MEMORY-EFFICIENT LOOPED TRANSFORMER: DECOUPLING COMPUTE FROM MEMORY IN LOOPED LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent LLM architectures have emerged as a promising approach for improving reasoning, as they enable multi-step computation in the embedding space without generating intermediate tokens. Models such as Ouro (Zhu et al., 2025) perform reasoning by iteratively updating internal representations while retaining a standard Key-Value (KV) cache across iterations, causing memory consumption to grow linearly with reasoning depth. Consequently, increasing the number of reasoning iterations can lead to prohibitive memory usage, limiting the practical scalability of such architectures. In this work, we propose **Memory-Efficient Looped Transformer (MELT)**, a novel architecture that decouples reasoning depth from memory consumption. Instead of using a standard KV cache per layer and loop, MELT maintains a single KV cache per layer shared across reasoning loops, while retaining per-loop KV caches for a small sliding window. This cache is updated over time via a learnable gating mechanism. To enable stable and efficient training under this architecture, we propose to train MELT using *chunk-wise training* to align training and inference. Empirically, we show that MELT models fine-tuned from pretrained Ouro parameters maintain the original model performance and **outperform standard LLMs of comparable size**, while maintaining a memory footprint comparable to those models and dramatically smaller than Ouro's. Overall, MELT achieves constant-memory iterative reasoning without sacrificing LoopLM performance, using only a lightweight post-training procedure.

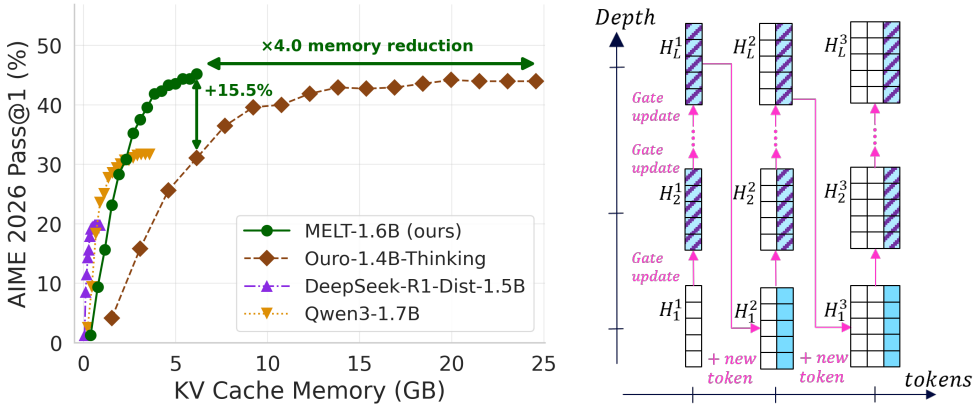

Figure 1: **MELT Memory Reduction and Architecture Overview. Left:** AIME26 accuracy as a function of the total memory footprint. At ~ 6.5 GB KV footprint, MELT improves performance by $\sim 15.5\%$ compared to Ouro. MELT achieves $\sim \times 4.0$ memory compared to Ouro at same performance. **Right:** KV cache dynamics. H_t^n denotes the latent state at time step t for token n in a given layer, before being projected into KV. Our single shared KV cache adds one row per token.

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.

1 INTRODUCTION

Large Language Models (LLMs) increasingly rely on inference-time compute to improve reasoning, shifting away from purely scaling training-time compute. A dominant approach is Chain-of-Thought (CoT) prompting, where models generate intermediate "thinking" tokens before producing a final answer. While effective, this couples reasoning depth to output length, increasing latency and memory usage. An alternative is latent reasoning, where models perform additional internal computation without generating extra tokens.

A prominent instantiation of latent reasoning is looped transformers, which perform recurrence at the architecture level by repeatedly passing hidden states through the same transformer stack. This approach was first explored in Universal Transformers (Dehghani et al., 2019) and has recently shown impressive gains with Ouro (Zhu et al., 2025), demonstrating that looped models can match or surpass transformers nearly twice their size. However, these approaches suffer from a key limitation: memory grows linearly with the number of loops due to Key-Value (KV) states. To address this, we propose **Memory-Efficient Looped Transformer (MELT)**, which decouples reasoning depth from memory consumption by maintaining a single KV entry per token and layer, updated across loops via a learnable gating mechanism. This design preserves full attention while keeping memory usage fixed as iterative depth increases.

We demonstrate this approach by training a MELT model initialized from pretrained Ouro (Zhu et al., 2025) weights. Empirically, we show that MELT outperforms similarly sized standard transformers on reasoning benchmarks while preserving the performance of the originating LoopLM, but with dramatically lower memory than looped baselines that retain per-loop KV growth. The main contributions of this paper are:

- We introduce **MELT**, a memory-efficient looped transformer architecture that decouples reasoning depth from memory consumption by sharing a single KV-cache per layer across reasoning loops and updating it with a learnable gating mechanism.
- We propose a data-efficient procedure for adapting pretrained LoopLMs to MELT through **chunk-wise training** and knowledge distillation on all loops.
- We empirically show that a MELT model initialized from pretrained Ouro parameters **outperforms standard LLMs** of comparable size, while matching their memory footprint and using substantially less memory than Ouro.

All the code to replicate our experiments and the models will be released soon.

2 RELATED WORK

This section provides a concise overview of related works, see Appendix A for an extended version.

Looped transformers. While CoT (Wei et al., 2022) emphasizes horizontal reasoning, a complementary line of work explores vertical reasoning via recurrent architectures. Early approaches such as HRM and TRM (Wang et al., 2025; Jolicoeur-Martineau, 2025), as well as adaptive-depth methods that dynamically skip or repeat layers (Li et al., 2025; Fu et al., 2026), highlight the benefits of iterative computation. More broadly, looped transformers have emerged as a strong architectural paradigm, outperforming similarly sized vanilla transformers on multi-hop reasoning, length generalization, and algorithmic tasks (Saunshi et al., 2025; Kohli et al., 2026; Fan et al., 2025; Yang et al., 2024). Despite classical optimization challenges such as instability and vanishing gradients (Dehghani et al., 2019), recent work demonstrates stable training at scale (Zhu et al., 2025; Geiping et al., 2025; Prairie et al., 2026) across different designs, including fully looped stacks and middle-cycle architectures (Geiping et al., 2025; Zeitoun et al., 2026). These results establish looped transformers as a promising direction for scalable reasoning through iterative compute.

KV cache compression and vertical sharing. Efficient KV cache management is critical in looped and long-context models, where memory typically scales with effective depth. Prior work has explored redundancy across heads, layers, and recurrence steps, including MQA/GQA for head sharing (Shazeer, 2019; Ainslie et al., 2023) and cross-layer reuse methods such as CLA

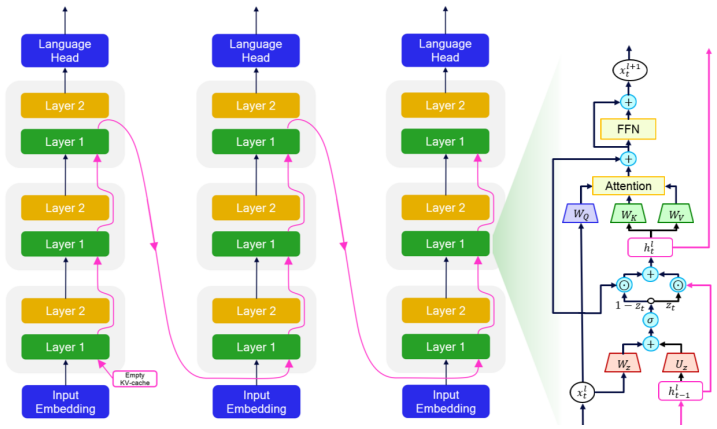

Figure 2: **Visualization of the MELT architecture and its KV cache dynamics.** The pink arrows highlight the flow of the KV cache for layer 1. The schematic emphasizes the shared-cache path and abstracts away the small local sliding window over the last W generated tokens.

and MLA (Brandon et al., 2024; DeepSeek-AI et al., 2024). In looped transformers, several approaches reduce KV growth by selectively reusing or compressing cached states, including hybrid global–local attention (Wu et al., 2025), recursion-aware caching and sharing (Bae et al., 2025), and untrained reuse across loops (Geiping et al., 2025; Zhu et al., 2025). While some of these methods can reduce memory costs in constrained settings, their effectiveness remains limited on long, complex reasoning tasks, where they often lead to performance degradation when applied to stronger models and longer reasoning traces (see Appendix B).

3 Memory-Efficient Looped Transformer

3.1 Preliminaries

Notation. Throughout the paper, we use the following notation. The model has N *layers*, each a distinct transformer block with its own parameters, and uses a *hidden dimension* d for internal representations. The *sequence length*, L, corresponds to the number of tokens in the input. The *reasoning depth* or *time index*, T, refers to the number of *reasoning loops* or *time steps* applied to a single token.

