheng et al., 2024); HQQ with 2 bits (groups of 32 in column dimension) and 4 bits (groups of 64) (Badri & Shaji, 2023); and asymmetric AWQ with 4 bits (groups of 128) (Lin et al., 2024). The `lm_head` is always stored in `BF16` but compressed as well. We also considered EXL3 (turboderp, 2025) based on Tseng et al. (2024) and GGUF (ggml-org, 2023) i-quants (Kawrakow, 2024), but they would have required specialized fused kernels and did not exhibit any entropy headroom (so tuning would most likely have picked the default kernels). Tune times ranged from 14.3 minutes (HQQ, 4 nominal bits, groups of 64; RTX 4060 Ti) to 262.3 minutes (EntQuant, 2.1 bpw; RTX 5090). Generally, `FP8` checkpoints take the longest due to the two payload formats and the additional search dimension over t (especially if there are many escapes).

Steady-State Decode Throughput. We evaluate MINE's steady-state single-stream decoding throughput on the 19 checkpoints and compare it to performance-optimized baselines where available: vLLM 0.27.1 (Kwon et al., 2023) with its cuBLAS (NVIDIA Corporation, 2026b) / CUTLASS

[2] We consider three-operand operations that have a throughput of 64 per SM per cycle. Further, we used frequencies measured during our actual workload: 2790 MHz on the 4060 Ti and 2917 MHz on the 5090.

(NVIDIA Corporation, 2026a), Marlin (Frantar et al., 2025), and Humming (inclusionAI, 2026) kernels. For HQQ, we did not find any high-performance baseline. We also include memory-optimized baselines that have source code available: DFloat11 (Zhang et al., 2025), ZipServ (Fan et al., 2026), and two engines which use nvCOMP (NVIDIA Corporation, 2020) for entropy coding: NeuZip (Hao et al., 2024) and EntQuant (Putzky et al., 2026). All of these required a different software stack configuration which we omit for brevity. The throughput was determined using the runtime difference between a 64- and 128-token generation following a random 512-token prompt. The runtimes were measured as the median of 3 runs.

In Figure 3 we summarize the results. For each quantization method, we list the checkpoints (CPs), and for each checkpoint the inference engines that serve it. Then, starting from the left, we give the size of dequantization side streams, entropy bits per weight, achieved bits per weight, and single-stream (batch 1) steady-state decode throughput. The color of the bar corresponds to the engine.

We highlight two key observations. First, MINE achieves the best throughput on all checkpoints. This is expected behavior because MINE can pass layers on which it cannot improve to vLLM. The maximum speedup over the fastest competitor is 2.52× / 1.48× and is achieved on EntQuant with 2.1 bits. As expected, this is the checkpoint with the most relative entropy headroom and we highlight that MINE's throughput is on par with conventional 2-bit quantization schemes on the RTX 4060 Ti. Generally, the true entropy coding baselines (DFloat11, NeuZip, and EntQuant) are at least 3.52× slower than MINE. Only ZipServ shows comparable performance. While making strong assumptions about the distribution of values, achieving slightly worse bpw, and being limited to `BF16`, it only achieves 0.88× / 0.72× MINE's throughput. However, it builds on vLLM 0.9.1, which reaches 91.8 tok/s on the RTX 5090, resulting in a 2.6% speedup.

