STAPLE: STATIC SPARSE UPCYCLING OF PRETRAINED CHECKPOINTS VIA EMBEDDING MODULES

Anonymous authors
Paper under double-blind review

ABSTRACT

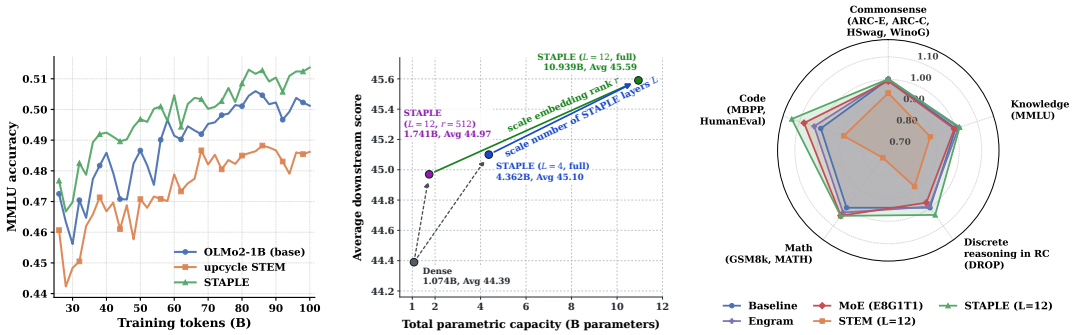
Sparse computation can scale the parametric capacity of large language models without proportionally increasing active computation, but pretraining sparse architectures remains expensive and data-hungry. Existing sparse upcycling methods mainly target Mixture-of-Experts (MoE) models, where expert homogeneity, sample inefficiency, and system overheads can limit practical scalability. Static sparse memory can address the efficiency and expert-homogeneity challenges, but naively integrating it into pretrained computation yields poor returns. In this work, we introduce STAPLE, a training recipe and architecture design for successfully integrating static sparse memory into a pretrained model. STAPLE preserves the pretrained model's computation and adds complementary token-specific knowledge in a scalable manner. On OLMo-2 1B, STAPLE improves over continued dense training by up to 2–3 points across math, code, discrete reasoning, and knowledge-intensive tasks. Average performance increases with static capacity, and improvements persist on the larger OLMo-2 7B backbone. STAPLE also outperforms static sparse baselines such as STEM and an Engram-style model, and achieves a higher average score than an MoE upcycling baseline while decoding 2$\times$ faster with less than half its GPU memory. Importantly, despite introducing billions of parameters late in training, our host-offloaded implementation keeps generation throughput within 1.5% and training throughput within 3% of the dense baseline, using a 1 GB GPU cache for inference. These results show that static sparse memory can provide a sample-efficient, scalable, and system-efficient path for expanding pretrained models without sparse pretraining from scratch.

1 INTRODUCTION

Scaling model capacity remains one of the most reliable ways to improve language-model capabilities (Kaplan et al., 2020; Hoffmann et al., 2022). Sparse computation makes this scaling attractive by increasing total parametric capacity without proportionally increasing active computation. Yet repeatedly pretraining larger sparse models from scratch is prohibitively expensive. This motivates *sparse upcycling* (Komatsuzaki et al., 2023; He et al., 2025; Vavre et al., 2024): expanding an already pretrained checkpoint through limited continued training such that the full potential of large parametric capacity is realized without training from scratch.

Most existing sparse upcycling methods rely on Mixture-of-Experts (MoE). A standard strategy clones pretrained FFNs into multiple experts, preserving the seed model at initialization (Komatsuzaki et al., 2023; He et al., 2025). However, cloned experts can remain functionally similar, limiting the benefit of further expert scaling. Breaking this *expert homogeneity* through domain-specialized branches, partial reinitialization, or additional optimization stages can improve specialization but at the cost of larger training budget (Nakamura et al., 2025) or additional training stages (Sukhbaatar et al., 2024; Mirvakhabova et al., 2025). Furthermore, MoE scaling also incurs routing, load-balancing, expert-fragmentation, memory, and communication overheads that can hurt practical training and serving efficiency (Doubov et al., 2024). The value of added expert capacity therefore depends on its performance gains relative to these costs. Hence, an effective upcycling method should be **sample-efficient**, preserve the **training and serving efficiency** of the seed model, and remain **scalable** as sparse capacity grows.

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.



(a) MMLU score vs train tokens (b) Scalability (c) Comparative Analysis

Figure 1: **Overview of STAPLE. (a)** STAPLE improves over the dense baseline early in training, while replacement-style STEM upcycling remains below it despite additional training. **(b)** Downstream performance improves as static sparse capacity is increased through embedding rank or the number of augmented layers. **(c)** Domain averages normalized to continued dense training; the largest gain is in code, with smaller gains in mathematics, knowledge, and reading comprehension.

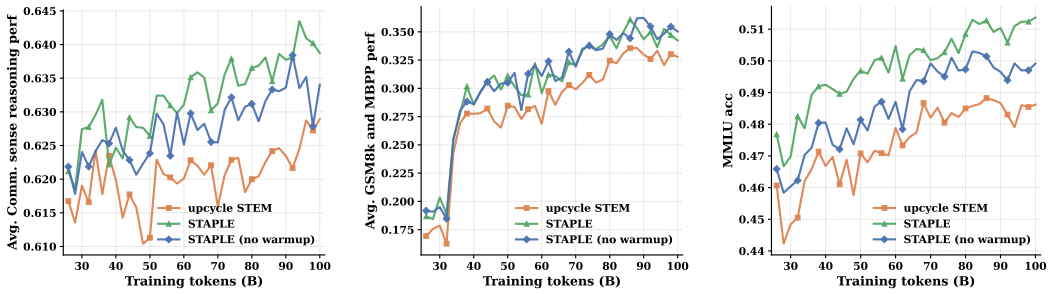
Recent emergence in static sparse architectures (Google DeepMind, 2025; Sadhukhan et al., 2026; Cheng et al., 2026; Liu et al., 2026; Tseng & Sa, 2026) has brought a breakthrough in scaling parametric capacity in a system-efficient manner. These architectures insert large token- or phrase-indexed embedding tables that are particularly amenable to CPU offloading and asynchronous prefetching, allowing substantial parametric expansion with little additional GPU-memory footprint or latency overhead. Moreover, we note that unlike cloned MoE experts, token-indexed parameters are intrinsically differentiated by their keys, naturally avoiding expert homogeneity problem. However, static sparse upcycling introduces its own challenges. *First*, replacing pretrained computation with newly initialized sparse memory discards useful features that must then be relearned. *Second*, existing static sparse architectures can suffer from poor sample efficiency: each embedding row is updated only when its corresponding token or phrase appears, producing highly *non-uniform training signals* under a limited token budget. *Finally*, even if these optimization challenges are addressed, it remains unclear whether static sparsity can continue to deliver performance gains as additional sparse capacity is introduced. Our key findings below directly address these concerns.

