ELASTIC DECODING: CONTROLLING LLM INFERENCE SPEED VIA CONTEXTUAL SPARSITY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) inference in memory-bound regimes is constrained by a fundamental trade-off between generation quality and decoding speed. In this regime, contextual sparsity (e.g., mixtures of experts) accelerates autoregressive generation by reducing the number of active parameters loaded per token, while preserving the model's total parameter count. We introduce *Elastic Decoding*, a contextual sparsity method that can be applied post-hoc to any trained LLM, dense or MoE, returns a single checkpoint with controllable sparsity levels, requires less than 5B finetuning tokens, and is at the Pareto frontier of speed and accuracy. To achieve this, we frame contextual sparsification as a multiple-choice knapsack problem with bytes-per-token as the budget, allocating capacity heterogeneously across the architecture before distilling the model into a single sparse checkpoint. Elastic Decoding can reduce the active number of parameters by up to $83\,\%$ and, paired with optimized on-device kernels, achieves end-to-end speed-ups of up to $4.6\times$ (e.g., on Gemma4-31B). Our sparsified Qwen3-14B runs faster and more accurately than Qwen3-8B and Qwen3-4B. Importantly, the quality-speed trade-off can be controlled during decoding in real-time, on a token-by-token basis, with negligible switching overhead.

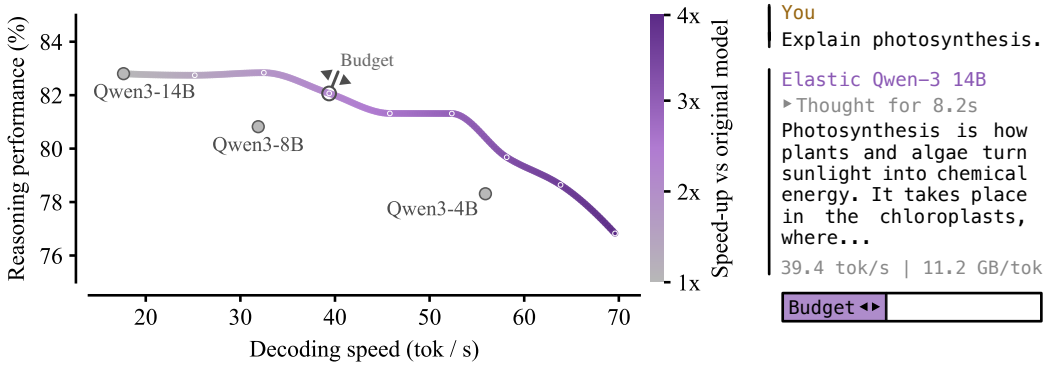


Figure 1: **Elastic Decoding enables decode-time control over the quality–speed trade-off.** Left: A single Elastic Decoding checkpoint (purple band) for Qwen3-14B enables switching between per-token budgets, achieving up to $4\times$ decoding speed-ups over `llama.cpp` with little loss in reasoning performance. Measurements on an Apple M4 Max at bfloat16 and batch size one. Right: Inference engine with token-by-token budget selection (see Supp. Video for a demonstration).

1 INTRODUCTION

Deploying large language models (LLMs) on local devices is fundamentally a constrained optimization problem where the goal is to maximize model quality under limits on memory, latency, power, and energy. In this setting, the objective is not to improve quality or efficiency in isolation, but to push the quality–efficiency frontier for effective on-device deployment. Moreover, these resource constraints are highly dynamic: available power, memory, and latency budgets constantly shift based

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.

on the current workload or device state. For instance, an application might demand a split-second execution for a routine, time–critical query, yet allocate a larger latency budget for complex reasoning tasks. Similarly, the operating system may restrict power or memory to manage thermal limits or preserve a low battery. To remain effective, the deployed model must be capable of seamlessly shifting along the frontier to match whatever resources it is currently allocated.

At small batch sizes, such as those commonly encountered during local inference, LLM decoding is constrained by memory bandwidth (Liu et al., 2023b; Song et al., 2023; Chen et al., 2023; Leviathan et al., 2023). Generating each token requires transferring the model weights from memory, yielding low arithmetic intensity. This bottleneck is especially severe on local devices (Xue et al., 2024), making it essential to reduce memory traffic while preserving model utility for on-device deployment.

To increase arithmetic intensity, existing methods either generate multiple tokens from a single memory read, as in speculative decoding (Leviathan et al., 2022), or reduce the weights loaded for each token. Among methods that reduce weight loads, some approaches permanently restrict the model's capacity or representation. For instance, serving families of distilled models restricts deployment to rigid capacity tiers using different checkpoints (Grattafiori et al., 2024), while weight quantization (Lin et al., 2024) reduces the numerical precision of parameters. Alternatively, static pruning (Ma et al., 2023) permanently removes weights, forcing all tokens to use the same fixed sparse subnetwork.

To preserve model capacity while reducing per-token cost, we focus on *contextual sparsity*, which dynamically routes each token to a subset of weights. Mixture-of-Experts (MoE) architectures are a prominent example (Shazeer et al., 2017; He, 2024). By fetching only a relevant subset of parameters per token, contextual sparsity reduces memory traffic and enables substantial wall-clock decoding speed-ups (Liu et al., 2023b; Song et al., 2023; Lee et al., 2024; Liu et al., 2025). Native MoEs learn this routing during training, often over trillions of tokens, but are typically tied to a fixed deployment budget. *Post-hoc* contextual sparsification applies token-level parameter selection to pretrained LLMs, introducing sparsity into dense models or further sparsifying existing MoEs. Early post-hoc methods exploited ReLU activation sparsity (Liu et al., 2023b; Song et al., 2023; 2024), while more recent approaches threshold SiLU activations (Lee et al., 2024) or sparsify matrices based on input magnitude (Liu et al., 2025). However, these methods incur substantial downstream degradation beyond $50\,\%$ sparsity.

To overcome these limitations, we propose *Elastic Decoding*, a post-hoc method that frames LLM sparsification as a joint optimization problem over sparsity configurations (the sparsity mode, routing mechanism and the activation budget per component) and model weights. Given any model, dense or MoE, Elastic Decoding produces a single checkpoint whose quality–speed trade-off can be controlled dynamically during decoding. Our main contributions are the following:

- **Contextual sparsity as an optimization problem**: We formalize contextual sparsification as a multiple-choice knapsack problem with bytes-per-token as the budget.
- **Elastic control over inference speed**: A single checkpoint spans an entire family of budgets. The model can switch between them at any token during decoding, with negligible overhead at up to 2 switches per second.
- **State-of-the-art post-hoc sparsity**: Across dense and MoE models (Qwen3-14B, Qwen3-30B-A3B, Gemma4-31B, and Gemma4-26B-A4B), Elastic Decoding is near-lossless at $67\,\%$ sparsity (e.g., Gemma4-31B) and reaches up to $83\,\%$ sparsity with modest degradation, substantially outperforming prior post-hoc methods. Our sparsified Qwen3-14B outperforms the dense Qwen3-8B and Qwen3-4B models in both decoding speed and downstream accuracy (Figure 1).
- **Hardware-validated speedups**: With optimized kernels and a custom on-device inference engine, Elastic Decoding translates contextual sparsity into a $4.6\times$ wall-clock decoding speed-up at $83\,\%$ sparsity, and a $2.5\times$ speed-up at near-lossless quality (e.g., Gemma4-31B).

2 Related work

Mixture of Experts. The most prominent examples of contextual sparsity are Mixture-of-Experts (MoEs), where a routing network dynamically directs each token to a subset of subnetworks of fixed sizes, effectively decoupling the model's total parameter capacity from its active inference compute (Shazeer et al., 2017; Fedus et al., 2022; Dai et al., 2024; Zhang et al., 2022; Jiang et al., 2024). While

MoEs can also be built post-hoc based on a pre-trained model (Zhang et al., 2022) or refined using fine-grained experts (Dai et al., 2024), they inherently restrict contextual sparsity to feed-forward networks (FFNs), bounding the maximum sparsity level.

Optimized quantization or pruning. Several works have proposed to frame optimization of quantization (Cai et al., 2020; Hubara et al., 2021; Sieberling et al., 2025) or pruning strategies (Frantar & Alistarh, 2022) as a multiple-choice knapsack problem. Our sparsity optimization is similar to these approaches, but specifically develops methods that optimize *contextual* sparsity. In addition, we will demonstrate that weight quantization is an orthogonal technique that can be combined with Elastic Decoding.

Training-free sparsification[1]**.** Several methods perform post-hoc contextual sparsification, without a subsequent fine-tuning phase to adapt the original model weights. ReLU FFNs exhibit natural sparsity and contextual-sparsity predictors omit inactive neurons at inference (Liu et al., 2023b). CATS thresholds the SiLU gate activations of gated FFNs, attaining roughly 25 % model-wide sparsity with little loss in the zero-shot setting (Lee et al., 2024). TEAL (Liu et al., 2025) and R-Sparse (Zhang et al., 2025) sparsify LLMs based on the input activations to matrices. R-Sparse uses an evolutionary algorithm to optimize sparsity levels, whereas TEAL uses a greedy algorithm to optimize sparsity levels within a layer.

Training-based sparsification. Because SwiGLU is considerably less sparse than ReLU, prior work *ReLUfies* models by substituting ReLU activations and continuing pre-training to recover sparsity (Mirzadeh et al., 2024; Song et al., 2024; 2025). However, these works perform cross-entropy training on massive datasets (e.g., 150B tokens in Song et al. (2024)) and typically do not sparsify attention or the LM-head, leading to low sparsity levels (Mirzadeh et al., 2024). Another thread of work globally ranks neurons or attention heads in their importance and trains routers to predict the size of the subset to use (Kudugunta et al., 2024; Cai et al., 2024; Taghibakhshi et al., 2026; Tastan et al., 2026). However, ranking neurons globally (i.e., without contextual information) limits expressivity, such that these methods either require extensive pretraining and fail to exceed $50\,\%$ sparsity.

Crucially, all these works consider a single sparsification method *in isolation*, and sometimes also sparsify layers *homogeneously* (Liu et al., 2025), which does not account for varying layer sensitivities (Section 4.3). In contrast, Elastic Decoding directly optimizes the heterogeneous sparsity configuration across layers.

