

Parallel Loop Transformer for Efficient Test-Time Computation Scaling

ByteDance Seed

Full author list in Contributions

Abstract

Large Language Models (LLMs) are powerful but often too slow and costly for real-world use during inference. Looped transformers save on parameters by reusing the same weights for multiple computational steps, or "loops." However, this approach has a major flaw: the loops run one after another, causing inference latency and memory requirements to increase with each added loop. This makes them impractical for fast applications. To solve this problem, we introduce the Parallel Loop Transformer (PLT). PLT is a new architecture that delivers the performance benefits of a deep, looped model but with the low latency of a standard, non-looped model. PLT works using two key techniques. First, Cross-Loop Parallelism (CLP) breaks the sequential dependency by computing different loops for different tokens at the same time, all within a single pass. Second, to prevent memory costs from growing, we use an Efficient Representation Enhancement strategy. This method shares the memory (KV cache) from the first loop with all other loops. It then uses a Gated Sliding-Window Attention (G-SWA) to combine this shared global information with local information, maintaining high accuracy. Our experiments show that PLT achieves the high accuracy of a traditional looped model but with almost no extra latency or memory cost compared to a standard transformer.

Date: October 30, 2025

Correspondence: Bohong Wu at bohongwu@bytedance.com, Xingyan Bin at binxingyan@bytedance.com

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of tasks [21, 26, 28, 29, 38, 39], yet their practical deployment is often constrained by substantial computational costs during inference [4, 5, 24, 45, 49]. As models scale, the latency and memory bandwidth required for token generation become significant bottlenecks. This challenge has spurred research into architectures that can achieve high performance while maintaining inference efficiency.

One promising direction is the use of looped transformers, such as the Universal Transformer [11], which reuses the same set of parameters across multiple computational steps or "loops." This weight-sharing mechanism makes them highly parameter-efficient, enabling them to achieve greater effective depth and stronger performance on complex reasoning tasks without increasing the model's storage footprint [13, 16, 30, 32, 35, 37, 43, 46]. However, this parameter efficiency comes at a severe cost: in their vanilla implementation, the loops are executed in a strictly sequential manner as shown in Figure 1a. Such sequential dependency means that per-token compute, wall-clock latency, and KV-cache size all scale linearly, $\mathcal{O}(L)$, with the number of loops (L). This scaling bottleneck renders vanilla looped transformers impractical for latency-sensitive

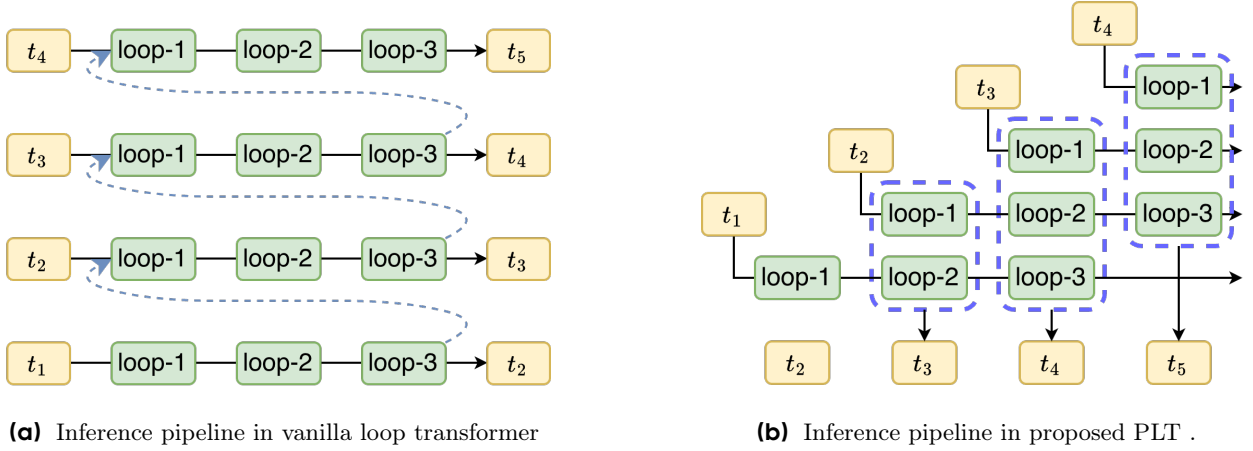


Figure 1 Illustration of the computation flow. (a) Vanilla loop transformer, where each loop in each token should be computed in serial manner. (b) Parallel loop transformer (PLT), where transformer loops within the same **blue dashed box** can be computed in parallel.

applications, effectively negating their primary advantages.

To resolve this tension between effective depth and inference speed, we introduce the **Parallel Loop Transformer (PLT)**. PLT is a novel architecture designed to unlock the performance benefits of deep, looped computation while maintaining the approximate inference latency of a standard, non-looped transformer. As shown in Figure 1b, the core principle of PLT is to break the sequential loop dependency by parallelizing the computation of different loops across different tokens.

Our approach consists of two key components. First, we introduce **Cross-Loop Parallelism (CLP)**, a technique that reformulates the training and inference pipelines. During decoding, CLP executes the l -th loop for the current token t_i concurrently with the $(l + 1)$ -th loop for the previous token t_{i-1} and so on, all within a single forward pass. This overlapping computation effectively collapses the L sequential steps into one, decoupling the model’s effective depth from its wall-clock latency. Second, to address the $\mathcal{O}(L)$ growth in KV cache, we propose an **Efficient Representation Enhancement** strategy. This strategy shares the KV cache from the first loop with all subsequent loops, reducing the KV cache memory footprint back to the level of Loop times as 1. To preserve high sliding accuracy, we augment this global shared representation with a local context mechanism, using a gated sliding-window attention (SWA) in non-first loops.

Our contributions are as follows:

- We propose the Parallel Loop Transformer (PLT), an architecture that, to our knowledge, is the first to successfully parallelize the computation of looped transformers to achieve scalable test-time computation with negligible latency overhead.
- We introduce Cross-Loop Parallelism (CLP) and an Efficient Representation Enhancement technique (KV-cache sharing with gated SWA) to overcome the critical latency and memory bottlenecks of traditional looped models.
- We demonstrate through extensive experiments on both in-house and open-source models that PLT significantly outperforms vanilla transformer baselines in accuracy while adding minimal latency.
- We show that PLT is far more efficient than vanilla looped transformers, and can even enable a shallower, more efficient PLT model (e.g., 1.7B activated parameters) to achieve superior performance and lower latency than a much larger vanilla model (e.g., 2.5B activated parameters).