LoopLM architecture. We adopt the LoopLM architecture (Zhu et al., 2025) for causal sequence modeling, following the formulation used in prior looped-reasoning models. This design increases per-token computation without expanding the parameter count. Let $\mathrm{emb}(\cdot) : \mathbb{R}^{|V|} \to \mathbb{R}^d$ denote the token embedding map, $\mathcal{T}_\theta(\cdot) : \mathbb{R}^{L\times d} \to \mathbb{R}^{L\times d}$ a causal Transformer layer with parameters θ, and $\mathrm{lmhead}(\cdot) : \mathbb{R}^d \to \mathbb{R}^{|V|}$ the output projection. A standard (non-looped) language model composes N layers as $\mathcal{M} = \mathcal{T}_{\theta_N} \circ \cdots \circ \mathcal{T}_{\theta_1}$. In the looped setting, this stack is applied repeatedly for T iterations, so the forward pass becomes:

$$F(\cdot) = \mathrm{lmhead} \circ \mathcal{M} \circ \mathrm{emb}(\cdot) \;\to\; F^{(T)}(\cdot) = \mathrm{lmhead} \circ \underbrace{\mathcal{M} \circ \cdots \circ \mathcal{M}}_{T \text{ iterations}} \circ \mathrm{emb}(\cdot).$$

3.2 Architecture

There are three key differences that separate our architecture from LoopLM:

- The per-layer KV cache has a fixed size independent of the reasoning depth, except for a small sliding window over the most recent tokens. Consequently, the total cache scales as $\mathcal{M}_{\text{MELT}} \propto \mathcal{O}(N \times L + W \times L \times T)$, where $W \ll N$ is the window size, compared to $\mathcal{M}_{\text{LoopLM}} \propto \mathcal{O}(N \times L \times T)$.
- Instead of appending a new state at every loop step, each loop *updates* the cached state of the token. A new state is added only at the first time step, and after all iterations these updated states are passed to subsequent tokens.

- Our gating mechanism enables each token, at each time step, to attend to keys and values that integrate information across *all* time steps of preceding tokens, rather than only the current step.

Preserving query-key alignment and memory dynamics. A key design choice in MELT is to maintain a separate latent state h that evolves across iterations, from which keys and values are derived through learned projections (W_K, W_V), rather than directly updating the KV cache at each loop step. This choice is motivated by preserving semantic integrity, decoupling memory updates from attention retrieval, and maintaining query–key alignment. By evolving a latent state and projecting it into K, V space, we preserve alignment across recurrent updates while separating memory dynamics from retrieval.

This design also leads to a fundamentally different memory behavior. Standard looped transformers follow an append-only strategy, where the per-layer KV cache grows linearly with both sequence length L and reasoning depth T, i.e., $\mathcal{M}^{(l)}_{\text{LoopLM}} \propto \mathcal{O}(L \times T)$, resulting in prohibitively large memory overhead for deep reasoning. In contrast, MELT maintains a latent state $h_t^{(l)}$ with size independent from depth, yielding $\mathcal{M}^{(l)}_{\text{MELT}} \propto \mathcal{O}(L)$. The latent state is updated via a learnable gated momentum mechanism:

$$\begin{aligned} z_t^{(l)} &= \sigma\left(x_t^{(l)} W_z^{(l)} + h_{t-1}^{(l)} U_z^{(l)} + b_z^{(l)}\right), \\ h_t^{(l)} &= z_t^{(l)} \odot h_{t-1}^{(l)} + (1 - z_t^{(l)}) \odot x_t^{(l)} \end{aligned} \tag{1}$$

where $x_t^{(l)}$ is the hidden state and $z_t^{(l)}$ the gating function. Outside the bounded sliding window, this reduces the depth-wise memory complexity to $\mathcal{O}(1)$ per layer, effectively recovering the footprint of non-looped transformers. As a result, the burden of retaining information shifts from explicit storage (KV cache) to the learned gating dynamics, which determine what information is preserved or overwritten over time.

Integration into the Transformer. This latent state is then used to generate the key and value representations for the current token,

$$k_t^{(l)} = h_t^{(l)} W_K^{(l)} \qquad v_t^{(l)} = h_t^{(l)} W_V^{(l)}$$

where $W_K^{(l)}, W_V^{(l)} \in \mathbb{R}^{d \times d}$ are learned projection matrices. The resulting $k_t^{(l)}$ and $v_t^{(l)}$ are appended to the KV-cache produced by earlier tokens at the same layer,

$$K_t^{(l)} = \left[K^{(l)}, k_t^{(l)}\right], \qquad V_t^{(l)} = \left[V^{(l)}, v_t^{(l)}\right],$$

which is then consumed by the attention mechanism to compute the updated hidden state $x_t^{(l+1)}$

$$x_{attn}^{(l)} = Attn^{(l)}(q^{(l)}, K_t^{(l)}, V_t^{(l)}) + x_t^{(l)} \qquad x_t^{(l+1)} = FFN^{(l)}(x_{attn}^{(l)}) + x_{attn}^{(l)}$$

Per-loop sliding-window KV cache. MELT also keeps a sliding window over the last W generated tokens. For tokens outside this window, the keys and values used by attention are computed from the final-loop latent state $h_T^{(l)}$. For tokens inside the window, attention at loop t instead computes keys and values from the current-loop latent state $h_t^{(l)}$. Therefore, only the last W generated tokens retain loop-specific keys and values, yielding a per-layer KV cache size of $\mathcal{M}^{(l)}_{\text{MELT}} \propto \mathcal{O}(L \times T)$ for this window. During prompt prefilling, each loop constructs its cache from previous-token hidden states at the same loop, preserving parallel processing across prompt tokens.

An overview of the MELT architecture is shown in Figure 2. Further theoretical analysis and analysis of gradient flow and stability is provided in Appendix F.

3.3 Training details

A key challenge in training MELT arises from its KV-cache computation, which introduces a sequential dependency across tokens: the KV cache for token $t{+}1$ can only be computed after completing the forward pass for token t. This contrasts with standard transformers (and Ouro), where KV caches depend only on per-layer activations, enabling parallel token processing during SFT. While fully autoregressive training would respect this dependency, it is prohibitively slow, whereas bypassing the final reasoning loop restores parallelism but introduces a mismatch with inference dynamics.

Chunk-wise training. To balance efficiency and fidelity, we propose *chunk-wise training*, illustrated in Figure 4. Sequences are split into fixed-length chunks processed sequentially, while computations within each chunk are performed in parallel using the current loop's latent state. Across chunks, the full computation is completed and the final latent state is propagated, better approximating autoregressive inference. The chunk size controls the fidelity–efficiency trade-off: smaller chunks more closely match inference at the cost of throughput, while larger chunks improve efficiency but introduce greater deviation.

Knowledge distillation on all loops. To preserve alignment with the pretrained model, we apply Knowledge Distillation (Hinton et al., 2015) using the initial LoopLM as teacher, applying supervision at all reasoning loops. This denser signal improves convergence and stabilizes training.

4 Experimental results

4.1 Experimental setup

We initialize our models, MELT-1.6B and MELT-3.1B, using the pretrained weights of Ouro-1.4B-Thinking and Ouro-2.6B-Thinking respectively (Zhu et al., 2025), except for the new gating parameters, which are initialized randomly. Because MELT modifies the KV cache structure and introduces randomly initialized gating parameters, this hybrid initialization leads to initially incoherent outputs. To address this, we fine-tune the full model using *chunk-wise* training, as described in Subsection 3.3, on the AceReason-1.1-SFT (Liu et al., 2025) and OpenThoughts3 (Guha et al., 2025) datasets, focused on mathematical reasoning and coding. A summary of all training hyperparameters is shown in Table 7. In total, training required 90 hours on a node with 8 H100 GPUs (80GB), corresponding to 720 GPU-hours. We use a sliding window of size $W = 50$ in all experiments. Further details on the compute used for preliminary experiments, ablations, and testing are provided in Appendix C. We emphasize that the training cost of MELT from Ouro is just a tiny fraction of the total model training, as it requires only 32K samples. This is negligible relative to Ouro's overall training budget; for instance, it uses 16.4M tokens in its final training stage alone. Even accounting for MELT slower training throughput, MELT training would still require less than 5% of the time spent on Ouro's final training stage.

