UNIVERSAL YOCO FOR EFFICIENT DEPTH SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

The rise of test-time scaling has remarkably boosted the reasoning and agentic proficiency of Large Language Models (LLMs). Yet, standard Transformers struggle to scale inference-time compute efficiently, as conventional looping strategies suffer from high computational overhead and a KV cache that inflates alongside model depth. We present Universal YOCO (YOCO-U), which combines the YOCO decoder-decoder architecture with recursive computation to achieve a synergistic effect greater than either alone. Built on the YOCO framework, YOCO-U implements a Universal Self-Decoder that performs multiple iterations via parameter sharing, while confining the iterative process to shallow, efficient-attention layers. This combination yields a favorable capability-efficiency tradeoff that neither YOCO nor recursion achieves independently. The YOCO architecture provides a constant global KV cache and linear pre-filling, while partial recursion enhances representational depth with limited overhead. Together, YOCO-U improves token utility and scaling behavior while maintaining efficient inference. Empirical results confirm that YOCO-U remains highly competitive in general and long-context benchmarks, demonstrating that the integration of efficient-attention architectures and recursive computation is a promising direction for scalable LLMs.

1 INTRODUCTION

Large Language Models (LLMs) have fundamentally transformed the landscape of artificial intelligence, particularly through the emergence of test-time scaling (Jaech et al., 2024; Guo et al., 2025). These techniques have significantly augmented the reasoning capabilities of models, enabling them to tackle intricate, multi-step problems that were previously beyond their reach. By shifting towards more intensive computation during inference, LLMs have demonstrated a heightened proficiency in autonomous planning and executing complex tasks in real-world environments. This paradigm shift underscores the critical importance of effectively scaling computation.

Despite these advances, current LLM architectures face significant limitations in efficiently supporting such computational scaling. First, from the perspective of inference-time scaling, relying solely on post-training strategies remains relatively inefficient, as it often fails to fully capitalize on the foundational knowledge and expressive depth inherent in the pre-training stage. Second, while implementing looping mechanisms within standard Transformers can theoretically extend computational depth, it incurs prohibitive costs, as the computational complexity remains high, and the memory footprint of the Key-Value (KV) cache grows linearly with the increasing depth.

In this work, we present Universal YOCO (YOCO-U), which combines the YOCO decoder-decoder architecture with recursive computation, showing that the two techniques are synergistic and yield benefits beyond what either provides alone. YOCO-U builds on the "You Only Cache Once" (YOCO) framework (Sun et al., 2024), which splits the model into a *Self-Decoder* and a *Cross-Decoder*. The Self-Decoder processes input tokens with efficient attention (e.g., sliding-window attention) and produces a compact global KV cache. The Cross-Decoder then reuses this shared cache via cross-attention across all its layers, avoiding the per-layer KV caches of standard Transformers and yielding substantial memory savings. In YOCO-U, the static Self-Decoder is replaced with a Universal Self-Decoder that iterates computation for multiple steps using shared parameters, enhancing representational capacity through recursive depth without increasing the parameter count.

The architectural innovation of YOCO-U offers distinct advantages over the standard Universal Transformer (Dehghani et al., 2018). While UT applies recursive computation to the entire network,

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.

Table 1: Inference complexity comparison. N, L, D, T denote sequence length, number of layers, hidden dimension, and loop iterations. W is the local window size in efficient self-attention. In YOCO-U, the global KV cache ($\mathcal{O}(ND)$) is independent of T. Since $W \ll N$, the additional overhead is negligible for long sequences.

Model	KV Cache Memory	Prefilling Time	Decoding Time
Transformer	$\mathcal{O}(LND)$	$\mathcal{O}(LN^2D)$	$\mathcal{O}(LND)$
YOCO	$\mathcal{O}((N+WL)D)$	$\mathcal{O}(\frac{L}{2}ND)$	$\mathcal{O}(\frac{L}{2}(N+W)D)$
Loop / Universal Transformer	$\mathcal{O}(LTND)$	$\mathcal{O}(LTN^2D)$	$\mathcal{O}(LTND)$
YOCO-U	$\mathcal{O}((N+WTL)D)$	$\mathcal{O}(\frac{L}{2}TND)$	$\mathcal{O}(\frac{L}{2}(N+WT)D)$

leading to redundant overhead and potential optimization difficulties, YOCO-U restricts the recursion to the shallow self-decoder modules. The partial recursion design allows the model to significantly enhance its expressive power through multiple iterations within a fixed FLOP budget. Crucially, by utilizing efficient attention (such as sliding-window attention) within the recurrent Self-Decoder, YOCO-U eliminates the need for additional KV cache memory and avoids the prohibitive time complexity associated with full-attention looping. Consequently, YOCO-U not only maintains robust long-context modeling capabilities but also preserves the "one-piece" KV cache advantage, providing a much more scalable and efficient path for recursive computation compared to its predecessors.

Extensive evaluations across benchmarks validate the effectiveness of our approach. Our scaling experiments reveal that YOCO-U offers a favorable capability efficiency tradeoff: by directing extra compute into recursive efficient-attention blocks, it improves token utility and achieves competitive or better performance at the same FLOP budget, with negligible KV cache overhead. Furthermore, compared with other Transformer variants, YOCO-U demonstrates comparable capability in both general and long-context scenarios. YOCO-U retains the hallmark inference advantages of the non-recursive YOCO, such as linear pre-filling and reduced memory footprint. These results suggest that partial recursion within efficient-attention-based modules represents a promising direction for developing the next generation of high-performance and cost-effective large language models.

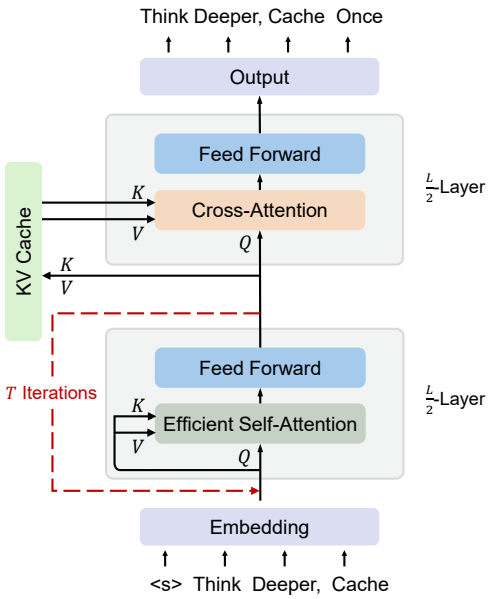


Figure 1: YOCO-U architecture: the Universal Self-Decoder runs recursive local-window attention for T iterations, while the Cross-Decoder reuses a shared global KV cache for autoregressive prediction.

