TACT: TIERED ACTIVATION SPARSITY

Anonymous authors
Paper under double-blind review

ABSTRACT

Deploying LLMs on consumer-grade GPUs with insufficient memory to hold their weights can result in prohibitively slow inference, because decoding repeatedly transfers offloaded weights from system RAM or flash storage into GPU at much lower bandwidth than local GPU-memory access. Activation sparsity reduces these transfers by skipping weights associated with zero or near-zero activations. However, as more activation contributions are omitted, model quality eventually degrades rapidly, indicating that weights associated with small-magnitude activations collectively influence model quality sharply. In this work, we improve the trade-off between model quality and decoding performance when exploiting activation sparsity. Our key idea is to replace the binary choice of whether or not to read a weight with three options: fully retain it, approximate it using a compressed weight representation, or omit it entirely. TACT skips weights associated with activations closest to zero, reads approximate weights for smaller-magnitude activations, and reads original weights for the largest-magnitude activations. Smaller-magnitude activations attenuate the errors introduced by approximate weights, while compressed weight representations require fewer bytes to be transferred. With weights offloaded to CPU memory, TACT speeds up decoding by $3.86\times$ on average (up to $5.57\times$) with 16-bit weights and $2.06\times$ (up to $2.74\times$) with 4-bit weights, at a WikiText-2 perplexity increase of at most 10%. With weights offloaded to flash storage, the speedups are $3.31\times$ on average (up to $4.81\times$) and $1.54\times$ (up to $2.03\times$).

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong capabilities in a wide range of tasks, from text understanding to agentic workflows such as code generation, computer use, and mathematical reasoning. Running capable LLMs locally on personal devices such as desktops or workstations gives users greater control over their data and computing. This has vital implications for task ownership and data privacy, and reduce dependence on cloud subscriptions and their dynamic pricing.

Unlike server-grade GPUs, desktop and workstation GPUs typically have limited GPU memory. We use *GPU memory* for VRAM, *CPU memory* for system DRAM, and *flash storage* for SSD-based storage (an NVMe SSD in our experiments). *Offloaded weights* reside in CPU memory or flash storage and are transferred into GPU memory as needed during decoding. These transfers are much slower than reading weights already resident in GPU memory. For example, on our workstation (Section 6), the NVIDIA RTX 4090's local memory bandwidth is about $80\times$ the bandwidth for transferring weights from CPU memory and about $150\times$ that for transferring weights from flash storage. Thus, transferring offloaded weights can dominate decoding latency (Alizadeh et al., 2024; Xue et al., 2024). Reducing this latency requires reducing the weight bytes transferred to generate each token. This requires targeting the transformer's attention projection and feed-forward network (FFN) weights. Within each projection, a *weight column* contains the weights multiplied by a single input activation entry (see Section 2). Activation sparsity reduces these transfers by skipping weight columns associated with zero activations. The columns that can be skipped differ across tokens.

Early methods such as Deja Vu (Liu et al., 2023) and PowerInfer (Song et al., 2024a) demonstrate that activation-dependent sparsity can accelerate LLM inference. These methods exploit sparse activations in the FFN of models that use the ReLU activation function (Nair & Hinton, 2010). However, this source of sparsity is largely absent in modern LLMs that use smooth activation functions such as SiLU and GELU (Shazeer, 2020). ReLUfication (Mirzadeh et al., 2024; Song et al., 2025; 2024b)

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.

addresses this limitation by replacing the smooth activation function with ReLU and performing continued pretraining to recover model quality. PowerInfer-2 (Xue et al., 2024) decodes such sparsified models on a smartphone with FFN weights offloaded to flash storage. Continued pretraining, however, requires access to representative training data and compute resources and produces a modified instance of the model. Training-free approaches instead induce sparsity by omitting reads of weight columns corresponding to low-magnitude activations. CATS (Lee et al., 2024) applies this approach to FFN weights in models using SiLU activation functions. CHESS (He et al., 2024) and TEAL (Liu et al., 2025a) apply magnitude-based sparsification to attention projection weights as well. These methods offer a sparsity level hyperparameter that is adjustable during deployment, allowing a trade-off between model quality and weights read per token. Increasing sparsity removes contributions of progressively higher-magnitude activations. However, beyond a certain sparsity level, even a small increase in leveraged sparsity causes model quality to drop sharply (see Section 4). This rapid degradation indicates that model quality is highly sensitive to omitting these intermediate-magnitude activations. This limits the sparsity level that can be used in practice (Szatkowski et al., 2026).

Our goal in this work is to accelerate autoregressive decoding in modern LLMs on consumer-grade GPUs where weights exceed GPU memory, without requiring continued pretraining. To this end, we propose TACT, a tiered activation-sparsity mechanism to reduce the *weight data* required per token, i.e., the number of weight bytes (compressed or otherwise) transferred to GPU memory, while limiting model-quality loss. The key idea of TACT is to replace the binary choice of whether or not to read a weight column with three options: read the original weights of the weight column, read an approximation of the weight column using a compressed representation, or omit it entirely. This design builds on the observation that an activation's magnitude scales the contribution of its weight column's approximation error to the projection's output error. Because smaller-magnitude activations attenuate these errors, approximating their weight columns with compressed representations loses little accuracy while transferring far fewer bytes. This motivates omitting weights associated with activations closest to zero, approximating those of smaller-magnitude activations, and retaining the original weights for the largest-magnitude activations.

Realizing this idea requires addressing three challenges: (1) providing compressed weight approximations that preserve useful contributions and allow each selected column to be read independently; (2) choosing which columns to retain, approximate, or omit at each token under a given read budget; and (3) translating byte savings into faster decoding despite the fragmented storage reads caused by token-dependent column selection. To address (1), TACT defines three weight approximation tiers: a top tier read in the deployment's weight format, a middle tier read from compressed representations of each weight that the GPU decompresses during decoding, and a bottom tier left unread. The middle tier encodes each weight column independently (Section 5.1). To address (2), TACT retains the magnitude ordering used by top-k selection and introduces a second boundary to divide activation entries among the three tiers. Offline calibration allocates the read budget among matrices and between the top and middle tiers (Section 5.2). To address (3), TACT groups matrices that share input activations and places their jointly accessed weight columns contiguously in storage. It merges adjacent selected ranges into larger read requests to reduce fragmentation (Section 5.3).

We evaluate TACT on three models in two weight formats against four training-free activation-sparsity methods. At a WikiText-2 perplexity increase of at most 10%, with weights offloaded to CPU memory, TACT decodes $3.86\times$ faster than dense on average (up to $5.57\times$) in BF16 and $2.06\times$ (up to $2.74\times$) with 4-bit weights, where TEAL reaches $1.94\times$ and $1.85\times$ on average. With flash offloading, the speedups are $3.31\times$ (up to $4.81\times$) and $1.54\times$ (up to $2.03\times$), where TEAL reaches $1.82\times$ and $1.48\times$. Our contributions are as follows:

- We propose TACT, a training-free method that leverages activation-sparsity to selectively read each offloaded weight in full precision, as compressed approximations, or not read at all, according to its activation's magnitude.
- At the same model quality, TACT requires significantly less weight data per token than prior work in all evaluated settings.
- TACT supports offloading to CPU memory or flash storage, with 16- or 4-bit weights.

2 Background

Weights of a Transformer Layer A gated feed-forward network (FFN; SwiGLU or GeGLU (Shazeer, 2020)) computes an intermediate activation $a = \sigma(W_{\text{gate}}\, x) \odot W_{\text{up}}\, x$ from its

input x, where σ is SiLU or GELU, and projects it back to the model dimension as $\mathrm{FFN}(x) = W_{\mathrm{down}}\, a$. An attention block (Vaswani et al., 2017) projects its input h into queries, keys, and values with W_q, W_k, and W_v, and projects the concatenated head outputs z back to the model dimension with W_o. TACT is applied to each of these seven matrices. Their inputs are h for W_q, W_k, and W_v, z for W_o, x for W_{gate} and W_{up}, and a for W_{down}.