3 METHOD

A core challenge in local deployment is that the optimal trade-off between model quality and decoding speed changes dynamically based on shifting device constraints. We therefore propose Elastic Decoding, which generates a *single model checkpoint* capable of running at multiple operating points, switching between them on a token-by-token basis. This checkpoint can be derived from any pretrained transformer-based model, either dense or MoE, via post-hoc contextual sparsity and lightweight fine-tuning, thereby minimizing the dependency on data-availability.

3.1 ELASTIC DECODING AS A CONSTRAINED OPTIMIZATION PROBLEM

Given a pretrained model, our objective is to reduce its active parameter count during inference to satisfy any target budget, $b \in \mathcal{B}$, with minimal accuracy degradation. We define *budget* as the number of bytes read from memory per generated token. To ensure hardware efficiency, we strictly enforce structured sparsity. This bypasses entire contiguous dimensions of the weight matrices and physically reduces the memory access volume. In this setting, our byte budget acts as a direct proxy for decoding speed in memory-bound regimes (Figure 2a).

For a single budget, b, this results in the following constrained optimization over two sets of parameters: the sparsity configuration (which defines which rows and columns to sparsify), parameterized by θ, and the neural network weights, denoted by ψ:

$$\psi^*, \theta^* \in \arg\min_{\psi,\theta} \mathcal{L}(\theta, \psi) \quad \text{s.t.} \quad \sum_l N_l(\theta) \leq b, \tag{1}$$

[1]Here, *training-free* refers to any approach that leaves the original model weights unmodified.

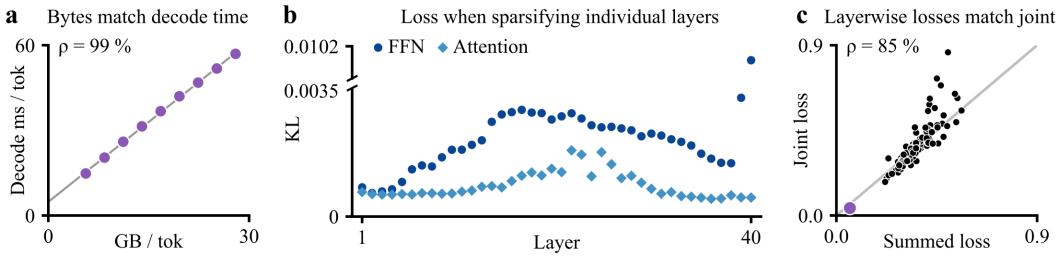


Figure 2: **Motivation for methodology.** Experiments performed on Qwen3-14B. **(a)** Decode time correlates with the decode bytes read per token. **(b)** KL divergence from the dense model when sparsifying the FFN or attention projections in each layer to 50% input sparsity. **(c)** Sparsifying layers one at a time and summing the loss correlates with the joint sparsification loss, across 128 random sparsity allocations with the same budget. Purple dot is the allocation found by our sparsity optimization.

where $\mathcal{L}$ is the optimization objective and $N_l(\theta)$ the number of active bytes read by layer[2] l under sparsity configuration θ. To allow the model to be deployed at multiple budgets using a single model checkpoint, we expand Equation (1) to

$$\psi^*, \{\theta_b^*\}_{b\in\mathcal{B}} \in \underset{\psi,\{\theta_b\}_{b\in\mathcal{B}}}{\arg\min} \sum_{b\in\mathcal{B}} \mathcal{L}(\theta_b, \psi) \qquad \text{s.t.} \quad \sum_l N_l(\theta_b) \le b \qquad \forall b \in \mathcal{B}. \tag{2}$$

This formulation jointly searches for a single set of weights ψ^* that performs well across all budgets and budget-specific sparsity configurations θ_b^* satisfying the corresponding byte-read constraint. Note that, contrary to the model's parameters, the sparsity configuration is lightweight and can be changed dynamically during decoding with negligible decoding overhead (Appendix B).

Solving Equation (2) would require repeatedly updating both the sparsity configurations and the shared model weights. In practice, we decouple the two optimizations in two stages. In Stage 1, we determine the optimal sparsity configuration for each budget using the original pretrained model weights ψ_0 as

$$\theta_b^\star \in \underset{\theta_b}{\arg\min}\, \mathcal{L}(\theta_b, \psi_0) \quad \text{s.t.} \quad \sum_l N_l(\theta_b) \le b \qquad \forall b \in \mathcal{B}. \tag{3}$$

In Stage 2, we hold these configurations fixed and optimize the model weights across all budgets:

$$\psi^\star \in \underset{\psi}{\arg\min} \sum_{b\in\mathcal{B}} \mathcal{L}(\theta_b^\star, \psi). \tag{4}$$

This approximation avoids repeated model training, while still producing a single high-quality checkpoint that can be deployed across the full set of target budgets (Section 4.1).

3.2 Parametrization of sparsity configurations

To solve the optimization defined in Equation (3), we must first define the set of all sparsity configurations to choose from. We consider two distinct sparsification methods (Figure 3):

Input sparsity: Applied to the FFNs, attention layers, and the LM head (Liu et al., 2025). Given a matrix-vector product $xW^\top$ and a sparsity level, k, this method selectively loads only the rows $W_i^\top$ corresponding to the k largest magnitude activations in $|x|$.

Neuron sparsity: Applied exclusively to the FFNs (Liu et al., 2023b). A lightweight router predicts the magnitude of each neuron's contribution to the final output, which for SwiGLU FFNs can be formulated as $\left|\mathrm{SiLU}(xG^\intercal)\,(xU^\intercal)\right| \cdot \|D\|$, where G, U, and D are the gate, up, and down projections, and $\|\cdot\|$ denotes the column-wise norm (i.e., one norm per neuron). Then, given a sparsity level k, only the neurons corresponding to the k largest predicted contributions are retained. As a result, only the corresponding columns of the gate and up projections, and the rows of the down projection, are loaded from memory. Importantly, the memory footprint of the router itself consumes a portion of the total budget b. Therefore, allocating a larger router trades available parameter budget for routing

[2]We define as layer every feed-forward network (FFN), attention layer and LM head of the architecture.

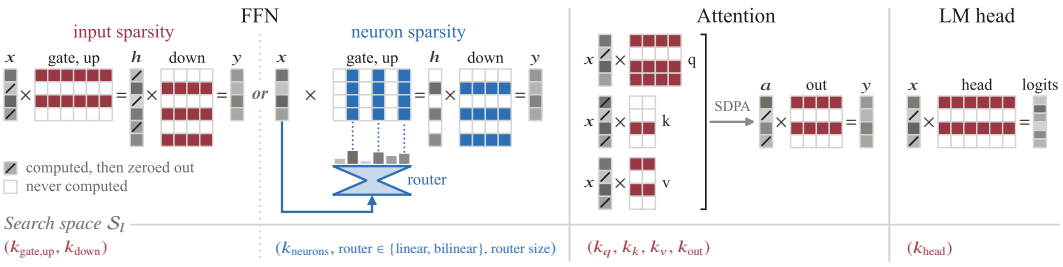


Figure 3: **Search space for sparsity configuration.** To sparsify FFNs, Elastic Decoding can choose between input sparsity and neuron sparsity, where the latter trains a router to predict the magnitude of each neuron contribution. For attention and the LM-head, Elastic Decoding uses input sparsity.

precision, and vice versa. To provide flexibility in this trade-off, we include two types of routers in our search space: a linear router (Alizadeh et al., 2024), and a bilinear router (details in Appendix C, ablation in Figure A1). The bilinear router is motivated by the gated FFN architecture: because the target neuron magnitude depends on the product of two projections of x (the gate and the up projection), a bilinear operation provides a natural structural match for predicting these interactions. Both routers are available at various low-rank bottleneck dimensions.

A single sparsity configuration dictates the number of active elements k_l for each layer l of the network, and, for every FFN, it also specifies the binary choice between input and neuron sparsity, alongside the type of router (linear or bilinear) and its rank. Consequently, our full search space (Figure 3) encompasses *all* possible combinations of the methods and variables (Table C1). We maintain this fine-grained, per-layer flexibility because we empirically observe that layers have distinct sensitivities to sparsification, allowing some to be sparsified much more aggressively than others (Figure 2b, Figure A2).

3.3 Stage 1: Sparsity optimization

Given the set of sparsity configurations, the first stage solves Equation (3) to identify the optimal sparsity configurations for a given budget b, while keeping the pretrained model weights ψ_0 fixed. As our loss function, we use the forward Kullback-Leibler (KL) divergence computed over a reference dataset of T tokens. We measure the divergence between the next-token logits $p_{\text{o}}(x)$ of the original model and the logits $p(x; \theta, \psi_0)$ of the sparsified model:

$$\mathcal{L}(\theta_b, \psi_0) = \frac{1}{T} \sum_{t=1}^{T} \mathrm{D}_{\mathrm{KL}}\big(p_{\text{o}}(x_t) \,||\, p(x_t; \theta_b, \psi_0)\big). \tag{5}$$

Solving Equation (3) exactly is computationally intractable. The sparsity configuration θ contains hundreds of discrete choices and a single evaluation requires a full pass over the data. We introduce two approximations for tractability. First, we search over a discrete grid by defining a finite set of configurations, $\mathcal{S}_l$, for each layer $\theta(l)$. Second, instead of evaluating the loss jointly across all components, we approximate it as the sum of independent, layer-wise losses. In other words,

$$\mathcal{L}(\theta_b, \psi_0) \sim \sum_{l} \mathcal{L}(\theta_b(l), \psi_0)$$

where the loss $\mathcal{L}(\theta_b, \psi_0)$ is approximated by the sum of layer-wise losses $\mathcal{L}(\theta_b(l), \psi_0)$, each computed by applying the sparsity configuration $\theta_b(l)$, which sparsifies only layer l, leaving the other layers untouched. We analyzed this assumption empirically and found that sparsifying layers one at a time and summing their losses strongly correlates with the true loss of sparsifying all layers jointly (Figure 2c). These approximations transform Equation (3) into a multiple-choice knapsack problem

$$\hat{\theta}_b \in \arg\min_{\theta_b} \sum_{l=1}^{L} \mathcal{L}(\theta_b(l), \psi_0), \quad \text{s.t.} \sum_{l=1}^{L} N_l(\theta_b) \le b, \quad \theta_b(l) \in \mathcal{S}_l, \quad \forall b \in \mathcal{B}, \tag{6}$$

where the items (i.e., sparsity variables) are partitioned into L classes (i.e., the number of layers) and exactly one item is chosen for each class. To solve this, we first pre-compute $\mathcal{L}(\theta(l), \psi_0)$ for every