Sparsity should complement existing architecture: Retaining pretrained computation is important for effective static sparse upcycling. STEM replaces the FFN up-projection with a token-indexed embedding table (Sadhukhan et al., 2026). As Fig. 1a illustrates, STEM upcycling continues to struggle with downstream performance even after 100B tokens, whereas retaining the pretrained FFN up-projection and adding static memory as a residual pathway performs substantially better. This allows the new parameters to focus on complementary token-specific information rather than relearning computation already present in the seed model.

Initialize for sample-efficient adaptation: Preserving the pretrained model's initial functionality through correct initialization is important for better sample efficiency. Additionally, a short warmup with selective freezing of pretrained components then encourages the embeddings to learn useful complementary knowledge, particularly improving knowledge-intensive and commonsense tasks (Figs. 2c and 2a).

Scalability: When allowed to learn complementary token-specific knowledge, static memory continues to improve performance as capacity grows without proportionally increasing systems cost (Fig. 4). As illustrated in Fig. 1b, STAPLE improves in average downstream performance as we increase either the number of augmented layers or the embedding dimension, while the added memory can remain largely offloaded to CPU memory.

Based on these observations, we introduce **STAtic uPcycLing via Embedding layers (STAPLE)**. Building on STEM (Sadhukhan et al., 2026), STAPLE retains the pretrained FFN up-projection and augments it with a zero-initialized token-indexed embedding. The dense pathway preserves shared pretrained computation, while the static pathway learns complementary token-specific information. We scale this capacity through both the number of augmented layers and the embedding dimension.



(a) Common sense reasoning ablation (b) Math & code ablation (c) MMLU ablation

Figure 2: Domain-specific performance comparison among STEM upcycling, STAPLE and STAPLE with no warmup. We use 4 STEM embedding layer in each model. Fig.(a,b,c) illustrate the importance of retaining the FFN up-projections in STAPLE as it consistently outperforms STEM in every aspect. Fig.(a,c) shows the importance of warmup in improving general knowledge and commonsense task performances. Conversely, Fig.(b) suggests that math and code reasoning performances are relatively less affected.

For systems efficiency, the large embedding tables remain in CPU memory and are accessed through asynchronous prefetching, LFU caching, token deduplication, and row-sparse optimization during training and serving.

On OLMo-2 1B, STAPLE improves over continued dense training across multiple domains, including math, code, discrete reasoning, and knowledge-intensive tasks, by up to 2–3 points (Fig. 1c); these gains grow with static capacity (Fig. 1b) and persist on the larger OLMo-2 7B backbone. Under the same budget, STAPLE also outperforms the parameter-matched static sparse baselines, STEM and an Engram-style model. Compared with a serving-efficient MoE upcycling baseline, STAPLE achieves a higher average score while decoding 2$\times$ faster with a much smaller GPU memory footprint, and it stays within 1.5% and 3% of dense generation and training throughput. Finally, Sec. 4.2 examines the interpretability of STAPLE's memory and its interplay with contextual signals. Together, these results suggest that existing pretrained models can gain new capabilities from *scalable* static sparse memory without expensive training or a prohibitive increase in serving cost.

2 METHOD

Our final method builds upon three key aspects. Sec. 2.1 studies the importance of sparse architecture selection. Sec. 2.2 discusses what initialization of the embedding tables can restore the seed model's performance and what its implications are in the downstream task performance. Finally, Sec. 2.3 illustrates different scaling axes of STAPLE and their scaling behavior.

2.1 ARCHITECTURE STUDY: IMPACT OF RETAINING THE UP-PROJECTION

STEM Layers. Consider a gated FFN with gate, up, and down projections W_ℓ^g, W_ℓ^u, and W_ℓ^d, respectively, and hidden state x_ℓ at layer ℓ. STEM (Sadhukhan et al., 2026) replaces the pretrained up-projection output $W_\ell^u x_\ell$ with a token-indexed embedding $U_\ell[t]$, where t is the input token id and $U_\ell \in \mathbb{R}^{V \times d_{\text{ff}}}$ is a layer-local table with vocabulary size V and FFN width d_{ff}:

$$y_\ell^{\text{STEM}} = W_\ell^d \left(\text{SiLU}(W_\ell^g x_\ell) \odot U_\ell[t]\right). \tag{1}$$

A key architectural choice is whether sparse memory should replace or augment pretrained computation. In our four-layer ablation, replacement-based STEM upcycling fails to match continued dense training even with additional warmup(Fig. 1a), whereas retaining the pretrained FFN up-projection and adding STEM embeddings as a residual pathway yields substantially better downstream performance under the same training budget (Fig. 2).

Token-indexed embeddings receive updates only when their corresponding tokens appear, producing highly non-uniform training signals under a limited token budget. Replacement-based STEM

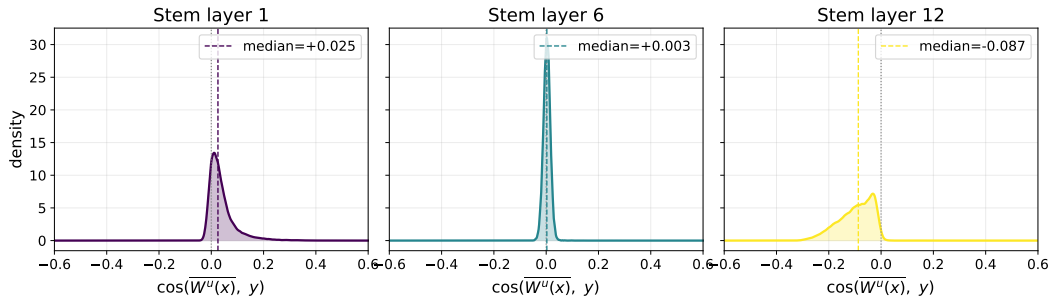


Figure 3: Pair-wise cosine similarity analysis between mean token-wise up-projection output and STEM embeddings in different layers of STAPLE with 12 STEM tables. Overall, the STEM embeddings are quite orthogonal to the respective mean up-projection output vectors.

compounds this general challenge of sparse memory by discarding useful pretrained up-projection features, which the new embeddings must then relearn. Moreover, it lacks a straightforward functionality-preserving initialization: random, token-wise mean, principal-component, and transformed input-embedding initializations did not improve long-term performance in our experiments.