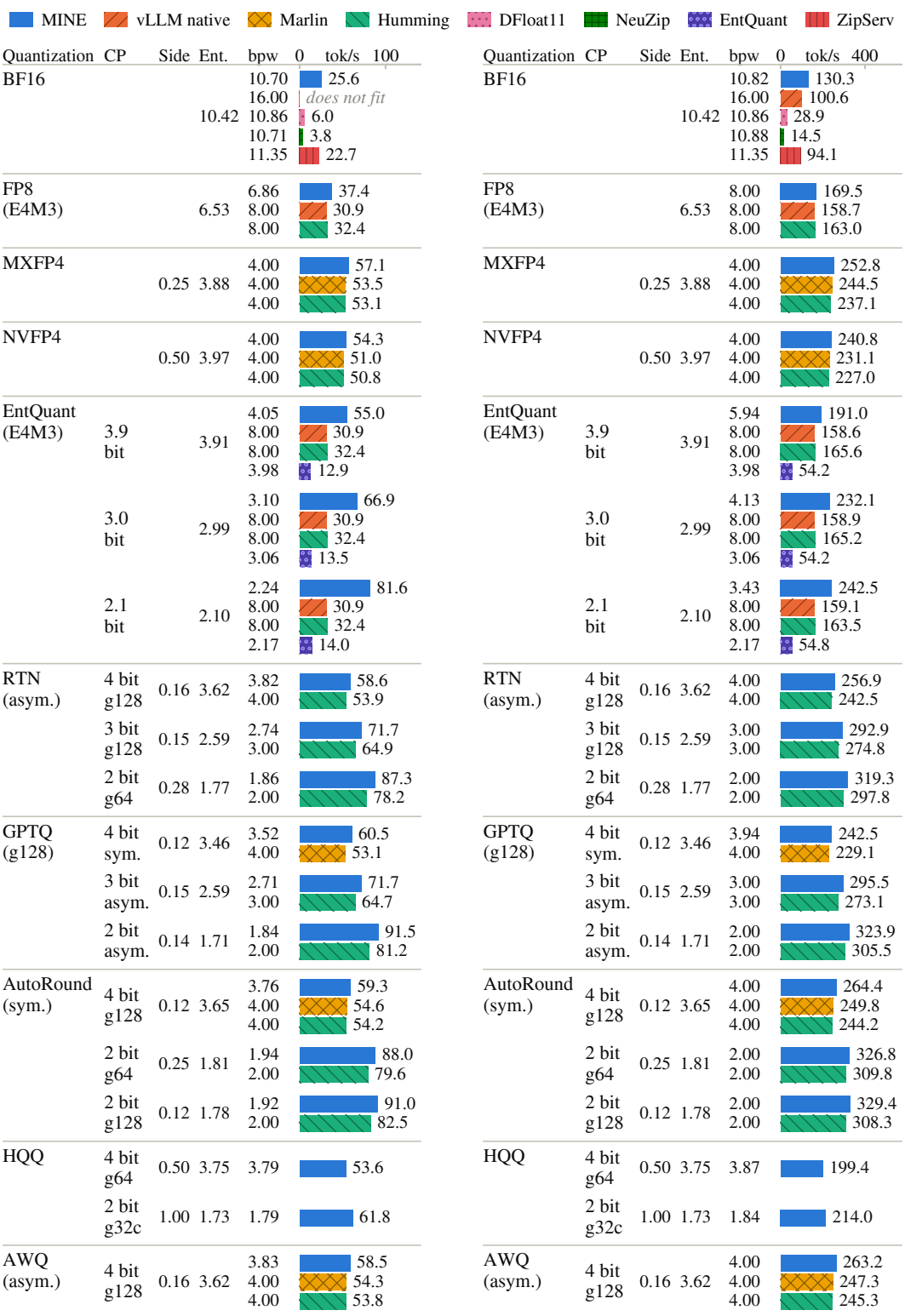
Second, MINE prefers smaller codings on the RTX 4060 Ti compared to the RTX 5090. This is because the RTX 5090 is less memory-bound, favoring parameters that lead to less work over those that lead to less memory transfer. Further, more launches fall back to vLLM because the RTX 5090 has more SMs that reduce the number of waves and make latency more critical (dtANS-LLM has to transfer an entire chunk first). For many checkpoints, MINE never chooses its own kernels on the RTX 5090 and its speedups result exclusively from combining native/Marlin/Humming kernels and compressing the `lm_head`. Note that MINE could still compress in these cases to free up memory but would suffer from a throughput penalty.