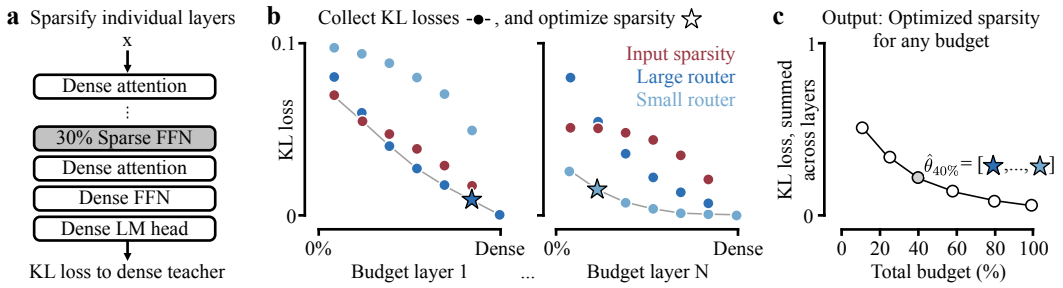


Figure 4: **Sparsity optimization with linear programming. (a)** We first sparsify individual layers and measure the resulting KL divergence from the original model, for every layer and for every possible sparsity hyperparameter $\theta \in \Theta$. **(b)** Given KL losses of all sparsity configurations (dots), we then use linear programming to optimize the sparsity pattern given a budget (stars). Pareto front highlighted as gray line. **(c)** Linear programming optimizes the sparsity in seconds, which enables us to obtain optimized sparsity configurations $\hat{\theta}_b$ at any budget b.

layer and sparsity configuration $\theta(l) \in \mathcal{S}_l$ (Figure 4a), which is highly parallelizable. We then solve the knapsack problem using the HiGHS solver for mixed-integer linear programming (Figure 4b) (Huangfu & Hall, 2018), which we found scales well to models with many layers and configurations. The solver outputs a heterogeneous sparsity allocation that optimally mixes different sparsification levels and methods across the architecture, substantially outperforming naive uniform allocations (Figure A3). As running the solver only takes seconds, we efficiently obtain the configuration for all budgets (Figure 4c).

3.4 Stage 2: Finetuning with distillation training

In the first stage, we identified the sparsity configurations $\{\hat{\theta}_b\}_{b\in\mathcal{B}}$ for each target budget. Applying these configurations directly to the pretrained weights yields strong zero-shot performance at moderate sparsity levels (e.g., retaining $> 50\%$ active parameters). However, aggressive sparsification ($< 50\%$ active) naturally degrades model capacity. To recover this performance gap and yield a highly capable model across the entire efficiency frontier, we further finetune the model's weights ψ.

During training, we enforce token-level budget transitions. For each answer token t, we apply the configuration $\hat{\theta}_{b_t}$ to all answer tokens, where the target budget is sampled uniformly and independently $b_t \sim \mathcal{U}(\mathcal{B})$. We define the search space as $\mathcal{B} = [\frac{1}{1}, \frac{1}{1.5}, \frac{1}{2}, ..., \frac{1}{M}]$, with M between 4 and 6 depending on the model, ensuring robust optimization in highly constrained regimes. This token-wise sampling exposes the model to a heterogeneous KV-cache, encouraging the attention mechanism to remain robust when budgets shift during deployment. We optimize ψ via gradient descent using the KL divergence over the T answer tokens in the sequence:

$$\mathcal{L}(\psi) = \frac{1}{T}\sum_{t=1}^{T} \mathrm{D}_{\mathrm{KL}}\big(p_{\mathrm{o}}(x_t) \,||\, q(x_t; \hat{\theta}_{b_t}, \psi)\big). \tag{7}$$

By jointly optimizing across budgets, the shared weights retain strong representations, outperforming models distilled independently at fixed, low budgets (Figure A4). To ensure a lightweight fine-tuning stage, we optimize the shared weights using Low-Rank Adaptation (LoRA). Additional details on how we train the resulting model are provided in Appendix C.

Despite the approximations introduced by decoupling the joint optimization problem and discretizing the search space, we demonstrated empirically that distillation robustly recovers model capacity (Figure A5), yielding performant models capable of traversing the quality–speed frontier (Section 4.1).

4 Experiments

We evaluated Elastic Decoding across a diverse suite of models, including dense (Qwen3-14B, and Gemma4-31B) and MoE architectures (Gemma4-26B-4B-it and Qwen3-30B-3B). For each model, Elastic Decoding generates a single checkpoint which is then evaluated at different decoding budgets.

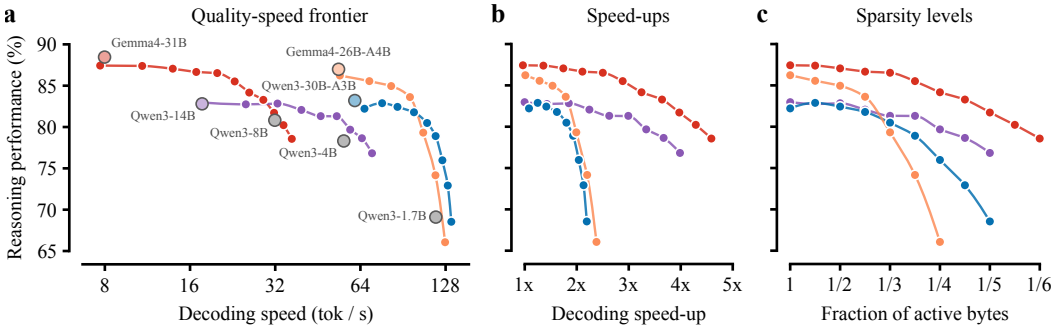


Figure 5: **Elastic Decoding is at the Pareto frontier of speed and accuracy.** Speed of original unsparsified models was measured with `llama.cpp`, speed of Elastic Decoding via our custom inference engine, all on Apple Silicon M4 Max in bfloat16 at batch size one. **(a)** Decoding speed and reasoning accuracy achieved by Elastic Decoding. Lines are Elastic Decoding models, evaluated at different budgets, colored large circles are the original models, gray circles are additional stock models used only for comparison. **(b)** Wall-clock acceleration of Elastic Decoding relative to the corresponding original model. **(c)** Reasoning accuracy versus active memory footprint, measured as the fraction of bytes read per token compared to the original model.

Metrics. To measure how well Elastic Decoding maintains reasoning performance, we measured accuracy with thinking enabled on nine reasoning benchmark tasks, spanning math (GSM8k, Minerva Math) (Cobbe et al., 2021; Hendrycks et al., 2021b), coding (HumanEval+, MBPP+) (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023a), and scientific knowledge (ARC-challenge, MMLU-pro, MMLU-COT, GPQA, GPQA-diamond) (Clark et al., 2018; Wang et al., 2024; Hendrycks et al., 2021a; Rein et al., 2023).

Datasets & Implementation Details. We optimized both sparsity configurations and shared weights on a reference dataset, generated by running the original models on prompts sourced primarily from Nemotron Post-Training splits (Appendix D). Since practical speed-ups from contextual sparsification are achieved only in the memory-bound regime, we processed the compute-bound prefill phase densely across all tasks and applied Elastic Decoding exclusively during generation. We then benchmarked Elastic Decoding using custom Metal kernels on an Apple Silicon M4 Max in bfloat16 at batch size one. To isolate decoding speed, our primary hardware measurements evaluate generation with an empty context. Furthermore, to ensure fair comparisons, we verified that our engine roughly matches `llama.cpp` speed on the dense baselines (Appendix E). We provide extended evaluations demonstrating performance at 8k context window in Figure A6.

4.1 Pareto-frontier of quality and speed

Elastic Decoding establishes a strictly superior quality–speed Pareto frontier compared to smaller-scale models distilled or pre-trained from scratch (Figure 5a, performance in science/math/coding in Figure A7). Despite requiring higher total VRAM, large models sparsified with Elastic Decoding dominate the smaller models in decoding speed. For example, the sparsified Qwen3-14B trajectory lies entirely above Qwen3-8B and Qwen3-4B. When calibrated to match the exact accuracy of the 8B variant, the 14B model requires $47\,\%$ fewer active bytes per token (Figure A8) and yields a $1.64\times$ end-to-end speed-up. These results are not distillation artifacts: fine-tuning the original 8B and 4B models on our reference data and distillation loss slightly degrades their accuracy (Figure A9), implying that Elastic Decoding could achieve even higher performance if given access to the original post-training data.

In comparison to the dense original models, Elastic Decoding achieves large speed-ups and sparsities at small losses in performance (Figure 5b,c). Sparsifying Gemma-4-31B to $33\,\%$ of parameters reduces reasoning performance by only $0.9\,\%$, but yields a speed-up of $2.5\times$ over the original dense model in `llama.cpp`. At aggressive sparsification to $16.6\,\%$ of active parameters, Elastic Decoding achieves a speed-up of $4.6\times$, with a modest $8.8\,\%$ reduction in reasoning performance. Native MoE architectures (e.g., Gemma-4-26B-A4B), on the other hand, exhibit narrower relative gains. Because MoEs operate on a smaller subset of active parameters, their baseline matrix multiplications are significantly faster, thus, the overhead induced by sparsification has a stronger impact on decoding

speed. Nonetheless, our sparsification method combined with our kernels still extracts significant gains, delivering a $2.0\times$ speed-up for the Gemma-4 MoE when sparsified to $33\,\%$ of active parameters. As reasoning length can increase at high sparsities, we further report time-to-answer results, which for Qwen3-14B still yield a $3.16\times$ end-to-end speed-up at $20\,\%$ of active parameters (Table A1).

Overall, Elastic Decoding pushes post-hoc sparsification to new extremes while maintaining strong reasoning performance. At these extremes, MoE architectures exhibit steeper accuracy degradation than dense models. Because MoEs inherently activate fewer parameters per token at baseline, they exhibit less representational redundancy to exploit during post-hoc sparsification. Furthermore, the model supports dynamic, token-by-token, budget switching at decode-time without degrading reasoning quality, incurring negligible switching overhead at rates up to 2 switches per second (Appendix B).

Takeaway. A single Elastic Decoding checkpoint effectively operates as an entire family of models, dominating the optimal frontier while adapting to real-time on-device constraints.

