MIDUS: Memory-Infused Depth Up-Scaling

Taero Kim TAERO.KIM@YONSEI.AC.KR
Yonsei University, South Korea

Hoyoon Byun HOYUN.BYUN@YONSEI.AC.KR
Yonsei University, South Korea

Yongjun Choi CHOIYJ9803@YONSEI.AC.KR
Yonsei University, South Korea

Sungrae Park SUNGRAE.PARK@UPSTAGE.AI
Upstage AI, South Korea

Kyungwoo Song* KYUNGWOO.SONG@YONSEI.AC.KR
Yonsei University, South Korea

Abstract

Depth Up-Scaling (DUS) expands pre-trained language models by duplicating Transformer blocks, but also duplicates FFN-heavy components and increases parameter and compute costs. Conditional computation and memory layers motivate lighter alternatives to dense FFN branches, while attention-head specialization suggests that added capacity can be allocated more effectively at finer head-level granularity. We propose Memory-Infused Depth Up-Scaling (MIDUS), replacing duplicated FFNs with memory layers for lightweight retrieval-based residual capacity. We instantiate the inserted memory layer as a Head-wise Memory Layer (HML), where each attention-head output queries a distinct product-key space, and Head-wise Implicit Value Expansion (HIVE) realizes retrieved latent values through head-specific projections from a shared latent bank. MIDUS with HML improves the performance–efficiency trade-off, and our head-importance and fixed-retrieval analyses suggest that its added residual capacity is organized around head-wise structure. [1]

1. Introduction

Large language models (LLMs) exhibit predictable gains as parameters scale, but training larger architectures from scratch remains prohibitively expensive. Model up-scaling offers a practical alternative by expanding a pre-trained backbone and adapting it through continual pre-training (CPT). Among up-scaling strategies, Depth Up-Scaling (DUS) increases depth by inserting Transformer blocks while preserving the backbone structure [16, 44, 46]. From the residual-architecture view, these inserted blocks act as correction paths between pre-trained blocks, making their parameterization central to how added capacity is introduced.

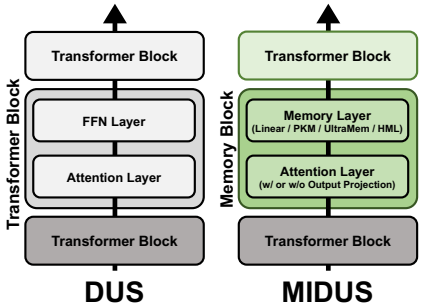

Figure 1: DUS expands model depth through duplicating Transformer blocks, while MIDUS integrates Memory blocks.

* Corresponding Author
1. Code: https://github.com/MLAI-Yonsei/midus-hml

© T. Kim, H. Byun, Y. Choi, S. Park & K. Song*.

Recent DUS methods refine depth expansion through selective block copying, learned initializers, or transport-guided initialization [5, 44, 47]. However, they still duplicate FFN-heavy Transformer blocks, tying added capacity to whole-block FFN replication with substantial parameter and compute overhead. This motivates rethinking how the inserted branch is parameterized and where its capacity is allocated. Studies on conditional computation, structured sparsity, and retrieval-based memory suggest that Transformer capacity can extend beyond dense FFNs [13, 19, 24, 25, 27, 50], while attention heads differ in function and importance [49, 51]. Together, these observations motivate a finer-grained DUS design where inserted capacity is lightweight through memory-based parameterization and aligned with head-level heterogeneity.

We propose Memory-Infused Depth Up-Scaling (MIDUS), which replaces inserted FFN branches in DUS with memory layers, turning added depth into lightweight retrieval-based residual capacity. Its Head-wise Memory Layer (HML) lets each attention-head output query a distinct product-key space, while Head-wise Implicit Value Expansion (HIVE) realizes retrieved latent values through compact head-specific projections from a shared latent bank. On Llama-3.2-1B and Llama-3.1-8B, MIDUS with HML (MIDUS-HML) improves performance and efficiency over DUS baselines, using fewer parameters, lower GPU memory, and lower computational cost. Head-importance and structural analysis further support the head-conditioned structure of the added residual correction.

2. Methodology

To parameterize the added residual capacity in DUS without duplicating FFN-heavy blocks, we introduce MIDUS with HML. MIDUS replaces inserted FFN branches with memory layers, while HML allocates the retrieval-based capacity at the attention-head level. We build on the DUS, memory-layer, and attention-head literature summarized in Appendix B, with preliminaries on Transformer blocks and linear memory layers provided in Appendix C. Below, we present the core formulation of MIDUS-HML and defer complexity analysis, and implementation details to Appendix D.

2.1. Memory-Infused Depth Up-Scaling

Let $x \in \mathbb{R}^{T\times d}$ be a length T sequence representation with hidden dimension d. In a pre-norm Transformer block, the attention update is followed by an FFN residual update. MIDUS replaces this FFN branch in an inserted block with a memory layer Mem, defining

$$a(x) = x + \mathrm{Attn}(\mathrm{LN}(x)), \qquad M(x) = x + \mathrm{Mem}(a(x)). \tag{1}$$

Here, LN denotes layer normalization and Attn denotes the attention layer. Let $f_0(x) = B_{L-1} \circ \cdots \circ B_0(x)$ be the pre-trained base model, where B_ℓ denotes the ℓ-th Transformer block. Given anchor indices $0 \le i_0 < \cdots < i_{D-1} \le L-1$, standard DUS inserts duplicated Transformer blocks *after* these anchors [44]. MIDUS uses the same anchors but inserts Memory blocks *before* the corresponding pre-trained blocks, yielding

$$f_{\mathrm{MIDUS}}(x) = B_{L-1} \circ \cdots \circ B_{i_{D-1}} \circ M_{D-1} \circ \cdots \circ B_{i_0} \circ M_0 \circ \cdots \circ B_0(x). \tag{2}$$

This preserves the DUS depth-expansion structure while changing the added residual path. The full composition, ordering, and placement policies are provided in Appendix D.1. We zero-initialize each memory value bank so that $\mathrm{Mem}(a) = 0$ at initialization, making every Memory block an identity map. We initialize each Memory block's attention parameters from the following pre-trained block,

providing a stable attention-updated input for learning retrieval-based residual corrections. Further details are given in Appendix D.2.

2.2. Head-wise Memory Layer

HML parameterizes the memory-based residual correction at the attention-head level. Let H be the number of heads and $d_h = d/H$. HML uses the pre-output-projection attention output

$$a'(x) = \mathrm{Attn}'(\mathrm{LN}(x)) = [a_1(x) \mid \cdots \mid a_H(x)], \tag{3}$$

where Attn' omits the output projection from Attn and $a_h(x) \in \mathbb{R}^{T \times d_h}$ is the output of head h. Thus, in the HML instantiation, the generic memory correction $\mathrm{Mem}(a(x))$ in Eq. (1) is implemented as $\mathrm{HML}(a'(x))$, so that memory queries remain head-partitioned. HML sets $q_h = a_h(x)$, so each head queries memory using its own head-specific representation.

Multi-Head Product-Key Lookup in HML. Instead of using a linear key bank for each head, HML uses a product-key memory [27] at the head level. Each query q_h is split into row and column parts:

$$q_h = [q_h^{\mathrm{row}} \mid q_h^{\mathrm{col}}], \qquad q_h^{\mathrm{row}}, q_h^{\mathrm{col}} \in \mathbb{R}^{T \times d_p},$$

where $d_p = d_h/2$. For each head h, HML maintains two sub-key banks $K_h^{\mathrm{row}}, K_h^{\mathrm{col}} \in \mathbb{R}^{n \times d_p}$. A composite key is specified by a pair (i, j), where $i \in 1, \ldots, n$ indexes a row sub-key and $j \in 1, \ldots, n$ indexes a column sub-key. The Cartesian product of the two sub-key banks therefore defines $N = n^2$ composite keys, with value index map

$$\pi(i, j) = (i - 1)n + j.$$

The row and column score matrices are

$$S_h^{\mathrm{row}} = q_h^{\mathrm{row}} K_h^{\mathrm{row}\top}, \qquad S_h^{\mathrm{col}} = q_h^{\mathrm{col}} K_h^{\mathrm{col}\top}, \qquad S_h^{\mathrm{row}}, S_h^{\mathrm{col}} \in \mathbb{R}^{T \times n}. \tag{4}$$

For each head h and token t, let $I_{h,t}$ and $J_{h,t}$ be the Top-k index sets of $S_{h,t}^{\mathrm{row}}$ and $S_{h,t}^{\mathrm{col}}$, respectively. These sets form k^2 candidate composite keys:

$$\widehat{\Omega}_{h,t} = I_{h,t} \times J_{h,t}. \tag{5}$$

For each candidate pair $(i, j) \in \widehat{\Omega}_{h,t}$, define the additive pair score

$$\sigma_{h,t}(i, j) = S_{h,t}^{\mathrm{row}}(i) + S_{h,t}^{\mathrm{col}}(j). \tag{6}$$

The final Top-k pair set $\Omega_{h,t} \subset \widehat{\Omega}_{h,t}$ is selected according to $\sigma_{h,t}$. We define $\alpha_{h,t}$ as the softmax weights over the selected pair scores $\{\sigma_{h,t}(i, j)\}_{(i,j) \in \Omega_{h,t}}$.

Head-wise Implicit Value Expansion. While head-wise product-key retrieval gives each head a distinct key space, using a single realized value bank still forces different heads to retrieve values from the same output value space. To address this, HML incorporates Head-wise Implicit Value Expansion (HIVE), which separates shared latent storage from head-specific value realization. Let $\bar{V} \in \mathbb{R}^{N \times r}$

Table 1: Performance of DUS and MIDUS across Llama-3.2-1B and Llama-3.1-8B backbones under CPT and SFT settings. Llama-3.1-8B† results are from [5], except for MIDUS-HML.