These observations motivate the additive design of STAPLE:

$$y_\ell^{\text{STAPLE}} = W_\ell^d \left(\text{SiLU}(W_\ell^g x_\ell) \odot (W_\ell^u x_\ell + U_\ell[t])\right). \tag{2}$$

The pretrained up-projection preserves shared dense computation, while the added embeddings learn complementary token-specific residual information, as further examined in Sec. 4.1 and Fig. 3.

2.2 Initialization

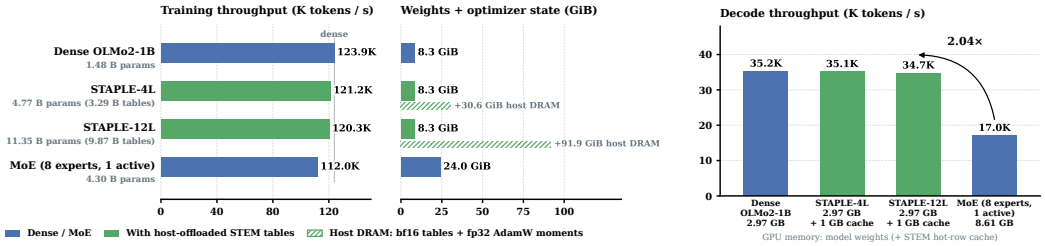
We initialize the embeddings with zero weights. This restores the seed model's computation graph and avoids discarding the knowledge embedded into the pretrained model weights.

Warmup. Furthermore, we find that a short warmup phase can better prepare the newly introduced STEM embeddings for subsequent mid-training. Before training on the high-quality mid-training mixture, we train STAPLE on a small subset of the pretraining corpus ($\approx 2.5\%$). Since the seed model has already been optimized on this corpus, we freeze the FFN up-projections during warmup. This preserves the pretrained dense computation while forcing the new embedding tables to learn useful complementary signals.

As shown in Fig. 2c and Fig. 2a, warmup consistently improves MMLU and average commonsense reasoning over direct upcycling throughout mid-training. This suggests that brief exposure to the pretraining distribution helps token-indexed embeddings acquire useful general-purpose memory before adapting to the mid-training mixture. In contrast, warmup provides no consistent gains on advanced math and programming tasks (Fig. 2b). To control this comparison, we exclude math and code sources from warmup, although they constitute a small fraction of the original pretraining corpus. These results suggest that sparse memory remains responsive to the subsequent mid-training distribution and can learn capabilities not emphasized during warmup.

2.3 Scalability

We scale STAPLE along two axes: the number of embedding-augmented layers L and the embedding dimension r. Under the same continued-training budget, increasing L from 4 to 12 raises the average downstream score from 45.10 to 45.59 (Table 2; Fig. 1b). The gains include reasoning-heavy tasks: from $L = 4$ to $L = 12$, MBPP improves by 1.80 points, HumanEval by 1.83 points, and DROP by 0.73 points, while MMLU is essentially unchanged (51.27 vs. 51.21). We place the embedding tables in early decoder layers (1–12), allowing token-specific residuals to be integrated into representations processed by subsequent layers.



(a) Training throughput and GPU memory

(b) Decode throughput

Figure 4: **Training and serving efficiency of host-offloaded STAPLE. (a)** Training throughput and GPU-resident weights plus optimizer state; parameter counts include the STEM tables. STAPLE keeps its bf16 tables and fp32 AdamW moments in host DRAM (hatched), matching the dense GPU footprint, whereas the MoE baseline (Sec. 3) keeps all experts and their optimizer states on the GPU. **(b)** Decode throughput on one NVIDIA B200 at batch size 64 and context length 256, labeled with GPU memory for weights plus, for STAPLE, a 1 GB cache of frequently accessed rows. Latency and prefill results are in Table 4.

To vary the memory capacity within each layer, we replace the full-rank table $U_\ell \in \mathbb{R}^{V \times d_{\text{ff}}}$ with $U_\ell^{(r)} \in \mathbb{R}^{V \times r}$ and learn a projection $P_\ell \in \mathbb{R}^{d_{\text{ff}} \times r}$:

$$y_\ell^{\text{STAPLE}(r)} = W_\ell^d \left(\text{SiLU}(W_\ell^g x_\ell) \odot \left(W_\ell^u x_\ell + P_\ell U_\ell^{(r)}[t] \right) \right). \tag{3}$$

The resulting module has $Vr + d_{\text{ff}} r$ parameters per layer and adds $2d_{\text{ff}} r$ projection FLOPs per token. For OLMo-2 1B, $r = 512$ reduces the token-indexed table by $16\times$. At $L = 12$, this low-rank variant reaches 44.97 average performance versus 45.59 for full-rank STAPLE and 44.39 for dense continuation (Table 2). Together, these results show that STAPLE benefits from scaling both the number of augmented layers and the capacity of each embedding table.

2.4 System Efficiency

Although STAPLE substantially increases the parametric memory footprint, the predictable token-indexed access pattern allows the large tables to remain in CPU DRAM, with only needed rows transferred to the GPU for training and inference. The challenge is to hide the host to device memory transfer cost behind model computation.

Inference-time offloading. We reduce retrieval overhead through three complementary optimizations. **Asynchronous prefetching** overlaps transfers for upcoming requests or prefill chunks with current computation; during decode, later-layer lookups overlap with earlier-layer execution. **Frequency-aware caching** exploits the Zipfian token distribution. Caching 5–6% of the vocabulary achieves roughly 80% hit rate, reducing the GPU-resident STAPLE-12L embedding footprint from 19.73 GB to a 1 GB cache. **Token deduplication** transfers each distinct row only once per prefill batch, reducing prefetch workload by roughly 50%. We omit it during decode, where the deduplication overhead exceeds the savings. As Fig. 4b shows, with 19.73 GB of learned tables offloaded, STAPLE-12L stays within 1.5% of dense decode throughput at batch size 64 (within 1.0% at batch size 1; Table 4). Prefill throughput is within 1.5% of dense at batch size 64, but is 6.9% lower at batch size 1. Relative to the MoE baseline, STAPLE uses about 54% less HBM model memory and improves decode throughput by 2.04× at batch size 64 and 58% at batch size 1.