Activation sparsity: omitting columns of weight matrices For any of these FFN and attention projection weights, let $W \in \mathbb{R}^{d\times n}$ be its weight matrix and $x \in \mathbb{R}^n$ its input activation. The product $Wx = \sum_{i=1}^{n} x_i W_{:,i}$ scales each *weight column* $W_{:,i}$ by its activation entry x_i and sums the results. Activation sparsity computes this product from a subset of the entries, so only their weight columns need to be transferred. In a ReLU FFN, exact-zero entries can be skipped without changing the output. Gated FFNs with smooth activation functions produce almost no exact zeros, so activation sparsity instead omits entries with small magnitudes (Lee et al., 2024; Liu et al., 2025a). A binary mask $m \in \{0,1\}^n$ selects the entries to retain:

$$W(m \odot x) = \sum_{i:m_i=1} x_i W_{:,i}. \tag{1}$$

Each omitted entry x_i adds an error e_i to the output:

$$e_i = x_i W_{:,i}, \qquad \|e_i\|_2 = |x_i|\,\|W_{:,i}\|_2, \qquad Wx - W(m \odot x) = \sum_{i:m_i=0} e_i. \tag{2}$$

The error terms scale with the activation entry's magnitude, while the total error also depends on the directions of the omitted weight columns. With at most k retained entries, the best mask is:

$$m^\star \in \operatorname*{arg\,min}_{m\in\{0,1\}^n,\ \|m\|_0\le k} \|Wx - W(m \odot x)\|_2^2. \tag{3}$$

A simple selection rule that works well in practice (though not optimal) is magnitude-based top-k, which retains the k entries with the largest $|x_i|$ and gives a sparsity level of $1 - k/n$. Top-k reduces the error due to discarded activations $\sum_{i:m_i=0} x_i^2$.

3 Related Work

Training-free activation sparsity. These methods reduce the weight data required during decoding without updating model parameters through fine-tuning or continued pretraining. CATS (Lee et al., 2024) omit loading weights of small-magnitude activations following the FFN's gate projection activation function. TEAL (Liu et al., 2025a) extends this technique to every weight matrix in the model. WINA (Chen et al., 2026) and LaRoSA (Liu et al., 2025b) refine the selection criterion: WINA scores an entry by its magnitude and the norm of the weight column it indexes, while LaRoSA ranks entries in a rotated activation vector. R-Sparse (Zhang et al., 2025) multiplies the largest-magnitude entries by the original weights and the remaining entries by a rank-r SVD approximation of the weight matrix. We compare TACT with TEAL, LaRoSA, WINA and R-Sparse in Section 6.

Training-based activation sparsity. Training-based approaches modify pretrained model weights to promote higher sparsity in input activations. ReLUfication (Mirzadeh et al., 2024; Song et al., 2024b) replaces smooth activation functions with ReLU variants and uses continued pretraining to recover model quality. Systems such as Deja Vu (Liu et al., 2023) and PowerInfer (Song et al., 2024a) exploit sparse activations in ReLU-based FFNs, using trained predictors to identify active neurons before computing the full activation. PowerInfer-2 uses sparsified checkpoints to enable inference on smartphones with limited memory (Xue et al., 2024).

Caching methods. Reducing the number of selected weights does not by itself ensure efficient storage access: decoding latency also depends on the locality of reads, cache reuse, and overlap between I/O and computation. LLM in a Flash (Alizadeh et al., 2024) stores FFN weights in flash storage and fetches only the columns associated with neurons that its predictor marks active. To reduce small random reads, LLM in a Flash and Neuralink place weights accessed together next to one another (Alizadeh et al., 2024; Wang et al., 2025b). Neuron Chunking incorporates storage cost into selection, choosing contiguous neuron chunks according to their importance and estimated I/O latency (Yang et al., 2025). DIP's cache-aware masking considers both activation magnitude and the weights already resident in memory to reduce flash reads (Federici et al., 2025). ActiveFlow overlaps storage I/O with computation by predicting and preloading active weights for subsequent

layers (Jia et al., 2025). TACT addresses the storage access cost of its tiered reads by placing jointly accessed columns contiguously and merging adjacent selected ranges into larger requests.

Offline weight compression. Weight quantization reduces data transfer by storing every weight at a lower precision. AWQ (Lin et al., 2024) uses activation magnitudes to identify salient weights and protects them through scaling. QTIP (Tseng et al., 2024b) codes weights as a path through a trellis instead of an index into a codebook, reaching a lower error at the same bitrate. Activation sparsity and weight quantization can be combined (Liu et al., 2025a; Zhang et al., 2025; Wang et al., 2025a). These approaches read every retained column from a single quantized representation. TACT instead stores both the deployment-format weights and a lower-bit copy, and decides at each token which columns to read from each. Concurrent work GRINQH (Oberländer et al., 2026) uses activation magnitudes at each token to select weight-column precision or skip columns. We omit direct comparison because we could not locate public code at submission time.

4 Model Quality on Omitting Weights

Figure 1 shows the speed–quality tradeoff of TEAL (Liu et al., 2025a) on Llama-3.1-8B with weights offloaded to CPU memory and flash storage. With CPU memory offloading, decoding takes 1.16 s per token, 97% of it on weight transfers. Omitting 40% of columns yields a $1.62\times$ speedup with under 7.8% perplexity increase; omitting 70% yields a $3.17\times$ speedup but sharply raises perplexity.

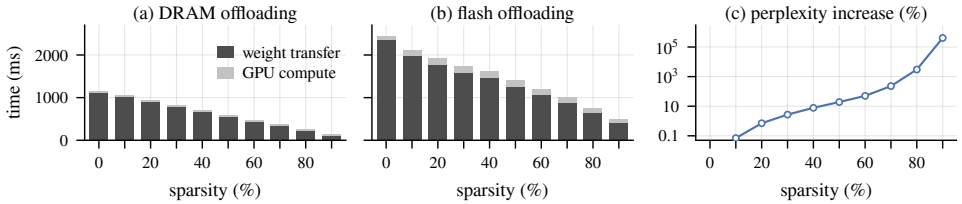


Figure 1: Llama-3.1-8B (BF16) with all weights offloaded, at different sparsity levels. **(a), (b):** Time per decoded token with the weights offloaded to CPU memory and to flash storage. **(c):** Perplexity increase relative to dense on the WikiText-2 test set. (See Section 6 for the experiment setup.)

As sparsity increases, top-k selection omits columns associated with larger activations. Each column's omission error scales with activation magnitude (Equation 2). AWQ likewise finds that protecting channels associated with large activations reduces quantization error (Lin et al., 2024). R-Sparse shows that even coarse approximations of omitted contributions can recover model quality (Zhang et al., 2025). This motivates reading compressed approximations of these columns to retain some of their contribution while transferring fewer bytes than the original weights.

5 TACT: Tiered Activation Sparsity

TACT reduces the weight data required per token by assigning weight columns to three tiers according to their activation magnitudes. It reads the *top tier* in the deployment's weight format, reads compressed approximations for the *middle tier*, and leaves the *bottom tier* unread. Figure 2 illustrates the offline preparation and per-token decoding steps. TACT encodes each weight column C_j into compressed codes C'_j (step 1; Section 5.1). It then calibrates the number of columns assigned to each tier for each weight matrix under a given read budget (step 2; Section 5.2). The original weights and their compressed codes are stored in CPU memory or flash storage. During decoding, at each token, TACT ranks the input activations by magnitude and assigns their weight columns to the three tiers (step 3). In the example, x_2 selects C_2 for the top tier, x_0 and x_3 select C_0 and C_3 for the middle tier, and the remaining columns belong to the bottom tier. TACT fetches C_2 and the compressed codes C'_0 and C'_3 into GPU memory, leaving C_1 and C_4 unread (step 4). The GPU reconstructs $\hat{C}_0$ and $\hat{C}_3$ from these codes and computes $x_2C_2 + x_0\hat{C}_0 + x_3\hat{C}_3$ (step 5).

5.1 Compressed Representation for the Middle Tier

For the middle tier, we consider two lossy weight compression schemes: scalar quantization and trellis-coded quantization (step 1 in Figure 2).