4.2 Comparison to baselines

We compared our method to state-of-the-art contextual sparsity approaches: the training-free method TEAL (Liu et al., 2025) and training-based ReLUfication (Mirzadeh et al., 2024). We omit unstructured sparsity methods (e.g., SparseGPT) as they yield marginal on-device speed-ups, and CATS (Lee et al., 2024) as it is strictly outperformed by TEAL (Liu et al., 2025).

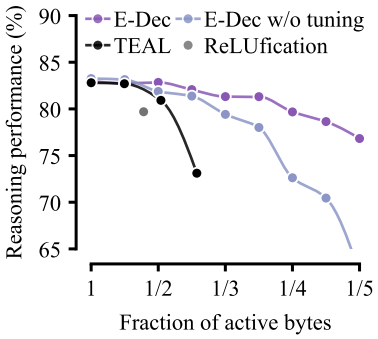


Figure 6: **Elastic Decoding outperforms baselines.** Reasoning performance of Qwen3-14B over fraction of active bytes of Elastic Decoding (E-Dec), Elastic Decoding before fine-tuning (E-Dec w/o tuning), TEAL, and ReLUfication.

Elastic Decoding achieves significantly higher reasoning performance while using fewer bytes per token than both baselines (Figure 6, other models in Figure A10 and Table A2). For example, for Qwen3-14B, our method strictly outperforms TEAL when using a budget of $33\,\%$ of bytes of the original model, and maintains strong performance at up to $25\,\%$, where TEAL completely degenerates. Furthermore, our method also vastly outperforms ReLUfication. The latter computes densely the FFN gate projection (to determine routing), attention, and the LM-head, it achieves only $45\,\%$ model-wide sparsity, and it cannot control the sparsity level. In addition, when training on our dataset of only 5B tokens, ReLUfication does not recover the performance of the original model.

Finally, we also evaluated Elastic Decoding after the sparsity optimization (stage 1), prior to any distillation finetuning. We found that our method still outperformed TEAL, also training-free, demonstrating that our heterogeneous knapsack allocator alone serves as a highly capable training-free sparsifier.

Takeaway. Elastic Decoding achieves state-of-the-art reasoning performance at all sparsity levels, significantly outperforming existing contextual sparsification methods.

4.3 Analysis of sparsified models

To understand how Elastic Decoding achieves such large sparsification rates, we analyzed the sparsity configurations selected by the multiple-choice knapsack solver (Figure 7, Figure A11). We found that the optimal parameter allocation is highly heterogeneous across layers. For example, in Qwen3-14B, the solver aggressively sparsifies early FFNs while preserving capacity in the middle layers, allocating $15\times$ more active parameters to Layer 14 than to Layer 2. This stark heterogeneity highlights a primary advantage of our sparsity optimization over baselines like TEAL, which enforce a rigid, uniform sparsity level across all layers.

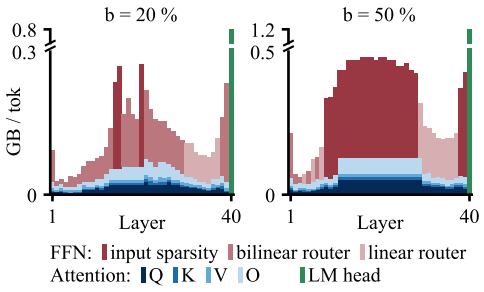


Figure 7: **Allocation analysis.** Sparsity configurations selected by the multiple-choice knapsack solver for Qwen3-14B, for a byte budget b of $20\,\%$ and $50\,\%$ of the original model.

In addition, the *type* of sparsity selected was also heterogeneous. The knapsack often selected different sparsification methods for different layers, even within a single model and at a certain budget. We observed that, at low byte budgets (i.e., high sparsity), the knapsack tends to select neuron sparsity, whereas it selects input sparsity at high budgets. At low byte budgets, the choice of neuron sparsity requires the router itself to consume roughly $25\,\%$ of the active FFN bytes budget on average. Because our knapsack formulation explicitly penalizes this memory footprint, the solver's preference shows that paying a heavy "router tax" for precise neuron selection yields a better global loss trade-off and suggests that neuron sparsity is important to achieve high sparsification rates.

Takeaway. Achieving high-quality contextual sparsity requires a global optimization solver rather than naive, layer-wise greedy heuristics.

4.4 Compatibility with quantization

Since Elastic Decoding selectively bypasses parameter loads, it should be orthogonal to weight quantization techniques. While quantization reduces the precision of individual weights, contextual sparsity decreases the total volume of parameters fetched per token, enabling multiplicative reductions in memory bandwidth requirements. We verified this hypothesis empirically by first applying Elastic Decoding to Qwen3-14B and then quantizing the resulting model weights to 8-bit (RTN) and 4-bit (AWQ) (Figure 8). We found that contextual sparsity combines effectively with both low-precision schemes: quantizing to 8-bit has negligible effects on reasoning performance, whereas quantizing to 4-bit leads to a constant decrease of $1.5\,\%$ reasoning performance across sparsities.

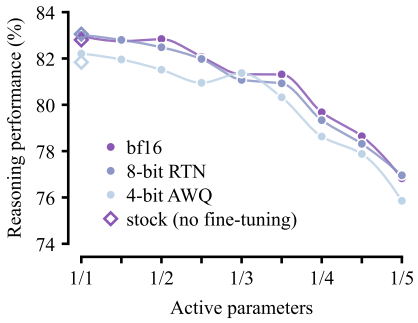


Figure 8: **Elastic Decoding is compatible with quantization.** Reasoning performance of Elastic Decoding for Qwen3-14B, when model weights are quantized to 8 bit (RTN) or 4 bit (AWQ) vs bfloat16.

5 Conclusion

We introduced Elastic Decoding, a post-hoc contextual sparsification method that enables a single LLM checkpoint to traverse the quality–speed Pareto frontier. By framing sparsification as a joint constrained optimization over sparsity configurations and weights via a KL divergence distillation loss, we obtain a heterogeneous, architecture-specific sparsity allocation. The resulting models achieve up to $83\,\%$ sparsity and $4.6\times$ decoding speed-ups on Apple Silicon (e.g., Gemma4-31B), outperforming smaller dense models in both speed and reasoning quality. Crucially, Elastic Decoding accelerates both dense and MoE architectures and enables token-by-token budget switching with negligible hardware latency. Ultimately, Elastic Decoding challenges the paradigm of training separate models for different compute tiers. While smaller models still hold an advantage in static memory footprint, a single elastic checkpoint now dominates the frontier for dynamic, on-device inference.

Limitations and future works. Our two-stage optimization is an approximation that requires offline router training and up to 5B distillation tokens, still significantly less compute than training models from scratch or performing ReLUfication. While the Stage 1 configuration alone already serves as a strong training-free sparsifier, future work exploring true joint sparsity optimization could bridge the gap between our decoupled approach and the theoretical global minimum. The distillation phase also requires alignment between the reference data and the model original post-training distribution. On the hardware side, achieving negligible budget switching overhead currently requires trading resident RAM capacity for speed by keeping routers and dual weight layouts in memory. Future optimizations can eliminate this overhead by enforcing consistent sparsity types within the solver or by implementing predictive disk prefetching. Additionally, sparse memory gathers currently prevent acceleration during the compute-bound prefill phase: adopting block-structured sparsity could resolve this bottleneck. Finally, promising directions for future research include extending evaluations to agentic and multimodal tasks and developing an internal budget predictor. This would enable the model to autonomously modulate its active parameters based on sequence difficulty, instead of relying on external signals, further lowering the average bytes read per token.

AI USE STATEMENT

In this work, we used generative AI tools for research execution, specifically, for the implementation of well-defined mechanical concepts, optimizing on-device kernels, and automating workflow scripts to monitor and relaunch failed jobs. We have not used generative AI tools for research ideation, generating core ideas, or formulating hypotheses. Additionally, we used AI-assisted search for retrieval and discovery to efficiently navigate related work. Other required disclosure tasks are not applicable to this work.

Furthermore, we used generative AI tools iteratively to aid and polish the writing of this manuscript on a sentence-by-sentence or paragraph-by-paragraph basis. We have reviewed all AI-assisted work. To ensure quality in execution, all AI-generated code and optimizations were validated for correctness through an extensive test suite, and performance improvements were verified using rigorous benchmarks. Regarding the text, AI was strictly used for targeted polishing rather than wholesale generation; all suggestions were actively evaluated and integrated by the authors to ensure scientific accuracy and original intent were strictly preserved. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work introduces a method to accelerate local LLM inference. By enabling practical on-device deployment of large models, our approach supports data privacy by allowing users to process sensitive information locally rather than via cloud APIs. Environmentally, our post-hoc sparsification requires significantly less compute (under 5B tokens) than training sparse models from scratch. The primary ethical consideration of our specific method is its potential impact on model fairness: as with other compression techniques, aggressive sparsification can disproportionately degrade performance on tail knowledge, rare dialects, or underrepresented languages. Because Elastic Decoding inherits the baseline biases of the underlying models (e.g., Qwen3, Gemma4), we recommend that practitioners explicitly evaluate safety and fairness at the specific sparsity budgets they intend to deploy.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we have fully documented the mathematical formulation of Elastic Decoding and the multiple-choice knapsack optimization algorithm in Section 3 and Table C1. Complete experimental details (including distillation hyperparameters, routing configurations, calibration datasets, and exact hardware benchmarking conditions) are provided in Section C and Table D1. The source code will be released separately. We provide comprehensive algorithmic descriptions to enable independent implementation. Additionally, to substantiate our hardware efficiency claims, we include a supplementary video demonstrating the custom engine performing real-time, token-by-token budget switching on-device.

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 *ACL*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Akhiad Bercovich et al. Llama-nemotron: Efficient reasoning models, 2025.

Ruisi Cai, Saurav Muralidharan, Greg Heinrich, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. Flextron: Many-in-one flexible large language model. In *ICML*, 2024.

Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. ZeroQ: A novel zero shot quantization framework. In *CVPR*, 2020.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

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:1803.05457v1*, 2018.

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

Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models. In *ACL*, pp. 1280–1297, 2024.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*, 23(120):1–39, 2022.

Elias Frantar and Dan Alistarh. SPDY: Accurate pruning with speedup guarantees. In *ICML*, 2022.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Xu Owen He. Mixture of a million experts. *arXiv preprint arXiv:2407.04153*, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *ICLR*, 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021b.