Prefill. As `mvm`, which is crucial for single-stream decoding, is the main optimization goal and only implemented for batch size 1, larger batch sizes (for prefill) have to be handled by `dequant`/`decode` and a GEMM. For instance, the latency of materialization is only 1.3–2.9× that of `mvm` and the subsequent GEMM at batch size 2048 is another 14.7–20.2× (on the launches of Llama 3.1 8B Instruct in `BF16`). This overhead of 8.1–15.4% is partially compensated by the materialized weights being L2 resident and thus prefill reaches 12,860 tok/s compared to vLLM's 13,650 tok/s (94%, 4096-token prompt on the RTX 5090; the RTX 4060 Ti has no uncompressed reference).

7 Conclusion

We showed with MINE that fast entropy decoding can be used to accelerate inference even in the challenging single-stream decoding setting. At the same time, MINE can adapt to tighter memory budgets, trading performance for better compression and thus fit larger models into less RAM. Thus, it would be promising to extend MINE beyond single-stream decoding where optimization objectives become more heterogeneous: fused kernels for small batch sizes to avoid the full cost of materialization, compressing the KV cache, and switching codings dynamically as the optimal coding might depend on the workload (e.g., between prefill and decode).

Lastly, our work motivates a new metric for quantization methods. As MINE can store weights close to their entropy and provide inference at equivalent speeds as well, the entropy of the quantized weights gets more important than their nominal rate and finding quantization schemes with low entropy becomes more important.



(a) RTX 4060 Ti (16 GB) (b) RTX 5090 (32 GB)

Figure 3: Evaluation of MINE (blue) and other inference engines on checkpoints of Llama 3.1 8B Instruct and two GPUs (left and right half). Columns state quantization method, checkpoint (CP), size of dequantization side streams, entropy bits per weight, achieved bits per weight, and single-stream (batch 1) steady-state decode throughput.

8 AI USE

We used generative AI to *implement methods* (coding) and *design or provide feedback on research methodology or experiments* (evaluation). We also used it to *create or modify scientific figures or images* (all figures and plots), *suggest experimental parameters* (identify hyper-parameters to tune), *create or edit software code*, *creation of artifacts* (reproducibility artifact, statement, and appendix), *draft parts of a research paper* (abstract and reproducibility statement and appendix), *identify relevant literature*, *summarize or analyse existing literature*, *sourcing/searching for information*, *edit a research paper to improve readability* (also finding disagreements between paper and code), *format references* (we checked all details manually), and *propose a title or keywords for a research paper*.

9 REPRODUCIBILITY STATEMENT

The supplementary material is an anonymous reproduction package, the single archive `mine-artifact.zip`. It contains the MINE source code (codec, CUDA kernels, auto-tuner, and vLLM plugin) and one numbered script per experiment, and it downloads everything else: third-party code at pinned commits, Python packages from exact lock files, and the model and datasets at fixed Hugging Face revisions. Appendix A maps Figure 3 and every number of Section 6 to the script that produces it, describes how to serve MINE under a memory budget, and lists the software versions, the recipes of all quantized checkpoints, the measurement protocols, the hardware requirements, and all changes we made to third-party code. The dtANS-LLM format and the tuning procedure are specified in Section 4, the hardware and software environment in the Experimental Setup paragraph of Section 6. All scripts detect the GPU at runtime, so the complete pipeline runs on any NVIDIA GPU with compute capability 8.9 or higher and at least 16 GB of memory (24 GB to also run the baselines that need the uncompressed `BF16` model on the GPU); since MINE tunes for the GPU it runs on, absolute throughput is only expected to match ours on the same GPU models.

REFERENCES

Hicham Badri and Appu Shaji. Half-quadratic quantization of large machine learning models. Blog post, https://dropbox.github.io/hqq_blog/, November 2023.

Feng Cheng, Cong Guo, Chiyue Wei, Junyao Zhang, Changchun Zhou, Edward Hanson, Jiaqi Zhang, Xiaoxiao Liu, Hai Li, and Yiran Chen. Ecco: Improving memory bandwidth and capacity for LLMs via entropy-aware cache compression. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, pp. 793–807, New York, NY, USA, 2025a. Association for Computing Machinery. ISBN 9798400712616. doi: 10.1145/3695053.3731024. URL https://doi.org/10.1145/3695053.3731024.