Training-time offloading. Training also requires gradients and optimizer states, making full-table GPU residency costly. Yet each step accesses only the **distinct token ids** in its batch, $U = |\text{uniq}(\mathcal{B})|$, rather than the full vocabulary V. With non-uniform token frequencies, typically $U \ll V$, and U grows sublinearly with batch size (Takahashi & Tanaka-Ishii, 2019). We keep the full STEM tables and fp32 AdamW states in CPU DRAM and materialize only the active $U \times d_{\text{ff}}$ rows on GPU. This compact buffer acts as the autograd leaf, avoiding dense $V \times d_{\text{ff}}$ gradients; row-restricted AdamW updates only the accessed rows. To improve throughput, we overlap host gathering, host-to-device/device-to-host transfers, and optimizer work with backbone computation. As host work

scales with U but transformer compute scales with the total token count, overlap becomes more favorable at larger batches. STAPLE-12L trains within 3% of dense throughput (120.3K vs. 123.9K tokens/s) with 91.9 GiB of tables and optimizer moments in host DRAM; its GPU-resident weights and optimizer state match the dense model, and the compact row buffers add only about 3% peak HBM, whereas the MoE baseline trains 9.6% slower and needs 2.9× the GPU memory for weights and optimizer state (Fig. 4a). Appendix A.3 details both pipelines.

3 Experimental Setup

Models. We use OLMo-2 1B (OLMo et al., 2024) as the primary dense seed model because its training recipe, datasets, and optimizer-state checkpoints are publicly available. The released checkpoint is pretrained on 4T tokens and contains 16 decoder layers. We construct STAPLE variants by augmenting the early decoder layers with 4 or 12 STEM embedding tables, all zero-initialized so that the upcycled model initially preserves the computation and performance of the dense checkpoint. Additionally, we upcycle the same OLMo-2 1B backbone with a single-stream Engram-style model using memory modules at layers 2 and 7, hashed 2-gram and 3-gram memories with eight 64-dimensional heads per order, and approximately 9.87B additional embedding-memory parameters matched to STAPLE-12L; this implementation uses context-conditioned key/value projections with sigmoid gating and a depthwise causal convolution, but omits Engram's multi-branch hyperconnections. We also include a relatively serving-efficient MoE upcycling baseline, which replaces the FFNs in eight interleaved decoder layers with MoE layers of eight full-width experts and top-1 routing; we refer to it as the MoE baseline.

Datasets. For mid-training, we use the Dolma 3 Dolmino Mix (Olmo et al., 2025) high-quality data mixture, which contains training sequences from web, non-web, and synthetic sources. This mixture is designed to improve academic question answering, mathematical reasoning, and programming abilities, and to prepare models for subsequent post-training (Ouyang et al., 2022; Wei et al., 2022). For the additional warmup phase, we train on 100B tokens sampled from a 2.5% subset of the DCLM Baseline 1.0 dataset. The original OLMo-2 1B pretraining corpus contains 5T tokens and includes a small fraction of code and math data (OLMo et al., 2024); these sources are not included in our warmup dataset. This choice allows us to isolate the effect of warmup on STEM initialization without introducing additional math or code-specific supervision before mid-training.

Training. For all dense backbone parameters, we follow the OLMo-2 1B training hyperparameters (OLMo et al., 2024) and initialize the optimizer from the released pretrained optimizer states. Newly introduced STEM embeddings use a peak learning rate 5× larger than that of the dense parameters, while all parameter groups follow a linear decay to zero. Full training hyperparameters are listed in Table 3 (Appendix A.1). During the STAPLE warmup phase, we freeze the up-projections in the augmented layers so that the newly added static memory can adapt without perturbing this pretrained pathway. When the separate STEM warmup stage is omitted, we instead apply a 2% warmup ratio to the embedding-specific learning-rate schedule during mid-training. For fair compute comparison, we also continue training the dense baseline for the same number of pretraining tokens using the original pretraining learning-rate schedule. The 7B experiments use a reduced budget of approximately 30B warmup tokens followed by 70B mid-training tokens due to their higher computational cost.

Evaluations. We evaluate our models on a broad benchmark suite spanning commonsense reasoning, general knowledge, reading comprehension, mathematics, and coding. For commonsense reasoning, we report ARC-Easy (Clark et al., 2018) (0-shot, accuracy), ARC-Challenge (Clark et al., 2018) (0-shot, normalized accuracy), HellaSwag (Zellers et al., 2019) (0-shot, normalized accuracy), and WinoGrande (Sakaguchi et al., 2020) (0-shot, accuracy). We evaluate general knowledge on MMLU (Hendrycks et al., 2021a) (0-shot, accuracy) and reading comprehension on DROP (Dua et al., 2019) (3-shot, F1). For mathematics, we use GSM8K (Cobbe et al., 2021) (5-shot, strict exact match) and Hendrycks MATH (Hendrycks et al., 2021b) (0-shot, exact match). For coding, we evaluate MBPP (Austin et al., 2021) (5-shot, pass@1) and HumanEval (Chen et al., 2021) (0-shot, pass@1).

Table 1: **Upcycling comparison and design ablations.** All models in the first block use the OLMo-2 1B seed; the Engram-style baseline's added embedding memory is parameter-matched to STAPLE-12L. The replacement-style STEM and additive STAPLE models both use 12 embedding layers. The second block reports the OLMo-2 7B backbone with $L = 16$ for STAPLE. Best values are bolded within each backbone block.

	Commonsense				Knowledge	RC	Math		Code		
Method	**ARC-E**	**ARC-C**	**HSwag**	**WinoG**	**MMLU**	**DROP**	**GSM8K**	**MATH**	**MBPP**	**HEval**	**Avg.**
OLMo-2 1B: baselines and architecture/initialization comparisons											
Baseline	75.42	**47.44**	65.18	**66.22**	50.30	41.91	37.76	7.68	28.80	23.17	44.39
MoE upcycling (8 experts, top-1)	75.04	45.73	66.82	64.72	49.96	40.74	39.80	7.64	28.00	**28.05**	44.65
Engram-style	**77.40**	47.35	66.05	64.01	50.98	41.68	38.21	**8.44**	29.20	24.39	44.77
STEM (L=12)	74.37	42.06	60.79	61.56	44.30	37.00	26.00	6.94	22.60	23.78	39.94
STAPLE (L=12)	76.60	45.99	**68.27**	63.77	**51.21**	**43.56**	**40.03**	7.44	**31.60**	27.44	**45.59**
OLMo-2 7B: larger-backbone evaluation											
Baseline	82.07	55.97	77.52	**74.82**	62.53	62.79	67.79	11.08	41.80	34.76	57.11
STAPLE (L=16)	**82.58**	**57.76**	**77.62**	74.35	**62.67**	**63.55**	**70.13**	**12.02**	**43.40**	**35.98**	**58.01**

Table 2: **STAPLE scalability on OLMo-2 1B.** Increasing the number of augmented layers L and the embedding dimension r are separate capacity axes; $r = 512$ denotes the low-rank variant, while the other STAPLE rows use full-width embeddings. All rows use the same continued-training budget. Best values are bolded within this table.