QHJAJ Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical programming computation*, 10(1):119–142, 2018.

Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In *ICML*, 2021.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, and Prateek Jain. Matformer: Nested transformer for elastic inference. *Advances in Neural Information Processing Systems*, 37:140535–140564, 2024.

Donghyun Lee, Jaeyong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. CATS: Context-aware thresholding for sparsity in large language models. In *CoLM*, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *ICML*, 2022.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *ICML*, 2023.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device llm compression and acceleration. *MLSys*, 2024.

James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models. In *ICLR*, 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NeurIPS*, 2023a.

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 *ICML*, pp. 22137–22176. PMLR, 2023b.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-Pruner: On the structural pruning of large language models. *NeurIPS*, 36:21702–21720, 2023.

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 *ICLR*, volume 2024, pp. 25378–25400, 2024.

Dhruv Nathawani, Shuoyang Ding, Vitaly Lavrukhin, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Boris Ginsburg, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v2, 2025a. URL `https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v2`.

Dhruv Nathawani, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Ameya Sunil Mahabaleshwarkar, , Jian Zhang, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v1, 2025b. URL `https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v1`.

NVIDIA. Nemo gym: An open source library for scaling reinforcement learning environments for llm. `https://github.com/NVIDIA-NeMo/Gym`, 2025a. GitHub repository.

NVIDIA. Nvidia nemotron nano 2: An accurate and efficient hybrid mamba-transformer reasoning model, 2025b.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Oliver Sieberling, Denis Kuznedelev, Eldar Kurtic, and Dan Alistarh. Evopress: Accurate dynamic model compression via evolutionary search. In *ICML*, 2025.

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 *COLING*, 2025.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

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*, 2024.

Ali Taghibakhshi, Ruisi Cai, Saurav Muralidharan, Sharath Turuvekere Sreenivas, Ameya Sunil Mahabaleshwarkar, Marcin Chochowski, Akhiad Bercovich, Ran Zilberstein, Ran El-Yaniv, Yonatan Geifman, Daniel Korzekwa, Yoshi Suhara, Oluwatobi Olabiyi, Ashwath Aithal, Nima Tajbakhsh, and Pavlo Molchanov. Star elastic: Many-in-one reasoning LLMs with efficient budget control. In *ICML*, 2026.

Nurbek Tastan, Stefanos Laskaridis, Karthik Nandakumar, and Samuel Horváth. MoSE: Mixture of slimmable experts for efficient and adaptive language models. In *Forty-third International Conference on Machine Learning*, 2026.

Kimi Team. Kimi k2.5: Visual agentic intelligence, 2026.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024.

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.

Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. MoEfication: Transformer feed-forward layers are mixtures of experts. In *ACL*, 2022.

Zhenyu Zhang, Zechun Liu, Yuandong Tian, Harshit Khaitan, Zhangyang Wang, and Steven Li. R-sparse: Rank-aware activation sparsity for efficient llm inference. In *ICLR*, 2025.

A EXTENDED RESULTS AND ABLATIONS

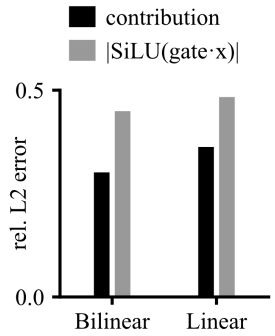


Figure A1: **Router architecture and supervision ablation.** We evaluated the impact of router architecture and training signals on the FFN. In SwiGLU-based networks, we compared routers trained to predict the post-SiLU gate activation (Lee et al., 2024) against those trained to predict the full neuron *contribution* (defined in Section 3.2, incorporating the gate, up, and down projections). Figure shows the relative L_2 reconstruction error of the local FFN output across different router configurations. We compare two routing architectures (linear vs. bilinear) and two training signals. Black bars represent routers supervised by the full neuron *contribution* (defined in Section 3.2, which incorporates the gate, up, and down projections), while gray bars represent routers trained to predict only the magnitude of the post-SiLU gate activation. The results demonstrate that the contribution-based metric provides a strictly superior training signal, and that the more expressive bilinear routers (detailed in Section C.1) consistently outperform standard linear routers across both targets.

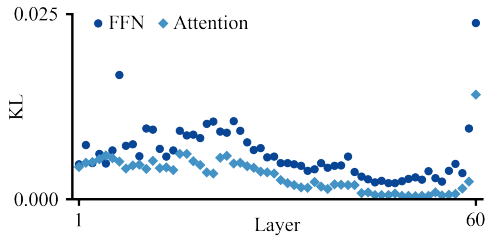


Figure A2: **Layer-wise sensitivity to sparsification.** KL divergence from the dense Gemma4-31B model when applying 50% input sparsity to either the FFN or the attention projections of a single layer. The stark variance in degradation across different depths justifies framing Elastic Decoding as a heterogeneous knapsack problem rather than enforcing uniform sparsity.

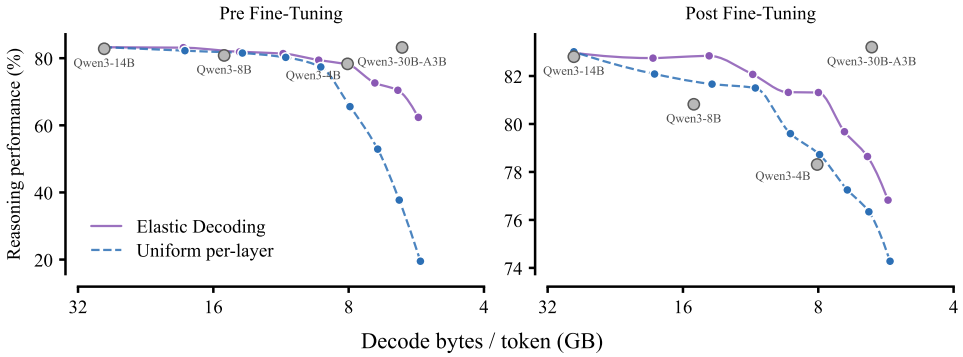


Figure A3: **Heterogeneous vs. uniform budget allocation.** Comparison of our globally optimized Elastic Decoding allocation (Qwen3-14B) against a uniformly budgeted baseline. In the uniform baseline, the total byte-read budget is divided equally across all transformer layers and the LM head. While the configuration *within* each layer is still optimally resolved, the budget cannot be traded off across different depths of the network. *Left:* Performance at initialization (prior to weight distillation). The globally optimized heterogeneous allocation substantially outperforms the uniform baseline out-of-the-box. *Right:* Performance after end-to-end distillation. Although training recovers capacity for both models, the heterogeneous checkpoint maintains a strict Pareto advantage over the uniform baseline across the entire budget frontier. This demonstrates that our global multiple-choice knapsack problem formulation, which allows the network to dynamically distribute parameters across its depth, is critical for maximizing model quality.

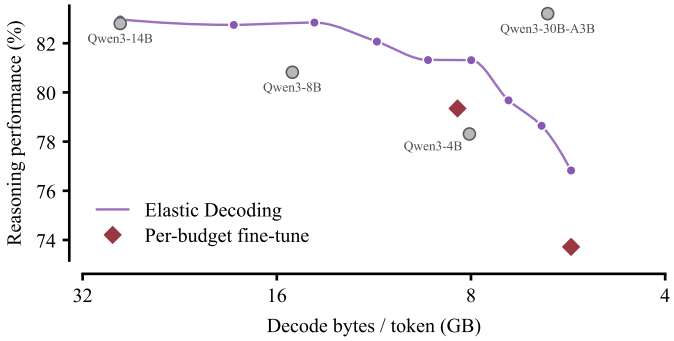


Figure A4: **Shared vs. independent weight distillation.** Comparison of Elastic Decoding (which optimizes a single shared LoRA across all budgets) against independently trained models (per-budget LoRAs) at specific operating points. Elastic Decoding consistently outperforms the independent baselines, requiring far less resources. Because our method dynamically samples high-capacity and dense budgets during distillation, the shared weights retain strong representations for processing the dense prompt (prefill) phase during inference. In contrast, training an independent model strictly at a low budget permanently degrades its ability to process the prompt effectively.

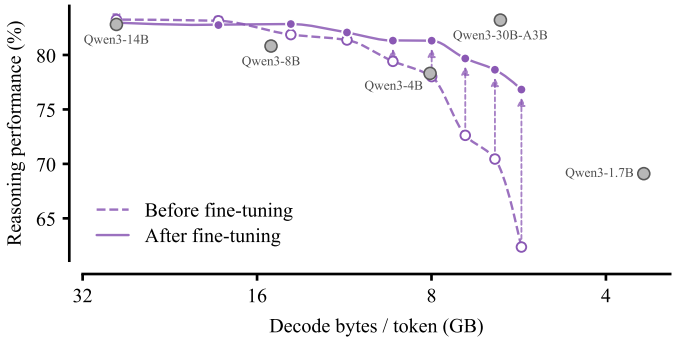


Figure A5: **Performance recovery via distillation.** Trajectory of the Qwen3-14B operating points from initialization (zero-shot sparsity) to their final state after end-to-end distillation. While aggressive sparsification naturally degrades the model's initial capacity at lower budgets, the distillation phase effectively recovers this performance gap, shifting the points upward to form a highly capable and continuous quality-efficiency frontier.

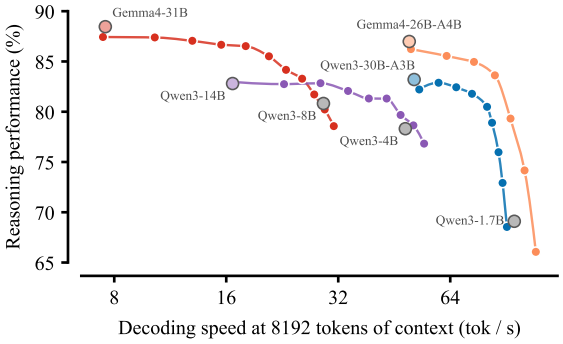


Figure A6: **Pareto frontier at longer context lengths.** Frontier of reasoning performance and decoding speed, measured at 8192 context length. Even under higher KV-cache memory pressure, Elastic Decoding maintains its strict Pareto advantage over the original models.