2 Method

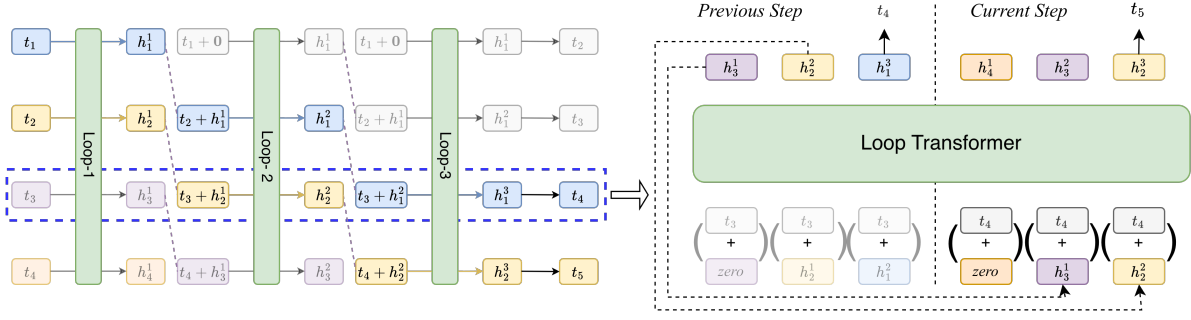


Figure 2 Training and inference pipeline of PLT with loop count $L=3$. **Training (Left):** Same Colored boxes trace how input tokens traverse the loops to predict their targets (e.g., token T_1 passes three loops to predict T_4 , consistent with Figure 1b). Training is parallel along the token dimension and serial along the loop dimension. **Inference (Right):** Parallelized forward pass of PLT when decoding T_4 and T_5 in a Loop Transformer with $L=3$. Because there are no horizontal (same-step, cross-loop) activation dependencies during training, computations within the same step (each row; see the blue dashed box) run in parallel during decoding.

2.1 Preliminaries: Vanilla Loop Transformer

We consider a **vanilla Loop Transformer** [11] with L loops. Let $T = (t_1, t_2, \dots, t_n)$ be a token sequence, $\mathbf{E} = (\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n)$ is the token sequence after embedding. For token index $i \in \{1, \dots, n\}$ and loop index $l \in \{1, \dots, L\}$, let $h_i^{(l)}$ denote the hidden state at position i after l forward loops, with $h_i^{(0)}$ the initial state.

The computation flow for i -th token in vanilla loop transformer is as follow:

$$h_i^{(0)} = t_i, \quad h_i^{(l)} = f^{(l)}(h_i^{(l-1)}), \quad l = 1, \dots, L, \quad (1)$$

where $f^{(l)}$ denotes the l -th forward loop. Finally, $h_i^{(L)}$ feeds into the classifier head to predict the $(i+1)$ -th token.

Challenge in vanilla loop transformer. In a vanilla loop transformer, loops run strictly sequentially, so per-token compute, wall-clock latency, and KV-cache size scale as $\mathcal{O}(L)$ with the number of loops L as shown in Table 1. While weight sharing makes it parameter-efficient (it achieves greater effective depth with fewer stored weights), it does not reduce latency. Therefore, it mainly helps under equal-parameter comparisons; under equal-latency budgets—typical in practical inference—the vanilla loop transformer offers no inherent advantage and can be worse due to longer decode paths, higher memory-bandwidth pressure, and larger KV caches.

2.2 Parallel Loop Transformer

Motivation. Vanilla loop transformers are parameter-efficient but rely on strictly serial computation per token, which increases the forward times and the KV-cache footprint. We aim to keep the parameter savings while shortening the forward times by enabling parallelism across loops and tokens without breaking causality.

As shown in Figure 1b, we convert the vanilla sequential loop transformer into a parallel loop transformer that executes L loops in parallel. The design has two key components: (i) cross-loop parallelism (Sec. 2.2.1) for parallel loop inference; (ii) efficient representation enhancement (Sec. 2.2.2) to improve accuracy with light time and KV cache overhead.

2.2.1 Cross-Loop Parallelism

Algorithm 1 Decoding of PLT with loop times as 3

Require:

Loop times $L = 3$
Transformer block function f ;
Input sequence $T = (t_1, \dots, t_n)$;
Maximum new tokens M .
1: $K_{share}, V_{share}, h_n^1, h_{n-1}^2, h_{n-2}^3 \leftarrow f(T)$ ▷ Prefilling
2: $\text{logits} \leftarrow \text{ClassifierHead}(h_{n-2}^3)$
3: $t_{k+1} \leftarrow \text{argmax}(\text{logits})$ ▷ Predict the next token
4: **for** $i = n + 1$ **to** $n + M - 1$ **do**
5: $\mathbf{e}_i = \text{Embedding}(t_i)$
6: $\mathbf{B} \leftarrow B_0, B_1, B_2 \leftarrow \mathbf{e}_i, \mathbf{e}_i + h_{i-1}^1, \mathbf{e}_i + h_{i-2}^2$
7: $K_{share}, V_{share}, h_i^1, h_{i-1}^2, h_{i-2}^3 \leftarrow f(\mathbf{B}, K_{share}, V_{share})$
8: $\text{logits} \leftarrow \text{ClassifierHead}(h_{i-2}^3)$
9: $t_{i+1} \leftarrow \text{argmax}(\text{logits})$
10: **end for**

Algorithm 2 Training for PLT

Require:

Loop count L
Transformer block function f
Classification head $\text{ClassifierHead}(\cdot)$
Input token sequence $T = (t_1, \dots, t_n)$
1: $\mathbf{E} \leftarrow \text{Embedding}(T)$
2: $K_{share}, V_{share}, \mathbf{H}^{(1)} \leftarrow f(\mathbf{E})$
3: **for** $i = 2$ **to** L **do**
4: $\mathbf{H}^{(i-1)} \leftarrow \text{concat}(\mathbf{0}, \mathbf{H}^{(i-1)}[: -1])$
5: $\mathbf{B} \leftarrow \mathbf{E} + \mathbf{H}^{(i-1)}$
6: $\mathbf{H}^{(i)} \leftarrow f(\mathbf{B}; K_{share}, V_{share})$
7: **end for**
8: $\text{logits} \leftarrow \text{ClassifierHead}(\mathbf{H}^{(L)})$
9: $\text{loss} = \text{CrossEntropy}(\text{logits}, T)$
10: **return** loss