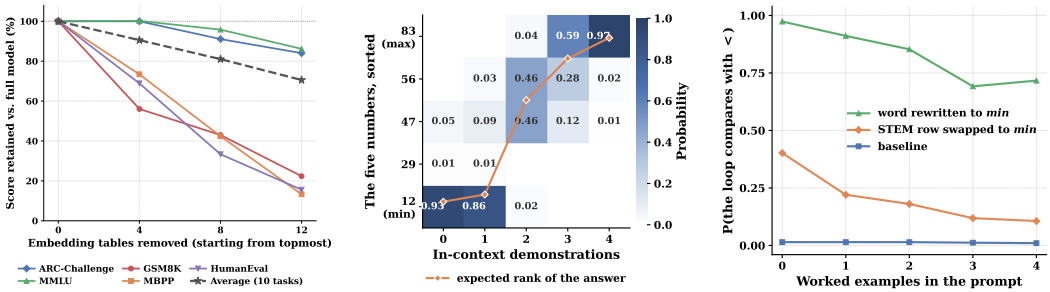
	Commonsense				Knowledge	RC	Math		Code		
Method	**ARC-E**	**ARC-C**	**HSwag**	**WinoG**	**MMLU**	**DROP**	**GSM8K**	**MATH**	**MBPP**	**HEval**	**Avg.**
Baseline	75.42	**47.44**	65.18	**66.22**	50.30	41.91	37.76	**7.68**	28.80	23.17	44.39
STAPLE (L=4)	**76.98**	46.25	66.00	65.82	**51.27**	42.83	39.35	7.08	29.80	25.61	45.10
STAPLE (L=12)	76.60	45.99	**68.27**	63.77	51.21	**43.56**	**40.03**	7.44	**31.60**	**27.44**	**45.59**
STAPLE (L=12, r=512)	76.18	46.67	66.99	65.59	50.49	41.91	39.80	6.28	30.20	25.61	44.97

4 Results & Analysis

Table 1 shows that STAPLE-12L raises the ten-task average from 44.39 to 45.59 over continued dense training. Although large parametric memory is usually associated with memorization, the gains extend to reasoning-heavy tasks: HumanEval improves by 4.27 points, MBPP by 2.80, GSM8K by 2.27, and DROP by 1.65.

Under the same upcycling budget, STEM struggles to even match continued dense training performance, with notable performance degradations in knowledge-based tasks as well as GSM8K. This establishes the benefits of retaining the pretrained model's computation graph while upcycling with static memory. Furthermore, STAPLE exceeds the parameter-matched Engram-style baseline across math, code, and discrete reasoning, supporting the effectiveness of its architecture and adaptation recipe. Compared with the MoE baseline, STAPLE gains 0.94 points on average while delivering higher decode throughput and using less GPU-resident model memory (Fig. 4). These comparisons establish a favorable performance–efficiency tradeoff for the evaluated configurations.

Table 2 separates the effects of augmenting more layers and increasing the embedding dimension. STAPLE improves average performance along both scaling axes, and the added capacity has negligible impact on model throughput. We additionally evaluate STAPLE on an OLMo-2 7B backbone to test whether the gains persist beyond the 1B setting. We augment the first 16 decoder layers with STEM tables and, due to the substantially higher training cost, restrict training to approximately 30B warmup tokens followed by 70B mid-training tokens. As shown in Table 1, STAPLE improves the average score from 57.11 to 58.01 and improves 9 of the 10 evaluated tasks. The gains are particularly encouraging on reasoning and code: average mathematics performance improves by 4.16% relative to the 7B baseline, while average code performance improves by 3.68%. The modest MMLU gain may reflect the shorter warmup, consistent with the 1B warmup ablation, although this explanation is not isolated in the 7B experiments. These results suggest that the benefits of static sparse memory extend to larger backbones even under a substantially reduced training budget.



(a) Embedding layer ablation (b) Order-statistics intervention (c) Code-generation intervention

Figure 5: **Static memory ablation and controlled embedding interventions.** **(a)** Benchmark-score retention after removing the last 4, 8, or all 12 embedding tables from trained STAPLE-12L. **(b)** Order statistics: with the STEM embedding for 'maximum' swapped for that of 'minimum', probability over the five sorted candidates (heatmap) and expected answer rank (line) as correct demonstrations are added. **(c)** Code generation: probability that the generated maximum-finding loop compares with $<$ under the same swap, alongside the unmodified model and the instruction rewritten to 'minimum'.

4.1 Ablation studies

Figure 3 compares learned token embeddings with their corresponding mean up-projection outputs. Their low cosine similarity suggests that the two feature streams occupy different directions, but geometry alone does not establish the memory's usefulness. We therefore test the model's dependence on these embeddings through removal ablations.

Specifically, we progressively remove the last 4, last 8, or all 12 embedding tables from a STAPLE-12L checkpoint at evaluation time, without retraining. As shown in Fig. 5a, the average score across ten benchmarks falls from 45.59 to 41.28, 36.94, and 32.17, respectively. The sharp decline shows that the model relies on the learned static memory after mid-training. The fully ablated model is not equivalent to the original dense seed, since the remaining weights have co-adapted with static memory.

Dependence on embedding depth. The dependence is especially pronounced on math and code tasks: removing all tables lowers GSM8K, MBPP, and HumanEval by 31.08, 27.40, and 23.17 points, compared with 7.34 on ARC-Challenge and 7.08 on MMLU. Even removing only the last four tables reduces GSM8K by 17.59 points, MBPP by 8.40 points, and HumanEval by 8.54 points. Thus, the learned memory contributes beyond the earliest augmented layers, particularly to generation-heavy mathematical and programming tasks, whereas the knowledge-intensive tasks benefit more from the bottom-most embedding layers.

4.2 Interpretability: Interplay between static memory and contextual reasoning

The removal ablations show that STAPLE relies on its static memory, but not how that memory affects individual predictions. Following STEM (Sadhukhan et al., 2026), we probe this with causal interventions on the embeddings. Because each row of U_ℓ is selected only by the input token id, swapping one row for another alters the memory at a single position without changing the input text, so any change in the output is attributable to the embeddings. At the same time, Eq. (2) scales $U_\ell[t]$ by the context-dependent gate $\mathrm{SiLU}(W_\ell^g x_\ell)$, so context may counteract a conflicting memory signal. We test this on two tasks, finding the maximum of five numbers and writing Python code that finds the maximum of an array. In both, we replace the STEM embedding for 'maximum' with that for 'minimum' and vary the number of correct demonstrations in the prompt.

Order statistics. Without demonstrations, the swap overrides the instruction: the model assigns 0.93 probability to the minimum (Fig. 5b). Each demonstration shifts probability toward larger answers, and with four the model assigns 0.97 probability to the maximum.