		Llama-3.2-1B									Llama-3.1-8B†								
		PPL ↓	Zero-shot Accuracy ↑								PPL ↓	Zero-shot Accuracy ↑							
	Method	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
CPT	Base	13.22	**68.69**	22.27	60.22	25.88	63.61	75.08	29.85	49.37	8.35	79.97	26.88	72.06	65.19	81.83	78.84	58.61	66.20
	SOLAR	13.41	68.81	22.58	60.22	26.29	61.16	**75.24**	30.14	49.21	9.90	79.88	26.88	71.59	57.41	80.70	78.56	54.37	64.20
	Llama Pro	12.33	66.75	22.43	59.75	35.14	64.86	74.54	31.22	50.67	7.81	81.61	**29.49**	73.72	70.93	81.65	79.98	62.56	68.56
	LESA	12.06	66.33	21.97	60.22	45.21	64.37	74.59	36.11	52.69	7.73	82.07	27.96	74.11	**72.40**	81.93	80.30	62.63	68.77
	OpT-DeUS	11.72	66.46	**23.35**	**61.56**	**46.44**	62.42	74.54	36.56	53.05	7.73	82.07	27.34	**74.74**	71.91	82.26	**80.79**	62.96	68.87
	Avg-DeUS	12.04	66.84	22.73	60.38	41.93	64.01	74.27	34.40	52.08	7.95	82.15	27.50	73.48	71.09	82.17	80.20	62.11	68.39
	MIDUS-HML	**11.64**	66.16	23.20	**61.56**	46.27	**65.29**	75.08	**36.91**	**53.50**	**7.40**	**82.37**	28.57	74.59	70.84	**82.87**	80.25	**63.40**	**68.98**
SFT	Base	13.09	**69.65**	21.20	58.96	26.37	62.66	75.46	30.43	49.25	8.32	81.10	24.58	72.14	68.30	82.14	79.71	59.17	66.73
	SOLAR	13.23	69.28	**23.81**	58.33	27.44	60.64	**75.90**	31.05	49.49	9.68	80.68	25.19	71.19	61.18	81.19	79.16	55.03	64.80
	Llama Pro	12.55	67.68	22.89	60.06	42.59	62.75	75.46	33.62	52.15	7.81	83.33	27.19	74.11	72.07	82.26	80.79	62.32	68.87
	LESA	12.37	65.99	23.35	60.38	47.91	**66.33**	75.24	36.43	53.66	7.72	83.84	26.57	75.53	73.05	83.00	80.69	63.57	69.47
	OpT-DeUS	11.82	66.62	**23.81**	60.14	49.55	63.43	75.52	37.64	53.81	7.73	83.80	26.73	**76.09**	73.05	83.36	80.85	63.84	69.67
	Avg-DeUS	12.18	67.09	23.66	60.30	47.01	65.63	74.97	35.62	53.47	7.91	**83.88**	26.42	75.45	72.89	83.18	80.47	63.10	69.34
	MIDUS-HML	**11.75**	66.33	21.51	**60.77**	**50.04**	66.21	**75.90**	**37.82**	**54.08**	**7.50**	83.50	**28.11**	74.90	**73.63**	**83.39**	**80.96**	**64.54**	**69.86**

be a shared latent value table with latent dimension r, and let $W_h \in \mathbb{R}^{r \times d_h}$ be a head-specific value projection. HIVE computes

$$\bar{M}_{h,t} = \sum_{(i,j) \in \Omega_{h,t}} \alpha_{h,t}(i,j)\, \bar{V}_{\pi(i,j)} \in \mathbb{R}^r, \quad M_{h,t}^{\mathrm{HIVE}} = \bar{M}_{h,t} W_h \in \mathbb{R}^{d_h}. \tag{7}$$

Concatenating token-stacked head outputs $M_h^{\mathrm{HIVE}} \in \mathbb{R}^{T \times d_h}$ defines

$$\mathrm{HML}(a'(x)) = [M_1^{\mathrm{HIVE}} \mid \cdots \mid M_H^{\mathrm{HIVE}}] \in \mathbb{R}^{T \times d}, \qquad M^{\mathrm{HML}}(x) = x + \mathrm{HML}(a'(x)). \tag{8}$$

This enables head-specific realization while sharing latent storage. Details are in Appendix D.3.

2.3. Structural Analysis of HIVE

We analyze MIDUS-HML under fixed retrieval coefficients to isolate the value-side role of head-specific realization, by comparing a shared realized value bank with head-specific value banks under the same retrieved memory slots. Fix a' and the retrieval weights. For each head h, collect the Top-k retrieval-weight vectors into $A_h = [\alpha_{h,1}, \ldots, \alpha_{h,T}]^\top \in \mathbb{R}^{T \times N}$, where each $\alpha_{h,t}$ is treated as an N-dimensional sparse vector with nonzero entries only on the selected Top-k value indices. Let $Y_h \in \mathbb{R}^{T \times d_h}$ be the target correction for head h, and let $F(a') = [F_1(a') \mid \cdots \mid F_H(a')]$ be a head-partitioned correction. Under the same retrieval weights, the shared-realized-value family uses one bank $V_{\mathrm{sh}} \in \mathbb{R}^{N \times d_h}$ with $F_h(a') = A_h V_{\mathrm{sh}}$, whereas the head-specific family uses banks $V_h \in \mathbb{R}^{N \times d_h}$ with $F_h(a') = A_h V_h$. Using the Frobenius norm, define the loss and optimal values as

$$\mathcal{L}(F) = \frac{1}{2} \sum_{h=1}^{H} \|F_h(a') - Y_h\|^2, \quad \mathcal{L}_{\mathrm{Share}}^\star = \inf_{V_{\mathrm{sh}}} \mathcal{L}(F), \quad \mathcal{L}_{\mathrm{Ind}}^\star = \inf_{\{V_h\}_{h=1}^H} \mathcal{L}(F). \tag{9}$$

Theorem 1 *In the fixed-retrieval setup, suppose $A_h = A$ for all $h \in [H]$, and let Π_A be the orthogonal projection onto the column space of A. Assume that there exists an optimal head-specific solution $\{V_h^\star\}_{h=1}^H$ that admits a shared latent factorization $V_h^\star = \bar{V} W_h$, where $\bar{V} \in \mathbb{R}^{N \times r}$, $W_h \in \mathbb{R}^{r \times d_h}$, and $r \leq d_h$. If the projected targets are non-identical across heads, i.e., $\Pi_A Y_h \neq \Pi_A Y_{h'}$ for some $h \neq h'$, then the corresponding HIVE correction $\widehat{F}_h(a') = A\bar{V}W_h$ satisfies $\mathcal{L}(\widehat{F}) = \mathcal{L}_{\mathrm{Ind}}^\star < \mathcal{L}_{\mathrm{Share}}^\star$.*

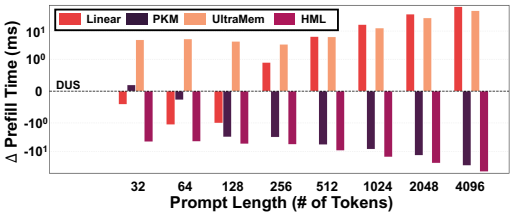

Figure 2: Change in prefill time over prompt length for each memory layer relative to DUS. HML is the fastest across all prompt lengths.

Table 2: Efficiency for Llama-3.1-8B with parameters, GPU memory, training speed, and throughput.

	Method	B ↓ Params	GB ↓ GPU Memory	s/iter ↓ Train	Tokens/s ↑ Throughput
	Base	8.03 / 8.03	81.6 / 19.2	6.12	61.0
DUS	Llama Pro	3.49 / 11.52	63.8 / 27.0	6.89	41.3
DUS	OpT-DeUS	3.49 / 11.52	59.8 / 27.0	5.78	41.3
MIDUS	Linear	1.48 / 9.51	51.3 / 23.1	11.05	42.4
MIDUS	PKM	1.21 / 9.24	48.3 / 22.5	9.88	47.3
MIDUS	UltraMem	1.29 / 9.32	56.9 / 23.6	8.56	36.9
MIDUS	HML	**0.42 / 8.45**	**34.8 / 20.1**	**5.05**	**50.5**

The theorem shows that HIVE shares latent storage while preserving head-specific value realization. If optimal head-specific banks admit a shared latent factorization, HIVE recovers the head-specific optimum through compact per-head projections. By contrast, a shared realized value bank must fit different projected head corrections with one value matrix, causing an irreducible gap when projected targets differ. Thus, HIVE separates storage sharing from head-wise realization in the added residual branch. Proofs and FFN-related discussion are provided in Appendix M.

3. Experiments

3.1. Experiment Settings

Following Cao et al. [5], we evaluate DUS and MIDUS on Llama-3.2-1B and Llama-3.1-8B, adding 8 and 16 blocks, respectively. MIDUS-HML uses multi-head product-key memory with $n = 64$ row/column sub-keys, Top-k retrieval with $k = 4$, and HIVE dimension $r = d_h$. With 32 heads per block, this yields 1.05M and 2.10M head-specific addressable memory slots for Llama-3.2-1B and Llama-3.1-8B, respectively. DUS baselines follow their original placement policies, while MIDUS uses *Distributed* placement by default (Appendix G.4). Training and evaluation are provided in Appendix E. We perform CPT on `FineWeb-Edu` and `MathPile`, and apply SFT to the `FineWeb-Edu` CPT model using `Alpaca-GPT4` and `Databricks-Dolly-15k`. General-purpose language and reasoning are evaluated with `lm-evaluation-harness`, while math-domain, retrieval-heavy, and long-context evaluations are reported in the Appendix.

3.2. Experiment Results

Main results. Table 1 reports general-purpose language and reasoning performance after CPT on `FineWeb-Edu` and SFT on `Alpaca-GPT4`, using Llama-3.2-1B and Llama-3.1-8B backbones. Across both backbones, MIDUS-HML achieves the lowest WikiText PPL and the highest average zero-shot accuracy after CPT, with gains largely preserved after SFT. These results support replacing inserted FFN branches with HML as head-conditioned memory-based residual correction. Additional SFT, `MathPile`, retrieval-heavy, and long-context results are reported in Appendix H, J, and I.

Table 3 compares different memory layers within MIDUS. All variants use equal memory-head counts and distinct key memories per head, while Linear, PKM [27], and UltraMem [19] share value parameterization across heads. HML achieves the best average accuracy, supporting HIVE-based head-specific realization. Additional ablations are provided in Appendix K.4.