2 METHOD

The proposed architecture YOCO-U, is a recursive foundation architecture for autoregressive modeling, such as large language models (LLMs). The design of YOCO-U inherits from YOCO (Sun et al., 2024), where the decoder-decoder architecture has two parts, i.e., self-decoder and cross-decoder, which splits the whole model into two $\frac{L}{2}$-layer modules. In YOCO-U, the non-recursive Self-Decoder is replaced by a Universal Self-Decoder with a fixed recursion count T, shared between training and inference.

Compared with Universal Transformer, YOCO-U applies recursion only to a subset of model layers. As a result, it preserves key inference advantages, including linear-complexity pre-filling and one-piece KV cache.

2.1 YOCO-U ARCHITECTURE

Given an input sequence $x = x_1 \cdots x_{|x|}$, the input embeddings are packed into $X^0 = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{|x|}] \in \mathbb{R}^{|x| \times d_{\text{model}}}$, where d_{model} is hidden dimension. The Self-Decoder consists of $\frac{L}{2}$-layer modules:

$$\text{Self-Decoder}^{L/2}(X) = \underbrace{L_{S_{L/2}} \circ \cdots \circ L_{S_1}}_{\frac{L}{2} \text{ layers}}(X) \tag{1}$$

The Universal Self-Decoder performs recursive computation T times on the Self-Decoder module:

$$\text{USD}(X) = \underbrace{\text{Self-Decoder}^{L/2} \circ \cdots \circ \text{Self-Decoder}^{L/2}}_{T \text{ iterations}}(X).$$

$\text{USD}(X)$ is used to produce KV caches $\hat{K}, \hat{V}$ for the cross-decoder. The Cross-Decoder also consists of $\frac{L}{2}$-layer modules:

$$\text{Cross-Decoder}^{L/2}(X, \hat{K}, \hat{V}) = \underbrace{L_{C_{L/2}} \circ \cdots \circ L_{C_1}}_{\frac{L}{2} \text{ layers}}(X, \hat{K}, \hat{V})$$

The final output is obtained by applying the Cross-Decoder to the representation produced by $\text{USD}(X)$ as $Y = \text{Cross-Decoder}(\text{USD}(X))$.

2.2 UNIVERSAL SELF-DECODER

Self-decoder layers leverage Efficient Self-Attention (ESA) to achieve $\mathcal{O}(1)$ inference memory and a constant number of KV caches:

$$Y^l = \text{ESA}(\text{LN}(X^l)) + X^l, \quad X^{l+1} = \text{SwiGLU}(\text{LN}(Y^l)) + Y^l \tag{2}$$

where $\text{ESA}(\cdot)$ represents efficient self-attention, i.e., subquadratic, state-efficient causal module suitable for recurrent shallow blocks. $\text{SwiGLU}(X) = (\text{swish}(XW_G) \odot XW_1)W_2$, and RMSNorm (Zhang & Sennrich, 2019) is used for $\text{LN}(\cdot)$. We adopt sliding-window attention (SWA) (Child et al., 2019) as the default for its simplicity and engineering stability. Linear attention variants from three generations of subquadratic modeling, such as RetNet (Sun et al., 2023), Mamba (Gu & Dao, 2023), and gated DeltaNet (Yang et al., 2024), are also compatible, though they perform similarly to SWA within hybrid architectures.

Universal Self-Decoder uses the recursive computation pattern in Universal Transformer (Dehghani et al., 2018), where the output is iterated multiple times within the same model parameters. The recursive computation enhances the model's representation capability without additional parameters. Looping only the shallow modules of a Transformer has been shown to provide improved computational efficiency (Alabdulmohsin & Zhai, 2025), and this design aligns with the position of the Self-Decoder in YOCO.

2.3 CROSS-DECODER

First, the output of the Universal Self-Decoder generates global KV caches $\hat{K}, \hat{V}$ for the cross-decoder:

$$\hat{K} = \text{LN}(\text{USD}(X))W_K, \quad \hat{V} = \text{LN}(\text{USD}(X))W_V \tag{3}$$

The KV caches $\hat{K}, \hat{V}$ are reused by all cross-decoder layers, and queries are computed layer-wise as $\hat{Q}^l = \text{LN}(X^l)W_Q^l$:

$$Y^l = \text{Attention}(\hat{Q}^l, \hat{K}, \hat{V}) + X^l, \quad X^{l+1} = \text{SwiGLU}(\text{LN}(Y^l)) + Y^l \tag{4}$$

where $\text{Attention}(\cdot)$ is standard multi-head attention (Vaswani et al., 2017), and $W_Q^l \in \mathbb{R}^{d \times d}$ is a learnable matrix. We use NoPE position embedding (Yang et al., 2025) in Cross-Decoder to enhance global retrieval capability. After obtaining Y, a softmax classifier performs next-token prediction.

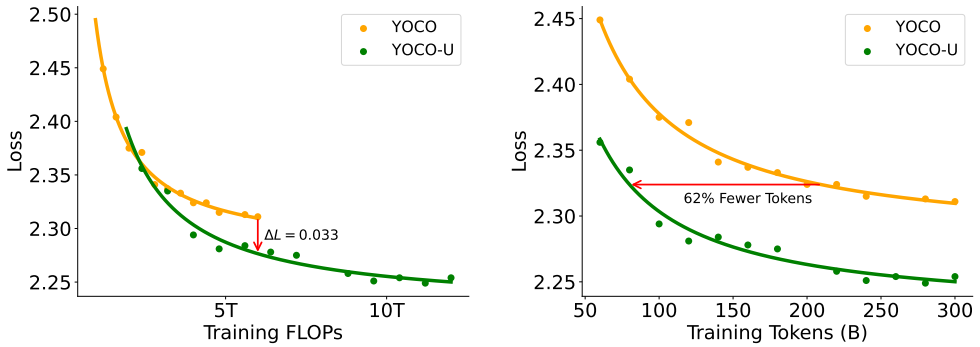


Figure 2: Scaling behavior of language modeling loss with the same model size. **(Left)** Loss versus training FLOPs: YOCO-U achieves competitive or lower loss (ΔL=0.033) at the same FLOPs budget, while incurring negligible KV cache overhead. **(Right)** Loss versus training tokens: YOCO-U also improves data efficiency, requiring approximately 62% fewer tokens to reach comparable performance.