Wenhua Cheng, Weiwei Zhang, Haihao Shen, Yiyang Cai, Xin He, Lv Kaokao, and Yi Liu. Optimize weight rounding via signed gradient descent for the quantization of LLMs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 11332–11350, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.662. URL https://aclanthology.org/2024.findings-emnlp.662/.

Wenhua Cheng, Weiwei Zhang, Heng Guo, Haihao Shen, and Zaner Ma. SignRoundV2: Toward closing the performance gap in extremely low-bit post-training quantization for LLMs, 2025b. URL https://arxiv.org/abs/2512.04746.

Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Jarek Duda. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540, 2013. URL http://arxiv.org/abs/1311.2540.

Ruibo Fan, Xiangrui Yu, Xinglin Pan, Zeyu Li, Weile Luo, Qiang Wang, Wei Wang, and Xiaowen Chu. ZipServ: Fast and memory-efficient LLM inference with hardware-aware lossless compression. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 2264–2280, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790250. URL https://doi.org/10.1145/3779212.3790250.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The eleventh international conference on learning representations*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, pp. 239–251, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714436. doi: 10.1145/3710848.3710871. URL https://doi.org/10.1145/3710848.3710871.

ggml-org. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. AI and memory wall. *IEEE Micro*, 44(3):33–39, 2024. doi: 10.1109/MM.2024.3373763.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary,

Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa,

Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Greg. Reply to "cannot achieve max shared memory bandwith". NVIDIA Developer Forums, https://forums.developer.nvidia.com/t/cannot-achieve-max-shared-memory-bandwith/273109/10, November 2023. NVIDIA staff statement on shared-memory throughput of TU10x, GA10x and AD10x.

Yongchang Hao, Yanshuai Cao, and Lili Mou. NeuZip: Memory-efficient training and inference with dynamic compression of neural networks, 2024. URL https://arxiv.org/abs/2410.20650.

Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Or Ozeri, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, and Danny Harnik. ZipNN: Lossless compression for AI models. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 186–198, 2025. doi: 10.1109/CLOUD67622.2025.00028.

inclusionAI. Humming: a JIT-compiled GEMM kernel library for quantized inference. https://github.com/vllm-project/humming, 2026. humming-kernels 0.1.10, as shipped with vLLM 0.27.1.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference . In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2704–2713, Los Alamitos, CA, USA, June 2018. IEEE Computer Society. doi: 10.1109/CVPR.2018.00286. URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00286.

Jeff Johnson. DietGPU: GPU-based lossless compression for numerical data, 2022. URL https://github.com/facebookresearch/dietgpu. Meta Platforms. MIT licence. Repository archived (read-only).

Iwan Kawrakow. 2-bit quantizations. ggml-org/llama.cpp pull request #4897, https://github.com/ggml-org/llama.cpp/pull/4897, January 2024.

Hans Kellerer, Ulrich Pferschy, and David Pisinger. *The Multiple-Choice Knapsack Problem*, pp. 317–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24777-7. doi: 10.1007/978-3-540-24777-7_11. URL https://doi.org/10.1007/978-3-540-24777-7_11.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Sohan Lal, Jan Lucas, and Ben Juurlink. E2MC: Entropy encoding based memory compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1119–1128, 2017. doi: 10.1109/IPDPS.2017.101.

Sylvie Liberman, Tianyi Zhang, and Daniel Y Fu. NyoomFloat12: Lossless 12-bit weight compression for post-training inference. In *The 1st Workshop on Scaling Post-training for LLMs*, 2026. URL https://openreview.net/forum?id=yYFj1E1mSZ.

Fangzheng Lin, Kasidis Arunruangsirilert, Heming Sun, and Jiro Katto. Recoil: Parallel rANS decoding with decoder-adaptive scalability. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP 2023, pp. 31–40. ACM, August 2023. doi: 10.1145/3605573.3605588. URL http://dx.doi.org/10.1145/3605573.3605588.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In P. Gibbons, G. Pekhimenko, and C. De Sa (eds.), *Proceedings of Machine Learning and Systems*, volume 6, pp. 87–100, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf.

Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, Lin Xiao, Yuandong Tian, Bilge Soran, Raghuraman Krishnamoorthi, Tijmen Blankevoort, and Vikas Chandra. ParetoQ: Improving scaling laws in extremely low-bit LLM quantization. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 91311–91336. Curran Associates, Inc., 2025. doi: 10.52202/085713-3055. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/83b17fb3369b1effa97ca5409526b02e-Paper-Conference.pdf.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit LLMs: All large language models are in 1.58 bits, 2024. URL https://arxiv.org/abs/2402.17764.

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

NVIDIA, Felix Abecassis, Anjulie Agrusa, Dong Ahn, Jonah Alben, Stefania Alborghetti, Michael Andersch, Sivakumar Arayandi, Alexis Bjorlin, Aaron Blakeman, Evan Briones, Ian Buck, Bryan Catanzaro, Muya Chang, Jinhang Choi, Mike Chrzanowski, Eric Chung, Victor Cui, Steve Dai, Bita Darvish Rouhani, Carlo del Mundo, Deena Donia, Burc Eryilmaz, Henry Estela, Abhinav Goel, Oleg Goncharov, Yugi Guvvala, Robert Hesse, Russell Hewett, Herbert Hum, Ujval Kapasi, Brucek Khailany, Mikail Khona, Nick Knight, Alex Kondratenko, Ronny Krashinsky, Ben Lanir, Simon Layton, Michael Lightstone, Daniel Lo, Paulius Micikevicius, Asit Mishra, Tim Moon, Deepak Narayanan, Chao Ni, Abhijit Paithankar, Satish Pasumarthi, Ankit Patel, Mostofa Patwary, Ashwin Poojary, Gargi Prasad, Sweta Priyadarshi, Yigong Qin, Xiaowei Ren, Oleg Rybakov, Charbel Sakr, Sanjeev Satheesh, Stas Sergienko, Pasha Shamis, Kirthi Shankar, Nishant Sharma, Mohammad Shoeybi, Michael Siu, Misha Smelyanskiy, Darko Stosic, Dusan Stosic, Bor-Yiing Su, Frank Sun, Nima Tajbakhsh, Shelby Thomas, Przemek Tredak, Evgeny Tsykunov,

Gandhi Vaithilingam, Aditya Vavre, Rangharajan Venkatesan, Roger Waleffe, Qiyu Wan, Hexin Wang, Mengdi Wang, Lizzie Wei, Hao Wu, Evan Wu, Keith Wyss, Ning Xu, Jinze Xue, Charlene Yang, Yujia Zhai, Ruoxi Zhang, Jingyang Zhu, and Zhongbo Zhu. Pretraining large language models with NVFP4, 2025. URL https://arxiv.org/abs/2509.25149.

NVIDIA Corporation. nvCOMP: High-speed data compression using NVIDIA GPUs. https://developer.nvidia.com/nvcomp, 2020. Version 5.3.0 (EntQuant) and 4.1.1 (NeuZip).

NVIDIA Corporation. CUDA programming guide. https://docs.nvidia.com/cuda/archive/13.1.1/cuda-programming-guide/, December 2025. Version 13.1.

NVIDIA Corporation. CUTLASS, 2026a. URL https://github.com/NVIDIA/cutlass. Version 4.4.2 as built into vLLM 0.27.1.

NVIDIA Corporation. cuBLAS: Basic linear algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas, 2026b. Version 13.1.1.

NVIDIA Corporation. nvCOMPDx: nvCOMP device extensions, 2026c. URL https://docs.nvidia.com/cuda/nvcompdx/. Part of the MathDx package. Device-side (in-kernel) LZ4 and ANS decompression, shipped as LTO-IR. Accessed 2026-09-25.

Open Compute Project. OCP microscaling formats (MX) specification, version 1.0. Technical report, Open Compute Project Foundation, 2023. URL https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf.

Patrick Putzky, Martin Genzel, Mattes Mollenhauer, Sebastian Schulze, Thomas Wollmann, and Stefan Dietzel. Float8@2bits: Entropy coding enables data-free model compression. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2601.22787.