Code generation. The swap also alters the generated algorithm, since comparing with $<$ rather than $>$ turns the loop into a minimum finder. The probability of $<$ is 0.40 without demonstrations and falls to 0.11 with four (Fig. 5c). For reference, the unmodified model almost never uses $<$, while rewriting the instruction to 'minimum' makes $<$ the most likely choice at every demonstration count.

In both tasks, the static embeddings causally change the output, from the selected answer to the program's logic, yet consistent context progressively overrides them. This is consistent with the context-dependent gate controlling how much the token-indexed memory contributes.

5 Related Works

Static sparse memory. Token-indexed memory expands language-model capacity with little additional computation, as explored by Gemma 3n (Google DeepMind, 2025), STEM (Sadhukhan et al., 2026), and JTok (Yang et al., 2026). MeKi (Ding et al., 2026) adds memory parallel to the FFN, while L^3 (Tseng & Sa, 2026) contextually aggregates statically selected embeddings. Engram (Cheng et al., 2026) uses hashed n-gram memory with contextual gating; LongCat-Flash-Lite (Liu et al., 2026) also studies n-gram embedding scaling, and Qwen3.8-Flash-Next (Qwen Team, 2026a;b) demonstrates host-offloaded memory at scale. STAPLE studies how to make newly added static capacity useful in mature pretrained checkpoints under limited continued training. It retains the pretrained up-projection that STEM replaces and adds zero-initialized token residuals within the FFN. Our architecture and warmup studies examine this adaptation recipe, and our parameter-matched Engram-style baseline compares an alternative memory design under the same upcycling budget.

Memory-based adaptation. Learned key–value memory layers provide another form of sparse capacity (Lample et al., 2019; Berges et al., 2024). Sparse memory finetuning selectively updates memory slots to acquire knowledge while reducing forgetting (Lin et al., 2025), and subsequent work retrofits pretrained Transformers with sparse memory and recovery training (Goyal et al., 2026). These studies emphasize knowledge acquisition and retention. STAPLE instead examines how deterministically indexed memory scales broad downstream performance during continued training, combining preservation of pretrained FFN computation with predictable retrieval for training and serving offload.

Sparse upcycling. Sparse upcycling reuses pretrained dense weights to initialize MoEs (Komatsuzaki et al., 2023; He et al., 2025; Vavre et al., 2024). Domain-specialized branches (Sukhbaatar et al., 2024), informed router initialization (Jiang et al., 2025), and partial expert reinitialization (Nakamura et al., 2025) address expert specialization. Other studies characterize training-budget tradeoffs (Liew et al., 2025) and serving overheads (Doubov et al., 2024). STAPLE expands token-indexed memory while retaining dense FFN computation, avoiding cloned expert pools and learned routing. Its central challenge is making sparsely and unevenly updated parameters useful within a limited training budget; we study architecture, initialization, and warmup choices that enable this expansion while largely preserving the seed model's throughput.

6 Conclusion

We introduced STAPLE, a simple and sample-efficient method for continually expanding the parametric capacity of pretrained dense models during mid-training. STAPLE shows that the system-efficient scaling of static sparsity can translate into practical performance gains even in the upcycling setting. The key idea is to preserve the pretrained model's learned knowledge and to further complement it with scalable memory. Despite introducing a large number of new parameters, STAPLE remains highly token-efficient and provides consistent gains with increasing memory scale, notably on reasoning-heavy tasks. Our training recipe enables static sparse upcycling of pretrained models, and compared with a serving-efficient MoE upcycling baseline, STAPLE achieves a higher average score while training faster and decoding $2\times$ faster with a much smaller GPU memory footprint. An exciting direction for future work is whether jointly scaling static memory with MoE can continue to push the performance–efficiency frontier.

AI USE STATEMENT

Generative AI tools assisted with manuscript drafting and editing, consistency checks between the text, tables, and figures, and verification of bibliographic references. The authors are responsible for the final text, claims, experimental results, and accompanying artifacts.

REPRODUCIBILITY STATEMENT

All models are upcycled from publicly released OLMo-2 1B and OLMo-2 7B checkpoints and their optimizer states, using the publicly available DCLM BASELINE 1.0 and DOLMA 3 DOLMINO MIX corpora. Section 3 specifies the seed models, the STAPLE, Engram-style, and MoE configurations, the warmup and mid-training data and budgets, and the evaluation benchmarks with their shot counts and metrics. Eqs. (2) and (3) define the full-rank and low-rank STAPLE layers. Appendix A.1 lists the sequence lengths, batch sizes, step counts, and learning-rate schedules for every parameter group in both training stages. The training and serving measurements use the settings stated in Fig. 4 and Table 4, and Appendix A.3 details the inference- and training-time offloading pipelines, including Algorithm 1.

ETHICS STATEMENT

This work studies how to add static sparse memory to existing, publicly released language models through continued training on public corpora, evaluated on public benchmarks. It involves no human-subject study and collects no personal data. The web-derived training corpora may contain biased, harmful, or personal content, and the upcycled models inherit these risks from their seed models and data.

REFERENCES

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.

Vincent-Pierre Berges, Barlas Oğuz, Daniel Haziza, Wen tau Yih, Luke Zettlemoyer, and Gargi Ghosh. Memory layers at scale, 2024. URL https://arxiv.org/abs/2412.09764.

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.

Xin Cheng, Rui Tian, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Chengqi Deng, Shangyan Zhou, Chenggang Zhao, Zhewen Hao, Yukun Li, Han Zhang, Zhengyan Zhang, Yixu Wei, M. Y. Xu, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. Conditional memory via scalable lookup: A new axis of sparsity for large language models, 2026. URL https://arxiv.org/abs/2601.07372.

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.

Ning Ding, Fangcheng Liu, Kyungrae Kim, Linji Hao, Kyeng-Hun Lee, Hyeonmok Ko, and Yehui Tang. MeKi: Memory-based expert knowledge injection for efficient LLM scaling, 2026. URL https://arxiv.org/abs/2602.03359.

Sasha Doubov, Nikhil Sardana, and Vitaliy Chiley. Sparse upcycling: Inference inefficient finetuning. In *Proceedings of the 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pp. 194–205. PMLR, 2024. URL https://proceedings.mlr.press/v262/doubov24a.html.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proc. of NAACL*, 2019.

Google DeepMind. Gemma 3n model overview, 2025. URL https://ai.google.dev/gemma/docs/gemma-3n. Accessed: 2025-09-04.

Satyam Goyal, Anirudh Kanchi, Garv Shah, and Prakhar Gupta. Improving sparse memory finetuning, 2026. URL https://arxiv.org/abs/2604.05248.