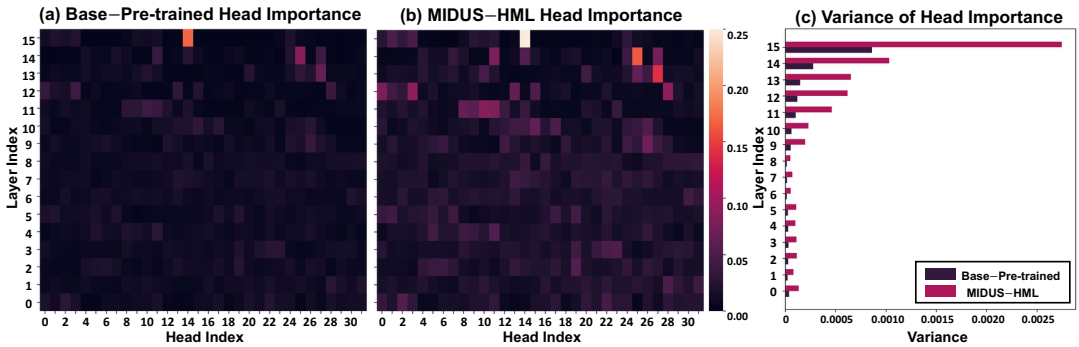

Figure 3: Head-importance score IS_h on PIQA for Llama-3.2-1B. (a) Heatmap of IS_h over layer $\times$ head index for the pre-trained base model before CPT. (b) Heatmap for MIDUS-HML after CPT. Head importance rises overall, with the largest gains on heads already high in (a). (c) Per-layer variance of IS_h across heads is consistently higher under MIDUS-HML, reflecting that important heads become more dominant. This pattern is consistent with head-conditioned capacity allocation in HML, where each head retrieves from its own key space and realizes head-specific values.

Table 3: CPT results with Llama-3.2-1B for MIDUS with different memory layers.

	PPL ↓	Zero-shot Accuracy ↑							
Mem	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
Linear	11.82	65.61	22.73	**62.04**	42.42	64.71	75.08	35.89	52.64
PKM	11.97	65.70	21.35	61.40	42.18	**65.47**	74.65	33.93	52.10
UltraMem	**11.64**	**66.46**	21.81	61.09	44.31	64.83	**75.63**	36.49	52.94
HML	**11.64**	66.16	**23.20**	61.56	**46.27**	65.29	75.08	**36.91**	**53.50**

Efficiency of MIDUS-HML. Table 2 reports parameters, peak GPU memory, training latency, and generation throughput on Llama-3.1-8B. MIDUS-HML gives the strongest efficiency profile among up-scaling methods, with the fewest parameters, the lowest training and inference memory, faster training than OpT-DeUS, and the highest throughput. Among memory-layer variants, HML is also the most efficient, training nearly twice as fast as Linear, PKM, and UltraMem. Figure 2 shows that MIDUS-HML remains faster than DUS across prompt lengths, while UltraMem incurs larger prefill overhead from virtual value indexing and Tucker-based query–key scoring [19]. All DUS methods share one inference reference curve due to identical inference-time structure. Since placement policy also affects training efficiency [5], additional ablations are provided in Table 12 and Appendix G.4.

Head-importance concentration under HML. We further measure gradient-based head importance [3] on PIQA for pre-trained Llama-3.2-1B and MIDUS-HML after CPT on `FineWeb-Edu`. Figure 3 shows that MIDUS-HML yields a more uneven head-importance profile with higher within-layer variance, indicating more concentrated head-level contributions. This pattern is consistent with head-conditioned allocation of the added residual capacity. The score definition is in Appendix E.3.

4. Conclusion

We introduced MIDUS, a DUS framework that replaces FFN-heavy duplicated branches with memory layers for lightweight retrieval-based residual capacity. HML realizes this capacity at the attention-head level by combining multi-head product-key retrieval with head-specific value realization by HIVE. Experiments show improved performance and efficiency over DUS baselines and memory layer variants, while head-importance and fixed-retrieval analyses support its head-wise structure.

5. Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00457216) and by Upstage AI.

References

[1] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers)*, pages 2357–2367, 2019.

[2] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 3119–3137, 2024.

[3] Hritik Bansal, Karthik Gopalakrishnan, Saket Dingliwal, Sravan Bodapati, Katrin Kirchhoff, and Dan Roth. Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11833–11856, 2023.

[4] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[5] Mingzi Cao, Xi Wang, and Nikolaos Aletras. Progressive depth up-scaling via optimal transport. *arXiv preprint arXiv:2508.08011*, 2025.

[6] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[7] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert's attention. In *Proceedings of the 2019 ACL workshop BlackboxNLP: analyzing and interpreting neural networks for NLP*, pages 276–286, 2019.

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

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

[10] Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. Free dolly: Introducing the world's first truly open instructiontuned llm. 2023.

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[12] Wenyu Du, Tongxu Luo, Zihan Qiu, Zeyu Huang, Yikang Shen, Reynold Cheng, Yike Guo, and Jie Fu. Stacking your transformers: A closer look at model growth for efficient llm pre-training. *Advances in Neural Information Processing Systems*, 37:10491–10540, 2024.

[13] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[14] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Zenodo*, 2021.

[15] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495, 2021.

[16] Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pages 2337–2346. PMLR, 2019.

[17] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[18] 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. *arXiv preprint arXiv:2103.03874*, 2021.

[19] Zihao Huang, Qiyang Min, Hongzhi Huang, Defa Zhu, Yutao Zeng, Ran Guo, and Xun Zhou. Ultra-sparse memory network. *arXiv preprint arXiv:2411.12364*, 2024.

[20] Zihao Huang, Yu Bao, Qiyang Min, Siyan Chen, Ran Guo, Hongzhi Huang, Defa Zhu, Yutao Zeng, Banggu Wu, Xun Zhou, et al. Ultramemv2: Memory networks scaling to 120b parameters with superior long-context learning. *arXiv preprint arXiv:2508.18756*, 2025.

[21] 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.

[22] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

[23] Dahyun Kim, Chanjun Park, Sanghoon Kim, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeonwoo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, et al. Solar 10.7 b: Scaling large language models with simple yet effective depth up-scaling. *arXiv preprint arXiv:2312.15166*, 2023.

[24] Gyuwan Kim and Tae-Hwan Jung. Large product key memory for pretrained language models. *arXiv preprint arXiv:2010.03881*, 2020.

[25] 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. *arXiv preprint arXiv:2212.05055*, 2022.

[26] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, pages 785–794, 2017.

[27] Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. *Advances in Neural Information Processing Systems*, 32, 2019.

[28] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 6086–6096, 2019.

[29] Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *arXiv preprint arXiv:2007.08124*, 2020.

[30] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[31] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[32] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[33] Yu Pan, Ye Yuan, Yichun Yin, Jiaxin Shi, Zenglin Xu, Ming Zhang, Lifeng Shang, Xin Jiang, and Qun Liu. Preparing lessons for progressive training on language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18860–18868, 2024.

[34] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 30811–30849. Curran Associates, Inc., 2024. doi: 10.52202/079017-0970. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/370df50ccfdf8bde18f8f9c2d9151bda-Paper-Datasets_and_Benchmarks_Track.pdf.

[35] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*, 2023.

[36] David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*, 2024.

[37] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[38] Nikunj Saunshi, Stefani Karp, Shankar Krishnan, Sobhan Miryoosefi, Sashank Jakkam Reddi, and Sanjiv Kumar. On the inductive bias of stacking towards improving reasoning. *Advances in Neural Information Processing Systems*, 37:71437–71464, 2024.

[39] 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.

[40] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*, 2018.

[41] Zhen Tan, Daize Dong, Xinyu Zhao, Jie Peng, Yu Cheng, and Tianlong Chen. Dlo: Dynamic layer operation for efficient vertical scaling of llms. *arXiv preprint arXiv:2407.11030*, 2024.

[42] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 5797–5808, 2019.

[43] Zengzhi Wang, Xuefeng Li, Rui Xia, and Pengfei Liu. Mathpile: A billion-token-scale pretraining corpus for math. *Advances in Neural Information Processing Systems*, 37:25426–25468, 2024.

[44] Chengyue Wu, Yukang Gan, Yixiao Ge, Zeyu Lu, Jiahao Wang, Ye Feng, Ying Shan, and Ping Luo. Llama pro: Progressive llama with block expansion. *arXiv preprint arXiv:2401.02415*, 2024.

[45] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International conference on machine learning*, pages 10524–10533. PMLR, 2020.

[46] Cheng Yang, Shengnan Wang, Chao Yang, Yuechuan Li, Ru He, and Jingqiao Zhang. Progressively stacking 2.0: A multi-stage layerwise training method for bert training speedup. *arXiv preprint arXiv:2011.13635*, 2020.

[47] Yifei Yang, Zouying Cao, Xinbei Ma, Yao Yao, Zhi Chen, Libo Qin, and Hai Zhao. Lesa: Learnable llm layer scaling-up. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 22463–22476, 2025.

[48] Kazuki Yano, Sho Takase, Sosuke Kobayashi, Shun Kiyono, and Jun Suzuki. Efficient construction of model family through progressive training using model expansion. *arXiv preprint arXiv:2504.00623*, 2025.

[49] Kayo Yin and Jacob Steinhardt. Which attention heads matter for in-context learning? *arXiv preprint arXiv:2502.14010*, 2025.

[50] Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. Moefication: Transformer feed-forward layers are mixtures of experts. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 877–890, 2022.

[51] Youxiang Zhu, Ruochen Li, Danqing Wang, Daniel Haehn, and Xiaohui Liang. Focus directions make your language models pay more attention to relevant contexts. *arXiv preprint arXiv:2503.23306*, 2025.

Appendix A. Limitations and Broader Impact.

This work focuses on efficient DUS for Llama-3.2-1B and Llama-3.1-8B under CPT and SFT settings. While we evaluate MIDUS across general-purpose, math-domain, retrieval-heavy, and long-context benchmarks, further validation on other model families, larger scales, and different training regimes remains future work. Due to the computational cost of repeated CPT runs, our main results are reported as single-run point estimates rather than with error bars or confidence intervals. Our structural analysis should also be interpreted as a fixed-retrieval explanation of HIVE, not as a complete characterization of all FFN-based or memory-based expansion methods. MIDUS may reduce the parameter, memory, and computational costs of model up-scaling, improving accessibility and resource efficiency. However, more efficient LLM scaling may also lower the cost of deploying models that inherit broader LLM risks, including misuse, biased outputs, privacy concerns, and unsafe generations. These risks should be managed through the responsible-use policies and safeguards of the underlying models and deployment settings.

Appendix B. Related Work

B.1. Depth Up-Scaling

Model up-scaling expands capacity along different axes, such as the width-wise route taken by MoE that selectively activates expert FFNs [21, 39]. DUS instead expands models depth-wise by duplicating and reorganizing Transformer blocks. Early work studies block selection and growth schedules through stacking and progressive training [12, 16, 33, 38, 46, 48]. Llama Pro selectively copies blocks under a stabilizing recipe [44], while SOLAR stacks contiguous block groups [23]. Recent methods focus on initialization. LESA learns per-layer initializers [47], while OpT-DeUS uses transport-guided initialization and Avg-DeUS initializes new layers by averaging adjacent blocks [5].