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

Emil Schätzle, Tommaso Pegolotti, and Markus Püschel. Fast entropy decoding for sparse MVM on GPUs. In *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 416–432, 2026. doi: 10.1109/IPDPS65963.2026.00044.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27 (3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

Hongshi Tan, Yao Chen, Gustavo Alonso, Weng-Fai Wong, and Bingsheng He. Approaching Shannon bound with lossless LLM weight compression. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 116–130, 2026. doi: 10.1109/ISCA66397.2026.00024.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: quantization with trellises and incoherence processing. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

turboderp. ExLlamaV3: An optimized quantization and inference library for running LLMs locally on modern consumer-class GPUs. https://github.com/turboderp-org/exllamav3, 2025.

André Weißenberger and Bertil Schmidt. Massively parallel Huffman decoding on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365109. doi: 10.1145/3225058.3225076. URL https://doi.org/10.1145/3225058.3225076.

André Weißenberger and Bertil Schmidt. Massively parallel ANS decoding on GPUs. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. doi: 10.1145/3337821.3337888. URL https://doi.org/10.1145/3337821.3337888.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Naoya Yamamoto, Koji Nakano, Yasuaki Ito, Daisuke Takafuji, Akihiko Kasagi, and Tsuguchika Tabaru. Huffman coding with gap arrays for GPU acceleration. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388160. doi: 10.1145/3404397.3404429. URL https://doi.org/10.1145/3404397.3404429.

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

Tianyi Zhang, Mohsen Hariri, Shaochen (Henry) Zhong, Vipin Chaudhary, Yang Sui, Xia Hu, and Anshumali Shrivastava. 70% size, 100% accuracy: Lossless LLM compression for efficient GPU inference via dynamic-length float (DFloat11). In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 98966–98994. Curran Associates, Inc., 2025. doi: 10.52202/085713-3306. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/8e8b4449c80879215ff2ee47b6dd01a6-Paper-Conference.pdf.

A REPRODUCING THE EXPERIMENTS

The supplementary material is a single archive, `mine-artifact.zip`, that reproduces all experiments of Section 6 from scratch. It holds only our code: the MINE package (codec, CUDA kernels, auto-tuner, vLLM plugin) and the scripts that drive the experiments. The setup scripts download all other code at pinned commits and apply our changes (`setup/fetch.sh`), build the Python environments from exact lock files (`setup/envs.sh`), and download the model, the DFloat11 checkpoint, and the evaluation data at fixed revisions (`setup/download.sh`). The numbered steps of Table 1 are then run in order; every step is resumable, and the package's `README.md` gives the commands, the requirements, and a quick check on a single checkpoint.

Table 1: Results of the paper and the steps of `mine-artifact.zip` that produce them. Steps `10`–`15` run once per GPU and store their results under a label derived from the GPU's name; step `20` needs no GPU.

Result	Step
The 18 quantized checkpoints	`run/01_checkpoints.sh`
Entropy $H(P)$; `BF16` perplexity (reference of ZipServ's gate)	`run/02_stats.sh`
Tune files and tune times	`run/10_tune.sh`
Decode throughput of MINE, vLLM (native / Marlin), and Humming	`run/11_serve.sh`
DFloat11, NeuZip, EntQuant (nvCOMP), ZipServ, and its vLLM 0.9.1	`run/12_baselines.sh`
Prefill paragraph	`run/13_prefill.sh`
GPU clocks under load (Experimental Setup)	`run/15_clocks.sh`
Figure 3 (side streams, $H(P)$, bpw, tok/s) and every number quoted in Section 6	`run/20_figures.sh`
Serving under a memory budget (Figure 1; optional, no reported result)	`run/14_budget.sh`

Serving under a memory budget. The evaluation serves every checkpoint with its fastest tuned selection. `run/14_budget.sh` serves MINE instead with the fastest selection whose weights fit a given budget, as in the right half of Figure 1: from the per-launch codings in the tune file, `mine/tools/pick_budget.py` chooses one per launch so that the summed batch-1 time is minimal while the weights (coded linears, embeddings, norms, and all other resident weights) fit the budget, a multiple-choice knapsack that it solves by dynamic programming and certifies against a lower bound. The selection is passed to the plugin in `MINE_TUNE_SELECT` and served with the decode protocol below. The budget is given in bytes or as a fraction of the fastest selection's footprint, and may lie anywhere between the smallest and the fastest selection; the package's `README.md` gives the commands.