2.4 Inference Advantages

In addition to the enhanced capability, YOCO-U preserves the low serving costs of the YOCO architecture. The complexity analysis is attached in Table 1.

Inference Advantages Inherited from YOCO YOCO-U preserves the key inference benefits of YOCO because the overall computation layout remains unchanged, where the universal self-decoder produces a single set of global KV caches, and the Cross-Decoder reuses them without recomputation.

Consequently, when the sequence length is long, the computation and memory cost is dominated by the global attention component. In YOCO-U, the recursive updates apply only to the efficient-attention module in the shallow block, whose overhead is negligible compared to global attention. More importantly, the KV cache still needs to be materialized only once, resulting in an overall memory cost of $\mathcal{O}(N)$ rather than $\mathcal{O}(NL)$ as in standard Transformer decoders. As a result, YOCO-U preserves YOCO's ability to serve very long contexts under a fixed GPU memory budget.

YOCO-U and Looping in Transformers A key distinction between looping in YOCO and looping in standard Transformer decoders lies in how the recursion interacts with global attention and KV caches. In a Transformer, introducing looped computation requires re-running all layers, which in turn regenerates a new set of KV caches and repeatedly executes global attention—leading to both increased memory usage and substantially higher compute cost.

In contrast, YOCO-U applies recursion only within the module based on efficient attention, while the cross-decoder and its global attention remain unchanged. As a result, the global KV cache is produced once and reused across iterations, and no additional global-attention computation is introduced. The recursive steps therefore incur only the small overhead of efficient attention, rather than the full cost of Transformer layers.

3 Experiments

Unless otherwise specified, we default to looping the Self-Decoder 3 times, resulting in approximately 2$\times$ the total FLOPs of the non-recursive baseline.

3.1 Language Model Evaluation

Training Recipe We train non-recursive and recursive YOCO models by scaling up the number of training tokens. The hidden dimension is 2560. The number of layers is 20. We leverage fine-grained MoE with shared experts (Dai et al., 2024), where we activate 8 among 64 experts and 1 additional shared expert. The expert dimension is 1024. Therefore, the total parameter is 10B and the activated

Table 2: Performance comparison of recursive and non-recursive YOCO. YOCO-U converts additional compute in recursive shallow blocks into consistent gains across tasks, with negligible KV cache overhead.

Model	ARC-C	Winogrande	HellaSwag	MMLU	BBH	GSM8K	HumanEval	DROP	Average
YOCO	46.50	61.72	63.44	49.59	33.13	38.06	9.15	32.62	41.78
YOCO-U (Equal FLOPS)	47.87	68.67	66.80	54.63	35.49	50.49	**10.98**	34.94	46.23
YOCO-U (Equal Steps)	**48.72**	**69.85**	**67.12**	**55.63**	**36.31**	**50.57**	10.37	**38.07**	**47.08**

parameter is 1.3B. For the YOCO setting, the Self-Decoder and Cross-Decoder both have 10 layers. We choose SWA for Self-Decoder. The window size is 512. For positional encoding, we choose NoPE (Yang et al., 2025) in Cross-Decoder and RoPE (Su et al., 2021) in Self-Decoder. The training length is 8192 and the batch size is 4M tokens. We use the AdamW (Loshchilov & Hutter, 2019) optimizer with $\beta = 0.9, 0.95$. The maximal learning rate is 1e-3 and consistent across the entire training process. We train the model with 75k steps (i.e., 300B tokens) given the resource budget. The training is running on AMD MI300X GPUs. The details are attached in Appendix A. Notably, the training process of YOCO-U exhibits high stability with a smooth loss trajectory and no significant spikes across the entire training regime.

Token Scaling As shown in Figure 2, we compare the training dynamics from two perspectives. We measure the validation loss every 20B tokens and remove 1 outlier point for each model. We use tokens $\times$ layers as a proxy for training compute, counting each recursive layer execution separately; Appendix F provides supplementary FLOP accounting at a 2048-token reference length. When we align the training FLOPs, YOCO-U achieves a lower loss (ΔL=0.033) at the same compute budget, demonstrating superior computation efficiency from recursive computation. Moreover, when we align the training tokens, YOCO-U shows better token utilities, where YOCO-U trained with 80B tokens is comparable with non-recursive YOCO with 210B tokens.

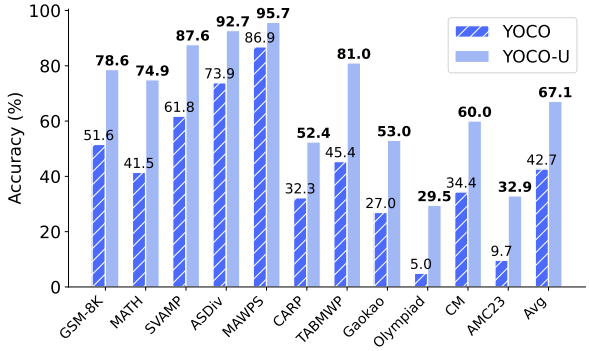


Figure 3: Accuracy comparison on 11 math benchmarks. YOCO-U consistently outperforms the YOCO baseline across all tasks, achieving a significant boost in accuracy.

End-Task Evaluation LM Eval-Harness (Gao et al., 2023) is used to evaluate performance in various downstream tasks. The models are trained with 300B tokens. Table 2 shows that YOCO-U achieves much better performance from recursive computation. Notably, even under the equal-FLOPs setting, YOCO-U already surpasses the baseline by a large margin (+4.45 average), confirming that the gains are not merely from additional compute.