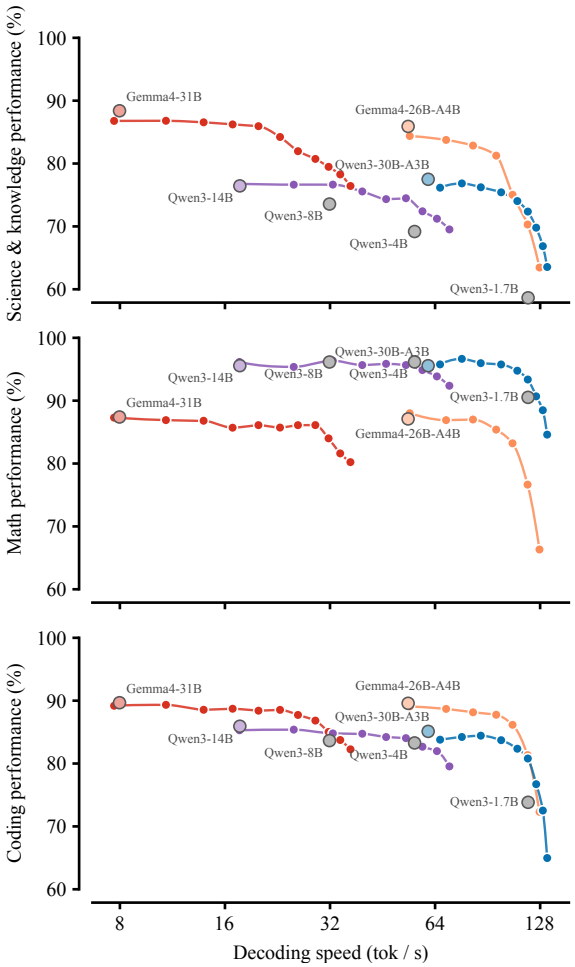


Figure A7: **Elastic Decoding performance across task categories.** Frontier of decoding speed and reasoning performance disaggregated by domain: Science and knowledge (ARC-challenge, MMLU-pro, MMLU-COT, GPQA, GPQA-diamond), Math (GSM8k, Minerva Math), and Coding (HumanEval+, MBPP+). Elastic Decoding delivers consistent Pareto improvements across all evaluated domains, demonstrating that the sparsification does not disproportionately degrade specific reasoning capabilities.

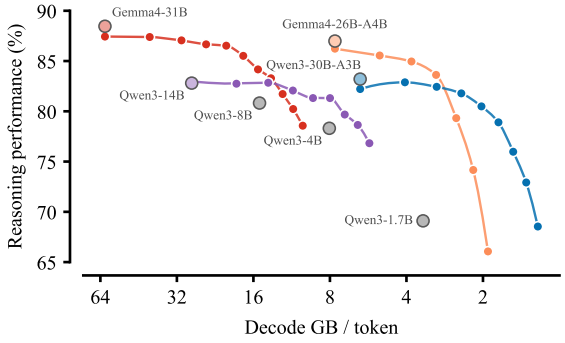


Figure A8: **Absolute memory bandwidth efficiency.** Reasoning performance is plotted against the absolute number of bytes read per token (in GB). Heavily sparsified large models (e.g., Qwen3-14B) consistently outperform inherently smaller dense models (e.g., Qwen3-8B and Qwen3-4B), requiring fewer active bytes to be read to achieve their quality level.

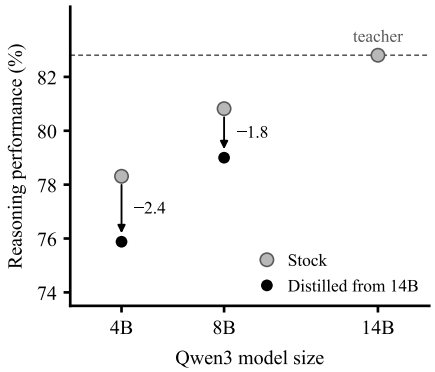


Figure A9: **Reference data does not artificially inflate performance.** Reasoning performance of original Qwen3 14B, 8B, and 4B models (grey), compared to the 8B and 4B models after fine-tuning on our 14B-generated reference dataset using the exact KD loss applied in Elastic Decoding (black). Applying this distillation recipe degrades the dense 4B and 8B models by 2.4 and 1.8 points, respectively. The inability of the 14B reference data to improve these smaller dense baselines demonstrates that the high quality retained by our sparsified 14B model is not a byproduct of data memorization or evaluation overfitting.

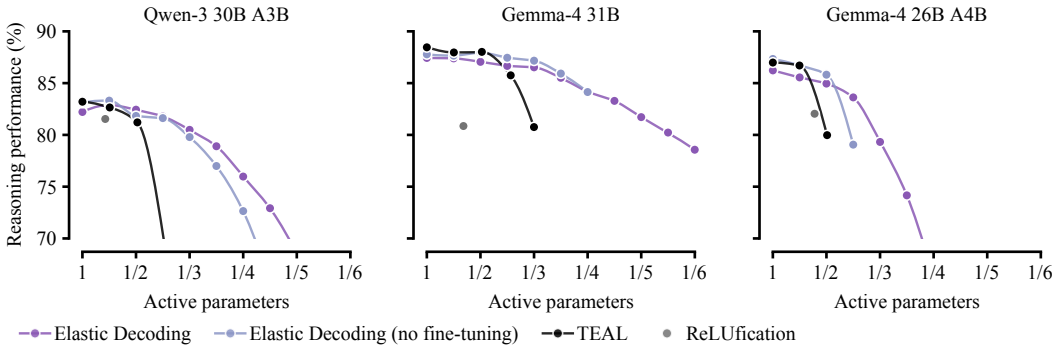


Figure A10: **Elastic Decoding outperforms baselines across models.** Reasoning performance over fraction of bytes used for Elastic Decoding, Elastic Decoding before fine-tuning, TEAL, and ReLUfication.

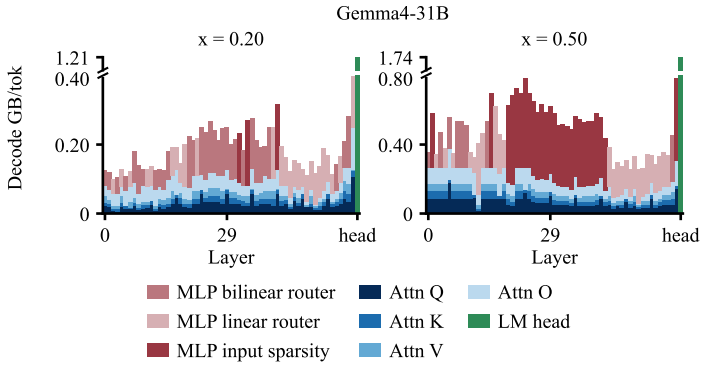


Figure A11: **Allocation analysis for Gemma4-31B.** Sparsity configurations selected by the multiple-choice knapsack solver. Similar to Qwen3-14B (see Figure 7), the optimal allocation is highly heterogeneous across network depth, highlighting the necessity of our global solver over uniform layer-wise heuristics.

Table A1: **Output length and time-to-answer.** Results for Elastic Decoding on Qwen3-14B at each decode budget (measured in active GB read per token). Crucially, the mean output represents the average answer length, across our entire reasoning benchmark suite. Speed-ups are relative to the dense operating point (28.0 GB/tok). While the model produces longer reasoning traces at tighter budgets (below 11.2 GB/tok) to compensate for mild reasoning degradation, the increased decoding speed easily offsets the overhead of these extra tokens, yielding a strictly faster end-to-end time-to-answer.

					Speed-up	
GB/tok	Active bytes (%)	Mean output (tok)	Speed (tok/s)	Time to answer (s)	tok/s	Time
28.0	100	2,020	17.6	114.7	1.00×	1.00×
18.7	67	2,010	25.2	79.8	1.43×	1.44×
14.0	50	2,040	32.6	62.6	1.85×	1.83×
11.2	40	2,059	39.6	52.0	2.25×	2.21×
9.3	33	2,120	46.3	45.8	2.63×	2.51×
8.0	29	2,168	52.8	41.1	3.00×	2.79×
7.0	25	2,311	58.9	39.2	3.34×	2.92×
6.2	22	2,383	64.8	36.8	3.68×	3.12×
5.6	20	2,543	70.3	36.2	3.99×	3.17×

Table A2: Reasoning performance over fraction of bytes used for Elastic Decoding, Elastic Decoding before fine-tuning, TEAL, and ReLUfication. Dashes indicate that the model was degenerate. Percentage after ReLUfy indicates its actual model-wide sparsity.

	Sparsity (%)							
Method	0	33.3	50	60	66.7	75	80	83.3
Qwen3-14B								
Elastic decoding	83.0	82.7	**82.8**	**82.1**	**81.3**	**79.7**	**76.8**	–
Elastic decoding (w/o tuning)	**83.2**	**83.1**	81.9	81.4	79.4	72.6	62.4	–
TEAL	82.8	82.7	80.9	73.1	–	–	–	–
ReLUfy (43.9%)	–	–	79.7	–	–	–	–	–
Qwen3-30B-A3B								
Elastic decoding	82.1	82.9	**82.4**	**81.8**	**80.5**	**76.0**	**68.5**	–
Elastic decoding (w/o tuning)	83.1	**83.3**	81.8	81.6	79.8	72.6	52.6	–
TEAL	**83.2**	82.6	81.2	69.3	–	–	–	–
ReLUfy (29.9%)	–	81.5	–	–	–	–	–	–
Gemma4-31B								
Elastic decoding	87.4	87.4	87.1	86.7	86.5	**84.2**	**81.7**	**78.6**
Elastic decoding (w/o tuning)	87.8	87.7	88.0	**87.5**	**87.2**	84.1	–	–
TEAL	**88.5**	**88.0**	**88.0**	85.8	80.8	–	–	–
ReLUfy (40.6%)	–	–	80.9	–	–	–	–	–
Gemma4-26B-A4B								
Elastic decoding	86.2	85.6	85.0	**83.6**	**79.3**	**66.1**	–	–
Elastic decoding (w/o tuning)	**87.3**	**86.7**	**85.8**	79.1	45.4	12.1	–	–
TEAL	87.0	86.7	80.0	5.1	–	–	–	–
ReLUfy (43.8%)	–	–	82.0	–	–	–	–	–

B CONTROLLABLE DECODING BUDGET

A core capability of Elastic Decoding is the ability to adjust the parameter budget dynamically during generation. To verify that token-by-token switching does not degrade output quality, we evaluated reasoning performance under a stochastic budget (Figure B1). For each generated token, we sample an operating point from a discretized Gaussian distribution (Table B1) such that the sequence's average footprint perfectly matches a target static budget.