Algorithm 3 Efficient representation enhancement for non-first loop in PLT

Require:

Input hidden state $\mathbf{H}$
QKV linear layer f_{qkv} , Output linear layer f_o
Gate linear layer f_{gate}
shared Key-Value cache from first Loop
(K_{share}, V_{share})
window size w of sliding window attention
(SWA)
1: $Q, K, V = f_{qkv}(\mathbf{H})$
2: $y_{\text{global}} = \text{Attn}(Q, K_{share}, V_{share})$
3: $y_{\text{local}} = \text{SWA}(Q, K, V, w)$
4: $g = \text{Sigmoid}(f_{gate}(Q))$
5: $\tilde{y} = g \odot y_{\text{local}} + (1-g) \odot y_{\text{global}}$
6: $o = f_o(\tilde{y})$
7: **return** o

We introduce Cross-Loop Parallelism (CLP): overlapping later-loop computation on earlier tokens with earlier-loop computation on later tokens. As shown in Figure 1b, taking decoding token t_4 with 3 loops as example, first loop on token t_3 , second loop on token t_2 and third loop on token t_1 are executed simultaneously, and similar cross-loop parallelism are executed by the following tokens. The detailed inference and training pipeline of this method are proposed as follows:

Training. The training flow of PLT is illustrated in Figure 2 and Algorithm 2. In the first loop, PLT matches a vanilla Transformer: it feeds the token embeddings $\mathbf{H}^{(0)} = \mathbf{E} = (\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n)$ into the model and obtains the last-layer hidden states $\mathbf{H}^{(1)} = (\mathbf{h}_1^{(1)}, \mathbf{h}_2^{(1)}, \dots, \mathbf{h}_n^{(1)})$. Before the second loop, PLT shifts $\mathbf{H}^{(1)}$ to the right by one position, from $(\mathbf{h}_1^{(1)}, \dots, \mathbf{h}_n^{(1)})$ to $(\mathbf{0}, \mathbf{h}_1^{(1)}, \dots, \mathbf{h}_{n-1}^{(1)})$, and then adds back the original embeddings: $\mathbf{B} = \mathbf{E} + \text{shift}(\mathbf{H}^{(1)})$. This shift removes direct dependence between the states with the same index across consecutive loops, which enables parallel processing during decoding; we refer to this as **cross-loop parallelism**. PLT repeats this process for $i = 2, \dots, L$: at each loop, it right-shifts the previous loop’s states by one position, adds the embeddings, and applies the transformer while reusing the shared K_{share} and V_{share} from the first loop.

Table 1 Complexity comparison of the vanilla Transformer, a looped Transformer, and the proposed *PLT*. The vanilla Transformer serves as the baseline with parameter count P , per-token compute cost C , and single-token decoding latency t under a memory-bound setting. The attention KV cache scales as $\mathcal{O}(nd)$, where n is the sequence length and d is the embedding dimension. Here, L denotes the loop count and w the sliding-window size used by *SWA* (sliding-window attention).

	Loop Times	Param.	Compute.	KV cache	Decoding latency
(1) Vanilla transformer	1	P	C	$\mathcal{O}(nd)$	t
(2) Vanilla loop transformer	L	P	LC	$\mathcal{O}(Lnd)$	Lt
(3) <i>Loop</i> + <i>CLP</i>	L	P	LC	$\mathcal{O}(Lnd)$	$\sim t$
(4) <i>Loop</i> + <i>CLP</i> + <i>KV share</i>	L	P	LC	$\mathcal{O}(nd)$	$\sim t$
(5) <i>Loop</i> + <i>CLP</i> + <i>KV share</i> + <i>G-SWA</i>	L	P	LC	$\mathcal{O}(nd + (L-1)wd)$	$\sim t$

Inference. As shown in Figure 2 (right), **cross-loop parallelism** in PLT processes L tokens with a single forward pass. This parallel design leverages the memory-bound nature [44] of LLM decoding: adding parallel test-time computation FLOPs improves accuracy, while the extra decoding latency is negligible. Algorithm 1 illustrates the decoding flow of PLT for $L=3$. At decoding step i , we construct a displaced micro-batch $\mathbf{B}=\{B_0, B_1, B_2\}$, where B_0 is the first loop of token i , B_1 is the second loop of token $i-1$, and B_2 is the third loop of token $i-2$. We then predict token $(i+1)$ from the final loop state h_{i-2}^3 . Compared with a vanilla loop transformer (which applies L sequential passes per token), PLT performs the same logical compute with one parallel pass per token. Compared with a standard non-loop decoder, it adds FLOPs that improve accuracy, yet the latency increase is negligible because decoding is memory-bound [44].

2.2.2 Efficient Representation Enhancement

Cross-loop parallelism addresses the inference-time scaling with loop count L in a vanilla loop transformer, but it still incurs an L -fold KV-cache memory cost, which limits long-context use. We introduce **efficient representation enhancement** with two components: (i) first-loop KV-cache sharing to provide a single global representation, and (ii) gated sliding-window attention to strengthen local context. Details follow.

KV-cache sharing from the first loop. In a standard loop design, each loop maintains its own KV cache, so memory grows linearly with L . To reduce KV memory, we share the first loop’s KV cache with all later loops. As shown in Algorithm 3, non-first loops keep their private queries but perform global attention on K_{share} and V_{share} from the first loop. Thus, only one global KV cache needs to be stored. This design preserves global information for non-first loops and removes the L -dependent KV-cache growth.