Despite this progress, DUS still adds capacity through FFN-heavy Transformer blocks, increasing compute and memory with duplicated depth. Token-routing methods [36, 41] alleviate runtime cost but still keep FFN parameters at inference. MIDUS instead changes the inserted block itself, replacing duplicated FFN branches with memory layers, so the added depth carries lightweight retrieval-based capacity rather than additional dense FFN branches.

B.2. Memory Layer

A memory layer stores key–value slots and retrieves values through dot-product lookup, forming a linear memory in its basic form. Linear memories scale poorly because lookup cost and key parameters grow linearly with memory size. PKM [27] factorizes the key space into two codebooks, enabling lookup over many composite keys with fewer dot products and improving compute and parameter efficiency over linear memory. Prior work uses PKM to augment or replace FFNs in pre-training, enabling retrieval-based computation in place of wide dense layers [24]. Ultra-sparse memory (UltraMem) [19, 20] further expands memory capacity through implicit value expansion and compressed representations while keeping per-token compute low.

MIDUS brings this memory-layer view to DUS by replacing the FFN branches that DUS would otherwise duplicate in inserted blocks. HML allocates memory capacity at the attention-head level by combining head-wise retrieval with HIVE-based value realization. Each head queries a distinct key space, while HIVE compactly realizes head-specific values from a shared latent bank. This aligns inserted memory capacity with head-level heterogeneity while keeping value storage tractable.

B.3. Specialization and Importance of Attention Heads

Attention heads specialize and vary in importance across layers, inputs, and tasks. Prior analyses show that many heads can be pruned with limited degradation, while a smaller subset plays a disproportionate role in model behavior [32, 42]. Other studies identify functional patterns such as local syntax, long-range dependency tracking, and task-dependent context use [7, 49]. Recent attribution analyses further show highly uneven head contributions, with a few heads dominating specific decisions while many remain inactive [51]. These findings suggest that added capacity need not be allocated only through block-level dense updates. HML instead allocates memory capacity at the attention-head level, giving each head a distinct key space and head-specific value realization.

Appendix C. Preliminaries

We formalize the standard pre-norm Transformer block [45] and a linear memory layer.

Transformer block. A pre-norm Transformer block is a mapping $B : \mathbb{R}^{T\times d} \to \mathbb{R}^{T\times d}$. Given $x \in \mathbb{R}^{T\times d}$, it is defined as

$$B(x) = a(x) + \mathrm{FFN}(\mathrm{LN}(a(x))), \qquad a(x) = x + \mathrm{Attn}(\mathrm{LN}(x)). \tag{10}$$

Here, LN denotes layer normalization and Attn denotes the attention layer. The block first applies an attention residual update and then an FFN residual update. We denote the ℓ-th Transformer block by B_ℓ.

Linear memory layer. Given an input representation $a \in \mathbb{R}^{T\times d}$, a memory layer $\mathrm{Mem} : \mathbb{R}^{T\times d} \to \mathbb{R}^{T\times d}$ returns a retrieval-based representation $m = \mathrm{Mem}(a)$. It is specified by a query map and a key–value memory bank. For a linear memory layer, define $q(a) = aW_q \in \mathbb{R}^{T\times d_q}$ with learnable query projection $W_q \in \mathbb{R}^{d\times d_q}$, keys $K \in \mathbb{R}^{N\times d_q}$, and values $V \in \mathbb{R}^{N\times d}$ for N memory slots. The similarity matrix is

$$S(a) = q(a)K^\top \in \mathbb{R}^{T\times N}. \tag{11}$$

For each token $t \in \{1, \dots, T\}$, let $\Omega_t \subset \{1, \dots, N\}$ be the Top-k index set of the row $S(a)_t$. Let $\alpha_t \in \Delta^{N-1}$ be the softmax weights over the selected scores, with zero mass outside Ω_t. The retrieved representation is

$$m_t = \sum_{j=1}^{N} \alpha_{t,j} V_j, \qquad t = 1, \dots, T. \tag{12}$$

Appendix D. Additional Method Details

This section provides the details summarized in the main methodology. We include the full DUS and MIDUS interleaving equations, the initialization rationale, the product-key lookup procedure used in HML, and the parameter and complexity analysis.

D.1. MIDUS Interleaving and Initialization

Let $f_0(x) = B_{L-1} \circ \cdots \circ B_0(x)$ be the pre-trained base model. DUS selects anchor indices $0 \le i_0 < \cdots < i_{D-1} \le L-1$ and interleaves D additional Transformer blocks $\widetilde{B}_0, \dots, \widetilde{B}_{D-1}$ after

the corresponding anchor blocks [44], yielding

$$f_{\text{DUS}}(x) = B_{L-1} \circ \cdots \circ \widetilde{B}_{D-1} \circ B_{i_{D-1}} \circ \cdots \circ \widetilde{B}_0 \circ B_{i_0} \circ \cdots \circ B_0(x). \tag{13}$$

MIDUS uses the same anchor indices, but inserts Memory blocks $M_0, \ldots, M_{D-1}$ before the corresponding pre-trained blocks:

$$f_{\text{MIDUS}}(x) = B_{L-1} \circ \cdots \circ B_{i_{D-1}} \circ M_{D-1} \circ \cdots \circ B_{i_0} \circ M_0 \circ \cdots \circ B_0(x). \tag{14}$$

The ordering in Eq. (14) is deliberate. Each Memory block is placed before the corresponding pre-trained block so that its attention parameters can be initialized from the following block. This aligns the attention-updated representation used by the memory layer with the context processed by the corresponding pre-trained block.

D.2. Initialization Details

Recall that a Memory block is defined as

$$M(x) = x + \text{Mem}(a(x)), \tag{15}$$

where $a(x)$ is the attention-updated representation and $\text{Mem}(a(x))$ is the retrieval-based residual correction. We initialize the value bank of each memory layer to zero. Therefore, at initialization,

$$\text{Mem}(a(x)) = 0, \qquad M(x) = x.$$

This makes every inserted Memory block an identity map at the start of training. As a result, the expanded model initially matches the pre-trained base model and provides a stable starting point for optimizing the newly inserted memory parameters.

The attention parameters inside each Memory block are initialized from the pre-trained Transformer block that follows it in Eq. (14). Since the Memory block is placed before this corresponding block, the copied attention module produces a contextualized representation aligned with the following pre-trained block. The memory layer then learns an additional retrieval-based correction through the residual path in Eq. (15).

D.3. Parameter and Complexity Analysis

We compare the lookup and value-parameter costs of the memory designs used in HML. A multi-head linear memory with N keys computes scores for all T tokens at cost

$$\mathcal{O}(TNd_q).$$

Product-key memory factorizes the key space into two sub-key banks of size n, with $N = n^2$ composite keys. It computes two n-way score matrices per head. Since each query slice has width $d_p = d_q/(2H)$, the total score-computation cost becomes

$$\mathcal{O}(Tnd_q),$$

which gives a $\sqrt{N}$ reduction over linear lookup when $N = n^2$. The number of key parameters also decreases from Nd_q to nd_q, because each head stores only two sub-key banks of size $n \times d_p$.

For value parameters, a naive head-wise value bank requires

$$HNd_h$$

parameters. HIVE instead uses a shared latent value table and head-specific projections, requiring

$$Nr + Hrd_h$$

parameters. When $r = d_h$, this becomes

$$Nd_h + Hd_h^2.$$

Thus, HIVE keeps head-specific value realization while reducing storage when $N \gg d_h$. The additional computation is only the projection of the retrieved latent value by W_h, rather than a full-dimensional value lookup for each head.

Appendix E. Experimental Details

E.1. Model and Evaluation Configuration

Following Cao et al. [5], we use Llama-3.2-1B and Llama-3.1-8B as base models, adding 8 and 16 blocks, respectively, for all DUS and MIDUS variants. Unless otherwise specified, each MIDUS-HML memory layer uses product-key memory [27] with $n = 64$ row and column sub-keys, yielding $N = n^2 = 4096$ composite keys per head. We use Top-k retrieval with $k = 4$ and set the HIVE latent dimension to $r = d_h$. Since both backbones have 32 attention heads, the inserted HML layers provide 1.05M and 2.10M head-specific addressable memory slots for Llama-3.2-1B and Llama-3.1-8B, respectively.

For general-purpose language and reasoning, we adopt the knowledge-centric suite used by Cao et al. [5]: ARC-Easy [8], LogiQA [29], Winogrande [37], CSQA [40], BoolQ [6], PIQA [4], and MMLU [17], using `lm-evaluation-harness` [14]. We also report WikiText perplexity [31]. These benchmarks use zero-shot evaluation. For math-domain reasoning, we evaluate GSM8K and GSM8K-CoT [9], MATH [18], and MathQA [1] with 5-shot evaluation. Retrieval-heavy and long-context evaluations use TriviaQA [22], NQ-Open [28], RACE [26], and LongBench [2], as reported in Appendix I. Boldface and underlining denote the best and second-best scores, respectively.

E.2. Training Configuration

We conduct `FineWeb-Edu` experiments on NVIDIA RTX 6000 Blackwell GPUs and `MathPile` experiments on NVIDIA H200 GPUs. Across all experiments, we use AdamW [30] with a cosine learning rate schedule, a warmup ratio of 0.1, and a weight decay sweep over $\{0, 10^{-6}, 10^{-2}\}$. All models are trained in `bfloat16` with `FlashAttention-2` [11]. For MIDUS-HML, we fix the learning rates of key and value parameters to the maximum learning rate without scheduling and set their weight decay to zero, following the sparse memory usage motivation of Lample et al. [27]. During CPT, following prior DUS studies [5, 44, 47], we train newly inserted blocks for DUS and MIDUS variants while freezing the pre-existing base blocks. For the base model and SOLAR [23], all parameters are updated. During SFT, all parameters are updated for every method, following prior work [5, 47].

For Llama-3.2-1B on `FineWeb-Edu`, we follow the hyperparameter configuration of prior work and apply the same search protocol to all baselines, including ours. During CPT, we search with a maximum learning rate of 10^{-4}, and during SFT, we fix the maximum learning rate to 10^{-5}. For Llama-3.1-8B on `FineWeb-Edu`, baseline results are taken from Cao et al. [5]; for MIDUS-HML, we fix the CPT learning rate to 5×10^{-5} and search the SFT learning rate over $\{10^{-6}, 5 \times 10^{-6}\}$. For `MathPile`, we use a learning rate of 10^{-4} for all methods and both backbones. Unless otherwise stated, we use a global batch size of 64 and a sequence length of 2,048 tokens.