We find that our checkpoint remains highly robust under this continuous switching regime (triangles), experiencing only a minimal drop in accuracy compared to standard static decoding (circles). This resilience in such an extreme scenario confirms that exposing the model to a heterogeneous KV-cache during training, as described in Section 3.4, successfully preserves generation quality.

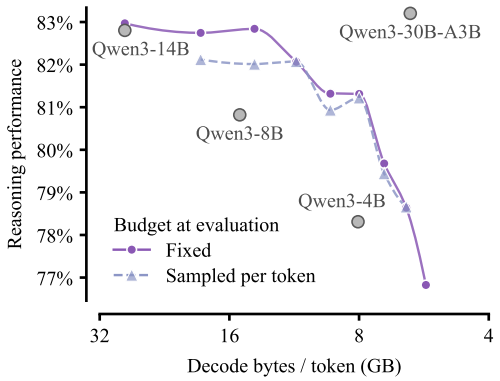


Figure B1: **Robustness to dynamic budget switching.** Reasoning performance of Elastic Decoding under standard static budgets (circles) versus stochastic token-by-token budget sampling (triangles). Despite sampling a new operating point at every generated token, the model remains highly robust, experiencing only a minimal drop in performance compared to the static evaluation.

Beyond reasoning quality, dynamic switching must not compromise speed. We evaluate decoding speed while varying the budget transition frequency (Table B2). To isolate the hardware cost of switching, we enforce an identical mix of operating points across all runs. The runs differ exclusively in how often the budget transitions to an adjacent step, ranging from 2 % to 100 % of the generated tokens. Because every run performs the exact same computational work, its speed is directly comparable to a static decoding baseline averaged over this identical budget mix.

Table B1: **Per-token budget sampling distributions.** The percentage of generated tokens allocated to each operating point during stochastic evaluation. For each target budget, the sampling probabilities form a discretized Gaussian distribution (with a standard deviation of one operating step) centered such that the expected memory footprint exactly matches the target. Distributions are naturally truncated at the boundaries. The prefill phase is always processed densely.

	Sampled operating point (fraction of dense bytes)								
Target budget	0.20	0.22	0.25	0.29	0.33	0.40	0.50	0.67	dense
0.22	34.5	**42.6**	19.4	3.2	0.2	—	—	—	—
0.25	7.1	27.8	**39.8**	20.9	4.1	0.3	—	—	—
0.29	0.7	7.2	27.7	**39.5**	20.7	4.0	0.3	—	—
0.33	—	0.7	7.5	28.3	**39.3**	20.1	3.8	0.3	—
0.40	—	—	0.8	8.0	29.2	**39.0**	19.2	3.5	0.2
0.50	—	—	—	0.9	8.6	30.1	**38.8**	18.3	3.2
0.67	—	—	—	—	0.7	7.6	29.1	**41.2**	21.4

The switching logic itself incurs no measurable overhead. When both weight layouts are resident, changing the budget at every token perfectly matches static speed. The sole hardware cost comes from the FFN. Input and neuron sparsity require orthogonal memory layouts (column-wise vs. row-wise), and the sparsity optimization (Section 3.3) may assign a layer different sparsity types at different budgets. If only one layout is kept resident, changing a layer's sparsity type mid-generation requires physically transposing its weights. This generates roughly 2.2 GB of memory traffic and adds 7 ms per switch on average. For realistic use cases with infrequent adjustments (e.g., OS-level thermal throttling), this penalty is negligible: even at ≈ 2 switches per second, speed drops by only 0.5 tok/s. However, this cost scales directly with frequency, reaching an 8 tok/s penalty when the budget changes at every token. For deployments that switch continuously, keeping both weight layouts resident ($+9.3$ GB for Qwen3-14B) removes the transpose entirely. Alternatively, constraining the Stage 1

optimization (Section 3.3) to assign a single sparsity type per layer across all budgets would avoid both the transpose and the extra memory.

Table B2: **Budget switching overhead.** Qwen3-14B decoding speed on an Apple M4 Max, generating 512 tokens. All rows compute an identical mix of budgets and differ only in the frequency of transitions. *Static* decodes each budget separately and averages the per-token time over this identical mix. With dual layouts resident, switching at every token incurs no measurable penalty. Transposing weights on the fly costs a roughly constant 7 ms per switch. This is negligible for realistic use cases with infrequent adjustments but scales to an 8 tok/s penalty when switching continuously.

Decoding regime	Bytes/Tok	Switch freq.	Switches/s	Speed (tok/s)	Δ vs. static
Static	11.4 GB	0 %	0	38.1	—
Dual layout in RAM	11.4 GB	100 %	38.1	38.1	$\sim$0 %
On-the-fly transpose	11.4 GB	2 %	0.74	37.9	$-0.4\,\%$
	11.4 GB	5 %	1.9	37.6	$-1.2\,\%$
	11.4 GB	10 %	3.7	37.1	$-2.6\,\%$
	11.4 GB	25 %	9.0	35.9	$-5.8\,\%$
	11.4 GB	50 %	16.8	33.5	$-12.0\,\%$
	11.4 GB	100 %	30.1	30.1	$-20.9\,\%$

Takeaway. Elastic Decoding supports token-by-token budget switching that preserves generation quality. Holding both weight layouts in memory makes switching entirely free. Otherwise, the transpose penalty scales with frequency but remains negligible for practical, infrequent adjustments.

C Extended Methodology

C.1 Derivation of the Bilinear router

For a SwiGLU FFN with input $x \in \mathbb{R}^{1\times d}$, the true contribution magnitude of the i-th neuron to the output is given by $c_i(x) = \left|\mathrm{SiLU}(xG_i^\top)\,(xU_i^\top)\right| \|D_{:,i}\|_2$, where $G, U \in \mathbb{R}^{M\times d}$ and $D \in \mathbb{R}^{d\times M}$ are the gate, up and down projections. Predicting c requires approximating the bilinear term $(xG^\top) \odot (xU^\top)$, which no linear function of x can represent.

We project x onto a rank-r subspace via $V \in \mathbb{R}^{r\times d}$ ($r \ll d$) and define:

$$z = xV^\top, \qquad s_p = zP^\top \approx \tfrac{1}{2}(xG^\top + xU^\top), \qquad s_q = zQ^\top \approx \tfrac{1}{2}(xG^\top - xU^\top) \tag{8}$$

with $P, Q \in \mathbb{R}^{M\times r}$. By the polarization identity, we reconstruct the bilinear component as $s_p^2 - s_q^2 = (s_p + s_q)(s_p - s_q) \approx (xG^\top) \odot (xU^\top)$. Since $\mathrm{SiLU}(a) = a\,\sigma(a)$, modulating this difference of squares by the logistic sigmoid $\sigma(s_p + s_q) \approx \sigma(xG^\top)$ recovers the SiLU gate, and we apply a per-neuron scale vector $s \in \mathbb{R}^M$ (where $s_i \approx \|D_{:,i}\|_2$) to account for the down projection. We refer to this formulation as the Bilinear router, whose score is:

$$\hat{c}(x) = \left|\sigma(s_p + s_q) \odot (s_p^2 - s_q^2)\right| \odot s. \tag{9}$$

The router selects the k neurons with the largest $\hat{c}_i(x)$. Per token, the routed forward pass thus reads from memory $r(d + 2M) + M$ router parameters plus $3kd$ for the selected neurons, instead of the $3Md$ of dense evaluation.

Initialization. V is set to the top-r right singular vectors of $[G; U] \in \mathbb{R}^{2M\times d}$, $P = \frac{1}{2}(G + U)V^\top$, $Q = \frac{1}{2}(G - U)V^\top$ and $s_i = \|D_{:,i}\|_2$. Then $s_p + s_q = \tilde{x}G^\top$ and $s_p - s_q = \tilde{x}U^\top$ with $\tilde{x} = xV^\top V$, so at initialization $\hat{c}(x) = c(\tilde{x})$ exactly: the router computes the true contribution of the input projected onto the best rank-r subspace of $[G; U]$. At initialization the router makes no modeling error beyond the compression. It computes the true contribution formula, applied to x projected onto the r directions that matter most to the gate and up weights.

Fitting and inference. V, P, Q and s are then trained jointly with Adam to minimize the MSE between $\log(1 + \hat{c})$ and $\log(1 + c)$ (details in Table D1). At inference the router is used only as a top-k selector: the selected neurons are evaluated with their original, unmodified weights and a router gate value of 1, so the approximation decides which neurons are active but never changes their outputs.

C.2 Mixture-of-Experts k-redistribution

In a Mixture-of-Experts layer, the sparsity configuration assigns a single sparsity level k shared by all routed experts. A token t routed to its top-n experts $\mathcal{E}(t)$ therefore activates nk neurons in total.

A uniform k ignores how much each expert contributes to the output. The MoE layer output is the weighted sum $\sum_{e\in\mathcal{E}(t)} w_e\, f_e(x)$, where w_e are the routing weights, so every neuron's contribution is scaled by its expert's weight w_e, and heavily weighted experts deserve a larger share of the budget. We therefore keep the per-token budget of nk neurons and redistribute it across the active experts according to their routing weights:

$$k_e = nk\,\frac{w_e^2}{\sum_{j\in\mathcal{E}(t)} w_j^2}. \tag{10}$$

Each k_e is capped at the expert width M_{expert}, and the budget a capped expert cannot absorb is iteratively redistributed among the remaining experts by the same rule. The allocations are rounded to integers with the largest-remainder method, so that exactly $\sum_{e\in\mathcal{E}(t)} k_e = nk$ neurons are active for every token.

Each expert e then retains its top-k_e neurons. Under neuron sparsity, the selection follows the sparsity router's scores; under input sparsity, k_e replaces k_{down}, while $k_{\mathrm{gate,up}}$ is unchanged. Since the total number of active neurons is unchanged, redistribution leaves the byte-read cost of every budget unchanged and only shifts bandwidth toward the most impactful experts. Because k_e depends only on

the routing weights, which the MoE router computes before any expert runs, each expert still scores and selects its neurons independently. We apply k-redistribution during both Stage 2 distillation (Section 3.4) and inference.