Alan Gray. Getting started with CUDA graphs. NVIDIA Technical Blog, 2019. URL https://developer.nvidia.com/blog/cuda-graphs/.

Ethan He, Abhinav Khattar, Ryan Prenger, Vijay Korthikanti, Zijie Yan, Tong Liu, Shiqing Fan, Ashwath Aithal, Mohammad Shoeybi, and Bryan Catanzaro. Upcycling large language models into mixture of experts, 2025. URL https://arxiv.org/abs/2410.07524.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (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, 2021b. URL https://arxiv.org/abs/2103.03874.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Wangyi Jiang, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. Improved sparse upcycling for instruction tuning. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 9485–9498, Abu Dhabi, UAE, 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.636/.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. Sparse upcycling: Training mixture-of-experts from dense checkpoints, 2023. URL https://arxiv.org/abs/2212.05055.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys, 2019. URL https://arxiv.org/abs/1907.05242.

Seng Pei Liew, Takuya Kato, and Sho Takase. Scaling laws for upcycling mixture-of-experts language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 37682–37704. PMLR, 2025. URL https://proceedings.mlr.press/v267/liew25a.html.

Jessy Lin, Luke Zettlemoyer, Gargi Ghosh, Wen-Tau Yih, Aram Markosyan, Vincent-Pierre Berges, and Barlas Oğuz. Continual learning via sparse memory finetuning, 2025. URL https://arxiv.org/abs/2510.15103.

Hong Liu, Jiaqi Zhang, Chao Wang, Xing Hu, Linkun Lyu, Jiaqi Sun, Xurui Yang, Bo Wang, Fengcun Li, Yulei Qian, Lingtong Si, Yerui Sun, Rumei Li, Peng Pei, Yuchen Xie, and Xunliang Cai. Scaling embeddings outperforms scaling experts in language models, 2026. URL https://arxiv.org/abs/2601.21204.

Leyla Mirvakhabova, Babak Ehteshami Bejnordi, Gaurav Kumar, Hanxue Liang, Wanru Zhao, and Paul Whatmough. Dirichlet-prior shaping: Guiding expert specialization in upcycled MoEs, 2025. URL https://arxiv.org/abs/2510.01185.

Taishi Nakamura, Takuya Akiba, Kazuki Fujii, Yusuke Oda, Rio Yokota, and Jun Suzuki. Drop-Upcycling: Training sparse mixture of experts with partial re-initialization, 2025. URL https://arxiv.org/abs/2502.19261.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 Furious, 2024. URL https://arxiv.org/abs/2501.00656.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, Jacob Morrison, Jake Poznanski, Kyle Lo, Luca Soldaini, Matt Jordan, Mayee Chen, Michael Noukhovitch, Nathan Lambert, Pete Walsh, Pradeep Dasigi, Robert Berry, Saumya Malik, Saurabh Shah, Scott Geng, Shane Arora, Shashank Gupta, Taira Anderson, Teng Xiao, Tyler Murray, Tyler Romero, Victoria Graf, Akari Asai, Akshita Bhagia, Alexander Wettig, Alisa Liu, Aman Rangapur, Chloe Anastasiades, Costa Huang, Dustin Schwenk, Harsh Trivedi, Ian Magnusson, Jaron Lochner, Jiacheng Liu, Lester James V. Miranda, Maarten Sap, Malia Morgan, Michael Schmitz, Michal Guerquin, Michael Wilson, Regan Huff, Ronan Le Bras, Rui Xin, Rulin Shao, Sam Skjonsberg, Shannon Zejiang Shen, Shuyue Stella Li, Tucker Wilde, Valentina Pyatkin, Will Merrill, Yapei Chang, Yuling Gu, Zhiyuan Zeng, Ashish Sabharwal, Luke Zettlemoyer, Pang Wei Koh, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. Olmo 3, 2025. URL https://arxiv.org/abs/2512.13961.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

Qwen Team. On the design of Qwen3.8-Next architecture: Evaluation, efficiency, and training stability. Technical report, Alibaba Group, August 2026a.

Qwen Team. Qwen3.8-Flash-Next: A new architecture, towards ultimate cost-efficiency, August 2026b. URL https://qwen.ai/blog?id=qwen3.8-flash-next.

Ranajoy Sadhukhan, Sheng Cao, Harry Dong, Changsheng Zhao, Attiano Purpura-Pontoniere, Yuandong Tian, Zechun Liu, and Beidi Chen. STEM: Scaling transformers with embedding modules, 2026. URL https://arxiv.org/abs/2601.10639.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial Winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8732–8740, 2020.

Sainbayar Sukhbaatar, Olga Golovneva, Vasu Sharma, Hu Xu, Xi Victoria Lin, Baptiste Rozière, Jacob Kahn, Daniel Li, Wen tau Yih, Jason Weston, and Xian Li. Branch-Train-MiX: Mixing expert LLMs into a mixture-of-experts LLM, 2024. URL https://arxiv.org/abs/2403.07816.

Shuntaro Takahashi and Kumiko Tanaka-Ishii. Evaluating computational language models with scaling properties of natural language, 2019. URL https://arxiv.org/abs/1906.09379.

Albert Tseng and Christopher De Sa. L^3: Large lookup layers, 2026. URL https://arxiv.org/abs/2601.21461.

Aditya Vavre, Ethan He, Dennis Liu, Zijie Yan, June Yang, Nima Tajbakhsh, and Ashwath Aithal. Llama 3 meets MoE: Efficient upcycling, 2024. URL https://arxiv.org/abs/2412.09952.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022. URL https://arxiv.org/abs/2109.01652.

Yebin Yang, Huaijin Wu, Fu Guo, Lin Yao, Xiaohan Qin, Jingzhi Wang, Debing Zhang, and Junchi Yan. JTok: On token embedding as another axis of scaling law via joint token self-modulation, 2026. URL https://arxiv.org/abs/2602.00800.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of the Eighth Conference on Machine Learning and Systems*, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/dbf02b21d77409a2db30e56866a8ab3a-Paper-Conference.pdf.

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*, 2019.

A Appendix

A.1 Training hyperparameters

Table 3: Training hyperparameters for warmup and mid-training. *STEM/STAPLE tables* are the token-indexed static sparse embeddings; *Engram n-gram tables* are the hashed n-gram embeddings of the Engram-style baseline. During STAPLE warmup, pretrained up-projections in augmented layers are frozen. The backbone warmup schedule resumes the base model's pretraining cosine schedule at its final checkpoint. MoE expert weights follow the backbone schedule at a higher peak learning rate.