Scalar quantization. We represent each weight using a two-bit code and store two scaling factors, $s_{j,0}$ and $s_{j,1}$, shared by all weights in column C_j. The two bits of the code specify signs $e_1, e_2 \in$

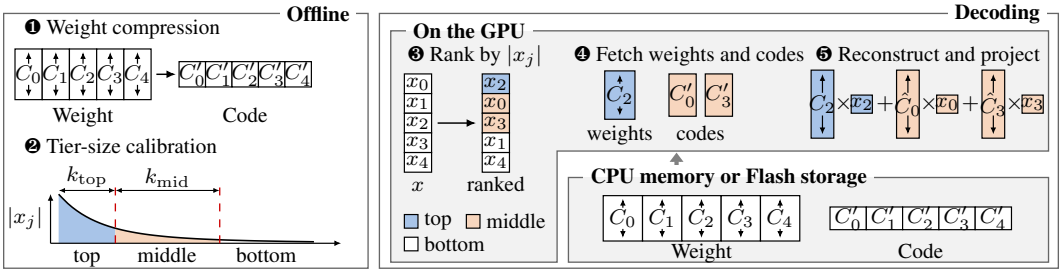


Figure 2: Overview of TACT. Offline encoding and calibration prepare compressed weight codes and determine tier sizes. During decoding, activation magnitudes determine which columns to read in the original format, reconstruct from compressed codes, or skip.

$\{-1, +1\}$, and the weight is reconstructed as $e_1 s_{j,0} + e_2 s_{j,1}$. In TACT, we encode all weight columns offline using symmetric two-bit Lloyd–Max quantization (Lloyd, 1982; Max, 1960). We first determine lower and upper representative weight magnitudes, l_j and h_j, for each column C_j. We set $s_{j,0} = (h_j + l_j)/2$ and $s_{j,1} = (h_j - l_j)/2$, so that adding or subtracting the two scales yields h_j or l_j, respectively. We then select two signs $e_1, e_2 \in \{-1, +1\}$ for each weight W_{ij} to minimize its squared reconstruction error with the scales fixed as follows:

$$(e_1, e_2) = \underset{(a,b)\in\{-1,+1\}^2}{\arg\min} \left(W_{ij} - a s_{j,0} - b s_{j,1}\right)^2, \qquad \hat{W}_{ij} = e_1 s_{j,0} + e_2 s_{j,1}. \tag{4}$$

During decoding, TACT reads the codes and two scales of each selected middle-tier column and reconstructs its weights on the GPU.

Trellis-coding compression. Prior work in quantization has shown that jointly encoding weights across multiple columns reduces quantization error compared to compressing each weight independently (Tseng et al., 2024b;a; Egiazarian et al., 2024). Because activation sparsity reads only a few selected columns, TACT instead builds on bitshift trellis coding (Mao & Gray, 2010) to encode the weights within each column jointly, so that each column can be reconstructed from its own codes.

TACT represents each weight column as a sequence of two-bit codes and a column-specific scale. All columns share a codebook of 2^{16} values drawn once from a standard normal distribution. Each weight is represented by a window of eight consecutive codes, its own and those of the seven preceding weights in the column, which form a 16-bit index into the codebook (Figure 3). The selected value times the column's scale gives the approximate weight. To encode a column, TACT divides its weights by their root-mean-square and stores this value as the scale. We choose the code sequence that minimizes the total squared error between the normalized weights and their reconstructions. Because the windows overlap, each code affects up to eight weights, so choosing each code separately need not minimize the column's error. TACT therefore chooses the codes jointly with the Viterbi algorithm (Forney, 1973). During decoding, consecutive weights share seven of their eight codes. Moving to the next row drops the oldest code and appends the next one (row 7 uses the codes of rows 0–7, row 8 those of rows 1–8); codes before the start of the column are treated as zeros. Each weight thus needs one codebook lookup and one multiplication.

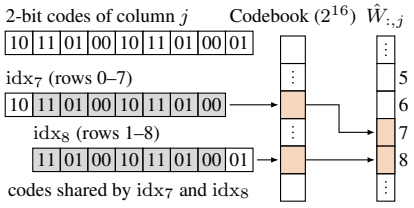


Figure 3: Reconstructing a middle-tier column from its own b-bit codes.

5.2 Calibrating the Number of Weight Columns in Each Tier

At each token, TACT ranks the input activation entries by magnitude and splits the corresponding weight columns into the three tiers (step 3 in Figure 2). The columns of the largest entries form the top tier (k_{top} in number), those of the next entries form the middle tier (k_{mid} in number), and the rest form the bottom tier. TACT calibrates the number k_{top} and k_{mid} offline within a given read budget (step 2 in Figure 2). Different choices of k_{top} and k_{mid} affect both the weight data required

and model quality. We express the per-matrix *read budget* as the weight data required to compute Wx, normalized by the size of the full weight matrix W:

$$\textbf{budget} = \frac{k_{\text{top}} + \rho\, k_{\text{mid}}}{n}, \tag{5}$$

where n is the number of columns of W and ρ is the compression ratio of the middle tier. Searching k_{top} and k_{mid} for every weight matrix creates a large search space, so TACT splits the search into two steps: For a given overall budget, Step 1 searches for read budget of each group of weight matrices that share an input activation, and Step 2 splits each group's budget between the top and middle tiers. Both steps minimize the output error of Equation 3 (Section 2), averaged over a calibration dataset.

Step 1. Greedy search for each group's budget. Weight matrices that share an input activation share its ranking, so TACT gives them one budget (Equation 5, normalized by their combined dense size) and one pair of tier sizes. Each layer thus has four groups: (W_q, W_k, W_v), (W_o), $(W_{\text{gate}}, W_{\text{up}})$, and (W_{down}). Starting with every group dense (budget 1) and $k_{\text{mid}} = 0$, TACT iteratively searches for the next group to reduce. In each iteration, it tries lowering each group's budget by 0.05 while holding the others fixed, and applies the reduction with the smallest ℓ_2 output error (Equation 3) on a calibration dataset. The search stops when the layer's read budget reaches its target; the intermediate steps give the group budgets for larger targets.

Step 2. Grid search for k_{top} and k_{mid}. With the group budgets from Step 1 fixed, each column removed from the top tier pays for $1/\rho$ middle-tier columns (Equation 5). From $k_{\text{mid}} = 0$, the grid lowers k_{top} in equal steps, raising k_{mid} to keep the budget fixed, until the bottom or the top tier is empty. TACT selects the setting with the smallest output error (Appendix B).

5.3 Efficiently Loading Offloaded Weights

Reads from flash storage proceed at a coarse granularity and serves large read requests faster than many small ones, but token-dependent column selection fragments each token's read into many small ranges. TACT therefore stores the columns of each group contiguously in the order of their activation entries and merges adjacent selected ranges into single read requests (Appendix E).

6 Results

Experiment setup We implement TACT using llama.cpp (Gerganov et al., 2023) and decode at batch one on a workstation with an NVIDIA RTX 4090 GPU. We evaluate two settings: (i) *CPU memory offloading*, with weights and TACT's codes held in CPU memory and transferred into GPU memory over PCIe, and (ii) *flash offloading*, with the weights and codes read from flash storage (an NVMe SSD). We define the *read budget* as the weight data required per decoded token, normalized by the size of the same weight matrices in the dense deployment format. This includes selected original weights, compressed codes, and any auxiliary data required by the compression format. Actual bytes transferred from flash storage can be larger because reads occur at coarser granularity.

We offload all weights targeted by each configuration to study the transfer bottleneck independently of GPU memory capacity. On our PCIe 4.0 x8 workstation, transfers account for 97% of dense LLAMA-ALL-BF16 decoding time with CPU memory offloading. Desktops with smaller-memory GPUs, such as an RTX 4060, face a similar transfer bottleneck, although differences in GPU computation may change the exact speedup. CA illustrates partial residency by keeping attention projection weights in GPU memory. Table 1 lists the models, configurations and weight formats we evaluate. Each case is named MODEL-CONFIG-FORMAT, e.g. LLAMA-ALL-BF16, and is compared against the dense model in the same format. We test release-format BF16 and llama.cpp's 4-bit Q4_K_M to compare the two-bit middle tier against different weight precisions. Most QWEN blocks use Gated DeltaNet linear attention. We evaluate QWEN only in CA, applying TACT to its FFN weights while keeping attention weights in GPU memory.