Thinking SFT We continue training the models with math thinking data to evaluate the compatibility between depth scaling and explicit reasoning. We initialize models from the 280B checkpoint and train 20B tokens with 32768 maximal length. We evaluate the models across 11 math benchmarks: GSM-8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), SVAMP (Patel et al., 2021), ASDiv (Miao et al., 2020), MAWPS (Koncel-Kedziorski et al., 2016), CARP (Zhang et al., 2023), TABMWP (Lu et al., 2023), Gaokao 2023 En (Liao et al., 2024), OlympiadBench (He et al., 2024), CollegeMath (Tang et al., 2024) and AMC23. As shown in Figure 3, YOCO-U outperforms YOCO on all benchmarks with an average accuracy gain of 24.4%. These results show that the benefits of depth scaling persist after Thinking SFT.

Table 3: Performance comparison among different architectures. RINS is a standard decoder-only Transformer with early-layer recursion. YOCO-U matches RINS in average accuracy (48.3). At 128K context, its prefilling and decoding throughput are $10.2\times$ and $5.35\times$ those of RINS (Figure 7).

Model	**Wiki.** ppl↓	**LMB.** ppl↓	**LMB.** acc↑	**PIQA** acc↑	**OBQA** acc↑	**Hella.** acc_n↑	**Wino.** acc↑	**ARC-E** acc_n↑	**ARC-C** acc_n↑	**Avg.** acc↑
Non-Recursive										
Transformer	22.52	22.26	38.4	**69.6**	22.6	45.7	**57.1**	59.6	36.6	47.1
YOCO	22.25	**18.30**	**41.2**	67.9	23.8	45.6	54.3	59.2	36.6	47.0
Recursive										
Loop/Universal TRM	21.56	22.56	37.7	69.0	23.6	47.7	56.3	**62.4**	38.1	47.8
ParScale	23.13	24.06	36.5	68.7	22.6	44.8	55.4	60.9	38.4	46.8
RINS	**20.98**	20.06	39.4	69.4	24.0	**49.0**	54.2	62.0	**39.9**	**48.3**
YOCO-U	21.01	18.32	**41.2**	68.7	**24.6**	48.9	55.3	62.2	37.0	**48.3**

3.2 Architecture Comparison

Setup In addition to the recursive analysis on YOCO, we further compare YOCO-U with various compute-scaling Transformer variants, including the standard Universal Transformer (Dehghani et al., 2018), RINS (Alabdulmohsin & Zhai, 2025) (a standard decoder-only Transformer with early-layer recursion), and ParScale (Chen et al., 2025). All models have 1.3B parameters with 20 layers and a hidden dimension of 2560, following the model configuration in Section 3.1 with dense layers instead of MoE layers. For non-recursive baselines, we leverage standard Transformer with RoPE position encoding. For Universal Transformer, we loop the entire model 2 times, aligning similar FLOPs. For ParScale, we also double the token numbers. We train the model for 20k steps with 1M batch size, i.e., 20B tokens. These dense experiments provide controlled architecture comparisons; the MoE experiments in Section 3.1 evaluate the benefits of recursion at a larger training scale.

General-Domain Tasks Table 3 compares various non-recursive and recursive architectures. We follow the evaluation procedure in Section 3.1. Overall, we observe three key points that take effect on our final design:

- **Scaling FLOPs on bottom blocks is more effective than all blocks.** Based on standard Transformer, RINS applies early-layer recursion and achieves better performance than vanilla UT, showing the rationale of YOCO-U on layout level. We will show further that scaling deeper layers leads to diminishing gains.
- **Recursive scaling is more effective than parallel scaling,** because parallel scaling methods (such as ParScale) do not increase modeling depth, and usually achieve less improvements than recursive scaling under same FLOPs.
- **Scaling efficient-attention blocks is equally effective.** YOCO-U achieves performance comparable to RINS, a standard Transformer with early-layer recursion. The findings demonstrate that recursive computation with efficient-attention is capable of extending the model capability without a proportional increase in computational cost and KV cache.

Long Context We evaluate the long-context modeling for the above architectures, analyzing whether recursive computation with efficient attention sacrifices long-context capability. We measure the last 512 token's perplexity given different prefix length. We evaluate both book- and repository-level code data. The results in Figure 4 show that YOCO-U utilizes a long context as effectively as RINS. The conclusion is also consistent with the common practice that hybrid attention generally does not harm long context performance (Lieber et al., 2024; Yang et al., 2025). Table 4 reports retrieval performance on the Needle In-A-Haystack test (Kamradt, 2023) with a context

Table 4: Accuracy on the Needle In-A-Haystack (Kamradt, 2023) test with a context length of 8192 tokens. N represents the number of needles.

Model Name	**S-NIAH-1**	**S-NIAH-2**
Transformer	0.87	0.82
YOCO	**1.00**	0.86
RINS	0.99	0.91
YOCO-U	**1.00**	**0.95**

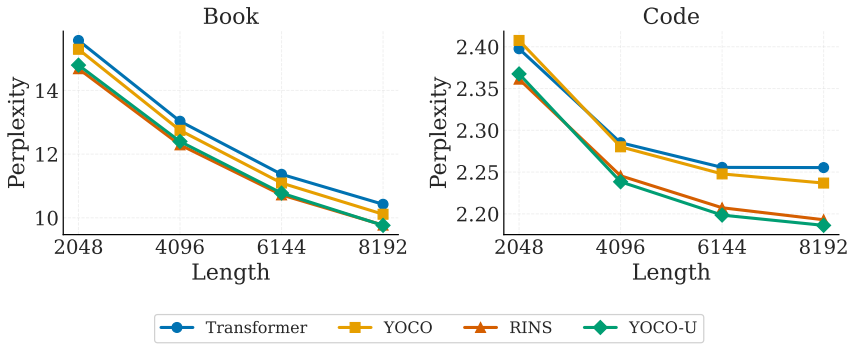


Figure 4: Perplexity drops with input length; YOCO-U outperforms non-recursive baselines and remains competitive with RINS.

length of 8192 tokens. The results demonstrate that YOCO-U maintains robust retrieval performance, confirming the effectiveness of our architecture in capturing and accessing information across extended sequences.