Parameter group		Training setup				Optimization schedule					
Stage	Parameters	Seq. Len.	Batch Size	Train Steps	Warmup Steps	Resume Step	End Step	LR Scheduler	Peak LR	Min LR Ratio	Weight Decay
Warmup	STEM/STAPLE tables	4096	512	50,000	500	0	50,000	cosine	4×10^{-4}	0.6	1×10^{-4}
	Engram n-gram tables	4096	512	50,000	500	0	50,000	cosine	4×10^{-4}	0.6	1×10^{-4}
	Backbone	–	–	50,000	0	1,907,359	2,384,186	cosine	4×10^{-4}	0.1	1×10^{-1}
	MoE expert weights	–	–	50,000	500	0	50,000	cosine	2×10^{-4}	0.91	1×10^{-1}
Mid-training	STEM/STAPLE tables	4096	512	50,000	0	0	50,000	linear	3×10^{-4}	0	1×10^{-4}
	Engram n-gram tables	4096	512	50,000	0	0	50,000	linear	3×10^{-4}	0	1×10^{-4}
	Backbone	–	–	50,000	0	0	50,000	linear	7.44×10^{-5}	0	1×10^{-1}
	MoE expert weights	–	–	50,000	0	0	50,000	linear	2×10^{-4}	0	1×10^{-1}

A.2 Serving latency and throughput

Table 4 reports the full serving measurements summarized in Fig. 4b, including batch-size-1 decode and prefill latency and throughput.

Table 4: **Serving efficiency on one NVIDIA B200.** All measurements use context length 256; latency and throughput are reported in ms and tok/s. HBM memory includes resident weights and GPU caches. STAPLE-12L offloads 12 embedding tables (19.73 GB in BF16 precision) to CPU DRAM with a 1 GB GPU cache. MoE denotes the upcycled MoE baseline of Sec. 3 (eight interleaved MoE layers, each with eight full-width experts and top-1 routing), which keeps all experts in HBM and incurs routing and token-reordering overhead.

Architecture	**HBM model memory**	**Linear FLOPs / token**	**Decode B1 ms / tok/s**	**Decode B64 ms / tok/s**	**Prefill B1 ms / tok/s**	**Prefill B64 ms / tok/s**
Dense	2.97 GB GPU weights	∼2.56 G	1.224 / 817	1.817 / 35.2K	1.419 / 180.5K	34.184 / 479.3K
STAPLE-12L	2.97 GB GPU weights + 1 GB cache	∼2.56 G	1.237 / 809	1.845 / 34.7K	1.524 / 168.0K	34.678 / 472.5K
STEM-12L	2.57 GB GPU weights + 1 GB cache	∼2.16 G	1.151 / 869	1.671 / 38.3K	1.413 / 181.1K	27.580 / 594.1K
MoE (8 experts, top-1)	8.61 GB GPU weights, all 8 experts	∼2.56 G + routing	1.951 / 513	3.761 / 17.0K	2.975 / 86.1K	36.179 / 452.9K

A.3 Implementation Details

We keep the large STEM tables in CPU DRAM and move only the rows required by the current workload to the GPU. Inference uses a small GPU-resident LFU cache, whereas training materializes a compact buffer over the distinct token set described in Sec. 2.4.

A.4 Inference-time embedding offload

Prefill and decode require different host-to-device embedding retrieval paths because of the difference in their prefetching workloads. Table 5 summarizes the implementation; in both cases, retrieval is overlapped with backbone execution. Our serving stack additionally uses CUDA graphs (Gray, 2019), a fused in-place KV-cache append kernel, and FlashInfer (Ye et al., 2025) for paged decode attention.

For very short prefills, where CPU-gather setup dominates, we use the same GPU-driven mapped-host path as decode. This avoids CPU gathers and explicit H2D copies on the latency-sensitive path.

Table 5: Inference-time retrieval paths for host-offloaded STEM tables. Prefill favors bulk deduplicated transfers, whereas decode uses GPU-driven fine-grained lookup to minimize exposed latency.

Phase	Retrieval path	Overlap / execution
Prefill	• Deduplicate prompt tokens once. • Probe the GPU LFU cache and gather only cache-miss rows from CPU. • Pack misses into pinned host buffers and issue bulk asynchronous H2D copies.	• Reuse the same inverse token map across all STAPLE layers. • Prefetch later-layer rows while earlier transformer layers execute. • Pipeline successive requests/chunks and fuse compact-row reconstruction with the FFN using `torch.compile`.
Decode	• Store every four adjacent STAPLE tables in a token-major CPU layout. • Launch a GPU lookup kernel that probes the LFU cache. • On a miss, read the row directly from CUDA-registered host memory; no CPU gather or explicit H2D copy is required.	• Capture grouped lookups and model execution in a multi-stream CUDA graph. • Launch later lookup groups on a side stream while earlier layers compute. • Synchronize only when a group is first consumed (layers 1, 5, and 9 for STAPLE-12L).

A.5 Training-time embedding offload

Algorithm 1 shows the training pipeline. The token union and inverse map are computed once per step and reused by every STEM layer; full tables and fp32 AdamW moments remain in host memory.

Algorithm 1: Training step with host-offloaded STEM tables.

Input: batch token ids $\mathcal{B}$; host tables/states $\{E_\ell, m_\ell, v_\ell\}$
$(\mathcal{U}, \pi) \leftarrow \mathrm{unique}(\mathcal{B})$ `// once per step`
for $\ell \in \mathcal{L}_{\mathrm{STEM}}$ *in forward order* **do**
  enqueue host gather $R_\ell \leftarrow E_\ell[\mathcal{U}]$ into pinned memory
  enqueue async H2D copy $R_\ell \rightarrow \widehat{R}_\ell$ and record ready event
end for
run forward; at each STEM layer, wait for its ready event and use $\widehat{R}_\ell[\pi]$
run backward
foreach STEM *layer* ℓ *when compact gradient* G_ℓ *becomes ready* **do**
  enqueue async D2H copy of G_ℓ into the alternate staging buffer
  update (E_ℓ, m_ℓ, v_ℓ) with row-restricted AdamW on rows $\mathcal{U}$
end foreach

The compact $U \times d_{\mathrm{ff}}$ row buffer is the autograd leaf, so backward never allocates a dense $V \times d_{\mathrm{ff}}$ gradient. Gradient staging is double buffered, while H2D and D2H transfers use separate streams. We issue next-step gathers in forward consumption order rather than inheriting backward order, which bounds the exposed pipeline-fill latency for the first augmented layer.

Finally, host-memory placement is important for sustaining bandwidth: table allocations are interleaved across NUMA memory controllers and optionally backed by huge pages, while the OpenMP worker pool uses a passive idle policy to avoid competing with the GPU-launch thread. Latency-critical CUDA-event waits are instead polled by the dedicated host worker.