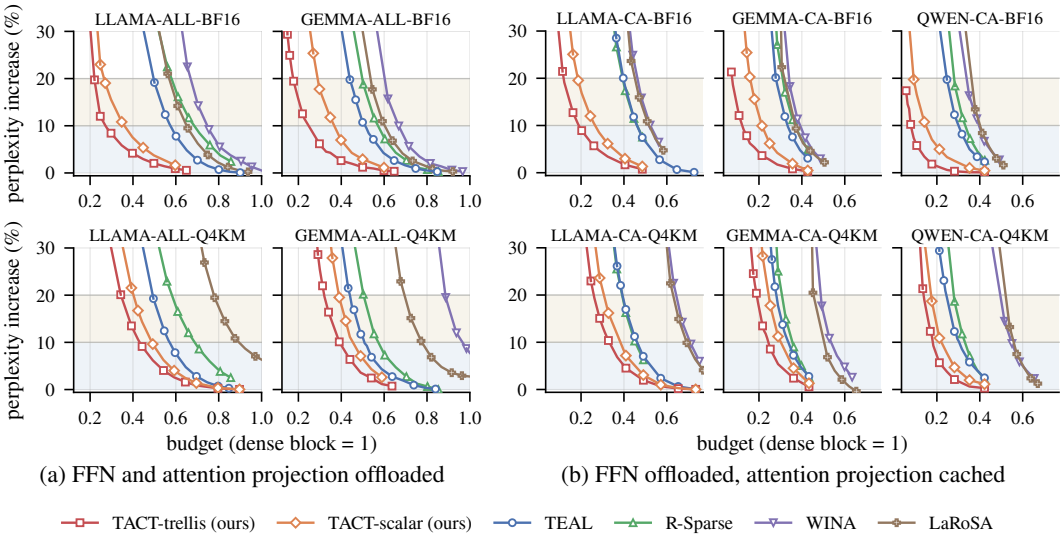
We compare against prior training-free activation sparsity methods, TEAL (Liu et al., 2025a), WINA (Chen et al., 2026), LaRoSA (Liu et al., 2025b) and R-Sparse (Zhang et al., 2025). TEAL selects by magnitude rank rather than against its calibrated threshold, so that it reads exactly the bytes it is given. Model quality is evaluated using perplexity measured over the WikiText-2 test data split (Merity et al., 2017) and downstream task accuracy measured using lm-eval (Gao et al., 2024) on

Table 1: The models, configurations and weight formats we evaluate, and their identifiers.

	Identifier	Name
MODEL	LLAMA	Llama-3.1-8B (Grattafiori et al., 2024)
	GEMMA	Gemma-4-31B (Gemma Team, 2026)
	QWEN	Qwen3.6-27B (Qwen Team, 2026)
CONFIG	ALL	FFN and attention projection weights offloaded
	CA	FFN weights offloaded, attention projection weights cached in GPU memory
FORMAT	BF16	16-bit
	Q4KM	Q4_K_M (Gerganov et al., 2023), 4-bit

ARC-Challenge, HellaSwag, PIQA, WinoGrande, MMLU and GSM8K (Clark et al., 2018; Zellers et al., 2019; Bisk et al., 2020; Sakaguchi et al., 2021; Hendrycks et al., 2021; Cobbe et al., 2021). We calibrate hyperparameters of TACT (k_{top}, k_{mid}, see Section 5.2) and all prior works we compare with using WikiText-2 train data split (disjoint from the test split). Full details are in Appendix A.

6.1 Byte-Accuracy Trade-off



(a) FFN and attention projection offloaded (b) FFN offloaded, attention projection cached

Figure 4: Perplexity increase over dense against the budget on WikiText-2 test. The shaded bands mark the two *tolerances*, a perplexity increase of 10% and 20%.

Figure 4 plots the perplexity increase against the budget for TACT and the four baselines. We observe that, first, at both perplexity tolerances, TACT requires a smaller budget than every baseline in all evaluated cases. TACT-trellis's budget advantage persists across the plotted range and narrows near zero perplexity loss. For LLAMA-ALL-BF16, TACT requires a 27% budget at the 10% tolerance, compared with 57% for TEAL. At the 20% tolerance, the corresponding budgets are 22% and 50%. Second, TACT-trellis requires a smaller budget than TACT-scalar at the same perplexity tolerance. At the 10% tolerance, its budget is 7 percentage points lower on average (up to 12). This is because trellis coding incurs less quantization error than scalar coding at the same two-bit rate (Section 5.1). Third, TACT reduces the budget relative to TEAL by less in Q4_K_M than in BF16. At the 10% tolerance, TACT's budget is 15 percentage points below TEAL's on LLAMA-ALL-Q4KM. On LLAMA-ALL-BF16, the difference is 30 points. This is because the middle-tier codes remain at two bits per weight in both formats, $1/8$ of a BF16 weight but about $2/5$ of a Q4_K_M weight, so replacing weights by codes saves a smaller fraction of bytes in Q4_K_M. Fourth, TACT requires a smaller budget than TEAL even when sparsity is applied only to the FFN (CA). On LLAMA-CA-BF16 at the 10% tolerance, TACT requires a 19% budget, against 46% for TEAL.

Table 2 reports the downstream accuracy at the two tolerances. At the 10% tolerance, TACT loses up to about 4 points relative to dense, and about 6 points on GSM8K. At the 20% tolerance, the accuracy loss is about twice as large. At either tolerance, TACT achieves accuracy comparable to TEAL while requiring less weight data per token.

Table 2: Downstream accuracy of LLAMA-ALL-BF16 for TACT and TEAL, the closest baseline in Figure 4, at the two tolerances. Appendix D gives all models and settings.

	Budget	PPL	Arc-C	HS	PIQA	WG	MMLU	GSM8K
LLAMA-BF16 (Dense)	100%	6.24	52.1	60.8	79.1	75.1	65.4	50.2
TEAL-ALL +10%	57%	6.89 (+10.4%)	48.4	58.5	78.8	72.3	62.3	42.2
TACT-ALL +10%	**27%**	6.87 (+10.1%)	49.1	58.5	78.6	73.4	62.1	43.9
TEAL-ALL +20%	50%	7.44 (+19.2%)	47.4	56.7	78.0	71.0	59.1	36.0
TACT-ALL +20%	**22%**	7.47 (+19.7%)	48.1	56.9	77.4	70.6	60.1	39.2

6.2 Decoding Speeds

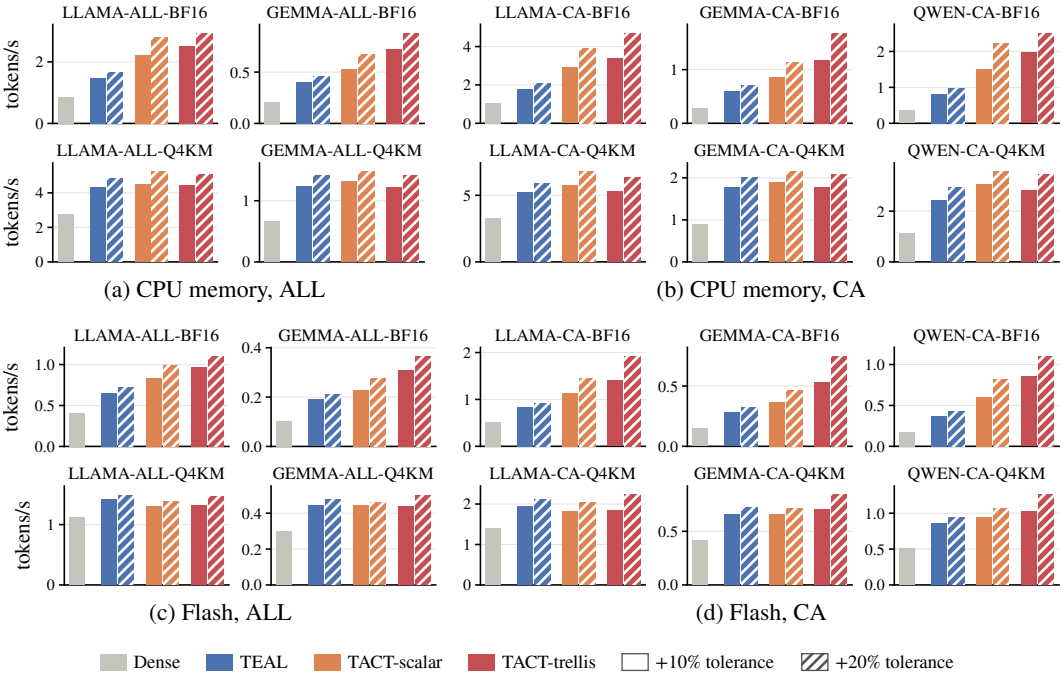


Figure 5: Decoding speed with the weights offloaded to CPU memory (a, b) and to flash storage (c, d), in the ALL and CA configurations of Table 1. Each method runs at its budget for each tolerance.