3.3 Ablation Studies

Table 5: Performance comparison among different design choices. "Deep (Instead of Wide)": double depth while using the same model size. "Upper Loop": loop cross-decoder instead of self-decoder. "Upper Loop w/o Shared KV": loop cross-decoder without using shared KV cache, i.e., employing self attention for loop cross-decoder.

Model	**Wiki.** ppl↓	**LMB.** ppl↓	**LMB.** acc↑	**PIQA** acc↑	**OBQA** acc↑	**Hella.** acc_n↑	**Wino.** acc↑	**ARC-E** acc_n↑	**ARC-C** acc_n↑	**Avg.** acc↑
Non-Recursive										
YOCO	22.25	18.30	41.16	67.90	23.80	45.64	54.30	59.22	36.60	46.95
Deep (Instead of Wide)	22.04	21.76	37.67	68.50	23.80	45.84	56.12	61.03	35.15	46.87
Recursive										
YOCO-U	21.01	18.32	41.18	68.66	24.60	48.89	55.33	62.16	36.95	48.25
Deeper (Instead of Wide)	21.42	18.45	41.39	69.27	24.40	48.23	56.75	61.49	38.57	48.59
Upper Loop (Cross-Decoder)	22.15	20.85	39.78	68.50	23.00	46.27	55.72	60.31	37.80	47.34
Upper Loop w/o Shared KV	22.06	21.56	38.21	68.66	23.60	45.69	53.35	59.64	35.75	46.41

In this section, we discuss other design options for introducing recursive computation on LLM architectures. By making analysis on loop position, KV cache management, and width-depth ratio, we further justify the rationale of the YOCO-U design. The experiments are shown in Table 5.

Loop Position We ablate the loop position based on the standard architecture. Since YOCO-U loops the shallow layers, we ablate two design choice, where Cross-Loop denotes the Cross-Decoder is looped and the KV cache is not shared, and Cross-Share shares all the KV cache generated from the Self-Decoder. The results demonstrate that recursive computation on deeper layer leads to diminishing improvement, regardless of the KV cache strategy. The conclusion is consistent with ETD (Koishekenov et al., 2025), where the final layers behave like a final decoder.

Model Layout We further discuss whether the computation depth and FLOPs contribute to the improved performance. We add a deeper model whose parameter is aligned with baseline but depth is doubled. The "*Deeper*" variant has 1792 hidden dimension and 40 layers. The results show that the training loss and downstream task performance are almost irrelevant to the model layout. Based on a deeper layout, YOCO-U can still have the same benefit of recursive computation.

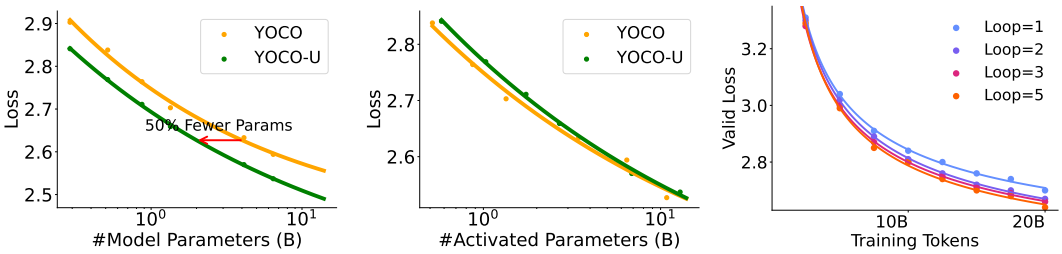


Figure 5: Parameter scaling properties. We keep the training steps the same. **Left:** YOCO-U achieves comparable performance with 50% fewer parameters than YOCO. **Right:** YOCO-U demonstrates scalable parameter utility as the activated parameter count increases.

Figure 6: Depth scaling with separately trained models at fixed $T \in \{1, 2, 3, 5\}$. YOCO-U improves as the trained depth increases.

3.4 Scaling Property

Figure 5 compares the scaling properties of non-recursive and recursive YOCO on language modeling. Except from the number of training token verified in Section 3.1, we scale up the model size and loop number respectively. We follow the experiment setting in Section 3.2.

Parameter Scaling We train language models with 300M, 500M, 870M, 1.3B, 3.4B, 6.4B, and 10.8B parameters. The details are attached in Appendix C. The scaling law (Kaplan et al., 2020) empirically fits well in this configuration. Critically, the entire scaling process remains highly stable, with all models exhibiting smooth convergence and no training instabilities as the parameter count increases.

First, YOCO-U shows consistent performance gain up to 6.4B model parameters. Second, when the activated parameter is aligned, YOCO-U narrows the performance gap with non-recursive models. When the activated parameter is more than 10B, YOCO-U even shows a performance near-comparable to non-recursive variants. The results suggest that YOCO-U eliminates parameter redundancy of LLMs.

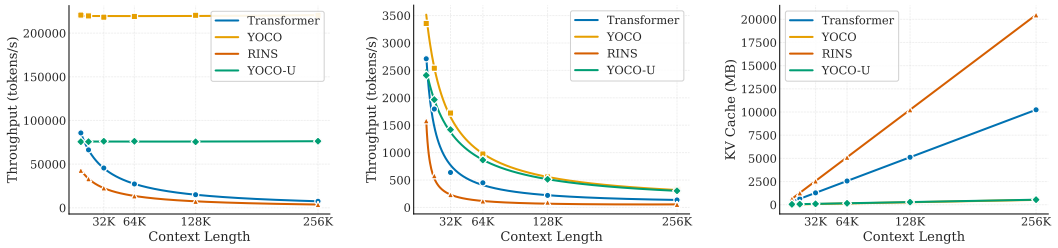
Loop Scaling We train separate 1.3B models with fixed loop counts $T \in \{1, 2, 3, 5\}$, where $T = 1$ corresponds to the non-recursive model and $T = 5$ increases FLOPs 3 times. Figure 6 shows that YOCO-U improves performance by scaling the trained depth.