Gated sliding-window attention (G-SWA) in non-first loops. To further enhance local information on top of the shared global representation, non-first loops apply sliding-window attention over their private Q , K , and V with window size w . In our experiments, we set $w=64$, and it does not increase with the overall sequence length. As shown in Lines 3–5 of Algorithm 3, we then fuse the outputs of SWA (local) y_{local} and full attention on the shared KV (global) y_{global} using a sigmoid gate:

$$g \leftarrow \text{Sigmoid}(f_{\text{gate}}(Q)), \quad \tilde{y} \leftarrow g \odot y_{\text{local}} + (1-g) \odot y_{\text{global}}. \quad (2)$$

The gate linear layer f_{gate} is head-wise with a scalar output per head, so the added parameters and computation are negligible. In addition, due to SWA, non-first loops only cache the most recent w KV entries during decoding. Overall, gated sliding-window attention makes the proposed PLT more KV-cache efficient than a vanilla loop transformer, since non-first loops store only w recent KV entries, while accuracy is maintained by combining both global and local information.

2.3 Inference Efficiency Analysis

As shown in Table 1, we analyze per-token inference in a memory-bound setting. The vanilla Transformer (baseline) has parameter count P , per-token compute C , KV cache $\mathcal{O}(nd)$, and decoding latency t . Introducing L loops increases compute to LC and, if each loop keeps its own cache, increases memory to $\mathcal{O}(Lnd)$; because the loops run sequentially, latency grows to Lt . With the proposed cross-loop parallelism, the loop iterations

Table 2 Performance evaluation of in-house Seed-MoE and *PLT* with the same number of activated parameters. Lat. and KV cache represents decoding latency (ms) and KV cache memory overhead under batch size as 4.

	MMLU	CEval	M.PRO	AGI.	BBH	DROP	GSM.	H.Eval	MBPP	TQA	Avg.	Lat. ₁	kv cache
(1)Seed-MoE 680M/13B	54.0	53.0	22.5	31.5	36.8	30.1	22.6	25.0	34.1	37.4	34.7	4.8	280M
(2)+ <i>loop-2</i>	59.1	61.6	26.0	37.7	44.2	36.8	29.0	25.0	35.7	41.6	39.7	9.4	560M
(3)+ <i>loop-2</i> + <i>CLP</i>	59.7	60.6	28.6	36.6	38.9	36.8	34.6	25.6	38.1	36.3	39.6	5.9	560M
(4)+ <i>loop-2</i> + <i>CLP</i> + <i>KV share</i>	55.7	57.2	22.1	33.4	35.3	33.5	27.3	23.8	34.4	39.1	36.2	4.8	280M
(5)+ <i>loop-2</i> + <i>CLP</i> + <i>KV share</i> + <i>G-SWA(PLT-2)</i>	59.6	58.9	27.0	34.9	41.6	36.4	33.6	26.8	37.8	40.0	39.7	4.9	284M
(6)+ <i>loop-3</i> + <i>CLP</i> + <i>KV share</i> + <i>G-SWA(PLT-3)</i>	62.5	61.4	27.1	35.7	40.3	41.7	39.3	23.8	34.4	41.3	40.8	5.0	287M

run concurrently, so latency is $\approx t$, compute remains LC , and the KV cache remains $\mathcal{O}(Lnd)$. A KV-cache sharing variant reuses the cache of the first loop and does not store separate caches for the non-first loops, which reduces the KV cache to $\mathcal{O}(nd)$ at the cost of a small accuracy drop. Finally, rather than using full-context attention or disabling attention in the non-first loops, we apply sliding-window attention of size w in those loops, which yields a total KV cache of $\mathcal{O}(nd + (L - 1)wd)$. Since typically $w \ll L$, the extra KV cache and compute overhead is small. Overall, PLT maintains near-baseline decoding latency and avoids the L -fold growth of the KV cache.

3 Experiments

We organize our experiments in two parts. Section 3.1 provides a comprehensive comparison of PLT against a vanilla Transformer baseline under identical parameter settings, demonstrating the contribution of each component introduced in PLT and examining scalability with respect to the number of parallelized loops. Section 3.2 evaluates inference efficiency at matched accuracy, showing that PLT achieves performance comparable to the vanilla Transformer with fewer parameters while delivering significantly improved inference efficiency.

3.1 Accuracy Comparisons under the Same Parameters

3.1.1 Settings

Training recipe. In this section, we compare Seed-MoE models with a vanilla looped Transformer and variants of PLT. We add each component—cross-loop parallelism, KV-cache sharing, and gated sliding-window attention (G-SWA)—to the vanilla looped Transformer one by one to assess the effect of each component. For PLT-related hyperparameters, we set the G-SWA window size w to 64 and vary the PLT loop count $L \in \{2, 3\}$ to study scalability. We train 680M/13B MoE (680M activated parameters with 13B total parameters) models on 150B high-quality tokens. Additional details are withheld due to confidentiality; a more detailed open-source training configuration appears in Section A.2.

Accuracy evaluation recipe. We evaluate accuracy on the following open-source benchmarks: MMLU [17], CEval [20], AGIEval (AGI.) [48], MMLU-Pro (M. Pro) [40], BBH [34], DROP [12], MBPP [1], HumanEval (H.Eval) [3], MATH [18], and GSM8k [9].

Inference efficiency evaluation recipe. We use FP8 self-attention [33] and W4A8 quantization of linear layers [19] to simulate real serving. To evaluate efficiency under both low- and high-throughput scenarios, we set the prefill context length to 5000 and vary the inference batch size in [4, 8, 16, 32, 64], which shifts decoding from low-throughput to high-throughput scenarios. We report per-token decoding latency, averaged over five independent runs that each decode 256 tokens on one single GPU.

3.1.2 Results and Analysis

Table 2 reports the performance of vanilla transformers, loop transformers, and variants of PLT. We summarize and explain the main findings below.