CPU memory offloading Figure 5(a, b) reports the decoding speed of the dense model, TEAL, TACT-trellis, and TACT-scalar. First, we observe that with BF16 weights, TACT-trellis decodes fastest across all evaluated models and configurations (up to 2.60×, 2.07× on average over TEAL, and up to 7.10×, 4.42× faster on average than dense model). Second, with Q4_K_M weights, TACT-scalar decodes fastest across the evaluated models and configurations (up to 1.25×, 1.10× on average over TEAL) although TACT-trellis reads fewer bytes. This is because of shorter GPU reconstruction times of TACT-scalar (Table 3). Third, caching the attention projections in GPU memory speeds up TACT more than TEAL. From LLAMA-ALL-BF16 to LLAMA-CA-BF16, TACT-trellis becomes 3.23–4.44× faster than dense, while TEAL becomes 1.69–1.95× over dense. This is because TACT saves more bytes on the FFN weights compared to attention projections.

Flash offloading Figure 5(c, d) reports the decoding speed of the same methods. First, as with CPU memory offloading, TACT-trellis decodes fastest with BF16 weights across all evaluated models and configurations. It is 1.92× faster than TEAL and 3.78× faster than dense on average (up to 2.57× and 6.22×). Second, unlike with CPU memory offloading, TACT-trellis outperforms TACT-scalar in most Q4_K_M cases. Because reading from flash is slower, the bytes TACT-trellis saves now outweigh its longer GPU reconstruction. Third, with Q4_K_M weights, TACT-trellis provides a smaller speedup over TEAL (1.07× on average). This is because flash reads have a coarser granularity than the reads of TACT (Table 3).

Decoding time breakdown Table 3 shows the breakdown of decoding time into transfer time, GPU computation (including middle-tier reconstruction), and the rest. We observe that, first, TACT-

trellis spends more GPU time (52–64 ms) than TACT-scalar (22–27 ms) due to longer reconstruction times (Section 5.1). In Q4_K_M with CPU memory offloading, TACT-trellis transfers fewer bytes than TACT-scalar and spends 24 ms less on the transfer, but its reconstruction takes 30 ms longer on the GPU. Second, unlike CPU memory offloading, flash offloading transfers more bytes than a token needs: about 1.52× in Q4_K_M. This is due to fragmentation: flash reads have a coarse granularity, while TACT's column ranges are often small, especially in Q4_K_M, where columns are smaller. Even so, TACT-trellis transfers fewer bytes than TEAL in all settings.

Table 3: Breakdown of decoding time per token of LLAMA-ALL in both formats with CPU memory and flash offloading. Appendix C gives all models and settings.

			CPU memory (ms)			Flash (ms)			
	needed (GB)	GPU (ms)	transfer	other	total	moved (GB)	transfer	other	total
BF16 (Dense)	13.96	26	1,118	12	1,156	13.96	2,388	60	2,474
TEAL +10%	7.95	14	637	29	680	7.99	1,382	142	1,538
TACT-trellis +10%	3.76	64	302	31	397	4.11	834	134	1,032
TACT-scalar +10%	4.83	27	388	34	449	5.25	1,016	146	1,190
Q4KM (Dense)	4.18	18	336	12	366	4.18	825	53	896
TEAL +10%	2.36	11	191	29	231	3.12	587	104	702
TACT-trellis +10%	1.76	52	142	33	227	2.67	588	116	756
TACT-scalar +10%	2.05	22	166	34	222	3.13	646	106	773

Comparison to weight quantization We compare TACT against Llama-3.1-8B quantized to 2 and 3 bits with QTIP (Tseng et al., 2024b) without fine-tuning (Figure 6). We find that TACT reaches a lower perplexity than QTIP at both budgets. At the budget of 2-bit QTIP, QTIP raises perplexity by 44.8%, against 13.5% for TACT-trellis and 21.5% for TACT-scalar. This is because QTIP lowers the precision of every column, while TACT keeps the columns of the largest activation entries in the deployment's weight format (Section 4).

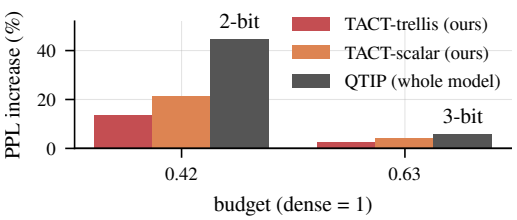


Figure 6: Perplexity increase at the budgets of 2- and 3-bit QTIP, on LLAMA-ALL-Q4KM.

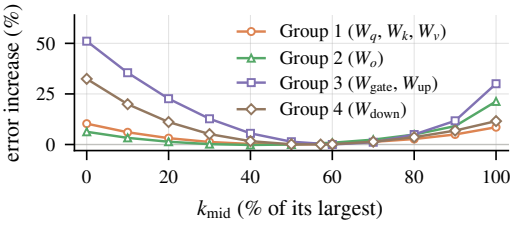


Figure 7: Layer output error of each k_{mid} for layer 15 of LLAMA-ALL-Q4KM.

6.3 Sensitivity to the Size of the Middle Tier

At a fixed budget, we sweep each weight matrix group's k_{mid} grid on layer 15 of LLAMA-ALL-Q4KM, with the other groups fixed (Figure 7). First, both ends of the grid leave more error than the k_{mid} TACT chooses. Setting $k_{\text{mid}} = 0$ raises the layer's error by 6–51%, and the largest k_{mid} raises it by 9–30%. This is because a larger k_{mid} recovers more of the omitted contributions but leaves a smaller k_{top}. Second, the best k_{mid} depends on the weight format. It lies at 50–60% of the largest k_{mid} in Q4_K_M and at 70–86% in BF16. Appendix B provides further ablations. For every weight group of a layer, we sweep k_{mid} at the group's fixed budget and report the tier sizes and layer error at each grid point, the group's share of the layer budget, and the tier sizes selected by Step 2 (Figure 2).

7 Conclusion

We present TACT, a training-free mechanism to accelerate autoregressive decoding in LLMs whose weights exceed GPU memory and must be fetched from CPU memory or flash storage. TACT reduces the weight data required per token by skipping weight columns associated with low-magnitude activations or reading compressed approximations of those columns. Across state-of-the-art LLMs, TACT requires less weight data per token than prior training-free activation-sparsity baselines at evaluated perplexity loss tolerances. At a WikiText-2 perplexity increase of at most 10%, with weights offloaded to CPU memory, it achieves decoding speedups over dense inference of 3.86× on average (up to 5.57×) in BF16 and 2.06× (up to 2.74×) with 4-bit weights. With flash offloading, the speedups are 3.31× (up to 4.81×) and 1.54× (up to 2.03×).

Reproducibility Statement

Section 5 describes our method and calibration procedure. Section 6 and Appendix A specify the evaluated models, datasets, baselines, evaluation protocols, and hardware setup.

We provide the source code of TACT as a supplementary ZIP archive. A README in the archive describes setup and execution. The code implements tiered reads with both middle-tier codes, the two-step calibration procedure, and the llama.cpp engine for offloading weights to CPU memory and flash storage. It also includes perplexity evaluation for TACT, TEAL, WINA, LaRoSA, and R-Sparse. The archive contains the calibrated k_{top} and k_{mid} values for every layer of each evaluated model, configuration, and weight format at the 10% and 20% tolerances. It also includes CSV files containing the data plotted in Figure 1 and every figure in Section 6.

AI Usage

We used Claude and ChatGPT to improve the clarity and readability of the manuscript, and Claude Code to assist with code implementation and figure generation. We did not use AI tools to develop the core research ideas or determine the research direction. We reviewed the AI-assisted text and thoroughly checked the code and experimental results for correctness. We take full responsibility for the final content of this paper, including all text, code, figures, and scientific claims.

References

Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12562–12584, 2024.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 7432–7439, 2020.