E.3. Head-Importance Analysis

We use the gradient-based head importance score IS_h from Bansal et al. [3]. For examples $(x, y) \sim \mathcal{D}$, let $\ell(x, y)$ denote the autoregressive negative log-likelihood, and let $a_{h,t}([x \mid y])$ be the output of head h at position t in the sequence $[x \mid y]$. The score is

$$IS_h(\mathcal{D}) := \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\left| \sum_t a_{h,t}([x \mid y])^\top \frac{\partial \ell(x, y)}{\partial a_{h,t}([x \mid y])} \right| \right]. \tag{16}$$

We compute IS_h on PIQA in the zero-shot setting for the pre-trained Llama-3.2-1B model and MIDUS-HML after CPT on `FineWeb-Edu`. The within-layer variance is computed over heads within each layer.

Appendix F. Dataset Details

Following Cao et al. [5], we construct a 1.5B-token `FineWeb-Edu` subset by sampling from the CC-MAIN-2024-51 slice of `FineWeb-Edu` [34] after the base model's pre-training cut-off. For SFT, we use `Alpaca-GPT4` with 52k examples [35] and `Databricks-Dolly-15k` with 15k examples [10]. To construct the `MathPile` subset, we extract math-related text from five components of the original MATHPILE corpus [43], namely Textbooks, Wikipedia, ProofWiki, CommonCrawl, and StackExchange, excluding arXiv. This yields approximately 1.1B tokens.

Appendix G. Implementation Details

MIDUS preserves the interleaved depth-expansion structure of DUS, but replacing duplicated FFN branches with HML changes the computational profile of the inserted blocks. Unlike FFNs, which are dominated by regular matrix multiplications (GEMMs), HML involves sparse retrieval with discrete addressing, per-sample weighting, and index-based accumulation. Existing deep learning frameworks, such as PyTorch, provide highly optimized kernels for GEMM-heavy workloads, but sparse lookup and weighted accumulation are less directly optimized. As a result, naïve HML implementations may suffer from irregular memory access, limited hardware utilization, and atomic operation overhead in CUDA kernels.

In this section, we describe three implementation-level optimizations used for HML to reduce the gap between theoretical and practical efficiency. These optimizations are orthogonal to the architectural design of HML and operate at the kernel and execution level. Further low-level engineering, such as fully fused CUDA kernels for the entire HML block, may yield additional gains beyond those reported in this work.

G.1. Fused Cartesian Top-k for Efficient Generation

The head-wise product-key selection used in HML avoids scoring all $N = n^2$ composite keys by using a two-stage selection procedure. As described in Section 2.2, for each head h and token t, the model first selects Top-k row and column index sets $I_{h,t}$ and $J_{h,t}$, forms the candidate pair set $\widehat{\Omega}_{h,t} = I_{h,t} \times J_{h,t}$, and then selects the final Top-k pair set $\Omega_{h,t}$ using the additive pair scores $\sigma_{h,t}$. This row–column factorization reduces the pair-selection work from scoring all $N = n^2$ composite keys to scoring two n-way sub-key sets per head. In practice, however, this hierarchical procedure can introduce non-trivial kernel-launch overhead and irregular memory access on GPUs, especially when the number of query tokens and the batch size are small, as in autoregressive decoding.

To reduce this overhead, we use a *Fused Cartesian Top-k* strategy for short-query inference. While the main HML formulation uses the two-stage head-wise product-key selection, the inference path optionally replaces this hierarchical selection with a fused operation. For each head h and token t, we construct the Cartesian score grid $\mathcal{S}_{h,t} \in \mathbb{R}^{n \times n}$ by

$$(\mathcal{S}_{h,t})_{ij} = S^{\text{row}}_{h,t}(i) + S^{\text{col}}_{h,t}(j).$$

We then let $\Omega_{h,t} \subset \{1, \ldots, n\} \times \{1, \ldots, n\}$ be the Top-k pair set over the flattened grid $\mathcal{S}_{h,t}$. This is equivalent to scoring all n^2 product-key pairs, but it is executed as a single fused operation that better utilizes GPU parallelism.

This fused strategy sacrifices the asymptotic $\mathcal{O}(n)$ pair-selection cost and uses $\mathcal{O}(N)$ work per head and query token. However, n is fixed and modest in our setting, and generation typically uses few query tokens or small batches. In this regime, reduced kernel-launch overhead and improved hardware utilization can lower end-to-end latency. For longer sequences and larger batches, we use the standard two-stage product-key selection described in Section 2.2.

G.2. Head-wise Value Caching for Inference

During training, HIVE retrieves latent values from the shared latent value bank $\bar{V}$ and applies a head-specific value projection W_h, as defined in Eq. (7):

$$\bar{M}_{h,t} = \sum_{(i,j) \in \Omega_{h,t}} \alpha_{h,t}(i,j)\, \bar{V}_{\pi(i,j)} \in \mathbb{R}^r, \qquad M^{\text{HIVE}}_{h,t} = \bar{M}_{h,t} W_h \in \mathbb{R}^{d_h}.$$

This factorization shares latent value storage while allowing head-specific value realization, reducing the training-time parameter cost compared with maintaining an explicit value table of size $N \times d_h$ for each head.

During inference, model weights are fixed, so repeatedly applying W_h to retrieved latent values is redundant. We remove this overhead by constructing head-specific cached value tables only for inference. Before decoding, for each head h, we compute

$$V^{\text{cache}}_h = \bar{V} W_h \in \mathbb{R}^{N \times d_h}.$$

This is equivalent to applying the head-specific value projection to every row of $\bar{V}$. At runtime, the memory layer retrieves from V^{cache}_h using the selected pair indices and weights, yielding the same output as applying W_h after latent-value aggregation. This equivalence follows from the linearity of the value projection.

This optimization trades additional inference-time storage for lower decoding latency. It does not change the training-time parameterization or storage footprint of HIVE. The learned parameters remain factorized as $(\bar{V}, \{W_h\}_{h=1}^{H})$, and the cached tables are constructed on demand only for inference. All inference-side efficiency metrics reported in our experiments, including peak GPU memory during decoding, generation throughput, and prefill time, are measured with value caching enabled.

G.3. Atomic Contention Mitigation via Deduplicated Gradient Accumulation

A practical bottleneck in training memory layers arises in the backward pass of value aggregation. When multiple retrieval queries access the same value-table row, standard implementations such as `torch.nn.EmbeddingBag` or `scatter_add` may issue many concurrent updates to the same memory address. On GPUs, these updates are typically serialized through atomic operations, which can reduce effective memory bandwidth and slow training.

This issue is particularly relevant for our memory layers, which combine sparse retrieval, per-sample weights, and `bfloat16` values. Current PyTorch kernels do not provide an efficient backward path for this combination that matches our requirements. We therefore implement a custom autograd function based on deduplication and pre-aggregation. The scheme applies to any value aggregation of the form

$$y_p = \sum_{u=1}^{k} \lambda_{p,u} V_{\mathcal{I}_{p,u}}, \qquad p = 1, \ldots, P,$$

where P is the number of retrieval queries, k is the number of retrieved entries per query, $\mathcal{I}_{p,u} \in \{1, \ldots, N\}$ is a value-table index, and $\lambda_{p,u}$ is the corresponding retrieval weight.

Given the output gradient G_Y, the gradient for value row j is

$$G_V[j] = \sum_{(p,u):\, \mathcal{I}_{p,u}=j} \lambda_{p,u} G_Y[p].$$

Algorithm 1 computes this gradient by first forming per-retrieval contributions, then deduplicating value indices, and finally aggregating all contributions for each unique value row before writing to the global gradient table. This reduces repeated atomic updates to the same global memory location.

This backward scheme computes the same value-table gradient as the naive implementation, up to floating-point accumulation order, while reducing repeated global atomic updates to the same value-table row. Gradients with respect to the retrieval weights are computed separately following the standard weighted-embedding rule,

$$\frac{\partial \mathcal{L}}{\partial \lambda_{p,u}} = \left\langle G_Y[p], V_{\mathcal{I}_{p,u}} \right\rangle,$$

and are omitted from Algorithm 1 for brevity.

G.4. DUS Placement Policy

DUS placement policy is a key design choice that affects both CPT performance and training efficiency. The Llama-3.2-1B and Llama-3.1-8B base models consist of 16 and 32 Transformer layers, respectively. For all DUS and MIDUS variants, we insert 8 and 16 additional modules into these backbones, yielding final depths of 24 and 48 blocks.

Algorithm 1 Deduplicated Backward Pass for Value Aggregation

Require: Output gradient $G_Y \in \mathbb{R}^{P \times d_v}$, value indices $\mathcal{I} \in \{1, \ldots, N\}^{P \times k}$, retrieval weights $\Lambda \in \mathbb{R}^{P \times k}$, value-table size N

Step 1: Form per-retrieval gradient contributions
$G_{\text{exp}} \leftarrow G_Y.\text{unsqueeze}(1) \odot \Lambda.\text{unsqueeze}(-1)$ $\{G_{\text{exp}} \in \mathbb{R}^{P \times k \times d_v}\}$
$G_{\text{flat}} \leftarrow G_{\text{exp}}.\text{reshape}(Pk, d_v)$
$\mathcal{I}_{\text{flat}} \leftarrow \mathcal{I}.\text{reshape}(Pk)$
Step 2: Deduplicate retrieved value indices
$\mathcal{I}_{\text{uniq}}, \eta \leftarrow \text{unique}(\mathcal{I}_{\text{flat}}, \text{return_inverse} = \text{True})$
Step 3: Pre-aggregate contributions for each unique index
$U \leftarrow |\mathcal{I}_{\text{uniq}}|$
$G_{\text{uniq}} \leftarrow \text{zeros}(U, d_v)$
$G_{\text{uniq}}.\text{index_add_}(0, \eta, G_{\text{flat}})$
Step 4: Write one aggregated gradient per touched value row
$G_V \leftarrow \text{zeros}(N, d_v)$
$G_V[\mathcal{I}_{\text{uniq}}] \leftarrow G_{\text{uniq}}$

Ensure: G_V

We distinguish between the underlying anchor schedule and the final-stack positions of inserted modules. DUS and MIDUS can use the same anchor schedule, but their inserted modules appear at different final-stack indices because they follow different insertion rules. In standard interleaved DUS, each duplicated Transformer block is placed after its anchor block. In MIDUS, as in Eq. (2), each Memory block is placed before the corresponding pre-trained block. Therefore, the *Llama Pro* placement used for DUS baselines and the *Distributed* placement used for MIDUS correspond to the same underlying anchor schedule, but their inserted-module indices are shifted in the final up-scaled stack. Although *Llama Pro* and *Distributed* have different final-stack indices, they correspond to the same underlying anchor schedule under different insertion rules.