Table 3 Inference efficiency evaluation of PLT variants across different batch size.

	bs=4	bs=8	bs=16	bs=32	bs=64
(1)Seed-MoE 680M/13B	4.8(1.00 $\times$)	5.6(1.00 $\times$)	6.7(1.00 $\times$)	8.1(1.00 $\times$)	10.9(1.00 $\times$)
(2)+ <i>loop-2</i>	9.4(1.96 $\times$)	11.1(1.98 $\times$)	13.2(1.97 $\times$)	16.1(1.99 $\times$)	21.4(1.96 $\times$)
(3)+ <i>loop-2</i> +CLP	5.9(1.23 $\times$)	6.9(1.23 $\times$)	8.5(1.27 $\times$)	10.9(1.35 $\times$)	16.4(1.50 $\times$)
(4)+ <i>loop-2</i> +CLP+KV <i>share</i>	4.8(1.00 $\times$)	5.6(1.00 $\times$)	6.8(1.01 $\times$)	8.3(1.02 $\times$)	11.1(1.02 $\times$)
(5)+ <i>loop-2</i> +CLP+KV <i>share</i> +G-SWA(PLT-2)	4.9(1.02 $\times$)	5.7(1.02 $\times$)	6.9(1.03 $\times$)	8.6(1.06 $\times$)	11.3(1.04 $\times$)

Observation 1: Cross-loop parallelism (CLP) preserves accuracy and reduces latency. A loop transformer with two loops (row (2)) raises average accuracy by +5.0 points over the vanilla model (34.7 $\rightarrow$ 39.7), but also increases latency and memory (4.8 $\rightarrow$ 9.4 ms, +96%; 280M $\rightarrow$ 560M, +100%). Adding CLP at the same loop count (row (3)) keeps the accuracy nearly unchanged (39.7 $\rightarrow$ 39.6, -0.1), while it cuts latency from 9.4 ms to 5.9 ms (-37%) by parallelizing loop computation. Compared to the vanilla transformer, CLP keeps most of the accuracy gain (+4.9 points) with only a modest latency factor (4.8 $\rightarrow$ 5.9 ms, $\approx$ 1.23 $\times$) and no extra KV-cache versus the naive loop (still 560M). Overall, CLP removes the sequential latency cost of looping while keeping the accuracy benefit.

Observation 2: Efficient representation enhancement reduces KV-cache with minimal accuracy loss. Efficient representation enhancement includes two parts: KV-cache sharing and gated sliding-window attention (G-SWA). KV-cache sharing (row (4)) removes the extra KV-cache footprint from looping (560M $\rightarrow$ 280M, -50%) and also reduces latency (5.9 $\rightarrow$ 4.8 ms, -19%) because of less KV-cache loading time during decoding, but also lowers average accuracy benefit by 3.4 points (39.6 $\rightarrow$ 36.2) because each loop no longer holds its own dedicated KV. Furthermore, Adding G-SWA (row (5)) restores per-loop specificity using a local window, raising accuracy from 36.2 to 39.7 (+3.5) while adding only 1.4% KV-cache overhead (280M $\rightarrow$ 284M) and nearly no latency penalty (4.8 $\rightarrow$ 4.9 ms). Therefore, KV-cache sharing solves the memory blow-up; G-SWA recovers accuracy at negligible cost.

Observation 3: PLT (PLT) delivers loop-level accuracy with near-vanilla efficiency and scales well. With two loops, *PLT-2* (row (5)) matches the accuracy of the naive loop transformer (39.7) but keeps efficiency close to the vanilla model (latency 4.8 $\rightarrow$ 4.9 ms, +2%; KV 280M $\rightarrow$ 284M, +1.4%). Scaling to *PLT-3* (row (6)) further improves average accuracy to 40.8 (+1.1 over *PLT-2*) with only a small latency increase (4.8 $\rightarrow$ 4.9 ms, +2%) and a minor KV change (284M $\rightarrow$ 287M, +1.1%). PLT decouples latency and memory from the loop count, so it preserves the accuracy gains of looping while keeping inference overhead close to vanilla, and it scales smoothly with more loops.

3.1.3 Inference Efficiency

Table 3 shows that PLT (*PLT*) improves latency in both low-throughput and high-throughput regimes while outperforming the vanilla loop transformer. In the low-throughput small-batch setting ($bs \in \{4, 8\}$), adding KV sharing on top of CLP (row (3) $\rightarrow$ (4)) reduces latency from 5.9 $\rightarrow$ 4.8 ms (-19%) at $bs = 4$ and 6.9 $\rightarrow$ 5.6 ms (-19%) at $bs = 8$, and *PLT-2* (row (5)) stays near vanilla (row (1)) at 1.02 $\times$ (4.8 $\rightarrow$ 4.9 ms; 5.6 $\rightarrow$ 5.7 ms) while G-SWA recovers accuracy with negligible latency change. In the high-throughput large-batch setting ($bs \in \{32, 64\}$), CLP’s concurrency is the main driver: relative to the vanilla loop transformer (row (2)), *PLT-2* cuts per-token latency by 47% at $bs = 32$ (16.1 $\rightarrow$ 8.6 ms) and 47% at $bs = 64$ (21.4 $\rightarrow$ 11.3 ms), with residual overhead vs. vanilla limited to 1.06 $\times$ (8.1 $\rightarrow$ 8.6 ms) and 1.04 $\times$ (10.9 $\rightarrow$ 11.3 ms), respectively. Overall, *PLT-2* reduces latency by 47% compared to naive looping across batch sizes and remains within 4–6% of vanilla for practical serving ($bs \geq 32$).

3.2 Latency Comparisons under Same Accuracy

We scale the *PLT* model size to match the accuracy of the vanilla Seed-MoE baseline while improving inference efficiency.

Training recipe. The baseline is an in-house 2.5B/60B Seed-MoE model trained on 1T tokens. For *PLT*, we use a shallower model by setting the number of layers to two-thirds of the baseline, yielding a 1.7B/40B MoE

Table 4 Performance comparison of in-house Seed-MoE 2.5B and PLT-2 1.7B.

	MMLU	CEval	M.PRO	AGI	BBH	MATH	TQA	DROP	MBPP	H.Eval	Avg.
Seed-MoE (2.5B/60B)	75.1	78.2	47.4	60.6	70.9	48.6	66.3	63.9	60.8	49.4	62.1
PLT-2 (1.7B/40B)	77.3	80.5	45.8	58.9	66.7	43.5	71.6	65.0	64.0	52.4	62.6

configuration.

Accuracy and inference efficiency evaluation Recipe Same as Sec.3.1.

Results. As shown in Table 4, the 1.7B/40B MoE model with *PLT* and two loops achieves an average accuracy of 62.6, outperforming the vanilla 2.5B/60B MoE model by 0.5 points. Figure 3 illustrates decoding latency across batch sizes from 4 to 64. The 1.7B/40B MoE model with *PLT* and two loops delivers about 30% lower latency than the vanilla 2.5B/60B MoE model. In addition, its KV cache is roughly two-thirds of the baseline due to the reduced depth. Overall, *PLT* improves the scalability of Transformers, achieving similar accuracy with fewer parameters, lower KV-cache memory, and lower inference latency.