Software. MINE runs as a plugin of vLLM 0.27.1 (commit `6e448d0`, built from source) with PyTorch 2.13.0 (CUDA 13.0 build); its kernels are JIT-compiled for the running GPU by nvcc 13.1 with `-O3 --use_fast_math`. The vLLM baselines run in a separate environment with the official vLLM 0.27.1 wheel and without the plugin; Humming is the `humming-kernels` 0.1.10 package vLLM 0.27.1 requires, enabled with `--quantization humming`. The checkpoints were produced with GPTQModel 7.3.4 (`fe8aecb`), auto-round (`f17d9cd`), llm-compressor (`fc2cf94`) with compressed-tensors 0.18.1a20260806, HQQ (`d88a488`), and EntQuant (`a957fec`). The baselines are DFloat11 0.5.0 with its published Llama 3.1 8B checkpoint (revision `8fcadb8`), NeuZip (`1224c60`) with nvCOMP 4.1.1.1, EntQuant's decoder with nvCOMP 5.3.0.16, and ZipServ (`f69cea9`) on its vLLM 0.9.1 fork (vLLM `7b1895e`) with the GEMM kernel of the ZipServ artifact (`6f8a209`). The model is Llama-3.1-8B-Instruct at revision `0e9e39f`; WikiText-2, C4, and pile-10k are pinned to revisions `b08601e`, `1588ec4`, and `127bfed`.

Checkpoints. All checkpoints quantize the 224 decoder linears and keep embeddings and `lm_head` in `BF16`. `E4M3`: llm-compressor's `FP8_DYNAMIC` scheme (weights per channel, activations dynamically per token); `MXFP4`/`NVFP4`: its weight-only `MXFP4A16`/`NVFP4A16` schemes. EntQuant: its entropy-regularized `E4M3` quantizer, data-free, with $(\lambda, \mathrm{lr}) = (58, 0.25)$, $(14.5, 1.0)$, and $(3.9, 1.0)$ for 2.1, 3.0, and 3.9 bits. RTN: auto-round with zero iterations (which, in this version,

applies its importance-weighted RTN on the calibration set), asymmetric, GPTQ format. GPTQ: GPTQModel with damping 0.05 and sequential quantization; activation reordering (`desc_act`) for the symmetric 4-bit checkpoint, GPTQModel's default group-aware reordering for the others. AutoRound: 200 iterations with `--enable_alg_ext`, symmetric. AWQ: llm-compressor's `W4A16_ASYM` AWQ recipe. HQQ: 4 bits with groups of 64 along rows and 2 bits with groups of 32 along columns, 16-bit scales and zeros, data-free. GPTQ and AWQ calibrate on 128 random windows of 2048 tokens from one C4 training shard; RTN and AutoRound on auto-round's default of 128 samples of 2048 tokens from pile-10k. The calibrated methods are not bit-reproducible across GPUs and driver versions; the others are deterministic. For serving, the scripts convert without changing any value: HQQ's pickle to safetensors (MINE), EntQuant to a compressed-tensors `FP8` checkpoint (vLLM, Humming), and the zero points of the asymmetric RTN and GPTQ checkpoints from the GPTQ v1 to the v2 convention (Humming). Humming is shown for the checkpoints on which it produced correct output in our runs; the scripts serve exactly these.