To evaluate the reasoning capabilities of MELT, we benchmark the model on six mathematical reasoning benchmarks (AIME24 (Mathematical Association of America, 2024), AIME25 (Mathematical Association of America, 2025), AIME26 (Mathematical Association of America, 2026), AMC23 (Mathematical Association of America, 2023), MATH500 (Lightman et al., 2023), OlympiadBench (He et al., 2024)) and four general reasoning benchmarks (GPQA (Rein et al., 2023), HLE (Phan et al., 2026), MMLU-Redux (Gema et al., 2024; Hendrycks et al., 2021b), HumanEval+ (Chen et al., 2021b)). For context, we compare MELT-1.6B performance with the state-of-the-art non-looped models of its size (Qwen3-1.7B (Yang et al., 2025), Gemma4-E2B (Google, 2024), Qwen3.5-2B (Team, 2026), DeepSeek-R1-1.5B (Guo et al., 2025)), as well as the looped model Ouro-1.4B-Thinking (Zhu et al., 2025), from which MELT-1.6B is derived. We evaluate all models with LightEval v0.8.1, using the default benchmark prompts, extraction procedures, and evaluation settings, as detailed in Appendix C. Following (Zhu et al., 2025), we use temperature 1.0 and top-p 0.7; all evaluations use a maximum completion length of 32k tokens.

4.2 Reasoning performance

Table 1 shows that MELT-1.6B consistently outperforms all non-looped baselines across both mathematical and general reasoning benchmarks, while maintaining a comparable memory footprint. In particular, in mathematical benchmarks MELT is only surpassed by Qwen3-1.7B and Gemma4-E2B on AMC23, and in other reasoning bencharks only by Qwen3.5-2B on GPQA. On all the other benchmarks, MELT achieves a superior performance, showing on average significant reasoning improvements, across both mathematical and general reasoning tasks.

MELT's performance is on par with Ouro's, despite retaining a full per-loop KV cache and therefore incurring substantially higher memory usage. Notably, MELT outperforms Ouro on AIME24 pass@1, AIME25 pass@10, and AIME26 on both pass@1 and pass@10. It also achieves better results across all non-math benchmarks, with an average improvement of 3%. Overall, these results

Table 1: Performance comparison of MELT-1.6B and similar sized SOTA models across benchmarks. We use **bold** and underlining to denote the best and second-best performance, respectively.

Dataset	Metric	Ouro-1.4B Thinking	MELT 1.6B	Qwen3 1.7B	Gemma4 E2B	Qwen3.5 2B	DeepSeek-R1 1.5B
AIME24	pass@1	50.2±1.6	**50.4**±1.6	43.1±1.5	40.6±1.8	19.0±1.3	32.1±1.6
	pass@10	**81.5**±1.9	77.5±2.4	76.2±2.6	68.5±2.8	47.0±2.5	75.7±3.5
AIME25	pass@1	**36.7**±1.5	35.8±1.3	33.1±1.3	26.5±1.3	16.9±1.1	20.4±1.2
	pass@10	69.0±2.5	**76.0**±2.7	58.6±2.7	50.1±2.9	37.1±2.8	46.0±2.7
AIME26	pass@1	44.0±1.5	**45.0**±1.6	31.7±1.4	36.0±1.7	16.0±1.3	19.8±1.3
	pass@10	73.2±2.4	**78.9**±2.0	61.5±2.9	58.3±2.6	46.7±2.4	48.7±2.7
AMC23	pass@1	81.2±1.2	77.2±1.2	79.2±1.2	**82.7**±1.1	64.4±1.4	70.9±1.3
	pass@10	96.6±1.0	96.5±1.5	**97.8**±1.5	95.0±1.9	92.0±1.2	96.1±1.4
MATH-500	accuracy	**94.4**±1.0	93.4±1.1	90.6±1.3	87.6±1.5	79.4±1.8	84.2±1.6
OlympiadB	accuracy	**67.5**±1.9	67.3±2.0	63.5±2.0	62.7±2.0	48.4±2.1	54.2±2.1
Avg math	pass@1	**62.3**	61.5	56.9	56.0	40.7	46.9
GPQA	accuracy	40.8±2.3	42.9±2.3	37.3±2.3	39.1±2.3	**45.1**±2.4	31.9±2.2
HLE	accuracy	**2.7**±0.9	1.3±0.8	1.3±0.7	2.0±0.8	1.7±0.7	2.0±0.8
MMLU-Redux	accuracy	74.2±0.6	**77.2**±0.6	73.8±0.6	75.3±0.6	75.3±0.6	53.3±0.7
HumanEval+	accuracy	76.8±3.3	**84.8**±3.0	71.3±3.5	61.6±3.6	26.2±3.4	57.3±3.9
Avg non-math	pass@1	48.6	**51.6**	45.9	45.5	37.1	36.1

Table 2: Performance of MELT-3.1B compared to Ouro-2.6B-Thinking.

	AIME24		AIME25		AMC23		MATH-500	Avg math
Model	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10	accuracy	pass@1
Ouro-2.6B-Thinking	**60.8**	81.9	**44.0**	**76.9**	**88.3**	**99.7**	**94.6**	**71.9**
MELT-3.1B	56.7	**86.7**	42.7	63.3	**88.3**	97.5	93.8	70.4

	GPQA	MMLU-R	HumanEval+	Avg non-math
Ouro-2.6B-Thinking	**50.0**	**84.8**	79.3	71.4
MELT-3.1B	46.2	83.4	**86.0**	**71.9**

highlight MELT's strong performance–efficiency trade-off, matching the performance of memory-intensive looped architectures while requiring only a fraction of their memory footprint.

MELT's performance is on-par with Ouro's, even though the latter retains a full per-loop KV cache and thus has substantially higher memory usage. Notably, MELT outperforms Ouro on AIME24 pass@1, AIME25 pass@10, and AIME26 on both pass@1 and pass@10. It also achieves better results across all non-math benchmarks, with an average improvement of 3%. Overall, these results highlight MELT's strong performance–efficiency trade-off, matching the performance of other looped architectures while requiring only a fraction of their memory footprint.

4.3 Scaling to a bigger model

We apply the same methodology to Ouro-2.6B-Thinking, without any model-specific modifications, to train MELT-3.1B. As shown in Table 2, MELT-3.1B achieves performance comparable to Ouro-2.6B across all benchmarks and outperforms it on average across non-math benchmarks, demonstrating that the MELT approach effectively scales to a larger looped models.

Table 3: Exact KV-cache memory measured with Hugging Face Transformers and derived VRAM requirements for generating a 32k-token sequence. MELT's KV-cache per token refers to the average of all tokens.

Model	Peak mem. (GB)	Model mem. (GB)	KV-cache (GB)	Avg. per-token KV-cache (MB)	Max batch
Ouro-1.4B-Thinking	26.58	2.87	23.71	0.759	11
MELT-1.6B	9.23	3.27	5.96	0.191	3
Memory improvement	$\times$2.88	–	$\times$3.98	$\times$3.98	–
Qwen3-1.7B	6.74	3.44	3.30	0.106	21

4.4 Memory-Efficient Reasoning

In this subsection, we report exact inference-time memory usage measured with Hugging Face Transformers to evaluate how MELT compares with existing baselines in terms of memory efficiency. This analysis highlights the substantial improvements achieved by MELT compared to Ouro, since for long-context generation the dominant contributor to memory usage is the KV-cache. For each model, we report the following memory measurements, obtained on a single H100 GPU using a 200-token prompt and a 32k-token generation:

- **Peak memory**: Maximum GPU memory usage observed during inference.
- **Model memory**: Memory required to store the model weights.
- **KV-cache**: Total KV-cache memory consumed at the end of the generation.
- **Average per-token KV-cache**: Average KV-cache memory consumed per generated token.
- **Max batch**: Maximum batch size that fits on a single H100 GPU.

As shown in Table 3, Ouro exhibits the largest KV-cache footprint, as its loop-specific KV growth causes memory to scale linearly with the number of reasoning loops. In contrast, MELT decouples reasoning depth from KV growth by maintaining a constant-size latent state instead of appending new KV entries, reducing memory by $\sim 3\times$. Although Qwen remains slightly more memory efficient in KV usage, the gap is small: for a 32k-token generation, Ouro exceeds Qwen by $\sim$ 20 GB, while MELT is only $\sim$ 2.5 GB more expensive. This difference stems from Qwen's use of Grouped-Query Attention (GQA), which reduces KV memory by sharing keys and values across query heads, whereas MELT does not employ GQA. Additional inference metrics, including TTFT, prefill throughput, decode throughput, total throughput, and prefill peak memory, are reported in Appendix D.

We next investigate how these memory savings translate into performance under fixed memory budgets. To study this setting, we cap the maximum sequence length available to each model and evaluate the resulting memory–accuracy trade-off. This reflects a practical deployment constraint: under a fixed memory budget, a model with a smaller KV cache can spend more tokens on solving a problem, while a model with a larger KV cache may be forced to stop earlier. As shown in Figure 1(a), MELT occupies most of the Pareto frontier between total memory and accuracy. In the low- and mid-memory regimes, MELT achieves higher accuracy than similarly sized non-looped baselines, and even surpasses Qwen3-1.7B under comparable memory budgets. Thus, MELT does not merely reduce memory consumption in isolation, but converts these memory savings into a stronger accuracy–memory trade-off across realistic sequence-length budgets.