Sihan Chen, Dan Zhao, Jongwoo Ko, Colby Banbury, Huiping Zhuang, Luming Liang, Pashmina Cameron, and Tianyi Chen. Wina: Weight informed neuron activation for accelerating large language model inference. In *International Conference on Learning Representations*, volume 2026, pp. 122858–122880, 2026.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Marco Federici, Davide Belli, Mart Van Baalen, Amir Jalalirad, Andrii Skliar, Bence Major, Markus Nagel, and Paul Whatmough. Efficient llm inference using dynamic input pruning and cache-aware masking. *Proceedings of Machine Learning and Systems*, 7, 2025.

G.D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973. doi: 10.1109/PROC.1973.9030.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Georgi Gerganov et al. llama.cpp, 2023. URL https://github.com/ggml-org/llama.cpp. Software, MIT licence.

Aaron Grattafiori et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Junhui He, Shangyu Wu, Weidong Wen, Chun Jason Xue, and Qingan Li. Chess: Optimizing llm inference via channel-wise thresholding and selective sparsification. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18658–18668, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Fucheng Jia, Zewen Wu, Shiqi Jiang, Huiqiang Jiang, Qianxi Zhang, Yuqing Yang, Yunxin Liu, Ju Ren, Deyu Zhang, and Ting Cao. Scaling up on-device llms via active-weight swapping between dram and flash. *arXiv preprint arXiv:2504.08378*, 2025.

Donghyun Lee, Jaeyong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. CATS: Context-aware thresholding for sparsity in large language models. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=v3w2a7EInO.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=dGVZwyq5tV.

Kai Liu, Bowen Xu, Shaoyu Wu, Xin Chen, Hao Zhou, Yongliang Tao, and Lulu Hu. La rosa: enhancing llm efficiency via layerwise rotated sparse activation. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025b.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023.

S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2): 129–137, 1982. doi: 10.1109/TIT.1982.1056489.

Mark Z. Mao and Robert M. Gray. Stationary and trellis encoding for iid sources and simulation. In *Proceedings of the 2010 Data Compression Conference*, DCC '10, pp. 3–12, USA, 2010. IEEE Computer Society. ISBN 9780769539942. doi: 10.1109/DCC.2010.8. URL https://doi.org/10.1109/DCC.2010.8.

J. Max. Quantizing for minimum distortion. *IRE Transactions on Information Theory*, 6(1):7–12, 1960. doi: 10.1109/TIT.1960.1057548.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. In *International Conference on Learning Representations*, volume 2024, pp. 25378–25400, 2024.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Jette Oberländer, Jan Finkbeiner, Catherine M Schöfmann, and Emre Neftci. Grinqh: Graded input-based quantization hierarchy for efficient llm generation. *arXiv preprint arXiv:2606.23419*, 2026.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27b dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, August 2021. ISSN 0001-0782. doi: 10.1145/3474381. URL https://doi.org/10.1145/3474381.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, et al. Prosparse: Introducing and enhancing intrinsic activation sparsity within large language models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 2626–2644, 2025.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, 2024a.

Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. Turbo sparse: Achieving llm sota performance with minimal activated parameters. *arXiv preprint arXiv:2406.05955*, 2024b.

Filip Szatkowski, Patryk Będkowski, Alessio Devoto, Jan Dubiński, Pasquale Minervini, Mikołaj Piórczyński, Simone Scardapane, and Bartosz Wójcik. Universal properties of activation sparsity in modern large language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=I9M5OoOBnI.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with hadamard incoherence and lattice codebooks. In *Forty-first International Conference on Machine Learning*, 2024a. URL https://openreview.net/forum?id=9BrydUVcoe.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: Quantization with trellises and incoherence processing. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b. URL https://openreview.net/forum?id=7sdkLVuYCU.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Rongxiang Wang, Kangyuan Shu, and Felix Xiaozhu Lin. Enabling dynamic sparsity in quantized llm inference. *arXiv preprint arXiv:2511.04477*, 2025a.

Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. Neuralink: Fast on-device llm inference with neuron co-activation linking. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 147–162, 2025b.

Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024.

Kichang Yang, Seonjun Kim, Minjae Kim, Nairan Zhang, Chi Zhang, and Youngki Lee. VLM in a flash: I/o-efficient sparsification of vision-language model via neuron chunking. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=3xwsD68F2K.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

Zhenyu Zhang, Zechun Liu, Yuandong Tian, Harshit Khaitan, Zhangyang Wang, and Steven Li. R-sparse: Rank-aware activation sparsity for efficient llm inference. In *International Conference on Learning Representations*, volume 2025, pp. 77452–77466, 2025.

Appendix

A Experimental Setup Details

Baselines. We compare against four training-free activation sparsity methods: TEAL (Liu et al., 2025a), WINA (Chen et al., 2026), LaRoSA (Liu et al., 2025b) and R-Sparse (Zhang et al., 2025). TEAL selects by magnitude rank rather than against its calibrated threshold, so that it reads exactly the bytes it is given.

Metrics. We report the *read budget*: the weight data required for one decoded token of a block, expressed in bytes and normalized by the block's dense weight size. TACT's codes, two bits per weight, are counted in those bytes, the codebook they index included: 256 KB for the whole model. The rotations WINA and LaRoSA read and R-Sparse's low-rank factors are counted too, at 16 bits in both weight formats, which is the precision their releases store them at.

Evaluation. Perplexity is measured over the whole WikiText-2 test split (Merity et al., 2017) in non-overlapping 2048-token windows, and downstream accuracy by lm-eval (Gao et al., 2024) on ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020) and WinoGrande (Sakaguchi et al., 2021) 0-shot, and on MMLU (Hendrycks et al., 2021) and GSM8K (Cobbe et al., 2021) 5-shot, each benchmark in full. Every method is applied to the whole of each scored sequence, except on GSM8K: the model generates its answer there, so the prompt is read densely and the method applies to the generated tokens. The two offline searches of TACT (Section 5.2) and the calibration of every baseline that needs one use the WikiText-2 train split, which is disjoint from the test split.

Offloading setup. We implement TACT in llama.cpp and decode at batch one, with every projection computed on the GPU. The embeddings, the output head and the KV cache stay in GPU memory, as do the attention projection weights in CA. In CPU memory offloading, the offloaded weights and TACT's codes are held in pinned CPU memory, and each decoding step transfers the selected ranges into GPU memory over PCIe. In flash offloading, they are read from flash storage (the NVMe SSD) with direct I/O, so that every read is served by the drive rather than by the page cache. Both settings run on one workstation with an RTX 4090 on a PCIe 4.0 x8 link, 128 GB of CPU memory and an NVMe SSD. A prompt is read densely, and the tiered read applies to the decoding steps. Each reported latency is the median over eight decoding steps, taken after four warm-up steps, and the bytes those steps read are metered on the same run.

B k_{top} and k_{mid} at One Layer

Table 4 gives the k_{top} and k_{mid} that the two searches of Section 5.2 choose for layer 15 of Llama-3.1-8B. Here the budget covers the seven weight matrices of one layer. Every layer runs at a budget of 24.6% (107.4 of 436.2 MB) in BF16 and about 39% (54.0 of 138.9 MB) in Q4_K_M. At these budgets, the perplexity of the model is 12.0% above dense in BF16 and 13.5% in Q4_K_M. Beside each group of matrices is the share of the layer's budget that Step 1 gives it. The top, middle and bottom columns give k_{top}/n, k_{mid}/n and the rest, in percent. Each row changes the k_{mid} of one group and holds the other groups at the k_{mid} Step 2 chose, shown in bold. Error is the layer's output error against reading every weight, on calibration data. The k_{top} and k_{mid} of every layer at the $+10\%$ and $+20\%$ tolerances are provided with our code in the supplementary ZIP archive.

Table 4: k_{top} and k_{mid} of each group at layer 15 of Llama-3.1-8B.