4 Related Works

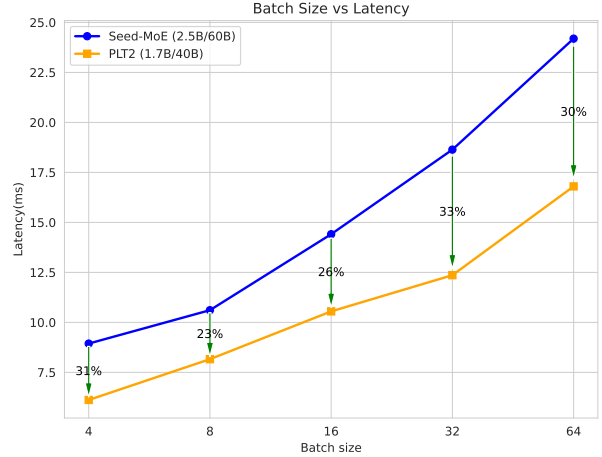
4.1 Latent Reasoning

While Chain-of-Thought (CoT) [41] improves reasoning through explicit intermediate step generation, researchers have also explored latent reasoning paradigms that internalize or extend such multi-step computation within the model itself – either vertically (looped Transformers) or horizontally (latent CoT).

Looped Transformers – vertical latent reasoning The initial studies by Dehghani et al. [11] and Lan et al. [25] introduced parameter sharing and recurrent adoption across layers, laying the foundation of looped Transformers. Saunshi et al. [32] highlighted the strong capabilities of looped Transformers in complex reasoning tasks. Meanwhile, Fan et al. [13], Yang et al. [43] demonstrated its parameter efficiency in data-fitting settings, and superior length generalization on RASP-L tasks. More recent works such as Chen et al. [7] and Geiping et al. [14] allocate greater computational depth to complex tokens through looping, enabling models to scale up inference-time reasoning capacity without increasing parameter count.

Latent CoT – horizontal latent reasoning Previous works [15, 46] enhance the model’s reasoning capability by inserting special discrete, non-semantic tokens into the sequence, thereby allocating additional computational steps to refine its intermediate representations. In contrast, recent studies by [16, 35] compress reasoning steps into continuous latent thoughts, achieving more efficient and expressive internal reasoning. These latent CoT methods demonstrate notable improvements across reasoning benchmarks such as GSM8K [9], NaturalQuestions [22], and CommonsenseQA [36], etc.

Both looped Transformers and latent CoT reasoning suffer from inferior inference efficiency due to their inherently sequential computation – whether across loops or by tokens, while our proposed PLT innovatively overlaps the computation of different loops in different tokens, leading to extreme inference efficiency.

**Figure 3** Batch size vs latency on Seed-MoE (2.5B/60B) and PLT-2 (1.7B/40B).

4.2 Parallelized Computation

The parallelized computation during inference in Large Language Models is a relatively new research area in recent years. We highlight three representative works including PHD [42], ParScale [6], and StagFormer [10]

PHD PHD [42] presents that utilizing parallel computation can lead to scalable performance improvement via parallelizing the forward of repeated tokens. To maintain minimum memory access overhead while achieve better performance, PHD introduces both KV cache sharing and chunk-wise sliding window attention. The drawback of PHD lies in the token repetition methodology, which is an inefficient method of utilizing parallel computation, since the hidden representations in the former transformer layers are very similar. Under high-throughput serving scenarios, the improved performance of PHD can not compensate for the loss of throughput due to increased decoding computation.

ParScale Latter, ParScale [6] presents that using sequence repetition with prefix tuning [27] can also lead to scalable performance improvement. The drawback of ParScale lies in its inefficiency by introducing $P \times$ KV cache when P inference streams are activated, leading to overhead both in KV cache footprint and inference latency, especially in the high-throughput serving scenarios.

StagFormer StagFormer [10] proposes a time-staggered decoding mechanism that parallelizes Transformer layer execution along the depth axis by splitting layers into multiple stacks, where the upper stack attends to the lower stack’s hidden states from the previous time step via cross-attention. This achieves partial layer-level parallelism but still suffers from incomplete parallelism in attention computation and memory access, leading to limited efficiency gains. Specifically, The separate-weight variant doubles hardware usage but achieves less than 50% throughput improvement, while the weight-sharing variant, though lighter in parameters, incurs extra KV-cache cost and additional cross-attention overhead compared to our *loop2+CLP* design, resulting in strictly worse inference efficiency. With completed parallelism and sharing strategy over the looped transformer, our proposed PLT further presents more impressive inference efficiency by reducing the KV cache footprint by over 50% with no performance degradation.

In this paper, our presented PLT improves the utilization of parallel computation to an unprecedented level. By discovering loop transformers are naturally suitable for parallel computation utilization, we propose cross-loop parallelism to improve looped transformer, maintaining the performance of loop transformers while achieving especially better inference efficiency.

5 Conclusion

In this paper, we present Parallelized Looped Transformer (PLT), which proposes **Cross-Loop Parallelism (CLP)** that overlaps the computation and the memory-access of *latter loops in previous tokens* and *previous loops in latter tokens*, and **gated sliding window attention (gated-SWA)** to achieve parallelism in the access of KV cache with no performance degradation. PLT presents impressive performance improvement with negligible latency overhead compared with the vanilla Transformer under memory-access bottle-neck, and clearly better inference efficiency with similar or even better performance compared with vanilla looped transformers with the same inference computation budget, showing the potential of parallel computation utilization.

6 Contributions and Acknowledgments

Names are Listed in the alphabetic order.

Project Lead

Bohong Wu

Core Contributors

Mengzhao Chen, Xiang Luo, Bohong Wu, Shen Yan

Infrastructure

Xiang Luo, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong

Model Architecture

Mengzhao Chen, Bohong Wu, Shen Yan, Qifan Yu, Xun Zhou

Supervision

Siyuan Qiao, Xingyan Bin

Acknowledgments

We thank Jianqiao Lu, Yunshui Li, Yao Luo, Jin Ma, Yiyuan Ma, Yutao Zeng, Chaoyi Zhang, Zhijian Zhuo as well as other colleagues at ByteDance for their support for this project.