C.3 Distillation Implementation Details

During the distillation phase, the target sparsity configurations $\{\hat{\theta}_b\}_{b\in\mathcal{B}}$ are held fixed. Furthermore, the routers used for FFN neuron sparsity make discrete "zero out" decisions. Because this represents a hard cutoff, these routers do not receive gradients and also remain fixed during training. To enforce sparsity during training, we computed the dense network with zeroed out sparsified values (e.g., unused inputs to attention or unused neurons).

C.4 Sparsity configurations search space

Table C1: **Search space of the sparsity optimization.** The discrete set of sparsity configurations $\mathcal{S}_l$ available to each layer l in Stage 1. Routers of every type and size r are trained offline. Each configuration is graded by the KL divergence from the original model with only its layer sparsified; attention configurations are first screened by the error of the layer's own output. The knapsack solver then selects one configuration per layer for each budget $b \in \mathcal{B}$. M_{dense} and M_{expert} are the FFN hidden widths, and all routed experts of an MoE layer share one configuration. Each k is a multiple of 32, given as a percentage of the dimension it selects from. On Gemma4's full-attention layers keys and values share one projection ($k_v = k_k$), hence the separate row.

	Qwen3		Gemma4	
	14B	30B A3B	31B	26B A4B
Model architecture				
Layers	40	48	60	30
Hidden width d	5120	2048	5376	2816
FFN width M_{dense}	17408	–	21504	2112
Expert width M_{expert}	–	768	–	704
Search grids: min–max (# points)				
FFN, neuron sparsity				
Linear router r	113–4765 (17)	80–768 (8)	8–608 (14)	8–608 (14)
Bilinear router r	64–4096 (20)	64–768 (9)	8–768 (14)	8–768 (14)
k_{neurons}	0.2–100% (22)	4.17–100% (24)	4.76–100% (21)	4.55–100% (22)
FFN, input sparsity				
$k_{\text{gate,up}}$	6.25–100% (16)	6.25–100% (16)	6.55–100% (16)	4.55–100% (22)
k_{down}	6.25–100% (16)	4.17–100% (24)	6.55–100% (16)	4.55–100% (22)
Attention				
k_q	1.25–100% (21)	1.56–100% (18)	1.19–100% (19)	1.14–100% (19)
k_k, k_v, k_{out}	1.5–100% (21)	1.5–100% (19)	1.5–100% (19)	1.5–100% (19)
LM head				
k_{head}	0.625–100% (160)	1.56–100% (64)	1.19–100% (84)	1.14–100% (88)
Configurations search space per layer $\lvert\mathcal{S}_l\rvert$				
FFN	1,070	792	844	1,100
Attention	194,481	123,462	130,321	130,321
Attention ($k_v = k_k$)	–	–	6,859	6,859
LM head	160	64	84	88

D EXPERIMENTAL SETUP

Table D1: **Hyperparameters of both optimization stages.** Stage 1 fits the routers offline and evaluates every sparsity configuration (for attention, after screening on the layer's output error) by its KL divergence from the original model. Stage 2 fine-tunes a single checkpoint across all target budgets $b \in \mathcal{B}$ by drawing b uniformly and independently per completion token, computing the distillation loss exclusively on these tokens. For neuron sparsity, c denotes a neuron's contribution $\left|\mathrm{SiLU}(xG^\top)\,(xU^\top)\right| \cdot \|D\|$. The router learning rate is selected via FFN output reconstruction on a validation set, with early stopping after 5 epochs without improvement. On mixture-of-experts models, the original expert choices are replayed during Stage 1 evaluation. Target budgets $\mathcal{B}$ follow $1/N$ for $N = 1, 1.5, 2, \ldots$ down to the specified minimum fraction. The learned routers and MoE expert gates remain frozen during Stage 2.

	Qwen3		Gemma4	
	14B	30B A3B	31B	26B A4B
Stage 1: sparsity optimization				
Calibration corpus	6M tokens, 6 sources			
Router optimizer	Adam			
Router learning rates, min–max (#)	10^{-4}–$3{\cdot}10^{-3}$ (4)		$3{\cdot}10^{-4}$–10^{-3} (2)	
Router epochs (max)	200		20	
Router minibatch (tokens)	4,096			
Router loss	MSE on $\log(1+c)$			
KL evaluation sequences × tokens	72 × 16,384			
Expert routing during evaluation	–	replayed	–	replayed
Stage 2: fine-tuning				
Hardware (nodes × GPUs)	8 × 8 B200			
Optimizer	AdamW, weight decay 0			
Learning rate	10^{-4}	10^{-5}	$3{\cdot}10^{-5}$	10^{-4}
Learning-rate schedule	constant, no warm-up			
Gradient clipping (norm)	1			
Batch, sequences × tokens	256 × 16,384			
Optimizer steps	1,000	1,032	1,000	
Loss (to the original model)	forward KL, $T = 1$			
Smallest budget M in $\mathcal{B}$ (#)	$1/5$ (9)		$1/6$ (11)	$1/4$ (7)
Prompt budget	dense			
LoRA rank, attention	256			
LoRA rank, dense FFN	256	–	256	64
LoRA rank, each routed expert	–	16	–	16
LoRA rank, LM head	256		256 (tied)	–
LoRA α, dropout	128, 0			
Precision	bf16			
Seed	42			

D.1 REFERENCE DATASET

To optimize both the sparsity configurations (Stage 1) and the shared model weights (Stage 2), we construct a diverse reference dataset focused on analytical reasoning, mathematics, and STEM domains. Crucially, we extract only the input prompts from these collections and process them through the original, uncompressed model to generate the target next-token logits (reference traces) required to compute the KL divergence loss. The final training mixture is sampled according to the following normalized distribution:

- **Kimi-K2.5 Distillation Prompts** (27%)**:** A general-purpose collection spanning coding, science, mathematics, logic, and creative writing, originally curated for distilling Kimi-K2.5 (Team, 2026). All prompts are strictly deduplicated prior to trace generation.
- **Nemotron-Post-Training v2** (51% **total):** Sourced from NVIDIA's post-training dataset (Nathawani et al., 2025a; NVIDIA, 2025b). To rigorously cover foundational analytical domains, we sample across three distinct subsets:

  - *Math split (*22.5%*):* Mathematics and formal logic problems.
  - *Code split (*15.0%*):* Programming and algorithmic reasoning prompts.
  - *STEM split (*13.5%*):* Broad science and engineering queries.
- **Nemotron-Science v1** (12.5%)**:** Incorporating both the multiple-choice (MCQ) and reasoning (RQA) splits, this subset provides graduate-level science evaluations (formatted similarly to the GPQA benchmark) alongside advanced chemistry reasoning tasks (Bercovich et al., 2025; Nathawani et al., 2025b).
- **Nemotron-3-Nano RL Blend (**9.0%**):** Utilizing the STEM multiple-choice slice, which consists of knowledge-intensive queries (spanning science and law with up to ten options per question) extracted from NVIDIA's reinforcement-learning mixture (NVIDIA, 2025a).

D.2 Details on baselines

We reimplemented TEAL and ReLUfication from scratch. For TEAL, we used a step size of 0.05. Following Liu et al. (2025), we sparsified all attention projection matrices and FFN matrices, but not the LM-head. In the fraction of bytes per token, we consider the global fraction of parameters (i.e., including the LM-head), which makes TEAL numbers appear slightly worse than in the original paper (which only considered sparsity in attention and FFN). We evaluated the TEAL reasoning performance based on top-k sparsity, since we found that it performs slightly better (especially at low budgets) than threshold-based sparsity. For ReLUfication, we used the same loss function, dataset, and training hyperparameters as for Elastic Decoding.

E System Implementation and Inference Engine

At deployment, Elastic Decoding requires only the single optimized model checkpoint ψ alongside the set of lightweight sparsity configurations $\{\hat{\theta}_b\}_{b\in\mathcal{B}}$. Because the model was distilled to withstand instantaneous budget shifts, the active operating point b can be dynamically adjusted by the user or the operating system on a token-by-token basis.

To generate a token under the active budget b, Elastic Decoding determines the active parameters according to $\hat{\theta}_b$ (via input magnitudes or router predictions) and selectively fetches only the necessary weights from memory. Specifically, for input sparsity, the engine loads only the weight rows corresponding to the retained input dimensions. For neuron sparsity, it loads only the gate and up-projection columns, and the down-projection rows that correspond to the activated FFN neurons.

Table E1: **Our inference engine is comparable to `llama.cpp` on the original models.** Decoding speed (tok/s) of each original model, measured with `llama.cpp` and with our custom inference engine, all on an Apple M4 Max in bfloat16 at batch size one. Our engine serves the full-budget operating point of the Elastic Decoding checkpoint. Both decode 128 tokens after a prompt of 1 or 8192 tokens.

	1-token prompt			8192-token prompt		
Model	`llama.cpp`	Ours	Δ	`llama.cpp`	Ours	Δ
Qwen3-14B	17.7	17.6	$-0.7\,\%$	16.7	16.6	$-0.1\,\%$
Gemma4-31B	8.0	7.7	$-3.5\,\%$	7.6	7.5	$-1.5\,\%$
Qwen3-30B-A3B	61.1	66.2	$+8.2\,\%$	51.2	52.8	$+3.2\,\%$
Gemma4-26B-A4B	52.8	54.1	$+2.5\,\%$	47.3	50.2	$+6.0\,\%$

Because standard dense inference engines cannot execute partial matrix loads efficiently, we developed a custom macOS inference engine with optimized Metal kernels (Table E1). These kernels natively exploit our structured sparsity patterns to physically bypass reading inactive parameter blocks. As a result, the theoretical reduction in memory footprint translates into end-to-end wall-clock speedups in the memory-bound regime, while providing real-time elasticity with negligible switching overhead across budgets. Figure E1 shows the same checkpoint answering one prompt at every budget, with responses that stay accurate and on-instruction down to 20 % of the dense model bytes per token.

Supplementary videos. We provide supplementary video demonstrations showcasing Elastic Decoding's dynamic budget switching in real time. The inference engine in these demos does not keep both weight layouts resident in memory, instead, it computes all necessary weight transposes on the fly, highlighting the negligible latency overhead of our approach during practical on-device deployment (Appendix B).



Figure E1: Responses of Qwen-3 14B and of Elastic Qwen3-14B at every budget to the same prompt (top). Each header gives the weight bytes read per decoded token and their share of the dense model's, which the Budget slider also shows. Thinking mode is enabled, the thinking block is omitted and responses are verbatim.