(a) BF16

	top	middle	bottom	error
W_q, W_k, W_v	30.0	0.0	70.0	0.1619
14.0%	29.0	8.0	63.0	0.1465
(15.1 MB)	28.0	16.0	56.0	0.1361
	27.0	24.0	49.0	0.1292
	26.0	32.0	42.0	0.1248
	25.0	40.0	35.0	0.1220
	24.0	48.0	28.0	0.1205
	23.0	**56.0**	**21.0**	**0.1197**
	22.0	64.0	14.0	0.1195
	21.0	72.0	7.0	0.1196
	20.0	80.0	0.0	0.1200
W_o	30.0	0.0	70.0	0.1438
9.4%	29.0	8.0	63.0	0.1346
(10.1 MB)	28.0	16.0	56.0	0.1285
	27.0	24.0	49.0	0.1244
	26.0	32.0	42.0	0.1219
	25.0	40.0	35.0	0.1205
	24.0	48.0	28.0	0.1198
	23.0	56.0	21.0	0.1197
	22.0	**64.0**	**14.0**	**0.1197**
	21.0	72.0	7.0	0.1200
	20.0	80.0	0.0	0.1204
$W_{\text{gate}}, W_{\text{up}}$	25.7	0.0	74.3	0.2837
56.3%	24.7	8.5	66.9	0.2417
(60.4 MB)	23.6	17.0	59.4	0.2071
	22.5	25.5	52.0	0.1786
	21.5	34.0	44.6	0.1564
	20.4	42.4	37.1	0.1400
	19.3	50.9	29.7	0.1290
	18.3	59.4	22.3	0.1226
	17.2	67.9	14.9	0.1200
	16.6	**72.8**	**10.6**	**0.1197**
	16.2	76.4	7.4	0.1200
	15.1	84.9	0.0	0.1212
W_{down}	18.6	0.0	81.4	0.2216
20.3%	17.4	9.3	73.3	0.1861
(21.8 MB)	16.2	18.6	65.1	0.1606
	15.1	27.9	57.0	0.1429
	13.9	37.2	48.9	0.1312
	12.8	46.5	40.7	0.1242
	11.6	55.8	32.6	0.1207
	10.4	65.1	24.4	0.1197
	10.3	**66.5**	**23.3**	**0.1197**
	9.3	74.4	16.3	0.1203
	8.1	83.8	8.1	0.1217
	6.9	93.1	0.0	0.1236

(b) Q4_K_M

	top	middle	bottom	error
W_q, W_k, W_v	43.3	0.0	56.7	0.1408
12.2%	39.3	9.7	51.0	0.1354
(6.6 MB)	35.4	19.3	45.3	0.1317
	31.4	29.0	39.7	0.1293
	27.4	38.6	34.0	0.1280
	23.4	**48.3**	**28.3**	**0.1277**
	19.4	57.9	22.7	0.1282
	15.4	67.6	17.0	0.1294
	11.5	77.2	11.3	0.1311
	7.5	86.9	5.7	0.1340
	3.5	96.5	0.0	0.1386
W_o	40.0	0.0	60.0	0.1357
7.0%	36.0	9.0	55.0	0.1319
(3.8 MB)	32.0	18.0	50.0	0.1294
	28.0	27.0	45.0	0.1279
	24.0	36.0	40.0	0.1274
	20.0	**45.0**	**35.0**	**0.1277**
	16.0	54.0	30.0	0.1288
	12.0	63.0	25.0	0.1308
	8.0	72.0	20.0	0.1339
	4.0	81.0	15.0	0.1393
	0.0	90.0	10.0	0.1549
$W_{\text{gate}}, W_{\text{up}}$	42.1	0.0	57.9	0.1929
51.5%	37.9	9.5	52.6	0.1730
(27.8 MB)	33.7	19.0	47.3	0.1566
	29.5	28.4	42.1	0.1439
	25.3	37.9	36.8	0.1347
	21.1	47.4	31.5	0.1294
	16.9	**56.9**	**26.2**	**0.1277**
	12.6	66.4	21.0	0.1291
	8.4	75.9	15.7	0.1340
	4.2	85.3	10.4	0.1426
	0.0	94.8	5.2	0.1660
W_{down}	32.9	0.0	67.1	0.1691
29.3%	29.9	9.7	60.4	0.1531
(15.8 MB)	27.0	19.3	53.7	0.1419
	24.0	29.0	47.0	0.1342
	21.1	38.6	40.3	0.1298
	18.1	48.3	33.6	0.1278
	16.0	**55.2**	**28.8**	**0.1277**
	15.2	57.9	26.9	0.1279
	12.3	67.6	20.1	0.1294
	9.3	77.3	13.4	0.1323
	6.4	86.9	6.7	0.1365
	3.4	96.6	0.0	0.1424

C Decoding Time Breakdown of Llama, Gemma and Qwen

Tables 5, 6 and 7 give the breakdown of Table 3 for Llama-3.1-8B in every configuration and weight format, Gemma-4-31B and Qwen3.6-27B. Qwen is evaluated in CA only.

Table 5: Breakdown of the decoding time per token of Llama-3.1-8B with CPU memory and flash offloading.

				CPU memory			Flash			
		needed (GB)	GPU (ms)	transfer (ms)	other (ms)	total (ms)	moved (GB)	transfer (ms)	other (ms)	total (ms)
LLAMA-ALL-BF16	Dense	13.96	26	1,118	12	1,156	13.96	2,388	60	2,474
	TEAL +10%	7.95	14	637	29	680	7.99	1,382	142	1,538
	TACT-trellis +10%	3.76	64	302	31	397	4.11	834	134	1,032
	TACT-scalar +10%	4.83	27	388	34	449	5.25	1,016	146	1,190
LLAMA-CA-BF16	Dense	11.27	31	903	11	945	11.27	1,844	60	1,935
	TEAL +10%	6.44	22	516	22	560	6.44	1,066	105	1,192
	TACT-trellis +10%	2.60	61	209	23	293	2.85	550	100	712
	TACT-scalar +10%	3.62	28	290	26	344	3.92	730	122	880
LLAMA-ALL-Q4KM	Dense	4.18	18	336	12	366	4.18	825	53	896
	TEAL +10%	2.36	11	191	29	231	3.12	587	104	702
	TACT-trellis +10%	1.76	52	142	33	227	2.67	588	116	756
	TACT-scalar +10%	2.05	22	166	34	222	3.13	646	106	773
LLAMA-CA-Q4KM	Dense	3.41	23	274	11	308	3.41	650	39	712
	TEAL +10%	1.88	18	152	21	191	2.39	422	78	518
	TACT-trellis +10%	1.37	53	110	25	188	1.94	408	80	540
	TACT-scalar +10%	1.60	21	129	25	175	2.33	462	68	550

Table 6: Breakdown of the decoding time per token of Gemma-4-31B with CPU memory and flash offloading.

				CPU memory			Flash			
		needed (GB)	GPU (ms)	transfer (ms)	other (ms)	total (ms)	moved (GB)	transfer (ms)	other (ms)	total (ms)
GEMMA-ALL-BF16	Dense	58.57	103	4,690	34	4,827	58.57	9,389	170	9,662
	TEAL +10%	29.20	44	2,339	74	2,456	30.55	4,868	343	5,255
	TACT-trellis +10%	13.30	226	1,067	84	1,377	15.18	2,631	374	3,231
	TACT-scalar +10%	21.40	76	1,716	88	1,880	23.77	3,913	396	4,385
GEMMA-CA-BF16	Dense	41.62	103	3,331	30	3,464	41.62	6,576	140	6,818
	TEAL +10%	19.61	63	1,571	52	1,686	20.61	3,252	242	3,557
	TACT-trellis +10%	7.11	222	570	59	851	8.18	1,392	256	1,870
	TACT-scalar +10%	12.54	84	1,006	64	1,154	14.19	2,334	290	2,709
GEMMA-ALL-Q4KM	Dense	17.51	75	1,403	32	1,510	17.51	3,110	144	3,330
	TEAL +10%	8.74	30	708	72	810	10.96	1,936	288	2,254
	TACT-trellis +10%	6.81	181	550	86	816	9.38	1,853	263	2,297
	TACT-scalar +10%	7.68	52	621	82	755	10.61	2,029	180	2,261
GEMMA-CA-Q4KM	Dense	12.60	90	1,009	29	1,128	12.60	2,180	130	2,400
	TEAL +10%	5.69	52	458	50	561	7.06	1,271	207	1,530
	TACT-trellis +10%	4.17	173	335	59	567	5.81	1,096	150	1,420
	TACT-scalar +10%	5.05	64	407	59	530	7.09	1,360	100	1,524

Table 7: Breakdown of decoding time per token of Qwen3.6-27B with CPU memory and flash offloading.