Measurement protocols. *Decode* (MINE, vLLM, Humming, ZipServ, and its vLLM 0.9.1): a random 512-token prompt followed by exactly G greedy tokens (EOS ignored) for $G = 64$ and $G = 128$, CUDA graphs enabled, prefix caching disabled, median of 3 runs after one warm-up; the steady-state rate $1/t_{\text{dec}}$ is solved from $t(G) = t_{\text{pre}} + G\,t_{\text{dec}}$. DFloat11, NeuZip, and EntQuant (nvCOMP) run under their own HF Transformers stacks with the same prompt and estimator, with three interleaved repetitions of both G after two warm-ups. Bits per weight are measured over the decoder linears for every engine (for DFloat11 from its published checkpoint; for NeuZip on each GPU, as nvCOMP's compressed size differs between GPUs). *ZipServ's gate*: ZipServ is timed only after its WikiText-2 perplexity (non-overlapping windows of 2048 tokens, all positions scored, prompt log-probabilities of its vLLM) was within $\pm 0.5\%$ of the `BF16` model under HF Transformers, and its greedy batch-1 tokens agreed with the same vLLM's dense `BF16` path (at least 90% of the tokens of 8 prompts and none of the first 8, or else every divergence a near-tie of dense `BF16`: ZipServ's token in its top 2 with a margin of at most 0.25 nats). *Tuning*: every launch is timed cold-cache (copies covering $2\times$ the L2 cache, CUDA-graph replays, median of 5 bursts of at least 400 μs); the `BF16` `lm_head` is identical in all checkpoints and is tuned once per GPU, outside the reported tune times. *Clocks*: the median SM clock over all 100 ms samples with at least 90% utilization during a MINE decode run.

Hardware requirements and portability. Nothing in the package is specific to our GPUs: the GPU is chosen by its index, and the label, architecture, JIT target, occupancy limits, and L2 size are read at runtime. MINE requires compute capability 8.9 or higher (its `FP8` conversion instructions and vLLM's `FP8` kernels) and at least 16 GB of memory: the served MINE models occupy 3.5–10.5 GB, tuning processes one launch at a time, and quantization needs up to about 20 GB (24 GB recommended). Plain vLLM on the `BF16` model, ZipServ's vLLM 0.9.1 on the `BF16` model, and the vLLM arm of the prefill experiment need the uncompressed weights on the GPU (24 GB or more); on smaller GPUs they are not run, as for the RTX 4060 Ti in Figure 3. ZipServ fits in 16 GB with the near-capacity settings of the MINE serves, but its batch-1 check then uses the dense `BF16` tokens of a run on a larger GPU as reference. For the decode runs, the vLLM 0.27.1 KV cache is fixed at 2 GiB on GPUs with less than 24 GB; otherwise, vLLM sizes it at 85% utilization. The prefill runs fix it at 1 GiB on every GPU. The setup builds vLLM for MINE from source for the GPUs present, as for the paper; the baselines use the official vLLM 0.27.1 wheel, which covers sm_75 to sm_120. The tuner's tile menus were chosen on our two GPUs; on others it still searches all legal configurations. Because tune files contain measured timings, a tune file must only be served on the GPU model it was tuned on. The complete pipeline needs about 300 GB of disk, 64 GB of host memory, CUDA toolkits 13.x and 12.8/12.9 (the latter only to build NeuZip and ZipServ), and takes 3–4 hours of setup, about 7 hours for the checkpoints and statistics, and 1.5–2 days per GPU.

Modifications of third-party code. We changed third-party code only where it did not build or run in our setting: one `constexpr` removed in NeuZip (nvcc 12 otherwise compiles its Python bindings for the device); ZipServ's compressor built for the local architecture instead of the hard-coded sm_89; in ZipServ's vLLM integration, its GEMM kernel sources replaced by those of the ZipServ artifact, because the integrated copy skips the computation for fewer than 8 tokens and thus at batch size 1, and in that kernel two function-local `static` shared-memory sizes computed per launch instead of once (the first layer launched fixed them; with the `lm_head` compressed,

later and larger layers overran shared memory), plus a build-only fix to locate NVTX3; auto-round's calibration set pinned to a dataset revision by a wrapper. Three runtime wrappers only change where memory is used, never a result: EntQuant's reported entropy statistic is computed with a histogram on the host (during quantization and when its nvCOMP baseline loads), its compressed buffers are moved to the host during quantization, and GPTQModel's CPU fallback after an out-of-memory error in its channel permutation, which would permute the weights twice, is refused.