4.5 Ablation studies

4.5.1 Gate mechanism variants

A core component of MELT is its gated update mechanism, which controls how loop-specific information is accumulated into the latent state. To assess the necessity and effectiveness of this design, we train a set of variants in which the proposed element-wise gating mechanism is replaced with simpler aggregation schemes. All other components are kept identical. Concretely, we compare the full MELT model against the following variants:

Table 4: MELT's gating mechanism ablation. **Bold** denotes best performance.

	AIME24		AIME25		AMC23		MATH-500	Avg
Model Variant	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10		
Mean	38.3	66.3	25.6	51.0	68.3	95.6	90.2	55.6
EMA-0.2	44.4	**78.7**	33.1	70.0	79.5	**97.1**	91.6	62.2
Last	44.6	77.7	33.5	65.8	**81.7**	96.5	93.4	63.3
Single-gated	46.3	73.8	30.2	64.2	78.4	96.8	**94.2**	62.3
MELT-1.6B	**50.4**	77.5	**35.8**	**76.0**	77.2	96.5	93.4	**64.2**

Table 5: Component removal ablation for MELT. Components are progressively removed one by one (top to bottom). **Bold** denotes best performance.

	AIME24		AIME25		AMC23		MATH-500
Component removed	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10	accuracy
MELT-1.6B	**50.4**	**77.5**	**35.8**	**76.0**	**77.2**	**96.5**	**93.4**
− GKD-allLoops	45.8	72.0	33.1	67.1	75.8	95.6	91.4
− Chunk-wise training	10.6	37.7	11.3	28.2	38.1	72.5	55.0
− Training	14.2	42.3	12.3	32.4	44.5	79.5	58.0

- **Mean**: the KV cache is computed as the average of the KV representations produced by all loops up to the current step.
- **Ema-0.2**: the KV cache is computed as an exponential moving average (EMA) of the KV representations up to the current step. The chosen decay factor (0.2) matches the average gate value observed in our trained MELT models.
- **Last**: the KV cache is constructed solely from the final reasoning loop, discarding information from earlier loops.
- **Single-gated**: the gated update is replaced with a scalar gate per token, such that a single gating value modulates the entire hidden state uniformly.

Table 4 shows that MELT (element-wise gating) achieves the best performance, being at least 1% better on average than the other variants, highlighting the importance of gating aggregation. Among variants without additional parameters, *Last* performs the best, with a big difference compared to *Mean* and slightly surpassing *EMA-0.2* and even *Single-gated*. Having into account that *Last* average is close to MELT's and that the mechanism is simpler and uses less parameters, it is a good alternative to take into consideration.

4.5.2 Component removal

We next perform a component removal ablation to assess the importance of individual training components in MELT. Starting from the full model, we progressively remove elements of the training procedure one by one, following their order of introduction, and fully retrain the model after each removal to ensure a fair comparison. Specifically, we remove training mechanisms one by one to isolate their contribution, following the sequence: (i) removing knowledge distillation on all loops, reducing training to standard SFT; (ii) replacing chunk-wise training with fully parallel SFT; (iii) removing all training, using Ouro weights with randomly initialized gates.

As shown by Table 5, the two main components yield a clear and consistent improvement over the preceding configuration across all benchmarks. Disabling knowledge distillation on all loops already causes a notable performance drop in most benchmarks, demonstrating its critical role in aligned the learned MELT representations. Further removing chunk-wise training leads to almost complete failure, halving the models capabilities. Interestingly, fully-parallel training degrades performance compared to using the original Ouro weights with random gate initialization, confirming that respecting MELT's sequential KV dynamics during training is indispensable. Overall, these results demonstrate that MELT's performance arises from the cumulative effect of these two capital components.

5 LIMITATIONS AND FUTURE WORK

A limitation inherited from Ouro is that the number of recurrent loops is fixed at inference time. While this provides a simple mechanism to control compute, it does not account for the fact that different inputs and tokens may require varying amounts of reasoning. Notably, MELT's memory remains independent of loop depth except for the small last-W sliding window, making it particularly well-suited for future extensions with adaptive loop depth, enabling dynamic allocation of reasoning steps based on input complexity. Another limitation carried over from Ouro is that our current implementation does not yet explore GQA. Extending MELT to grouped query attention remains an important direction, as GQA can reduce memory bandwidth and KV cache overhead during inference. Thanks to MELT's constant-memory design, combining it with GQA is especially promising and could further improve efficiency, potentially closing the remaining gap in memory usage with standard Transformer baselines. The current experiments adapt pretrained Ouro checkpoints; incorporating the MELT mechanism throughout pretraining remains future work and could enable models that are not constrained by the inherited design choices discussed above. Finally, MELT requires sequential KV updates during training, which constrains parallelism compared to standard transformer training. While our chunk-wise training and distillation procedure provides a practical adaptation path, developing more parallelizable training strategies remains an important direction for scaling MELT to larger models, longer reasoning horizons, and broader application domains.

6 CONCLUSION

We introduce MELT, an architecture that enables deep latent reasoning in looped transformers by decoupling memory usage from reasoning depth. By replacing the append-only KV-cache with a **gated, constant-size latent state**, MELT allows inference-time compute to scale without incurring linear memory growth. When integrated into Ouro, it achieves competitive reasoning performance relative to similarly sized baselines.

Training MELT, however, requires particular care. Its KV cache computation introduces a sequential dependency across tokens, preventing the fully parallel token processing typically used during training. We address this challenge through **chunk-wise training**, which provides a controllable trade-off between inference fidelity (smaller chunks) and training efficiency (larger chunks). In addition, we apply knowledge distillation with supervision on all loops, to keep MELT aligned with the original model despite the substantial architectural changes introduced by MELT. Importantly, given the minimal cost of adapting a looped transformer to MELT and the additional training overhead introduced by chunk-wise training, we propose MELT as a lightweight post-training adaptation of an existing looped model. Converting Ouro to MELT requires only 32K training samples, versus the 16.4M tokens used in Ouro's final training stage. Even with MELT's lower chunk-wise training throughput, the adaptation takes less than 5% of the time of Ouro's final training stage.

Empirically, MELT delivers strong and consistent performance across both mathematical and general reasoning benchmarks. Notably, it surpasses similarly sized standard Transformer baselines while operating under the same constant memory budget, demonstrating that improved reasoning capability can be achieved without increasing memory. To our knowledge, MELT is the first architecture to exceed the performance of standard models with the same memory footprint. These results highlight the effectiveness of looped architectures and demonstrating that iterative computation provides meaningful gains in reasoning even when memory is strictly constrained.

AI USE STATEMENT

We used a large language model as a writing and analysis aid: for prose editing of sections drafted by the authors, and for exploratory analysis scripts over result files that the authors then verified. Research questions, experimental design, model training, and all reported measurements are the authors' own. Every number in the paper was recomputed from the logged outputs by an author, and all LLM-assisted code was read and tested before use. We take responsibility for the final content of this work.

Ethics statement

All experiments are conducted using pretrained open-weight models and publicly available datasets for mathematical reasoning, coding, and general reasoning tasks. No human subjects, private user data, or personally identifiable information are collected or introduced as part of this work. The primary contribution of MELT is improved computational efficiency rather than a new capability class. By reducing the memory requirements of latent reasoning models, our method may facilitate broader experimentation with such architectures. At the same time, increased efficiency can also make powerful reasoning systems more accessible. We therefore encourage deployment and evaluation under the same safety, monitoring, and responsible-release practices that apply to the underlying language models.

Reproducibility statement

The architecture is fully specified in Section 3, including the shared KV-cache design, gated latent-state update rule, and sliding-window mechanism. The training procedure, including chunk-wise training and knowledge distillation on all loops, is described in Section 3.3. Model initialization, datasets, training compute, and evaluation protocols are reported in Section 4. Appendix C provides the complete training hyperparameters, benchmark settings, sampling parameters, and compute usage required to reproduce the main experiments and ablations. Appendix D reports additional inference and training efficiency metrics, while Appendix G documents all datasets, models, and software dependencies together with their licenses. Code, training scripts, and model checkpoints will be released upon publication.

References

Gustavo Aguilar, Yuan Ling, Yu Zhang, Benjamin Yao, Xing Fan, and Chenlei Guo. Knowledge Distillation from Internal Representations, January 2020. URL http://arxiv.org/abs/1910.03723. arXiv:1910.03723 [cs].

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models. *arXiv preprint arXiv:2305.13245*, 2023.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation, 2025. URL https://arxiv.org/abs/2507.10524.