				CPU memory			Flash			
		needed (GB)	GPU (ms)	transfer (ms)	other (ms)	total (ms)	moved (GB)	transfer (ms)	other (ms)	total (ms)
QWEN-CA-BF16	Dense	34.23	58	2,739	35	2,832	34.23	5,485	109	5,652
	TEAL +10%	14.41	25	1,156	56	1,236	14.90	2,452	205	2,682
	TACT-trellis +10%	3.78	145	303	60	508	4.62	836	193	1,174
	TACT-scalar +10%	6.87	44	551	65	660	7.79	1,374	230	1,648
QWEN-CA-Q4KM	Dense	10.36	42	830	25	897	10.36	1,848	87	1,978
	TEAL +10%	4.36	15	352	44	411	5.60	1,038	121	1,174
	TACT-trellis +10%	2.60	100	209	46	355	3.76	747	126	973
	TACT-scalar +10%	3.12	26	251	50	328	4.57	925	112	1,063

D DOWNSTREAM ACCURACY OF LLAMA, GEMMA AND QWEN

Tables 8, 9 and 10 give the downstream accuracy of Table 2 for Llama-3.1-8B, Gemma-4-31B and Qwen3.6-27B.

Table 8: Downstream accuracy of Llama-3.1-8B for TACT and TEAL at the two tolerances.

	Budget	PPL	Arc-C	HS	PIQA	WG	MMLU	GSM8K
LLAMA-BF16 (Dense)	100%	6.24	52.1	60.8	79.1	75.1	65.4	50.2
TEAL-ALL +10%	57%	6.89 (+10.4%)	48.4	58.5	78.8	72.3	62.3	42.2
TACT-ALL +10%	**27%**	6.87 (+10.1%)	49.1	58.5	78.6	73.4	62.1	43.9
TEAL-ALL +20%	50%	7.44 (+19.2%)	47.4	56.7	78.0	71.0	59.1	36.0
TACT-ALL +20%	**22%**	7.47 (+19.7%)	48.1	56.9	77.4	70.6	60.1	39.2
TEAL-CA +10%	46%	6.86 (+9.9%)	49.8	59.5	78.2	71.0	63.3	44.0
TACT-CA +10%	**19%**	6.88 (+10.2%)	48.5	59.3	78.8	71.9	62.7	44.7
TEAL-CA +20%	40%	7.49 (+20.0%)	47.5	57.7	78.6	70.6	60.5	36.4
TACT-CA +20%	**11%**	7.58 (+21.4%)	47.1	58.0	78.5	71.5	60.1	37.5
LLAMA-Q4KM (Dense)	100%	6.44	51.4	60.0	78.7	73.6	64.7	48.6
TEAL-ALL +10%	57%	7.11 (+10.4%)	48.5	57.6	77.9	71.1	60.9	37.9
TACT-ALL +10%	**42%**	7.12 (+10.6%)	48.3	57.9	78.9	72.2	60.5	39.7
TEAL-ALL +20%	50%	7.68 (+19.3%)	45.0	55.8	76.9	71.7	56.8	32.8
TACT-ALL +20%	**34%**	7.73 (+20.1%)	46.0	56.3	77.6	70.6	56.7	32.8
TEAL-CA +10%	45%	7.15 (+11.1%)	49.5	58.8	77.6	72.5	61.8	39.7
TACT-CA +10%	**33%**	7.10 (+10.2%)	50.0	59.4	78.3	72.5	61.9	39.6
TEAL-CA +20%	39%	7.72 (+19.9%)	47.8	58.0	77.4	71.9	59.6	34.6
TACT-CA +20%	**25%**	7.79 (+21.0%)	48.3	57.9	77.8	70.0	59.0	35.7

Table 9: Downstream accuracy of Gemma-4-31B for TACT and TEAL at the two tolerances.

	Budget	PPL	Arc-C	HS	PIQA	WG	MMLU
GEMMA-BF16 (Dense)	100%	6.39	64.8	65.8	82.9	80.1	84.5
TEAL-ALL +10%	50%	7.07 (+10.7%)	63.3	62.8	82.2	77.7	82.4
TACT-ALL +10%	**23%**	7.12 (+11.5%)	63.0	64.0	81.2	79.1	82.8
TEAL-ALL +20%	44%	7.65 (+19.8%)	59.2	60.8	81.6	76.5	80.6
TACT-ALL +20%	**18%**	7.63 (+19.4%)	61.2	63.2	81.5	76.2	82.0
GEMMA-Q4KM (Dense)	100%	6.54	63.7	65.7	82.8	79.6	83.6
TEAL-ALL +10%	50%	7.26 (+11.1%)	62.0	62.9	82.2	76.9	81.6
TACT-ALL +10%	**39%**	7.20 (+10.1%)	62.7	63.9	82.3	77.4	82.3
TEAL-ALL +20%	43%	7.94 (+21.5%)	59.5	60.8	80.8	75.4	80.1
TACT-ALL +20%	**32%**	7.82 (+19.7%)	61.3	62.7	81.9	75.1	80.7

Table 10: Downstream accuracy of Qwen3.6-27B for TACT and TEAL at the two tolerances.

	Budget	PPL	Arc-C	HS	PIQA	WG
QWEN-BF16 (Dense)	100%	7.40	60.3	64.0	81.6	77.2
TEAL-CA +10%	30%	8.18 (+10.6%)	58.7	62.2	80.9	75.3
TACT-CA +10%	**8%**	8.16 (+10.2%)	57.6	62.9	80.5	74.3
TEAL-CA +20%	25%	8.86 (+19.7%)	56.3	60.2	79.3	73.5
TACT-CA +20%	**6%**	8.69 (+17.4%)	56.4	62.2	80.3	73.2
QWEN-Q4KM (Dense)	100%	7.47	59.0	63.9	81.4	77.0
TEAL-CA +10%	30%	8.27 (+10.7%)	57.3	62.4	80.1	75.5
TACT-CA +10%	**18%**	8.32 (+11.4%)	58.3	62.0	80.2	75.6
TEAL-CA +20%	24%	9.04 (+20.9%)	56.4	60.6	79.8	73.6
TACT-CA +20%	**13%**	9.07 (+21.3%)	56.7	60.4	79.6	74.3

E EFFICIENTLY LOADING OFFLOADED WEIGHTS

TACT must fetch the selected columns with fewer and larger requests. Activation sparsity already splits a token's read into many small requests. Middle-tier columns are read in even smaller requests, because their codes occupy fewer bytes than their weights.

TACT fetches the weight columns one activation entry indexes in fewer and larger requests, by storing them in contiguous ranges of flash storage. It stores the codes of those columns the same way, in a separate region. To this end, TACT divides the seven weight matrices of the attention projections and the FFN into four groups by the input activation they share: (W_q, W_k, W_v), (W_o), $(W_{\text{gate}}, W_{\text{up}})$ and (W_{down}). An entry of that activation indexes one column of every matrix in its group, as entry x_2 indexes Q_2, K_2 and V_2 in Figure 8. To place the columns of one entry in the same tier, TACT ranks that activation once for the group, and the two searches of Section 5.2 give the group one budget and one setting $(k_{\text{top}}, k_{\text{mid}})$. In the figure, x_2 is in the top tier, so a single request reads Q_2, K_2 and V_2. If each matrix occupied its own region, the same three columns would take three requests. Entries x_0 and x_3 are in the middle tier, and the codes of each entry's three columns take one request. Each region holds its ranges in the order of the entries that index them. At each decoding step, TACT merges the adjacent ranges it reads, so a run of neighbouring entries in one tier takes one request instead of one each (Section 6.2).

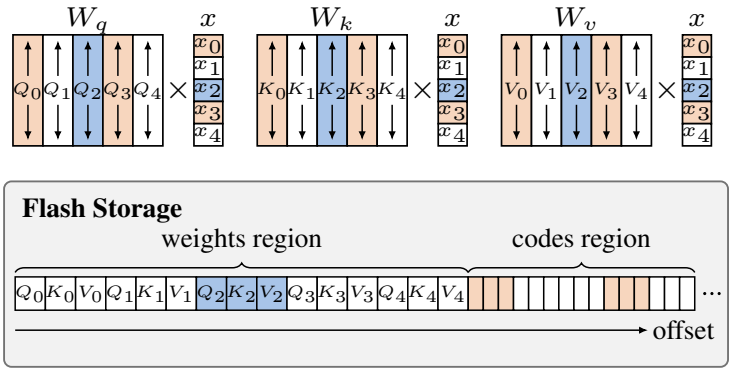


Figure 8: Weight columns of W_q, W_k and W_v arranged contiguously in flash storage.