References

- [1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.
- [2] 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.
- [3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.
- [4] Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, and Ping Luo. Efficientqat: Efficient quantization-aware training for large language models. arXiv preprint arXiv:2407.11062, 2024.
- [5] Mengzhao Chen, Chaoyi Zhang, Jing Liu, Yutao Zeng, Zeyue Xue, Zhiheng Liu, Yunshui Li, Jin Ma, Jie Huang, Xun Zhou, et al. Scaling law for quantization-aware training. arXiv preprint arXiv:2505.14302, 2025.
- [6] 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.
- [7] Yilong Chen, Junyuan Shang, Zhenyu Zhang, Yanxi Xie, Jiawei Sheng, Tingwen Liu, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. arXiv preprint arXiv:2502.13842, 2025.
- [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] Dylan Cutler, Arun Kandoor, Nishanth Dikkala, Nikunj Saunshi, Xin Wang, and Rina Panigrahy. Stagformer: Time staggering transformer decoding for running layers in parallel. arXiv preprint arXiv:2501.15665, 2025.
- [11] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In International Conference on Learning Representations, 2019. URL <https://openreview.net/forum?id=HyzdRiR9Y7>.
- [12] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. 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 2368–2378, 2019.
- [13] Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. arXiv preprint arXiv:2409.15647, 2024.
- [14] Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. arXiv preprint arXiv:2502.05171, 2025.
- [15] Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens. In The Twelfth International Conference on Learning Representations.
- [16] 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.
- [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] Huanqi Hu, Bowen Xiao, Shixuan Sun, Jianian Yin, Zhexi Zhang, Xiang Luo, Chengquan Jiang, Weiqi Xu, Xiaoying Jia, Xin Liu, et al. Liquidgemm: Hardware-efficient w4a8 gemm kernel for high-performance llm serving. arXiv preprint arXiv:2509.01229, 2025.
- [20] Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Yao Fu, et al. C-eval: A multi-level multi-discipline chinese evaluation suite for foundation models. Advances in Neural Information Processing Systems, 36:62991–63010, 2023.
- [21] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b. CoRR, abs/2310.06825, 2023. doi: 10.48550/ARXIV.2310.06825. URL <https://doi.org/10.48550/arXiv.2310.06825>.
- [22] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. Transactions of the Association for Computational Linguistics, 7:453–466, 2019.
- [23] 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 Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023, pages 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL <https://doi.org/10.1145/3600006.3613165>.
- [24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles, 2023.
- [25] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In International Conference on Learning Representations, 2020. URL <https://openreview.net/forum?id=H1eA7AEtvS>.
- [26] Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, et al. Minimax-01: Scaling foundation models with lightning attention. arXiv preprint arXiv:2501.08313, 2025.
- [27] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 4582–4597, 2021.
- [28] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. arXiv preprint arXiv:2405.04434, 2024.
- [29] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437, 2024.
- [30] Amirkeivan Mohtashami, Matteo Pagliardini, and Martin Jaggi. Cotformer: More tokens with attention make up for less depth. In Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023), 2023.
- [31] 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.
- [32] Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. arXiv preprint arXiv:2502.17416, 2025.
- [33] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. Advances in Neural Information Processing Systems, 37:68658–68685, 2024.

- [34] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051, 2023.
- [35] Jihoon Tack, Jack Lanchantin, Jane Yu, Andrew Cohen, Ilia Kulikov, Janice Lan, Shibo Hao, Yuandong Tian, Jason Weston, and Xian Li. Llm pretraining with continuous concepts. *arXiv preprint arXiv:2502.08524*, 2025.
- [36] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*, 2018.
- [37] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. In *International Conference on Machine Learning*, pages 47901–47911. PMLR, 2024.
- [38] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.
- [39] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.
- [40] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyang Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.
- [41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- [42] Bohong Wu, Shen Yan, Sijun Zhang, Jianqiao Lu, Yutao Zeng, Ya Wang, and Xun Zhou. Efficient pretraining length scaling. *arXiv preprint arXiv:2504.14992*, 2025.
- [43] Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023.
- [44] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.
- [45] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *CoRR*, 2024.
- [46] Eric Zelikman, Georges Raif Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah Goodman. Quiet-star: Language models can teach themselves to think before speaking. In *First Conference on Language Modeling*, 2024.
- [47] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- [48] Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. Agieval: A human-centric benchmark for evaluating foundation models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2299–2314, 2024.
- [49] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. A survey on efficient inference for large language models. *CoRR*, 2024.
- [50] Zhijian Zhuo, Yutao Zeng, Ya Wang, Sijun Zhang, Jian Yang, Xiaoqing Li, Xun Zhou, and Jinwen Ma. Hybridnorm: Towards stable and efficient transformer training via hybrid normalization. *CoRR*, 2025.

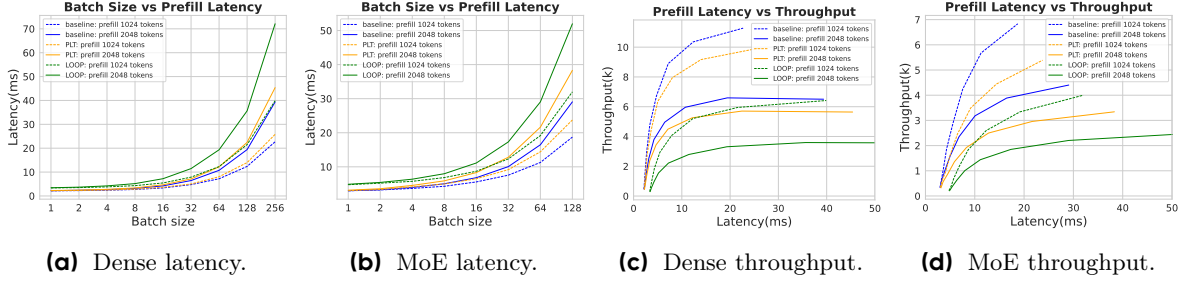


Figure 4 Inference efficiency analysis including latency and throughput for vanilla transformer, PLT and looped transformer over 1 billion activated parameters. We use FP8 quantization during inference based on VLLM [23].