Hugh Blayney, Álvaro Arroyo, Johan Obando-Ceron, Pablo Samuel Castro, Aaron Courville, Michael M. Bronstein, and Xiaowen Dong. A mechanistic analysis of looped reasoning language models, 2026. URL https://arxiv.org/abs/2604.11791.

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention, 2024. URL https://arxiv.org/abs/2405.12981.

Defang Chen, Jian-Ping Mei, Yuan Zhang, Can Wang, Yan Feng, and Chun Chen. Cross-Layer Distillation with Semantic Calibration, August 2021a. URL http://arxiv.org/abs/2012.03236. arXiv:2012.03236 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec

Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021c. URL https://arxiv.org/abs/2107.03374.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations (ICLR)*, 2019.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. *arXiv preprint arXiv:2409.15647*, 2025. doi: 10.48550/ARXIV.2409.15647. URL https://arxiv.org/abs/2409.15647. ICLR 2025.

Luyang Fang, Xiaowei Yu, Jiazhang Cai, Yongkai Chen, Shushan Wu, Zhengliang Liu, Zhenyuan Yang, Haoran Lu, Xilin Gong, Yufang Liu, Terry Ma, Wei Ruan, Ali Abbasi, Jing Zhang, Tao Wang, Ehsan Latif, Weihang You, Hanqi Jiang, Wei Liu, Wei Zhang, Soheil Kolouri, Xiaoming Zhai, Dajiang Zhu, Wenxuan Zhong, Tianming Liu, and Ping Ma. Knowledge Distillation and Dataset Distillation of Large Language Models: Emerging Trends, Challenges, and Future Directions, January 2026. URL http://arxiv.org/abs/2504.14772. arXiv:2504.14772 [cs].

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Selective latent iterations to improve reasoning language models, 2026. URL https://arxiv.org/abs/2511.08577.

Zitian Gao, Yilong Chen, Yihao Xiao, Xinyu Yang, Ran Tao, Joey Zhou, and Bryan Dai. Loop the Loopies!, July 2026. URL http://arxiv.org/abs/2607.16051. arXiv:2607.16051 [cs.CL].

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. doi: 10.48550/ARXIV.2502.05171. URL https://arxiv.org/abs/2502.05171.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile van Krieken, and Pasquale Minervini. Are we done with mmlu?, 2024.

Google. Gemma open models, 2024. URL https://ai.google.dev/gemma.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Wu, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Nathan Habib, Clémentine Fourrier, Hynek Kydlíček, Thomas Wolf, and Lewis Tunstall. Lighteval: A lightweight framework for llm evaluation, 2023. URL https://github.com/huggingface/lighteval.

Jitai Hao, Qiang Huang, Hao Liu, Xinyan Xiao, Zhaochun Ren, and Jun Yu. A Token is Worth over 1,000 Tokens: Efficient Knowledge Distillation through Low-Rank Clone, December 2025. URL http://arxiv.org/abs/2505.12781. arXiv:2505.12781 [cs].

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems, 2024. URL https://arxiv.org/abs/2402.14008.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021a. URL https://arxiv.org/abs/2009.03300.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring Massive Multitask Language Understanding, January 2021b. URL http://arxiv.org/abs/2009.03300. arXiv:2009.03300 [cs].

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network, March 2015. URL http://arxiv.org/abs/1503.02531. arXiv:1503.02531 [stat].

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks, 2025. URL https://arxiv.org/abs/2510.04871.

Harsh Kohli, Srinivasan Parthasarathy, Huan Sun, and Yuekun Yao. Loop, think, & generalize: Implicit reasoning in recurrent-depth transformers, 2026. URL https://arxiv.org/abs/2604.07822.

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.

Ziyue Li, Yang Li, and Tianyi Zhou. Skip a layer or loop it? test-time depth adaptation of pretrained llms. *arXiv preprint arXiv:2507.07996*, 2025. doi: 10.48550/ARXIV.2507.07996. URL https://arxiv.org/abs/2507.07996.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023. doi: 10.48550/arXiv.2305.20050. URL https://arxiv.org/abs/2305.20050.

Zihan Liu, Zhuolin Yang, Yang Chen, Chankyu Lee, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. AceReason-Nemotron 1.1: Advancing Math and Code Reasoning through SFT and RL Synergy, June 2025. URL http://arxiv.org/abs/2506.13284.

Mathematical Association of America. American mathematics competitions (amc) 10/12 2023, 2023. URL https://maa.org/.

Mathematical Association of America. American invitational mathematics examination (aime) 2024, 2024. URL https://maa.org/. Problems I and II.

Mathematical Association of America. American invitational mathematics examination (aime) 2025, 2025. URL https://maa.org/. Problems I and II.

Mathematical Association of America. American invitational mathematics examination (aime) 2026, 2026. URL https://maa.org/. Problems I and II.

Saurav Muralidharan, Sharath Turuvekere Sreenivas, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. Compact Language Models via Pruning and Knowledge Distillation, November 2024. URL http://arxiv.org/abs/2407.14679. arXiv:2407.14679 [cs].

Long Phan, Alice Gatti, Nathaniel Li, Adam Khoja, Ryan Kim, Richard Ren, Jason Hausenloy, Oliver Zhang, Mantas Mazeika, Dan Hendrycks, Ziwen Han, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, Aakaash Nattanmai, Gordon McKellips, Anish Cheraku, Asim Suhail, Luo, et al. A benchmark of expert-level academic questions to assess ai capabilities. *Nature*, 649 (8099):1139–1146, January 2026. ISSN 1476-4687. doi: 10.1038/s41586-025-09962-4. URL http://dx.doi.org/10.1038/s41586-025-09962-4.

Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y. Fu. Parcae: Scaling laws for stable looped language models, 2026. URL https://arxiv.org/abs/2604.12946.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

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. doi: 10.48550/ARXIV.2502.17416. URL https://arxiv.org/abs/2502.17416. ICLR 2025.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019. URL https://arxiv.org/abs/1911.02150.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL https://github.com/huggingface/trl.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical Reasoning Model, 2025. URL https://arxiv.org/abs/2506.21734. Version Number: 3.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art

natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Bohong Wu, Mengzhao Chen, Xiang Luo, Shen Yan, Qifan Yu, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong, Xun Zhou, Siyuan Qiao, and Xingyan Bin. Parallel loop transformer for efficient test-time computation scaling, 2025. URL https://arxiv.org/abs/2510.24824.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, and other. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2024. doi: 10.48550/ARXIV.2311.12424. URL https://arxiv.org/abs/2311.12424. Accepted at ICLR 2024.

Abbas Zeitoun, Lucas Torroba-Hennigen, and Yoon Kim. Hyperloop transformers, 2026. URL https://arxiv.org/abs/2604.21254.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models, 2025. URL https://arxiv.org/abs/2510.25741.

A Extended Related Work

Looped transformers. While CoT (Wei et al., 2022) and other ITC techniques have recently been highly influential, a complementary direction has emerged that focuses on vertical reasoning via recurrent architectures. Simple recurrent architectures such as HRM (Wang et al., 2025) and TRM (Jolicoeur-Martineau, 2025) have demonstrated strong performance on targeted reasoning tasks, while transformer-based models have also been modified to incorporate looping mechanisms. For instance, Li et al. (2025) propose a per-sample test-time depth adaptation, where a pretrained LLM's layers are treated as modules that can be skipped or repeated (looped) and reordered to form a sample-specific chain-of-layers. On a similar direction, Fu et al. (2026) introduced adaptive computation budgets, employing a classifier to dynamically allocate additional latent iterations for difficult tokens that can be skipped or repeated (looped) and reordered to form a sample-specific chain-of-layers.

Most notably within this line of work, looped transformers have emerged as a powerful architectural choice. Existing studies on simplified setups have shown that, compared to similar-sized vanilla transformers, looped transformers show superior capacity at multi-hop reasoning (Saunshi et al., 2025; Kohli et al., 2026), lenght generalization (Fan et al., 2025; Kohli et al., 2026) and learning algorithms (Yang et al., 2024). Despite the challenges of scaling such architectures when unrolled over many steps, most notably optimization instability and vanishing gradients (Dehghani et al., 2019), recent studies demonstrate that looped transformers can already be trained stably at the scale of several billion parameters (Zhu et al., 2025; Geiping et al., 2025), including reasoning-focused models (Zhu et al., 2025). Existing approaches generally follow two strategies: either looping transformer layers directly across iterations (Zhu et al., 2025), or looping only a central subset of layers using a middle-cycle design with fixed prelude and coda blocks (Geiping et al., 2025; Zeitoun et al., 2026). Additional work has reported favorable scaling behavior (Prairie et al., 2026), proposed architectural refinements (Zeitoun et al., 2026), and provided mechanistic insights into the internal dynamics of looped models (Blayney et al., 2026), further supporting looped transformers as a robust and promising research direction.