We consider four final-stack placement patterns:

- *Top-heavy*: inserted modules are concentrated in the upper half of the stack, alternating with pre-trained blocks, following the recipe of [5].
- *Llama Pro*: inserted Transformer blocks are placed every three positions starting from index 2 in the final stack, following the DUS insertion rule of [44].
- *Distributed*: Memory blocks are placed every three positions starting from index 1 in the final stack. This is the MIDUS counterpart of the Llama Pro anchor schedule, shifted by the before-anchor insertion rule in Eq. (2).
- *Bottom-heavy*: inserted modules are concentrated in the lower half of the stack, alternating with pre-trained blocks.

The following lists report the resulting positions of the inserted modules in the final up-scaled stack, using 0-indexing. For DUS baselines, the listed indices indicate the positions of duplicated Transformer blocks. For MIDUS variants, the listed indices indicate the positions of Memory blocks, with the corresponding pre-trained block placed immediately after each Memory block under Eq. (2). Thus, although *Llama Pro* and *Distributed* have different final-stack indices, they instantiate the same underlying anchor schedule under different insertion rules.

Up-scaled Llama-3.2-1B with 24 blocks

- *Top-heavy*: {8, 10, 12, 14, 16, 18, 20, 22}
- *Llama Pro*: {2, 5, 8, 11, 14, 17, 20, 23}
- *Distributed*: {1, 4, 7, 10, 13, 16, 19, 22}
- *Bottom-heavy*: {0, 2, 4, 6, 8, 10, 12, 14}

Up-scaled Llama-3.1-8B with 48 blocks

- *Top-heavy*: {16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46}
- *Llama Pro*: {2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47}
- *Distributed*: {1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46}
- *Bottom-heavy*: {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30}

When considering only the placement policy, *Top-heavy* is the most training-efficient design under our expanded-module training setup, since placing inserted modules closer to the output reduces the backward computation and activation memory associated with these modules. *Llama Pro* is also more efficient than *Distributed* placement for the same reason. In contrast, MIDUS-HML achieves its best performance under the *Distributed* policy, yet still provides substantially better overall efficiency than DUS baselines. MIDUS-HML can also adopt the *Top-heavy* policy to further improve efficiency; even in this extreme setting, its performance remains comparable to, or better than, that of DUS baselines. See Table 12 and Appendix L for further ablation studies.

Appendix H. SFT results on `Databricks-Dolly-15k`

Table 4: SFT results on the `Databricks-Dolly-15k` [10] with Llama-3.2-1B.

		Llama-3.2-1B								
		PPL ↓	Zero-shot Accuracy ↑							
	Method	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
SFT	Base	13.07	68.64	21.35	60.30	27.52	63.27	**75.24**	30.49	49.55
	SOLAR	13.19	**69.40**	23.04	59.04	27.52	60.03	75.14	30.77	49.28
	Llama Pro	12.47	68.10	**23.81**	60.54	41.93	63.70	**75.24**	34.03	52.48
	LESA	11.77	66.46	23.66	60.77	49.06	64.98	74.86	38.22	54.00
	OpT-DeUS	11.78	67.21	**23.81**	61.48	**49.14**	63.76	75.19	37.83	54.06
	Avg-DeUS	12.10	67.51	22.73	60.30	46.60	65.02	74.37	36.79	53.33
	MIDUS-HML	**11.72**	66.62	22.12	**61.72**	48.65	**66.51**	75.14	**38.59**	**54.19**

Appendix I. Additional Evaluation on Retrieval-Heavy and Long-Context Benchmarks

To test whether the gains of MIDUS-HML extend beyond the main knowledge-centric benchmark suite, we further evaluate retrieval-heavy and long-context tasks. We use the same CPT-trained Llama-3.2-1B models from the FineWeb-Edu experiments. For retrieval-heavy evaluation, we use TriviaQA [22], NQ-Open [28], and RACE [26] under 5-shot prompting. For long-context evaluation, we use LongBench [2] and report results over six categories.

Table 5: Retrieval-heavy evaluation of CPT-trained Llama-3.2-1B models.

Method	TriviaQA	NQ-Open	RACE	Average
Base	30.38	9.72	**38.28**	26.13
Llama Pro	34.33	10.28	37.99	27.53
OpT-DeUS	39.02	12.11	38.18	29.77
Avg-DeUS	38.25	11.75	37.42	29.14
MIDUS-HML	**39.75**	**12.24**	37.51	**29.83**

Table 6: Long-context evaluation of CPT-trained Llama-3.2-1B models on LongBench. Overall denotes the task-level average over the six categories.

Method	Single-Doc QA	Multi-Doc QA	Summarization	Few-shot	Synthetic	Code	Overall
Base	10.45	7.40	19.10	55.68	**4.15**	27.47	21.32
Llama Pro	10.52	7.32	21.09	58.08	3.35	35.70	23.07
OpT-DeUS	**10.92**	7.97	22.94	60.60	3.79	47.57	25.63
Avg-DeUS	10.66	7.74	21.04	60.38	3.54	46.96	25.03
MIDUS-HML	10.60	**8.48**	**23.23**	**63.52**	2.62	**54.64**	**27.00**

Appendix J. CPT results on `MathPile`.

`MathPile` is compiled from diverse mathematical corpora [43], and we perform CPT with Llama-3.2-1B and Llama-3.1-8B backbones. Table 7 reports 5-shot accuracy on math-domain benchmarks. Across both backbones, MIDUS-HML achieves the best average performance and remains competitive or better on most benchmarks. These results suggest that head-conditioned memory supports math-domain adaptation, including multi-step reasoning tasks.

Table 7: Performance of CPT on `MathPile` for Llama-3.2-1B and Llama-3.1-8B models.

	Llama-3.2-1B (5-shot Accuracy ↑)					Llama-3.1-8B (5-shot Accuracy ↑)				
Method	GSM8K	GSM8K–CoT	MATH	MathQA	Average	GSM8K	GSM8K–CoT	MATH	MathQA	Average
Base	4.32	4.93	5.66	29.88	11.20	35.33	33.36	11.42	37.99	29.53
Llama Pro	5.61	4.40	5.64	30.62	11.57	41.93	41.77	13.10	40.87	34.42
OpT-DeUS	6.07	6.22	**5.80**	30.95	12.26	**49.36**	50.04	13.84	**42.58**	38.96
MIDUS-HML	**6.60**	**6.60**	5.62	**31.39**	**12.55**	48.90	**51.78**	**14.34**	**42.58**	**39.40**

Appendix K. Ablation Studies

K.1. Ablation study on the number of memory slots

Table 8: Ablation study on the number of total memory slots and its effect on performance. The 66K, 262K, and 1M settings denote total addressable slots across inserted HML blocks and memory heads; each head uses $N = n^2$ composite keys.

	PPL ↓	Zero-shot Accuracy ↑							
# Memories	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
66K ($n = 16$)	**11.63**	66.12	22.89	60.06	46.27	65.02	74.81	36.74	53.13
262K ($n = 32$)	11.64	65.95	**23.35**	60.85	**47.17**	**65.44**	74.76	36.41	53.42
1M ($n = 64$)	11.64	**66.16**	23.20	**61.56**	46.27	65.29	**75.08**	**36.91**	**53.50**

In Table 8, we vary the total number of product-key memories from tens of thousands to one million, i.e., n=16, 32, 64 in the PKM factorization. Average zero-shot accuracy rises steadily with capacity, while PPL stays essentially flat. We hypothesize that a larger number of memory slots mainly helps tasks that benefit from retrieving facts or patterns, without disrupting the core language-modeling behavior. MIDUS gains an additional axis to scale quality, and rather than adding more Transformer blocks, we can improve results by increasing the number of memory slots.

K.2. Ablation study on Top-k

In Table 9, we vary the number of retrieved memory slots per token from k=1 to k=8 while keeping the memory size and all other hyperparameters fixed. Average zero-shot accuracy improves steadily as k increases from 1 to 4, with modest or no further gains at k=8, and PPL remains essentially unchanged across all settings. We therefore use k=4 as the default Top-k setting in all other experiments.

Table 9: Effect of the Top-k value k for Llama-3.2-1B with MIDUS-HML.

	Train Time ↓	TTFT ↓	PPL ↓	Zero-shot Accuracy ↑							
Top-k	s/iter	ms	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
1	**4.31**	9.33	11.67	65.74	21.97	60.54	46.19	64.95	74.86	36.95	53.03
2	4.41	9.40	**11.63**	65.87	**23.20**	60.69	46.11	65.11	74.54	37.04	53.22
4	4.54	**9.31**	11.64	**66.16**	**23.20**	**61.56**	46.27	**65.29**	**75.08**	36.91	**53.50**
8	4.81	10.71	11.64	66.04	23.04	60.46	**46.93**	64.98	75.03	**37.11**	53.37

K.3. Ablation study on the learning rate and weight decay

Table 10: Effect of learning rate (LR) and weight decay (WD) on the performance of Llama-3.2-1B with MIDUS-HML

		PPL ↓	Zero-shot Accuracy ↑							
LR	WD	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
1e-4	1e-6	11.64	66.16	**23.20**	**61.56**	46.27	65.29	**75.08**	36.91	**53.50**
1e-4	0	11.64	66.25	21.97	60.77	46.60	65.32	75.03	36.99	53.27
1e-4	1e-2	**11.62**	65.78	23.04	59.91	45.45	**65.35**	74.81	**37.25**	53.09
1e-5	1e-6	11.64	65.57	21.51	61.01	46.85	63.58	74.48	37.12	52.87
5e-5	1e-6	11.63	**66.46**	22.89	60.14	**47.58**	64.86	74.54	37.23	53.39

In Table 10, we vary the learning rate and weight decay over a range of reasonable values while keeping all other settings fixed. Across these configurations, PPL remains tightly clustered around 11.62–11.64 and the average zero-shot accuracy stays in a narrow band, indicating that MIDUS-HML is relatively robust to these optimization hyperparameters.

K.4. Ablation study on the Structure of HML

In Table 11, we evaluate four HML variants. The first applies BatchNorm1d to the queries, following PKM [27]. The second applies LayerNorm to the queries. The third adds an internal attention residual, $a' = x + \mathrm{Attn}'(\mathrm{LN}(x))$. The fourth enables the attention output projection. Neither query-normalized variant improves over the default HML, suggesting that normalization may disturb the score structure used for Top-k selection. The internal residual also degrades performance, which

Table 11: Ablation study on the structure of HML.