3.5 Inference Efficiency

We analyze inference efficiency from various perspectives, including prefilling and decoding throughput and KV cache memory. We quantify the inference overhead of YOCO-U relative to non-recursive YOCO and compare its efficiency with recursive Transformer variants. We compare two non-recursive models, Transformer and YOCO, and two recursive models, RINS and YOCO-U. We use 1.3B model configuration from Section 3.4. The generation batch size is 32 and the length is 128. The experiments are conducted with H100-80GB GPU cards. Our implementation is based on Nano-vLLM (Geeeek-Explorer, 2025), where common optimization techniques, such as Flash-Decoding (Dao et al., 2023), kernel fusion, and Paged Attention (Kwon et al., 2023) have been applied.

Prefilling Throughput In the prefilling stage, while Transformer requires $O(N^2)$ time to encode all tokens in parallel, YOCO only requires $O(N)$ time in Self-Decoder. As shown in Figure 7a, although YOCO-U brings additional depth, it still improves the prefilling throughput $10\times$ than vanilla Transformer and $20\times$ than RINS from the complexity advantage. Even at relatively short context such as 16K, YOCO-U still has $1.15\times$ acceleration than non-recursive Transformer. The linear complexity behavior guarantees that the cost of prefilling remains negligible within the overall inference pipeline.

Decoding Throughput The results are illustrated in Figure 7b. Recursive computation based on standard Transformer brings additional full-attention layers, where RINS only achieves 40% throughput due to more attention computation and smaller batch size. However, YOCO-U only



(a) Prefilling efficiency. (b) Decoding efficiency. (c) KV cache occupation.

Figure 7: Overall efficiency comparison across sequence lengths. The models have the same number of parameters, where YOCO-U achieves both better performance and competitive throughput, as well as KV cache memory consumption, compared with Transformer and loop Transformer (RINS). The curves of YOCO-U and YOCO in Figure 7c are overlapped.

introduces additional efficient attention layers, thus only sacrifices 5% throughput than non-recursive YOCO. Compared with non-recursive Transformer, YOCO-U achieves $1.1\times$ acceleration at 16K length and $2.21\times$ at 256K length.

KV Cache Memory We report the KV cache for a single sequence. Since inference engines with Paged Attention usually dynamically adjusts the batch size, the KV cache memory determines the data parallelism in each single decoding step. As shown in Figure 7c, YOCO-U only brings negligible additional memory cost due to the constant KV cache in efficient attention. However, RINS's memory cost increases with the model depth, where RINS requires $38\times$ memory than YOCO-U, almost proportional to the number of layers.

3.6 Representation Analysis

Figure 8 shows the angular distance $d(l, l+1)$ across layers. Within the Universal Self-Decoder, the angular patterns across different loops remain remarkably consistent. However, the gradual decrease in mean distance suggests diminishing marginal returns, indicating that representations approach a fixed point as the number of iterations in the loop increases.

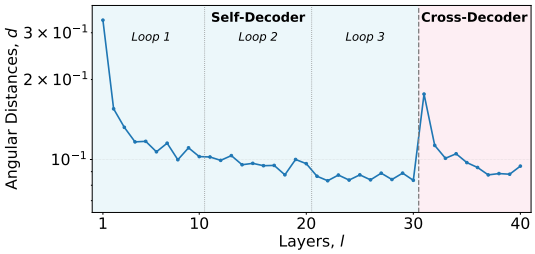


Figure 8: Angular distances between consecutive layers.

A sharp spike in the angular distance occurs at the interface between the blocks. This discontinuity suggests that the self-decoder and cross-decoder are responsible for different functionalities: the recursive self-decoder progressively refines intermediate representations, while the cross-decoder handles information retrieval and final decoding.

4 Conclusion

We introduce Universal YOCO (YOCO-U), a recursive architecture that decouples depth scaling from memory overhead. YOCO-U enables efficient depth scaling through parameter sharing and recursive computation without increasing the KV cache footprint. YOCO-U significantly enhances parameter utility, exhibiting superior scaling behaviors. Empirical results confirm that YOCO-U outperforms baselines in various benchmarks while preserving linear pre-filling efficiency, offering a scalable and cost-effective architecture for large language models.

AI USE STATEMENT

Generative AI was used only for manuscript preparation and editing. The authors take responsibility for the final content of this work.

ETHICS STATEMENT

This work improves the efficiency of large language models. It shares their general risks, including biased or harmful outputs, privacy concerns, and misuse. Deployment requires appropriate safeguards and compliance with applicable data licenses. Our experiments assess model capability and efficiency, without establishing safety guarantees or measuring environmental impact.

REPRODUCIBILITY STATEMENT

We describe the architecture in Sections 2.2 and 2.3 and provide training hyperparameters in Appendices A and C. Evaluation protocols and baseline configurations appear in Appendices B and E. Inference settings and FLOP accounting are detailed in Appendices D and F.

REFERENCES

Ibrahim Alabdulmohsin and Xiaohua Zhai. Recursive inference scaling: A winning path to scalable inference in language and multimodal systems. *arXiv preprint arXiv:2502.07503*, 2025.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.

Mouxiang Chen, Binyuan Hui, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Jianling Sun, Junyang Lin, and Zhongxin Liu. Parallel scaling law for language models. *arXiv preprint arXiv:2505.10475*, 2025.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse Transformers. *URL https://openai.com/blog/sparse-transformers*, 2019.

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.

Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*, 2024.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-Decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836.

GeeeekExplorer. nano-vllm. https://github.com/GeeeekExplorer/nano-vllm/tree/main, 2025. Accessed: 2025-09-09.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*, 2024.

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. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Greg Kamradt. Needle in a Haystack - pressure testing llms. https://github.com/gkamradt/LLMTest_NeedleInAHaystack/tree/main, 2023.

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. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Yeskendir Koishekenov, Aldo Lipani, and Nicola Cancedda. Encode, think, decode: Scaling test-time reasoning with recursive latent thoughts. *arXiv preprint arXiv:2510.07358*, 2025.

Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. Mawps: A math word problem repository. In *Proceedings of the 2016 conference of the north american chapter of the association for computational linguistics: human language technologies*, pp. 1152–1157, 2016.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Minpeng Liao, Wei Luo, Chengxi Li, Jing Wu, and Kai Fan. Mario: Math reasoning with code interpreter output–a reproducible pipeline. *arXiv preprint arXiv:2401.08190*, 2024.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid Transformer-Mamba language model. *CoRR*, abs/2403.19887, 2024. doi: 10.48550/ARXIV.2403.19887. URL https://doi.org/10.48550/arXiv.2403.19887.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *ICLR*, 2023.

Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. A diverse corpus for evaluating and developing english math word problem solvers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 975–984, 2020.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094, 2021.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. Mathscale: scaling instruction tuning for mathematical reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 47885–47900, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 6000–6010, 2017.

Bohong Wu, Mengzhao Chen, Xiang Luo, Shen Yan, Qifan Yu, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong, Xun Zhou, et al. Parallel loop transformer for efficient test-time computation scaling. *arXiv preprint arXiv:2510.24824*, 2025a.

Bohong Wu, Shen Yan, Sijun Zhang, Jianqiao Lu, Yutao Zeng, Ya Wang, and Xun Zhou. Efficient pretraining length scaling. *arXiv preprint arXiv:2504.14992*, 2025b.

Bowen Yang, Bharat Venkitesh, Dwarak Talupuru, Hangyu Lin, David Cairuz, Phil Blunsom, and Acyr Locatelli. Rope to nope and back again: A new hybrid attention strategy. *arXiv preprint arXiv:2501.18795*, 2025.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024.

Beichen Zhang, Kun Zhou, Xilin Wei, Xin Zhao, Jing Sha, Shijin Wang, and Ji-Rong Wen. Evaluating and improving tool-augmented computation-intensive math reasoning. *Advances in Neural Information Processing Systems*, 36:23570–23589, 2023.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

A HYPERPARAMETERS FOR SECTION 3.1

Table 6 presents the detailed hyperparameters for the models in Section 3.1. The only difference in the loop number, where $\text{Loop} = 1$ for non-recursive YOCO and $\text{Loop} = 3$ for YOCO-U.

Table 6: Hyperparameters used for the model in Section 3.1.

Params	**Values**
Layers	20
Hidden Size	2560
Expert Number	64
Expert Topk	8
Expert FFN Size	1024
Shared Expert FFN Size	1024
Vocab Size	151936
Heads	20
KV Heads	4
Self-Decoder Layers	10
Self-Decoder Window	512
Adam β	(0.9, 0.95)
LR	1×10^{-3}
Batch Size	4M
Warmup Steps	0
Weight Decay	0.1
Dropout	0.0

B THINKING SFT EVALUATION DETAILS FOR SECTION 3.1

To evaluate the mathematical reasoning capabilities of YOCO-U after Thinking SFT, we conduct evaluations on standard benchmarks using a consistent prompting template and decoding strategy.

Prompt Template We use a structured system and user message format to elicit step-by-step reasoning. The exact prompt used during evaluation is shown below:

> ***System:*** *You are a helpful and friendly AI assistant.*
> ***User:*** *Solve the following math problem. Explain your reasoning and put the final answer in \boxed{}.*
> *[Question Content]*

Decoding Strategy For all reasoning benchmarks, we employ greedy decoding to ensure reproducibility and focus on the model's primary reasoning path. The maximum generation length is set to 32,768 tokens to accommodate the extended "thinking" process and detailed step-by-step solutions required for complex mathematical problems.

C HYPERPARAMETERS FOR SECTION 3.4

Table 7 reports the hidden dimension, number of layers, and number of heads of for different model sizes. The training length is set to 8192. The batch size is set to 1.0M tokens. We use AdamW (Loshchilov & Hutter, 2019) with $\beta_1 = 0.9, \beta_2 = 0.95$. The weight decay is set to 0.1. The Self-Decoder Layers is set as half the total layers by default. The Self-Decoder Sliding-Window Attention size is 512. The FFN size is $3 \times d_{model}$, where d_{model} is the hidden dimension. We train the models with 20k steps, i.e., 20B tokens.

Table 7: Model size and hyperparameters used in Section 3.4.

Size	Hidden Dim.	#Layers	#Heads	LR
300M	1536	12	12	6×10^{-4}
500M	2048	12	16	6×10^{-4}
870M	2304	16	18	3×10^{-4}
1.3B	2560	20	20	3×10^{-4}
3.4B	4096	20	32	3×10^{-4}
6.4B	5120	24	40	3×10^{-4}
10.8B	6144	28	48	2×10^{-4}

D Inference Details for Section 3.5

The inference efficiency benchmarks are conducted on NVIDIA H100-80GB GPUs to evaluate the practical deployment performance of different architectures. All models are implemented using the Nano-vLLM framework and executed in BF16 precision. We adopt the 1.3B model configuration as specified in Section 3.4 for all comparisons. To optimize memory management and throughput, we employ Paged Attention with a block size of 256, and the GPU memory utilization is fixed at 0.7 to ensure a stable KV cache pool. The evaluation covers both prefilling and decoding phases, where we apply Flash-Decoding, kernel fusion, and other common optimization techniques to maximize hardware utilization. For the throughput measurements, we use a generation batch size of 32 with a sequence length of 128 tokens. This setup allows for a rigorous comparison between non-recursive models (Transformer and YOCO) and recursive variants (RINS and YOCO-U), specifically focusing on the trade-offs between computational overhead and memory efficiency in a unified high-performance inference stack.

Table 8: Prefill Throughput (tokens/s) across different sequence lengths.

Model	8K	16K	32K	64K	128K	256K
Transformer	85707	66342	45566	27276	14987	7475
YOCO	220662	219734	218247	219106	220416	220407
RINS	42905	33276	22767	13630	7397	3739
YOCO-U	75637	75694	76202	76148	75534	76301

Table 9: Decode Throughput (tokens/s) across different sequence lengths.

Model	8K	16K	32K	64K	128K	256K
Transformer	2712	1795	638	450	220	137
YOCO	3356	2539	1722	975	556	318
RINS	1582	580	234	118	96	56
YOCO-U	2410	1966	1419	865	514	303

E Experimental Details for Section 3.2

All baseline models are standardized to 1.3B parameters with 20 layers and a hidden dimension of 2560. We train each model for 20k steps with a 1M batch size (20B tokens total). To ensure a fair comparison, the total computational budget (FLOPs) of all variants is aligned to approximately $2\times$ that of a standard 20-layer Transformer.