KV cache compression and vertical sharing. Efficient KV cache management is central in scaling recurrent models and long-context regimes, where cache cost scales with effective depth. Beyond head-level sharing via Multi-Query Attention (MQA) (Shazeer, 2019) and GQA (Ainslie et al., 2023), recent work exploits vertical redundancy across layers or recurrence steps. Cross-Layer Attention (CLA) (Brandon et al., 2024) shows that KV representations remain relatively stable across adjacent layers, allowing multiple layers to read from a shared cache. DeepSeek-AI et al. (2024) further compresses this information via Multi-Layer Attention (MLA) using low-rank projections.

More specifically, there are some works that address the growing KV Cache issue in looped transformers. Wu et al. (2025) propose a KV-cache computation method that combines a global component from the first loop with a local component attending to a sliding window of recent tokens in the current loop. Bae et al. (2025) propose two key mechanisms for KV cache efficiency: Recursion-wise Caching, which selectively updates and attends only to the KV pairs of active "thinking" tokens at each depth, and Recursive KV Sharing, which reuses the initial cache from the first loop across all subsequent steps. Geiping et al. (2025) show that the looped transformer, without additional training, naturally reuses the first loop KV-cache entry from previous tokens, independent of when recurrence stops, and can further compress the cache by sharing entries periodically across recurrent steps. Finally, Zhu et al. (2025) show that, without additional training, retaining only the last or average KV-cache and reusing it across all loops can preserve performance on some tasks. Although some of these approaches achieve no or moderate performance drops in their respective settings, their applicability to practical long-reasoning remains unclear as their evaluations are mostly limited to weaker looped models and short or constrained generation settings. In fact, our analysis (Appendix B) suggests that directly applying untrained cache-sharing methods to Ouro, including those proposed by Geiping et al. (2025) and Zhu et al. (2025), significantly degrades performance on long reasoning tasks.

Activation-level knowledge distillation. An extensive body of literature exists on Knowledge Distillation (KD) (Hinton et al., 2015). We build on the line of work that aligns intermediate representations rather than relying solely on final output logits. Notable examples within this vast space

include foundational works that distill internal states to compress structural knowledge (Aguilar et al., 2020) and cross-layer mechanisms designed to calibrate semantic alignment between teacher and student (Chen et al., 2021a). In the context of large language models, Muralidharan et al. (2024) recently demonstrated the efficacy of combining structural pruning with layer-wise KD to derive highly accurate, compact models without full retraining. Furthermore, recent frameworks emphasize strict internal activation alignment to prevent catastrophic drift (Hao et al., 2025) and preserve complex, multi-step reasoning trajectories (Fang et al., 2026). Building on these foundations, our approach addresses representation drift in continuous recurrent architectures by applying knowledge distillation across all MELT loops.

B ANALYSIS OF EXISTING KV-CACHE SHARING METHODS ON LONG REASONING

¡think¿Okay, so I need to simplify tan 100 degrees plus 4 times sin 100 degrees. Hmm, let me think about how to approach this. I remember some trigonometric identities, maybe I can use them here.

First, let me write down the expression: tan(100°) + 4 sin(100°). Hmm, 100 degrees is in the second quadrant, so tan is negative there because tangent is sine over cosine, and cosine is negative in the second quadrant. Sin is positive. But maybe there's a way to combine these terms.

Alternatively, maybe I can use some angle addition formulas or convert tangent into sine and cosine? Let me see. Since tangent is sine over cosine, so tan(100°) = sin(100°)/cos(100°). So then the expression becomes [sin(100°)/cos(100°)] +4 sin(100°).

Hmm, so if I get a common denominator, maybe I can combine them? Let's see. The common denominator would be cos(100°). So that would be [sin(100°) +4 sin(100°) cos(100°)] divided by cos(100°). Wait, but that might not help much. Hmm.

Alternatively, maybe there is a clever way to write this as some sine or cosine function with some angle? Let me think.

Wait, 100 degrees is 80 degrees plus 20 degrees. 80 is 60 plus 20, but not sure. Maybe I can use some addition formulas?

Alternatively, perhaps there is a way to express 4 sin(100°) as some sine of another angle. Maybe with some multiple angle formulas?

Alternatively, maybe I can write 100° as 60° +40°, because h, 60+40 is h 100. Let me see. Let me think.

Wait, maybe I can use the identity for tan(A+B), but not sure. Alternatively, maybe I can write 100° as h 80°+20°, but maybe maybe I can't think of a direct identity.

Alternatively, maybe I can use the identity that tan(x) +4 sin(x) = something. Maybe there is a formula for that?

Wait, let me try tohe following:

Let me consider the expressionation: tan(100° +4 sin(100°). Maybe I can writet it as [tan(100°) +4 sin(100°)]. to see if I can findd some angle such that this expressionation is equal to some sine or cosine functiontian ofthat angle.

Hmm, maybe maybe maybe maybe maybe I know maybe that maybe the expressionation can be simplified using the identity:

tan(A) +4 sin(A)=sqrt(5) something.

Wait, I think I heard of a formula that tan((+4 sin()=sqrt(5) or something like like like. Let me check that

Let me try to see this. Let me think that if A=1 , then the expressionation would is tan()+=4 sin(), but probably that the value of thetan 100° plus h teh 4 sin100° is approximately equal to teh sqrt(5).

h, let me seech that.

h,

Hmmm, let me seeh theangleng in that.

h, maybe,

,

h,

Wait,
h, maybe,
h, h,
the expressionason tan(h +4 sin(h')= teh sqrt(5).

Maybe, (...)

Figure 3: Example reasoning trace in Ouro-1.4B-Thinking illustrating the failure mode of last-loop KV-cache sharing with prefill.

As discussed in the related work, several studies on KV-cache sharing in looped models suggest that caches can be reused to reduce memory footprint with minimal performance degradation. However, their applicability to practical long-reasoning settings remains unclear, as existing evaluations are largely restricted to weaker looped models and short or otherwise constrained generation regimes.

In this section, we investigate whether these methods extend to state-of-the-art looped models (Ouro (Zhu et al., 2025)) on long-horizon reasoning tasks.

To do so, we evaluate several untrained KV-cache sharing strategies in which the cache is reused across recurrent loops. Specifically, we consider sharing the KV cache from the last loop, following Zhu et al. (2025), and from the first loop, following Geiping et al. (2025). Since Zhu et al. (2025) observe that preserving the full prompt KV cache can be beneficial, we evaluate both variants for each strategy: one that keeps the original KV cache for the prompt and one that applies the first/last-loop sharing rule to the prompt as well.

All four untrained KV-sharing variants obtain zero performance on several reasoning benchmarks (full results can be seen in Table 6). This contrasts with Zhu et al. (2025), who report that last-loop KV sharing with prefill achieves performance comparable to Ouro on GSM8K and MATH-500. We hypothesize that this discrepancy is partly due to their few-shot CoT evaluation setup (Appendix C.1 in Zhu et al. (2025)), which constrains the generation format and reduces the likelihood of the model drifting during decoding. In our setting, the failure mode is apparent in qualitative outputs: as illustrated in Figure 3, generations with last-loop sharing and prefill often begin coherently but eventually degenerate during extended reasoning. This behavior is consistent with an accumulation of KV-cache mismatch errors: as generation proceeds farther from the prompt, the shared cache increasingly deviates from the cache produced by the original model. More specifically, queries from earlier loops attend to keys and values produced at the final loop, creating a recurrent-depth mismatch with the original model's depth-matched states. Because standard KV values are used during prompt prefilling, an increasing fraction of the context is generated under the mismatched cache as more tokens are generated, further amplifying this effect. These results motivate the need for a constant-memory KV-cache method that remains stable over long reasoning traces, which is precisely the goal of our approach.

Table 6: Performance of untrained KV-cache sharing strategies for Ouro-1.4B-Thinking. We compare first-loop and last-loop KV-cache sharing, with and without preserving the original prompt KV cache.

	AIME24		**AIME25**		**AMC23**		**MATH-500**
	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10	
Ouro-1.4B-Thinking	**50.2**	**81.5**	**36.7**	**69.0**	**81.2**	**96.6**	**94.4**
Last loop (w/ prefill)	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Last loop	0.0	0.0	0.0	0.0	0.0	0.0	0.0
First loop (w/ prefill)	0.0	0.0	0.0	0.0	0.0	0.0	0.0
First loop	0.0	0.0	0.0	0.0	0.0	0.0	0.0

C HYPERPARAMETERS

This section provides the hyperparameters required to reproduce our training and evaluation runs. Table 7 reports the hyperparameters used for MELT-1.6B and the ablation studies, while Table 8 summarizes the benchmark evaluation settings. Note that for HLE, we evaluate on a 300-sample subset of the original dataset.