	PPL ↓	Zero-shot Accuracy ↑							
Method	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
MIDUS-HML	**11.64**	66.16	23.20	**61.56**	46.27	**65.29**	75.08	36.91	53.50
w/ BatchNorm1d	11.74	**66.54**	22.43	60.77	**47.34**	64.80	74.92	36.94	53.39
w/ LayerNorm	11.68	66.08	22.43	**61.56**	46.68	65.17	75.19	**37.34**	53.49
w/ Residual	**11.64**	66.04	22.89	61.48	46.36	65.17	**75.35**	36.75	53.43
w/ Output Projection	**11.64**	66.12	**23.66**	61.09	46.52	65.23	74.86	37.07	**53.51**

indicates that directly using separated head outputs may provide more suitable memory queries than mixing them with the input representation. Although adding the output projection gives a marginal gain, we retain the projection-free design as the default because it preserves head-separated queries and avoids the additional projection cost.

K.5. Ablation study on DUS placement policy.

Table 12: MIDUS-HML under different DUS placement policies. Peak GPU Memory and Time are training requirements. *Top-heavy* places expanded blocks toward the top, *Distributed* interleaves, and *Bottom-heavy* places them near the input.

	GB ↓	s/iter ↓	PPL ↓	Zero-shot Accuracy ↑							
DUS Policy	GPU Memory	Time	Wiki	ARC	LogiQA	Wino	CSQA	BoolQ	PIQA	MMLU	Average
Top-heavy	**24.1**	**3.89**	11.63	66.12	22.27	61.25	**46.44**	64.28	74.92	36.23	53.07
Distributed	28.0	4.56	11.64	**66.16**	**23.20**	**61.56**	46.27	**65.29**	75.08	**36.91**	**53.50**
Bottom-heavy	28.6	4.67	**11.60**	66.12	21.81	60.30	45.05	65.05	**75.30**	36.35	52.85

Table 12 shows the efficiency and performance of MIDUS-HML with Llama-3.2-1B under different DUS placement policies. *Top-heavy* placement, as used by OpT-DeUS [5], is most efficient in memory and time, since many frozen lower blocks can be skipped during backpropagation. *Bottom-heavy* placement slightly improves PPL but does not maximize average zero-shot accuracy. *Distributed* policy yields the best overall accuracy.

Appendix L. Additional efficiency experiments

We measure all efficiency metrics on a single NVIDIA RTX 6000 Blackwell GPU. For training-related metrics, Table 2 and Figure 2 use a global batch size of 16 with 16 gradient-accumulation steps. Unless otherwise noted, the sequence length is fixed to 2048 in these efficiency experiments. For inference-related metrics, we report peak GPU memory and generation throughput measured during the generation stage.

Figure 4 further examines efficiency as a function of the global batch size. We fix the gradient-accumulation steps to 16 and the sequence length to 2048, and vary only the global batch size. For Llama-3.1-8B, we omit the training-time entries for Llama Pro and OpT-DeUS at a global batch size of 64, since they exceed the available GPU memory on a single RTX 6000 Blackwell GPU. Across Figure 4 and 5, which cover a range of batch sizes and sequence lengths, MIDUS-HML (*Dist.*) consistently achieves training times that are faster than or comparable to those of the DUS baselines. Notably, MIDUS-HML (*Top.*), which adopts the same DUS placement policy as OpT-DeUS, is

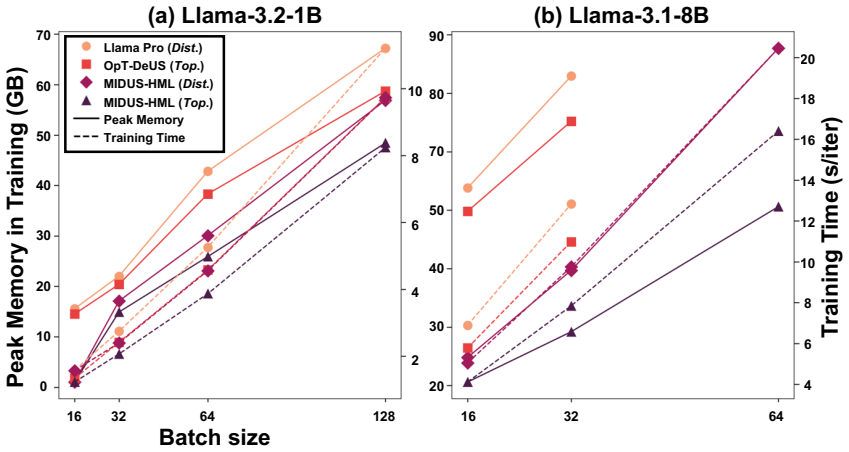

Figure 4: Efficiency of DUS baselines and MIDUS-HML as a function of batch size. *Dist.* denotes the *Distributed* and *Top.* the *Top-heavy* DUS placement policy.

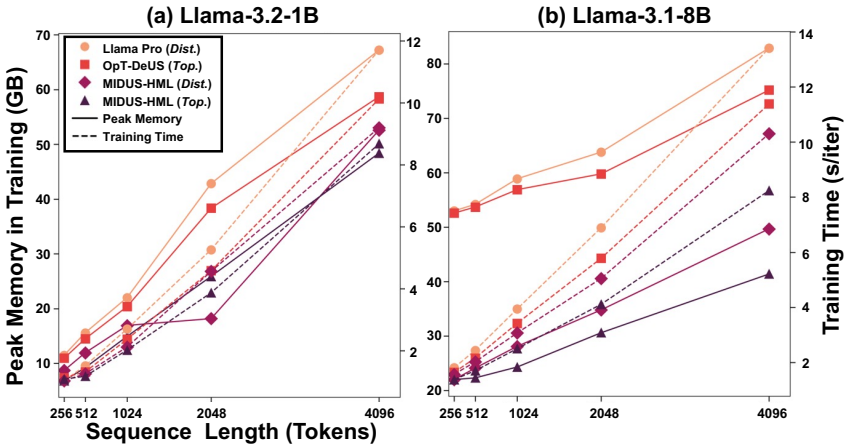

Figure 5: Efficiency of DUS baselines and MIDUS-HML as a function of sequence length. *Dist.* denotes the *Distributed* and *Top.* the *Top-heavy* DUS placement policy.

substantially faster than OpT-DeUS. In all cases, MIDUS-HML also requires less peak GPU memory than any of the DUS baselines. The Llama-3.1-8B results suggest that the efficiency advantage of MIDUS-HML remains pronounced on a larger and deeper backbone.

Appendix M. Structural Analysis of Head-wise Value Memory

This appendix analyzes the head-wise value parameterization used in MIDUS-HML under fixed retrieval coefficients. We focus on the following architectural question. When the desired residual corrections are heterogeneous across heads within a common retrieval span, when does head-specific value capacity reduce the empirical correction loss compared to a shared-realized-value alternative,

and when can HIVE preserve this advantage through shared latent storage with head-specific value realization?

M.1. Local Correction Setup

Consider fixed attention-head outputs

$$a' = [a_1 \mid \cdots \mid a_H] \in \mathbb{R}^{T\times d}, \qquad a_h \in \mathbb{R}^{T\times d_h},$$

where H is the number of attention heads, $d = Hd_h$, and $[H] = \{1, \ldots, H\}$. Unless specified otherwise, $\|\cdot\|$ denotes the Euclidean norm for vectors and the Frobenius norm for matrices.

We analyze value updates under fixed retrieval coefficients. For each head h, we collect the sparse Top-k retrieval-weight vectors $\alpha_{h,t} \in \Delta^{N-1}$ into the retrieval-weight matrix

$$A_h = [\alpha_{h,1}, \ldots, \alpha_{h,T}]^\top \in \mathbb{R}^{T\times N},$$

where N is the number of composite memory slots per head. Let $Y_h \in \mathbb{R}^{T\times d_h}$ denote the desired residual correction for head h, and let

$$Y = [Y_1 \mid \cdots \mid Y_H] \in \mathbb{R}^{T\times d}.$$

For a head-partitioned correction

$$F(a') = [F_1(a') \mid \cdots \mid F_H(a')],$$

define the empirical correction loss

$$\mathcal{L}(F) = \frac{1}{2}\sum_{h=1}^{H} \|F_h(a') - Y_h\|^2.$$

We compare two value parameterizations under the same retrieval-weight matrices $\{A_h\}_{h=1}^{H}$.

Shared realized value bank. The shared-realized-value family uses a single realized value bank $V_{\text{sh}} \in \mathbb{R}^{N\times d_h}$ for all heads and is defined by

$$F_h(a') = A_h V_{\text{sh}}.$$

Head-specific value banks. The head-specific-value family assigns an independent value bank $V_h \in \mathbb{R}^{N\times d_h}$ to each head and is defined by

$$F_h(a') = A_h V_h.$$

M.2. Loss Gap from Head-specific Value Capacity

For fixed retrieval, both families are linear in their value parameters. The optimal head-specific-value loss is

$$\mathcal{L}^\star_{\text{Ind}} = \inf_{\{V_h\}_{h=1}^{H}} \mathcal{L}(F) = \frac{1}{2}\sum_{h=1}^{H} \|(I_T - \Pi_{A_h})Y_h\|^2,$$

where Π_{A_h} is the orthogonal projector onto $\mathrm{col}(A_h) \subseteq \mathbb{R}^T$. In contrast, the shared-realized-value family must use one common bank V_{sh} to fit all heads simultaneously.