The specific configurations for each baseline are as follows:

- **Universal Transformer (UT) (Dehghani et al., 2018)**: We adopt the weight-sharing recurrence mechanism by looping the entire 20-layer block 2 times. This results in a total of 40 layers of

Table 10: KV Cache Occupancy (MB) across different sequence lengths.

Model	8K	16K	32K	64K	128K	256K
Transformer	320	640	1280	2560	5120	10240
RINS	640	1280	2560	5120	10240	20480
YOCO	26	42	74	138	266	522
YOCO-U	46	62	94	158	286	542

computation per forward pass, effectively doubling the depth while maintaining the same parameter count.

- **RINS (Alabdulmohsin & Zhai, 2025)**: A standard decoder-only Transformer with early-layer recursion. Following the spirit of recurrent intermediate networks, we divide the 20 layers into two segments, a 10-layer non-recursive block and a 10-layer recurrent block. The latter 10 layers are looped 3 times. This configuration also results in 40 layers of total computation ($10 + 10 \times 3$) and aligns with the block layout of YOCO-U for a direct comparison of recursive strategies.
- **ParScale (Chen et al., 2025)**: We follow the original setting by utilizing different KV cache prefixes to represent distinct parallel groups. By processing tokens across multiple parallel branches, ParScale achieves a $2\times$ compute scaling factor, matching the FLOPs of the other recursive baselines while maintaining its unique parallelized architectural characteristics.

The performance of models is evaluated across various benchmarks. The datasets used in Table 3 include WikiText-103 (Wiki.) for long-form language modeling, LAMBADA (LMB.) for predicting words based on long-range context, Physical Interaction QA (PIQA) for physical commonsense, OpenBookQA (OBQA) for fact-based reasoning, HellaSwag (Hella.) for everyday event continuation, Winogrande (Wino.) for coreference resolution, and the AI2 Reasoning Challenge (ARC-E/ARC-C) for easy and challenge-level science questions. The evaluation metrics are defined as follows:

- **Perplexity (ppl)** $\downarrow$: Measures the model's uncertainty in predicting the next token; a lower value indicates better language modeling performance.
- **Accuracy (acc)** $\uparrow$: The ratio of correctly predicted instances to the total number of test samples.
- **Normalized Accuracy (acc_n)** $\uparrow$: A length-normalized accuracy metric that adjusts for the varying lengths of candidate answers to ensure a fair comparison in multiple-choice tasks.

F FLOPs ACCOUNTING

We supplement the $\text{tokens} \times \text{layers}$ proxy in Section 3.1 with compute estimates that account for linear layers and attention separately. Table 11 reports these estimates for a reference sequence length of 2048 tokens. The per-token cost is obtained by dividing the per-sequence cost by 2048, and the total training-compute estimates are obtained by multiplying the per-token cost by the number of training tokens. Per-sequence and per-token costs are reported in GFLOPs (10^9 FLOPs), while total costs are reported in EFLOPs (10^{18} FLOPs).

Table 11: FLOP estimates at a 2048-token reference sequence length. The first two numerical columns give per-sequence and per-token costs in GFLOPs; the remaining columns give total compute in EFLOPs for the indicated training-token budgets. Totals are calculated before rounding the per-token values.

	GFLOPs		Total compute (EFLOPs)					
Model	/ sequence	/ token	50B	100B	150B	200B	250B	300B
YOCO	21,950.86	10.7182	535.91	1,071.82	1,607.73	2,143.64	2,679.55	3,215.46
Transformer	22,315.14	10.8961	544.80	1,089.61	1,634.41	2,179.21	2,724.02	3,268.82
YOCO-U	43,827.36	21.4001	1,070.00	2,140.01	3,210.01	4,280.02	5,350.02	6,420.02
RINS	44,920.19	21.9337	1,096.68	2,193.37	3,290.05	4,386.74	5,483.42	6,580.11

At this reference length, the estimated FLOP ratio of YOCO-U to YOCO is 1.997, and that of RINS to Transformer is 2.013. Both are close to the $2\times$ ratio given by the layer-count proxy, supporting its use as an approximation to the relative cost of recursion in this accounting setting. The estimates here use a reference length of 2048 tokens; the pre-training sequence length is 8192, as specified in Section 3.1 and Appendix C. Since attention costs depend on sequence length, absolute compute estimates must be evaluated at the sequence length of interest.

G Limitations

Our experiments use fixed recursion counts during training and inference; adapting the loop count of a trained model remains future work. Recursion adds computation and local-cache memory despite keeping the global KV cache unchanged. Measured speedups depend on hardware and workload, and the results may not generalize to larger models or other tasks. We do not quantify variability across training seeds.

H Related Work

Computation Scaling The efforts to scale the ratio of computation to model parameters have focused on several key aspects. Approaches such as Universal Transformer (Dehghani et al., 2018) significantly reduce parameter costs and enhance model capacity by sharing parameters across the depth dimension. Model performance can be further improved by selectively scaling the depth of certain layers (Alabdulmohsin & Zhai, 2025; Koishekenov et al., 2025; Bae et al., 2025). However, depth scaling typically introduces challenges, such as increased latency and a larger KV cache requirement. In contrast, parallel scaling methods (Chen et al., 2025; Wu et al., 2025b;a) increase computation with a smaller latency overhead, yet they usually exhibit suboptimal performance compared to depth scaling techniques.

Inference Scaling Increasing computation during the inference stage has demonstrated strong capabilities, particularly in complex reasoning tasks (Jaech et al., 2024; Guo et al., 2025). Furthermore, explicit chain-of-thought reasoning can be effectively compressed into a continuous representation (Hao et al., 2024). Crucially, the enhanced capability derived from inference scaling stems from the intrinsic capacity established after pre-training, rather than directly benefiting the pre-training process itself. Consequently, computation scaling strategies applied during pre-training are orthogonal to these inference scaling techniques.