Regarding compute, the main training run for MELT-1.6B required 90 hours on a node with 8 H100 GPUs (80GB), corresponding to 720 GPU-hours. The three ablation runs also required 60 hours each on the same 8-GPU node, for a total of 1,440 GPU-hours. Evaluation required approximately 500 GPU-hours, while preliminary experiments accounted for roughly 15,000 GPU-hours. Overall, the project used approximately 17,000 GPU-hours.

D EFFICIENCY ANALYSIS

We report additional efficiency measurements using Hugging Face Transformers.

Table 7: Training hyperparameters for **MELT-1.6B**. MELT-3.1B uses the same training hyperparameters, with only architecture-specific quantities (e.g., hidden dimension, number of layers, parameter initialization) adjusted accordingly.

Parameter	**Value**
Dataset-mix	50% AceReason-1.1-SFT, 50% OpenThoughts3
# layers	24
Hidden dimension	2048
Recurrent steps	4
# gating params.	$24 \times 2048^2 \times 2 \approx 0.2$B
Original params. init.	Ouro-1.4B-Thinking
Chunk size (tokens)	500
Sliding-window size (tokens)	50
Batch size (tokens)	320K
Seq. length (tokens)	10K
Gradient norm clipping	1.0
LR scheduler	Cosine decay with warmup
Optimizer	Adam (β_1=0.9, β_2=0.95)
Weight decay	1.0×10^{-4}
Learning rate	8×10^{-6}
Gate learning rate	5.0×10^{-4}
Training steps	1000
Training tokens	320M

Table 8: Benchmark evaluation settings. Default templates are from LightEval v0.8.1

Benchmark	**Hugging Face dataset**	**Prompt template**	**Metric**	**Samples**	**Completions**
AIME 2024	`HuggingFaceH4/aime_2024`	`MATH_QUERY_TEMPLATE` (default)	pass@1, pass@10	30	16
AIME 2025	`yentinglin/aime_2025`	`MATH_QUERY_TEMPLATE` (default)	pass@1, pass@10	30	16
AIME 2026	`MathArena/aime_2026`	`MATH_QUERY_TEMPLATE` (default)	pass@1, pass@10	30	16
AMC 2023	`math-ai/amc23`	`MATH_QUERY_TEMPLATE` (default)	pass@1, pass@10	40	16
MATH-500	LightEval built-in	`MATH_QUERY_TEMPLATE` (default)	accuracy	500	1
OlympiadBench	`math-ai/olympiadbench`	`MATH_QUERY_TEMPLATE` (default)	accuracy	581	1
GPQA Diamond	`Idavidrein/gpqa`	`GPQA_QUERY_TEMPLATE` (default)	accuracy	448	1
HLE	`cais/hle`	`MATH_QUERY_TEMPLATE` (default)	accuracy	300	1
MMLU-Redux	`edinburgh-dawg/mmlu-redux-2.0`	`GPQA_QUERY_TEMPLATE` (default)	accuracy	5700	1
HumanEval+	`evalplus/humanevalplus`	`HUMANEVAL_QUERY_TEMPLATE` (custom)	accuracy	164	1

Table 9: vLLM parameters used for benchmark evaluation

Parameter	**Value**
temperature	1.0
top-p	0.7
max new tokens	32768
dtype	bfloat16
tensor parallel size	1
pipeline parallel size	1
data parallel size	1
GPU memory utilization	0.9

D.1 Inference efficiency

Table 10 uses one H100 GPU, a 200-token prompt, and a 32k-token generation. MELT reduces decode memory and supports a larger batch than Ouro, at the cost of slower batch-1 decoding. MELT preserves parallel prefill by constructing each loop's cache from previous-token hidden states at the same loop.

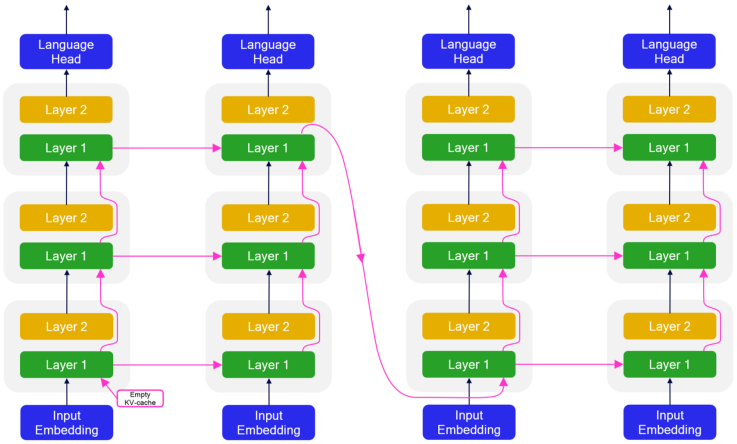

Figure 4: Visualization of *chunk-wise trianing*. Example with sequence length 4 and chunk size 2. MELT's KV cache is computed in parallel within each chunk and sequentially across chunks, balancing training efficiency with a closer approximation to autoregressive inference.

Table 10: Inference efficiency metrics for MELT-1.6B, Ouro-1.4B-Thinking, and Qwen3-1.7B.

Model	TTFT (s)	Prefill (tok/s)	Decode (tok/s)	Total (tok/s)	Prefill peak (GB)	Decode peak (GB)	Max batch (@32k)
MELT-1.6B	0.089	2254.5	8.5	8.5	3.15	9.23	11
Ouro-1.4B-Thinking	0.111	1795.7	13.5	13.5	2.86	26.58	3
Qwen3-1.7B	0.029	6973.9	41.6	41.6	3.27	6.74	21

D.2 Training efficiency

Table 11 compares MELT with a 100-step Ouro distillation run under the same configuration. Chunk-wise training results in an approximately $10\times$ wall-clock slowdown.

Table 11: Training efficiency metrics for MELT-1.6B and Ouro-1.4B.

Model	Tokens/s	Seconds/step	FLOPs/step
Ouro-1.4B	20,645.2	31.00	1.862×10^{15}
MELT-1.6B	2,168.1	295.19	2.236×10^{15}

E Notes on reproducibility and inference efficiency in Ouro

This appendix documents several technical observations we made while attempting to reproduce and analyze claims from Zhu et al. (Zhu et al., 2025). Our goal is not to diminish the contributions of Ouro, but to clarify practical aspects that are directly relevant when comparing memory usage and inference efficiency against MELT.

E.1 Reproducibility

Despite substantial effort, we were unable to fully reproduce the reported results of Ouro under the configurations described in the paper. While the authors provide the model's code and pretrained checkpoints, key implementation details and evaluation settings are either underspecified or differ from what is required to match the reported numbers. As a result, we observed non-trivial discrepancies between the performance reported in the original paper and the results obtained using the released artifacts. Therefore, throughout this work, we rely exclusively on values obtained from our own experimental measurements. We note that Gao et al. Gao et al. (2026) report similar findings when evaluating Ouro: their results are very close to ours, yet remain substantially below those reported in the original paper. We additionally observe that, under our evaluation setup, Ouro-2.6B-

Thinking does not outperform Qwen3-4B, contrary to the results reported in the original Ouro paper. Importantly, even under these independently measured evaluations, Ouro remains highly competitive and Ouro-1.4B-Thinking continues to outperform the state-of-the-art isoparametric vanilla baselines in our experiments, underscoring the strength of its underlying approach.

E.2 Early-exit gating and effective compute

A central component of Ouro is its learned gating mechanism, which is intended to enable adaptive computation by allowing tokens to exit early when additional recurrent steps are deemed unnecessary. While this mechanism is emphasized throughout the paper, we observed the following in practice:

- Ouro introduces an early-exit mechanism, but the released default configuration uses a threshold that effectively disables early exiting, and the paper does not specify how this threshold should be chosen in practice.
- Even when an early exit is triggered, the model still executes all recurrent loop computations up to the maximum depth; the gating affects which logits are selected, not whether subsequent loops are computed.

We verified this behavior by inspecting the released inference code. As a consequence, the gating mechanism does not reduce inference-time compute or memory usage under typical settings, despite its conceptual framing as an adaptive compute mechanism.

We hypothesize that this design choice is driven by KV-cache dependencies: since later tokens require access to the KV states produced at the final loop, it is not possible to terminate computation early for a given token without breaking autoregressive consistency. It is worth noting that this limitation would not apply to MELT's constant-memory KV update mechanism, although we leave a full investigation of early-exit strategies in MELT to future work.

F Theoretical analysis

As described in Section 3, MELT updates only the last row of the state matrix. Therefore, we specialize our proofs to take that aspect explicitly into account. Notwithstanding, our results are fully generalizable to the case of the full matrix update.