Proposition 2 *The shared-realized-value optimum is*

$$\mathcal{L}^{\star}_{\text{Share}} = \inf_{V_{sh}} \mathcal{L}(F) = \mathcal{L}^{\star}_{\text{Ind}} + \frac{1}{2}\sum_{h=1}^{H} \|A_h V^{\star}_{sh} - \Pi_{A_h} Y_h\|^2, \tag{17}$$

where

$$V^{\star}_{sh} = \Big(\sum_{h=1}^{H} A_h^{\top} A_h\Big)^{\dagger} \sum_{h=1}^{H} A_h^{\top} Y_h$$

is a minimum-norm optimal shared realized value bank, and $\dagger$ *denotes the Moore–Penrose pseudoinverse. Define* $\Delta_{\text{Share}} := \mathcal{L}^{\star}_{\text{Share}} - \mathcal{L}^{\star}_{\text{Ind}}$*. Then* $\Delta_{\text{Share}} \geq 0$*. Moreover,* $\Delta_{\text{Share}} = 0$ *if and only if there exists a single* $V_{sh} \in \mathbb{R}^{N \times d_h}$ *such that* $A_h V_{sh} = \Pi_{A_h} Y_h$ *for all* $h \in [H]$*. When retrieval is identical across heads, that is,* $A_h = A$ *for all* h*, the gap reduces to*

$$\Delta_{\text{Share}} = \frac{1}{2}\sum_{h=1}^{H} \|\Pi_A (Y_h - \bar{Y})\|^2, \qquad \bar{Y} = \frac{1}{H}\sum_{h=1}^{H} Y_h.$$

Thus, in this special case, the shared-realized-value penalty is the sum of squared deviations of head-specific correction targets within the common retrieval span $\mathrm{col}(A)$*.*

Proof The head-specific optimum follows by solving, independently for each head,

$$\inf_{V_h} \frac{1}{2}\|A_h V_h - Y_h\|^2,$$

which gives the projection residual

$$\frac{1}{2}\|(I_T - \Pi_{A_h}) Y_h\|^2.$$

For the shared-realized-value family, the optimization is

$$\inf_{V_{\text{sh}}} \frac{1}{2}\sum_{h=1}^{H} \|A_h V_{\text{sh}} - Y_h\|^2.$$

The normal equation is

$$\Big(\sum_{h=1}^{H} A_h^{\top} A_h\Big) V_{\text{sh}} = \sum_{h=1}^{H} A_h^{\top} Y_h,$$

and the minimum-norm solution is

$$V^{\star}_{\text{sh}} = \Big(\sum_{h=1}^{H} A_h^{\top} A_h\Big)^{\dagger} \sum_{h=1}^{H} A_h^{\top} Y_h.$$

For each head, $A_h V^{\star}_{\text{sh}}$ and $\Pi_{A_h} Y_h$ both have columns in $\mathrm{col}(A_h)$, whereas $(I_T - \Pi_{A_h}) Y_h$ has columns in $\mathrm{col}(A_h)^{\perp}$. Hence $A_h V^{\star}_{\text{sh}} - \Pi_{A_h} Y_h$ is orthogonal to $(I_T - \Pi_{A_h}) Y_h$ column-wise. Since

$$A_h V^{\star}_{\text{sh}} - Y_h = (A_h V^{\star}_{\text{sh}} - \Pi_{A_h} Y_h) - (I_T - \Pi_{A_h}) Y_h,$$

the Pythagorean identity yields

$$\|A_h V_{\text{sh}}^\star - Y_h\|^2 = \|A_h V_{\text{sh}}^\star - \Pi_{A_h} Y_h\|^2 + \|(I_T - \Pi_{A_h}) Y_h\|^2.$$

Applying this decomposition to the shared objective gives Eq. (17). The non-negativity and equality condition follow immediately.

If $A_h = A$ for all h, then

$$V_{\text{sh}}^\star = (A^\top A)^\dagger A^\top \bar{Y}, \qquad A V_{\text{sh}}^\star = \Pi_A \bar{Y}.$$

Substituting this into the gap expression gives

$$\frac{1}{2} \sum_{h=1}^{H} \|\Pi_A \bar{Y} - \Pi_A Y_h\|^2 = \frac{1}{2} \sum_{h=1}^{H} \|\Pi_A (Y_h - \bar{Y})\|^2.$$

■

Proposition 2 quantifies the penalty of forcing all heads to use a single realized value bank. The gap vanishes only when a common bank can match every projected head-specific target.

M.3. HIVE as Shared Latent Storage with Head-specific Value Projection

We now consider HIVE, which shares a latent value table $\bar{V} \in \mathbb{R}^{N \times r}$ while allowing each head to use its own value projection $W_h \in \mathbb{R}^{r \times d_h}$, so that the head-specific bank is represented as $\bar{V} W_h$.

The next proposition shows that if an optimal head-specific solution is approximately generated from a shared latent value bank with head-specific value projections, then HIVE inherits the independent head-specific loss up to a controlled residual penalty.

Proposition 3 *Let $\{V_h^\star\}_{h=1}^H \in \arg\min_{\{V_h\}_{h=1}^H} \mathcal{L}(F)$ be a selected optimal head-specific solution. Suppose there exist $\bar{V} \in \mathbb{R}^{N \times r}$, $W_h \in \mathbb{R}^{r \times d_h}$, and $R_h \in \mathbb{R}^{N \times d_h}$ such that*

$$V_h^\star = \bar{V} W_h + R_h \qquad \textit{for all } h \in [H],$$

and $\sum_{h=1}^H \|R_h\|^2 \le \varepsilon$, where $r \le d_h$. Let $\rho = \max_{h \in [H]} \|A_h\|_{op}^2$, where $\|\cdot\|_{op}$ denotes the operator norm. Then the corresponding HIVE correction $\widehat{F}_h(a') = A_h \bar{V} W_h$ satisfies

$$\mathcal{L}(\widehat{F}) \le \mathcal{L}_{\text{Ind}}^\star + \frac{\rho}{2} \varepsilon.$$

Proof Define

$$E_h := V_h^\star - \bar{V} W_h.$$

By assumption, $E_h = R_h$. Let

$$F_h^\star(a') := A_h V_h^\star.$$

The corresponding HIVE correction satisfies

$$\widehat{F}_h(a') - F_h^\star(a') = -A_h E_h.$$

By the operator-Frobenius bound,

$$\|\widehat{F}_h(a') - F_h^\star(a')\|^2 = \|A_h E_h\|^2 \le \|A_h\|_{\text{op}}^2 \|E_h\|^2 \le \rho \|E_h\|^2.$$

Since $V_h^\star$ is optimal for the head-specific least-squares problem, its fitted value is the column-wise projection

$$F_h^\star(a') = A_h V_h^\star = \Pi_{A_h} Y_h.$$

Hence $F_h^\star(a') - Y_h$ has columns in $\text{col}(A_h)^\perp$, while $A_h E_h$ has columns in $\text{col}(A_h)$. These two terms are orthogonal, and therefore

$$\|\widehat{F}_h(a') - Y_h\|^2 = \|F_h^\star(a') - Y_h\|^2 + \|\widehat{F}_h(a') - F_h^\star(a')\|^2.$$

Summing over heads gives

$$\mathcal{L}(\widehat{F}) \le \mathcal{L}_{\text{Ind}}^\star + \frac{\rho}{2} \sum_{h=1}^{H} \|E_h\|^2.$$

Since $E_h = R_h$ and $\sum_{h=1}^{H} \|R_h\|^2 \le \varepsilon$, we obtain

$$\mathcal{L}(\widehat{F}) \le \mathcal{L}_{\text{Ind}}^\star + \frac{\rho}{2}\varepsilon.$$

■

We now prove Theorem 1.

Theorem 1 *In the fixed-retrieval setup, suppose $A_h = A$ for all $h \in [H]$, and let Π_A be the orthogonal projection onto the column space of A. Assume that there exists an optimal head-specific solution $\{V_h^\star\}_{h=1}^H$ that admits a shared latent factorization $V_h^\star = \bar{V} W_h$, where $\bar{V} \in \mathbb{R}^{N \times r}$, $W_h \in \mathbb{R}^{r \times d_h}$, and $r \le d_h$. If the projected targets are non-identical across heads, i.e., $\Pi_A Y_h \ne \Pi_A Y_{h'}$ for some $h \ne h'$, then the corresponding HIVE correction $\widehat{F}_h(a') = A\bar{V}W_h$ satisfies $\mathcal{L}(\widehat{F}) = \mathcal{L}_{\text{Ind}}^\star < \mathcal{L}_{\text{Share}}^\star$.*

Proof [Proof] By the assumed shared latent factorization,

$$V_h^\star = \bar{V} W_h \qquad \text{for all } h \in [H].$$

This is a special case of Proposition 3 with

$$R_h = 0 \qquad \text{for all } h \in [H],$$

and hence $\varepsilon = 0$. Therefore,

$$\mathcal{L}(\widehat{F}) \le \mathcal{L}_{\text{Ind}}^\star.$$

Since every HIVE correction corresponds to a valid head-specific value choice with value banks $\bar{V}W_h$, it also belongs to the head-specific-value family. Therefore,

$$\mathcal{L}(\widehat{F}) \ge \mathcal{L}_{\text{Ind}}^\star.$$

Thus,

$$\mathcal{L}(\widehat{F}) = \mathcal{L}_{\text{Ind}}^\star.$$

It remains to compare this value with the shared-realized-value optimum. Under the identical-retrieval assumption $A_h = A$ for all h, Proposition 2 gives

$$\mathcal{L}^\star_{\text{Share}} = \mathcal{L}^\star_{\text{Ind}} + \frac{1}{2}\sum_{h=1}^{H} \|\Pi_A(Y_h - \bar{Y})\|^2, \qquad \bar{Y} = \frac{1}{H}\sum_{h=1}^{H} Y_h.$$

The assumption

$$\Pi_A Y_h \neq \Pi_A Y_{h'} \qquad \text{for some } h \neq h'$$

implies that not all projected targets are equal to their mean $\Pi_A \bar{Y}$. Hence

$$\sum_{h=1}^{H} \|\Pi_A(Y_h - \bar{Y})\|^2 > 0.$$

Therefore,

$$\mathcal{L}^\star_{\text{Share}} > \mathcal{L}^\star_{\text{Ind}}.$$

Combining this with $\mathcal{L}(\widehat{F}) = \mathcal{L}^\star_{\text{Ind}}$ yields

$$\mathcal{L}(\widehat{F}) = \mathcal{L}^\star_{\text{Ind}} < \mathcal{L}^\star_{\text{Share}}.$$

■

The theorem isolates a structured heterogeneity regime in which HIVE matches the head-specific optimum. The condition $\Pi_A Y_h \neq \Pi_A Y_{h'}$ for some $h \neq h'$ means that different heads require different projected corrections even under a common retrieval span, making a single shared realized value bank restrictive. Meanwhile, the shared latent factorization $V_h^\star = \bar{V} W_h$ is the zero-residual instance of Proposition 3, where the optimal head-specific banks share a common latent value table. HIVE matches this pattern by sharing latent storage while allowing head-specific value realization, thereby separating storage-level sharing from head-specific correction.

Connection to FFN-style correction blocks. Prior work interprets Transformer FFNs as implicit key–value memories, where the first linear map and nonlinearity induce key-like activations and the second linear map provides value-like outputs [15]. Under this view, duplicating FFN branches in DUS can be interpreted as increasing memory-like value capacity through a shared block-level correction over the full hidden representation.

The analysis above should not be read as a formal reduction of arbitrary FFNs to the shared-realized-value family, since FFNs include nonlinear activations and dense feature mixing. Instead, it isolates a simplified fixed-retrieval regime that reflects the distinction between shared realized values and head-specific value realization. HML with HIVE shares a latent value table while allowing each head to realize this shared storage through its own value projection. The theorem shows that this separation can matter when heads share the same retrieval-weight matrix but require different projected corrections.