Table 5 Performance evaluation of 1.2B dense models and 1B/7B MoE models across the baseline, vanilla looped transformer with $L = 2$ and our proposed PLT with $L = 2$. Evaluated benchmarks include: MMLU, Hellaswag (Hella.), ARC-Challenge (ARC-C), ARC-Easy (ARC-E), PIQA, Winogrande (Wino.), and Commonsense QA (Comm.).

	Loss	MMLU	Hella.	ARC-C	ARC-E	PIQA	Wino.	Comm.	Avg.
Vanilla Dense	2.577	35.5	62.5	38.1	71.4	74.9	60.6	44.7	55.4
+ <i>loop-2</i>	2.532	36.3	66.1	40.5	73.9	75.6	64.6	46.4	57.6
+ <i>PLT-2</i>	2.537	36.8	65.4	41.5	72.3	76.5	61.4	47.9	57.4
Vanilla MoE	2.342	37.3	67.2	40.5	72.1	76.3	62.7	48.2	57.8
+ <i>loop-2</i>	2.302	38.7	70.3	44.1	75.1	77.0	64.1	48.2	59.6
+ <i>PLT-2</i>	2.280	38.2	71.0	43.1	77.2	78.7	63.5	48.7	60.0

Appendix

A Experiments opensource

We arrange our opensource experiments in two-fold.

- We present the open source Experiments on both the dense model series and MoE model series, with over 1 billion activated Parameters in Section A.2.
- We present similar ablation study in Section A.3 on the dense models, as a complementary of the our in-house Seed-MoE experiments.

A.1 Evaluation Datasets

We used the following open-source datasets in our evaluation, including MMLU [17], HellaSwag [47], ARC [8], PIQA [2], Winogrande [31], CommonsenseQA [36].

A.2 Main Experiments

A.2.1 Training and Evaluation Settings

Our open-source implementation is based on OLMo and OLMoE. We compare our method with the vanilla transformer and vanilla looped transformer with the same activated parameters.

Dense Models Recipe For dense models, we set the number of transformer layers to 16. We set the hidden dimension to 2048. For MLP modules in these models, we set the MLP hidden dimension to 16384. We use FFN output norm. For Attention modules in these models, we use GQA with 32 query heads and 8 key/value heads. We use query/key/value layernorm at the same time, while no attention output norm. For LM head modules, we use weight tying that shares the parameter with the embedding layer.

Table 6 Component analysis of PLT . Compared with vanilla looped transformer, we introduce two extra components for the consideration of both inference speed and performance. *Lat.* is the abbreviation for latency(ms).

	Loss	MMLU	Hella.	ARC-C	ARC-E	PIQA	Avg.	Lat.@bs=1	Lat.@bs=16
(1) Vanilla Dense	2.880	29.2	45.7	26.4	61.8	69.7	46.6	1.68	2.49
(2) + <i>loop-2</i>	2.856	28.7	47.1	26.1	60.9	70.6	46.7	2.66	3.92
(3) + <i>loop-2</i> + <i>CLP</i>	2.840	30.0	48.3	30.8	59.5	69.7	47.8	1.73	3.23
(4) + <i>loop-2</i> + <i>CLP</i> + <i>KVshare</i>	2.858	29.6	46.5	28.4	60.2	69.8	46.9	1.73	2.69
(5) + <i>loop-2</i> + <i>CLP</i> + <i>KVshare</i> + <i>G-SWA</i>	2.844	29.7	47.4	30.1	62.3	69.6	47.8	1.73	2.70

For training dynamics, we train the model for 400B tokens in total, with global training batch size set to 1024. We use the cosine learning rate schedule, using 3e-4 as the peak learning rate and 2000 steps warmup.

Note that compared with the original setting of public available opensource model OLMo-1B, we have made slight modifications on the norm settings due to the consideration of training stability [50].

MoE Models Recipe For MoE models, we set the number of transformer layers to 16. We set the hidden dimension to 2048. For MLP modules in these models, we use SwiGLU experts and the 8 in 64 recipe. For Attention modules in these models, we use MHA with 16 attention heads in total. For LM head modules, we also use weight tying that shares the parameter with the embedding layer.

For training dynamics, we train the model for 400B tokens in total, with global training batch size also set to 1024. We use the cosine learning rate schedule, using 4e-4 as the peak learning rate and 2500 steps warmup.

Efficiency Evaluation Recipe Based on VLLM [23], we analyze the efficiency including both the latency and throughput, of all these baseline models. We vary the prefilling length in [1024, 2048] and the serving batch size in [1, 2, 4, 8, 16, 32, 64, 128, 256], which gradually shifts from memory-access bottleneck to compute bottleneck. For serving, we use FP8 quantization which is closer to the real industrial serving scenarios. The latency metric is averaged across 5 independent runs of decoding 256 tokens. We conduct the efficiency analysis experiments on one single GPU.

A.2.2 Analysis

Table 5 presents the performance of variants on Dense Models and MoE models. The observations are similar with our in-house experiments that PLT-2 achieves very similar performance with vanilla looped transformers. Figure 4 further presents the efficiency analysis, where we also obtain similar observations that PLT-2 presents obviously better efficiency than vanilla looped transformers.

A.3 Ablation study

A.3.1 Training/Evaluation Settings

Dense Models Recipe The training settings in this section mainly follows the setting in Section A.2.1. Except for the following two modifications.

- We change GQA with 16 query heads and 4 key value heads, and set hidden dimension to 1536 and MLP hidden size to 8192.
- We change the training token numbers to 100B, with 3e-4 as the peak learning rate and the cosine learning rate scheduler, warm-up 2000 steps.

MoE Models Recipe The training settings in this section mainly follows the setting in Section A.2.1. Different recipes are also presented as follows.

- We change the layer numbers to 12 and hidden dimension to 1536.

- For training dynamics, we also set the number of training tokens to 100B, with $3\text{e-}4$ as the peak learning rate and the cosine learning rate scheduler, warm-up 2000 steps.

Evaluation Settings The evaluation settings mainly follows Section [A.2.1](#), except for we use one H800 GPU for evaluation.

A.3.2 Analysis

Table [A.3.1](#) presents our ablation study on OLMo2, where the observation are similar with our in-house experiments.