F.1 Spectral stability of the gated update

The next proposition shows that, in the saturated-gate regime, gradients are preserved across loops, providing the foundation for the Gradient Superhighway described in Subsection 3.2.

This analysis is limited to dimensions in which the gate saturates near one and only characterizes the gated state update. It objective is not to establish stability of the full Transformer recurrence or bound non-saturated dimensions, attention, feed-forward layers, or normalization.

Proposition F.1 (Spectral Regulation in the Saturated Regime)**.** *Let $H_t^{(i)} \in \mathbb{R}^{L\times D}$ be the latent KV state at iteration t for layer i, where L is the sequence length and D is the feature size. Let us consider h_t as the $D \times 1$ state vector corresponding to the current token in the sequence (we omit the layer index for ease of notation). Consider MELT's element-wise update rule:*

$$h_t = z_t \odot h_{t-1} + (\mathbf{1} - z_t) \odot x_t \tag{2}$$

where $x_t \in \mathbb{R}^{D\times 1}$ is the hidden state at the current layer given input token x, and $z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \in \mathbb{R}^{D\times 1}$ is the gate vector. If the gating mechanism saturates such that $z_t \to \mathbf{1}$ (element-wise) for a set of latent dimensions, then the Jacobian $J_t = \frac{\partial h_t}{\partial h_{t-1}}$, restricted to these dimensions, converges to the identity matrix. Consequently, the spectral radius $\rho(J_t) \to 1$. Thus, ensuring that the gradient magnitude is preserved over any arbitrary number of iterations.

Proof. The Jacobian of the update rule with respect to h_{t-1} is derived via the product rule applied to the Hadamard product in Equation 2:

$$J_t = \underbrace{\text{diag}(z_t)}_{\text{Term 1}} + \underbrace{\text{diag}(h_{t-1} - x_t)\frac{\partial z_t}{\partial h_{t-1}}}_{\text{Term 2}} + \underbrace{\text{diag}(\mathbf{1} - z_t)\frac{\partial x_t}{\partial h_{t-1}}}_{\text{Term 3}} \tag{3}$$

We analyze the limit behavior in the shielding regime where $z_t \to \mathbf{1}$:

1. **Term 1:** Approaches the identity matrix: $\lim_{z \to \mathbf{1}} \text{diag}(z_t) = \mathbf{I}$.
2. **Term 2:** The derivative of the sigmoid function $\sigma'(u) = \sigma(u)(1 - \sigma(u))$ vanishes as $z_t \to \mathbf{1}$. Thus, $\frac{\partial z_t}{\partial h_{t-1}} \to \mathbf{0}$.
3. **Term 3:** The term $(\mathbf{1} - z_t)$ approaches $\mathbf{0}$, nullifying the contribution of the recurrent weight matrix in $\frac{\partial \tilde{h}_t}{\partial h_{t-1}}$.

Consequently: $\lim_{z \to 1} J_t = \mathbf{I} + \mathbf{0} + \mathbf{0} \implies J_t \approx \mathbf{I}$. Since the eigenvalues of the identity matrix are all 1, the spectral radius is $\rho(J_t) = 1$. □

Proposition F.1 gives more insights into the role of the gate z_t. Rather than simply selecting information, it acts as a structural stabilizer for the learning process. By explicitly controlling the decay rate of the hidden state, z_t maintains the spectral radius of the recurrence dynamics near unity. This allows gradients to propagate through long sequences without vanishing, while the strict boundedness of the gate prevents the instability associated with exploding gradients.

Remark F.2 (Relaxation to Continuous Regime). In practice, the gate z_t is modeled via a sigmoid function and takes values in $[0, 1]$. While it cannot reach exactly $\mathbf{1}$, the network can learn weights such that $z_t \geq \mathbf{1} - \epsilon$ for an arbitrarily small $\epsilon > 0$. In this regime, the gradient magnitude at step T scales as $(1 - \epsilon)^T$. Provided that $\epsilon \ll 1/T$, the signal degradation is negligible over the task's reasoning horizon. Thus, the continuous gate achieves *Effective Spectral Regulation*, approximating the ideal stability derived in Proposition F.1.

Our architecture establishes a retrieval hierarchy that decouples positional addressing from feature extraction. Because the state matrix H maintains the history of processed tokens as discrete columns $\{h_1, \ldots, h_L\}$, the model effectively possesses random access to the sequence axis. The attention mechanism first employs the Value projection W_V to address the Disentanglement problem: since each token vector encodes multiple attributes in linear superposition, W_V acts as a spectral filter, to isolate the specific feature subspace required for the current computation, ensuring that only the relevant signal is propagated while orthogonal interference is suppressed.

F.2 Gradient Superhighway

Stable gradient flow is essential for effectively optimizing the early loop iterations, so our architecture is designed to avoid both vanishing and exploding gradients. To analyze this behavior, we examine how the loss $\mathcal{L}$ backpropagates to the initial state $h_0^{(l)}$ across the T recurrent updates. By the chain rule, the gradient decomposes into a product of Jacobians:

$$\frac{\partial \mathcal{L}}{\partial h_0^{(l)}} = \frac{\partial \mathcal{L}}{\partial h_T^{(l)}} \prod_{t=1}^{T} \frac{\partial h_t^{(l)}}{\partial h_{t-1}^{(l)}} = \frac{\partial \mathcal{L}}{\partial h_T^{(l)}} \prod_{t=1}^{T} J_t$$

Leveraging Proposition F.1, we observe that for latent dimensions in the shielding regime ($Z_t \approx \mathbf{1}$), the local Jacobian effectively acts as the identity operator ($J_t \approx \mathbf{1}$). This simplifies the product significantly:

$$\prod_{t=1}^{T} J_t \approx \prod_{t=1}^{T} \mathbf{1} = \mathbf{1}$$

Such a behavior establishes a *Gradient Superhighway*: a direct path that allows error signals to traverse arbitrary depths. In contrast to standard recurrent dynamics, where gradient norms typically

scale as $\mathcal{O}(\lambda^T)$—inevitably leading to exponential decay for spectral radii $|\lambda| < 1$—our architecture ensures that $\|\frac{\partial \mathcal{L}}{\partial h_0^{(l)}}\| \approx \|\frac{\partial \mathcal{L}}{\partial h_T^{(l)}}\|$. This structural stability effectively alleviates the vanishing gradient problem, enabling the optimization of deeper looped transformer models.

G EXISTING ASSETS

In this appendix, we provide a comprehensive overview of all assets used throughout this work, along with their corresponding licenses to ensure transparency and reproducibility. Specifically, we list the evaluation benchmarks (Table 12), the models considered in our comparisons (Table 13), the training datasets employed (Table 14), and the main codebases used in our implementation (Table 15). All resources are referenced with links to their original sources.

Table 12: The list of benchmarks.

Benchmarks	**Link**	**License**
MATH500 (Lightman et al., 2023)	HuggingFace	MIT
AIME 2024 (Mathematical Association of America, 2024)	HuggingFace	Copyright MAA
AIME 2025 (Mathematical Association of America, 2025)	HuggingFace	Copyright MAA
AIME 2026 (Mathematical Association of America, 2026)	HuggingFace	Apache 2.0
OlympiadBench (He et al., 2024)	HuggingFace	MIT
AMC 2023 (Mathematical Association of America, 2023)	HuggingFace	Copyright MAA
GPQA (Rein et al., 2023)	HuggingFace	CC BY 4.0
HLE (Phan et al., 2026)	HuggingFace	MIT
MMLU (Hendrycks et al., 2021a)	HuggingFace	MIT
HumanEval (Chen et al., 2021c)	HuggingFace	MIT

Table 13: The list of models.

Model	**Link**	**License**
Ouro-Thinking-1.4B (Zhu et al., 2025)	HuggingFace	Apache 2.0
Ouro-Thinking-2.6B (Zhu et al., 2025)	HuggingFace	Apache 2.0
Qwen3-1.7B (Yang et al., 2025)	HuggingFace	Apache 2.0
Gemma4-E2B (Google, 2024)	HuggingFace	Apache 2.0
Qwen3.5-2B (Team, 2026)	HuggingFace	Apache 2.0
DeepSeek-R1-1.5B (Guo et al., 2025)	HuggingFace	MIT

Table 14: The list of training datasets.

Dataset	**Link**	**License**
AceReason-1.1-SFT (Liu et al., 2025)	HuggingFace	cc-by-4.0
OpenThoughts3 (Guha et al., 2025)	HuggingFace	Apache 2.0

Table 15: The main codebases used.

Codebase	**Link**	**License**
Transformers (Wolf et al., 2020)	GitHub	Apache 2.0
TRL (von Werra et al., 2020)	GitHub	Apache 2.0
vLLM (Kwon et al., 2023)	GitHub	Apache 2.0
LightEval (Habib et al., 2023)	GitHub	MIT